

z/OS



Language Environment Vendor Interfaces

Note

Before using this information and the product it supports, read the information in "Notices" on page 877.

Contents

Figures xi

Tables xv

About this document xvii

Using your documentation xviii

How to read syntax diagrams xix

z/OS information xxi

How to send your comments to IBM xxiii

If you have a technical problem xxiii

Summary of changes xxv

**Part 1. Language Environment
vendor interfaces for AMODE 31 /
AMODE 24 applications 1**

**Chapter 1. Common interfaces and
conventions 3**

Common runtime environment 3

Library not all linkable 3

Reentrancy 3

Recursion 3

AMODE/RMODE 4

Member code AMODE restrictions 4

External names 4

General register usage at entry to callable services 4

General register usage at exit from callable

services 5

Floating-point register conventions 5

Access register conventions 5

Program mask conventions 5

Routine layout. 6

Prolog information blocks 10

Epilog code 31

Base locator table 31

CEEYEPAF — locates an XPLINK or

non-XPLINK entry point PPA1 and PPA2 from a

passed DSA 33

__ep_find () — returns the address of the entry

point of the function owning the dsa_p DSA 34

CEEYPPAF — locates a field in the PPA1

optional area based on a passed pointer to the

PPA1 36

Language Environment dynamic storage area —

non-XPLINK 37

Language Environment dynamic storage area —

XPLINK 39

Language Environment common anchor area 40

Language Environment enclave data block 61

Language Environment process control block 69

Language Environment region control block 73

Example of a condition information block 78

Example of a machine state block 81

Language Environment member list and event

handler. 83

Language Environment callable services calling

conventions 85

Callable services syntax declarations 85

Optional parameter support 86

Data type definitions 86

ENTRY variable 87

LABEL variable 88

Callable service example 88

Invoking a callable service from C/C++ 88

Chapter 2. CALL linkage conventions 91

Terminology 91

Standard CALL linkage conventions 92

Register usage 92

Stack format 92

CEEVGTUN — next available byte locator

service 98

CEEVSSEG — return the stack segment bounds 99

Standard save area 100

Argument list format. 101

FASTLINK CALL linkage conventions 102

Register usage 102

Stack frame mapping. 103

Argument list format. 107

Leaf routines 110

Code sequences 110

Extra Performance Linkage (XPLINK) CALL

linkage conventions 114

Register usage 115

Stack frame mapping. 115

**Chapter 3. Program initialization and
termination 137**

Initialization overview 137

Termination overview 138

Enclave termination 138

Process termination 139

Putting initialization/termination together. 139

Member interfaces for initialization 139

CEESTART 140

CEEFMAIN 145

CEEMAIN 145

CEESTART operation. 146

CEESIOP — set interrupt option service 146

Signature CSECT 147

CEEBETBL — Language Environment externals

table 148

Event handler routines 149

CEEBLLST — language list. 149

CEEINT interface 153

CEEBCRLM — cancel/release load module 155

CEEBSENM — set the enclave name	156
CEEBSRCM — set the enclave return code modifier	157
CEEPGFD — get function pointer	159
CEEPRFD — release function pointer	160
CEE3ADDM — add new members to the enclave	161
CEE3CRE — create enclave.	163
CEE3CSYS — creating nested enclave	167
CEE3MBR — member bootstrap routine	168
CEE3SRSA — set return save area	170
CEE3DDBC — set dummy DSA back chain	171
CEE3PLST — PLIST manipulation	171
CEEGIN — obtain the program's invocation name	172
CEERELU — RCB lookup	173
Member interfaces for termination	174
CEETREC — explicit termination through HLL constructs	174
CEETREN — terminate without raising T_I_S	175
CEEATTRM — register event handler	176
Termination sequence	176
Termination failures	176
T_I_S condition.	176
Member event codes for initialization and termination	177
Language Environment abend summary	179
CEECOPP — runtime option compiler service	179
Options processing event	182
User exits	182
CEEBSHL — exit from/re-entry to Language Environment shell.	182
Language Environment interface validation exit	183
Structure of the Language Environment interface validation exit	184
CEEXVSEL — high-level selection criteria	184
Language-specific interface validation exit.	187
Interface for preinitialization	193
CEEPIPI — invocation for subroutine by address	193
Preinitialization environment and system request block mode	196
Chapter 4. Storage management	201
Dynamic storage (heap) services	201
Storage model	201
CWI to the heap services	202
Process-level heap storage management	203
Region-level heap storage management.	204
CEEVGTSB — unconditional get heap below	206
CEEV#GTS — get heap storage	207
CEEV#FRS — free heap storage	208
CEEVHRPT — obtain dynamic heap storage report	209
User-created heap services	210
CEEVUHCR — create a heap using user-provided storage	210
CEEVUHGT — allocate storage from a user-created heap	211
CEEVUHFR — return storage to a user-created heap	212

CEEVUHRP — produce a storage report for a user-created heap	212
Vendor heap manager interface	213
Requirements from the vendor	213
What the vendor should know	213
Activating the vendor heap manager	215
__vhm_event() API	216
XPLINK DSA extension services	216
CEEVXPAL — XPLINK DSA extension.	217
__alcaxp() — XPLINK DSA extension (alloca)	217
XPLINK compatibility stack swapping services	218
CEEVROND — run on downward-growing stack	218
CEEVRONU — run on upward-growing stack	219
CEEVH2OS — XPLINK to OS linkage on upward-growing stack	220
__stack_info() - stack segment ranges	221
Saving the stack pointer.	223

Chapter 5. Condition representation 225

Condition representation model	225
Data objects	226
Condition token data type (CEECTOK).	226
Feedback code	230
CEEGETFB — Construct a condition token given a facility ID and a message number	230

Chapter 6. National language support and message handler 233

National language support	233
Introduction to Language Environment message services	234
MSGFILE — related CWIs	235
CEECLDS — close ddname.	235
CEEODMF — open an input ddname	235
CEEOPMF — open the MSGFILE ddname	236
CEEQDMF — query an input ddname	237
CEEQUMF — query the MSGFILE ddname	238
CEECHMF — change the MSGFILE ddname	239
Relationship between date/time and COUNTRY settings	239
Message handling services	240
CEECMIB — create a message insert area entry	240
CEEMFNDM — return the MIB address	241
CEE3SMO — suppress printing of messages	243
C/C++-specific vendor interfaces.	244
__cttbl() — returns address of _LC_ctype_t structure	244
ASCII/EBCDIC mixed mode support for enhanced ASCII C-RTL	244
__ae_thread_setmode() — set character mode: ASCII or EBCDIC	246
__ae_thread_swapmode() — swap character mode to ASCII or EBCDIC	247
__isASCII() — determine character mode: ASCII or EBCDIC	248
__ae_autoconvert_state() — returns automatic conversion state of thread	249

Chapter 7. Condition management 251

Compiler-writer interfaces (CWIs)	251
CEE3ERP — support for user-provided error recovery	251
CEE3RSUM — resume an interrupted program	252
CEE3SGLN — signal invalid resume request	255
CEE3SGLT — signal a condition and terminate	256
CEE3SMS — set machine state	257
CEE3SMS2 — set machine state 2	259
CEEGOTO — restart execution at specified label	261
CEEHDHDL — register an event handler for stack frame zero processing	265
CEEMRCM — move the resume cursor	266
CEEYDSAF — find the previous DSA	267
__dsa_prev() — chain back to previous DSA	269
__far_jump() — perform far jump (C/C++ and XPLINK only)	272
__set_stack_softlimit() — set stack soft limit (C/C++ and XPLINK only).	274
Other Language Environment routines and handlers	275
Interface to the language-specific handlers.	275
DSA exit routines	276
Shunt routine	277
Attention handling	279
Error processing	280
Other Language Environment condition manager topics	283
Language Environment condition information block	283
Errors during condition handling.	283
HLL conventions and information	285
HLL condition handling conventions	285
HLL condition handling information	286
Language Environment-issued abends	286

Chapter 8. Program management 289

Loading and deleting programs in different environments	289
CWI to program management process services	290
CEEZLOD — process load service	290
CEEZDEL — process delete service	290
CWI to program management region services	291
CEEZLODR — region load service	291
CEEZDELR — region delete service	292
CWI to program management enclave services	292
CEEPLDOD — enclave level load service	293
CEEPLDOD2 — enclave/thread level load service	294
CEEPDEL — enclave level delete service	296
CEEPDEL2 — enclave level delete service	297
CEEPQLD — return information about loaded module	298
CEEPCB_DELETE — system dependent delete service.	299
CEEPCB_LOAD — system dependent load service.	300
CEEPLDODT — thread level load service	301
CEEPDEL2 — thread level delete service	303
Library subroutine access	303
LIBVECs	304
LIBPACKS	305

LIBVEC descriptor (LVD)	306
LIBVEC initialization	308
CWI to LIBVEC low-level services	309
CEEPLVI — LIBVEC initialization	309
CEEPLVE — verify load/delete	310
CEEPLVT — LIBVEC termination	311
CEEPPPOS — program object services	312
CWIs for explicit DLL reference	316
CEEPLDE — load DLL	316
CEEPFDE — DLL free	318
CEEPQDF — query DLL function	319
CEEPQDV — query DLL variable	320
CWIs for implicit DLL reference	322
CEETLOC — stub for trigger load on call	323
CEETHLOC — stub for trigger load on XPLINK call by name.	324
FDCB — function descriptor control block.	325
__bldxfd() — build an XPLINK compatibility descriptor	326
CEETLOR — stub for trigger load on reference	328
VDCB — variable descriptor control block.	329
CEETGTFN — stub for function invocation of old code	330
CWIs to find the writable static area (WSA)	331
CEEPFWSA — find writable static area (WSA)	331
__fnwsa() — CWI to find a writable static area	332
__static_reinit() — CWI to reinitialize writable static area	334
CEEDLLF — DLL failure control block	335

Chapter 9. Debugging and performance analysis 339

Language Environment-provided CWIs for the debug tool	339
__setHookEvents() — specify execute hook events for target process.	339
CEE3CBTS — pass component broker connector parameters	342
CEEBFBC — build feedback code routine	344
CEEKRGPM — register pattern match routine	345
CEEQFBC — query feedback code routine.	347
CEEQLDOD — query modules loaded with enclave level load service	348
CEETGCAA — get next CAA pointer	350
CEETSFB — translate standard feedback token	350
CEETSFC — translate standard feedback code	351
Debug tool-provided event handlers.	352
Debug tool event handler	352
Language Environment actions for the interactive debug tool	359
Language Environment interactive debug data areas	359
Execute hook support	360
Performance analysis support	361
Profile tool event handler	361
Language Environment actions for profiler	364

Chapter 10. DFSORT interface 365

DFSORT interface description	365
CEE3SRT — call DFSORT	365

ILC within SORT exits	367
Error handling within SORT exits	367
Messages and conditions	367

Chapter 11. Math library 369

Calling math services from an application	369
Math service condition handling requirements	369
Member-specific condition handling	370
Data types and their abbreviations	370
CWI conventions for scalar math services	370
Register interface	371
Conventional interface	371
Condition token values for math services	372
Math services	372
Scalar math services	373
Degree input/output trigonometry functions	380
Entry point names for scalar bit manipulation routines	381
Message ID — message text for math library	383
Language Environment math services — value of inserts	384
Language Environment conversion services	386
Terminology	386
CEEYCVHE — E-format output conversion routine	387
CEEYCVHF — F-format output conversion routine	389
CEEYCVHI — decimal to float input conversion routine	394

Chapter 12. Dump and tracing services 399

Dump services	399
CEE3DMP — runtime environment dump service	401
CEESDMP — symbolic dump of a routine	401
CEETRCB — traceback utility	401
CEETBCK — traceback utility (replaces CEETRCB)	404
Member language dump exit	414
CEELDMP — single line message dump service	414
CEEVDMP — variable dump service	415
CEEHDMP — hexadecimal storage dump service	418
CEEBDMP — control block dump service	419
Other dump-related CWIs	422
CEE3CDO — check dump options	422
CEEKSNP — produce a SNAP dump	423
CEEURTB — produce a user routine traceback	424
Tracing services	426
Global and member-specific tracing	427
CEEKCTRC — add a trace table entry	428

Chapter 13. Subsystem considerations 431

CICS and POSIX	431
Background information	431
Terminology	431
Running a program under CICS	433

Language Environment-CICS and Language Environment-batch program models	434
Language Environment-CICS interface	436
Languages supported	436
Extended runtime language interface	437
Flowchart of activities	440
Language Environment-CICS interface routines' DSA	441
Partition initialization (Language Environment enablement)	441
Partition termination (Language Environment disablement)	444
Establish ownership type call	445
Thread initialization	449
Thread termination	451
Run unit (program) initialization	451
Run unit (program) termination	456
Run unit (program) begin invocation	457
Run unit (program) end invocation	459
Error recovery	467
Determine working storage and static storage	468
Perform GOTO call	469
CEECTCB — set TCB+X'144' routine	471
CEECCICS — partition initialization changes	472
IMS considerations	472
IMS to Language Environment	472
Language Environment to IMS — CEETDLI	473
Implementation	474

Chapter 14. Anchor support 475

Anchor service	475
Fetch the anchor routine	475
Set the anchor routine	476
CEEARLU — anchor lookup	477
Anchor considerations	477
Bypassing anchor lookup, set, or reset	478

Chapter 15. Member language information 479

OS services — restricted use	479
Structure of executable programs	480
Central control blocks	480
Event handler	481
Event handler calls	481
Event code 1 — handle condition represented by the CIB event	482
Event code 2 — perform enablement for this stack frame event	483
Event code 3 — handle condition according to language defaults event	485
Event code 4 — runtime options event	486
Event code 5 — main-opts event	487
Event code 6 — event handler utilities event	487
Event code 7 — dump event handler event	492
Event code 8 — new load module event	495
Event code 9 — new condition event	496
Event code 10 — resume from a condition handler event	497
Event code 11 — DSA exit routines event	498

Event code 12 — national language change event	499
Event code 13 — country code change event	499
Event code 14 — main routine invocation event	500
Event code 15 — atterm event	501
Event code 16 — Debug Tool event	502
Event code 17 — process initialization event	502
Event code 18 — enclave initialization event	503
Event code 19 — enclave termination event	506
Event code 20 — query/build feedback code event	507
Event code 21 — process termination event	508
Event code 22 — DLL initialization event	509
Event code 23 — stack frame zero processing event	510
Event code 24 — POSIX events event	511
Event code 25 — static object constructor event	514
Event code 26 — region initialization event	516
Event code 27 — region termination event	517
Event code 28 — identify module entry point event	517
Event code 29 — determine enclave work area lengths event	518
Event code 31 — determine working storage (CICS only) event	519
Event code 32 — perform GOTO validation (CICS only) event	520
Event code 33 — member needs options processing event	520
Event code 34 — command line equivalent event	521
Event code 35 — default options event	522
Event code 36 — static destructor event	522
Event code 37 — preallocated storage event	523
Event code 38 — normal resume in DSA event	524
Event code 39 — interrupt received event	525
Event code 40 — get/release function pointer event	527
Event code 41 — cancel/release load module event	528
Event code 42 — automatic destructor event	529
Event code 44 — member program mask event	530

Chapter 16. z/OS UNIX System Services support 533

Thread management functions.	533
CEEOPAI.	533
CEEOPAD	534
CEEOPAGD	535
CEEOPAGS	535
CEEOPAGW	536
CEEOPASD	537
CEEOPASS	538
CEEOPASW	539
CEEOPC	540
CEEOPE	541
CEEOPEQ	542
CEEOPJ	543
CEEOPQ	544
CEEOPS	545
Signal handling CWIs	546

CEEOKILL	546
Thread keyed data CWIs	548
CEEOPGS	548
CEEOPKC	549
CEEOPKD	550
CEEOPSS.	551
Thread cancellation CWIs	552
CEEOPCPO	552
CEEOPCPU	553
Thread synchronization — mutex and read-write locks	555
CEEOPMD	555
CEEOPMI	557
CEEOPML	559
CEEOPML2	561
CEEOPMT	562
CEEOPMU	563
CEEOPMU2	564
CEEOPRL	565
CEEOPRL2	567
CEEOPRT	567
CEEOPRU	569
CEEOPRU2	570
CEEOPWL	571
CEEOPWL2	573
CEEOPWT	573
CEEOPXD	575
CEEOPXG	576
CEEOPXI.	578
CEEOPXS	580
Thread synchronization — condition variables	582
CEEOPCB	582
CEEOPCD	584
CEEOPCI.	585
CEEOPCS	586
CEEOPCT	587
CEEOPCW	590
CEEOPDD	592
CEEOPDG	593
CEEOPDI	594
CEEOPDS	595
Process control functions support.	597
CEEOEXEC	597
CEEOFORK	599
CEEOSPWN.	601
Miscellaneous utilities	604
CEEOEXIT	604
CEEOXEXE	604
Support for POSIX functions getenv(), setenv(), and clearenv().	605
Errors	605
CEEBENV	605

Chapter 17. COBOL-specific vendor interfaces 609

ILBOLLDX — OS/VS COBOL library load/delete exit.	609
IGZCXCC — COBOL call/cancel routine	611
IGZXAPI — COBOL file and runtime information query routine	613
IGZCXSF — COBOL extract side file routine	617

Chapter 18. PL/I-specific vendor interfaces	621
IBMPXSF — PL/I extract side file routine	621

Chapter 19. C/C++ special purpose interfaces for IEEE floating-point **625**

IEEE binary floating-point introduction	625
IEEE decimal floating-point introduction	626
Selection of fdlibm or fdlibm replacement functions	626
IEEE floating-point functions	627
__chkbfp() — check IEEE facilities usage	627
__fp_btoh() — convert from IEEE floating-point to hexadecimal floating-point	628
__fp_cast() — cast between floating-point data types	629
__fp_hctob() — convert from hexadecimal floating-point to IEEE floating-point	630
__fp_level() — determine type of IEEE facilities available	631
__fp_read_rnd() — determine rounding mode	631
__fp_setmode() — set IEEE or hexadecimal mode	632
__fp_swapmode() — set IEEE or hexadecimal mode	633
__fp_swap_rnd() — swap rounding mode	634
__fpc_rd() — read floating-point control register	635
__fpc_rs() — read floating-point control register and change rounding mode field	636
__fpc_rw() — read and write the floating-point control register	637
__fpc_sm() — set floating-point control register rounding mode field	638
__fpc_wr() — write the floating-point control register	639
__isBFP() — determine application floating-point mode	639
__to_xx() — C/C++ compiler casting support	640

Part 2. Language Environment vendor interfaces for AMODE 64 applications **645**

Chapter 20. Common interfaces and conventions for AMODE 64 applications **647**

Common runtime environment	647
Library not all linkable	647
Reentrancy	647
Recursion	647
AMODE/RMODE	647
Member code AMODE restrictions	647
External names	647
Routine layout	648
Prolog information blocks	650
Language Environment dynamic storage area	661
Language Environment control block mappings	662
Language Environment library anchor area	662
Language Environment library control area	664

Language Environment common anchor area	666
Language Environment debugger interfaces area	668
Language Environment enclave data block	671
Language Environment process control block	672
Language Environment region control block	673

Chapter 21. Compiler-writer interfaces (CWIs) supported for AMODE 64 applications **675**

Chapter 22. CALL linkage convention for AMODE 64 applications **677**

Terminology	677
XPLINK CALL linkage conventions for AMODE 64 applications	678
Register usage and linkage	678
Stack format	679

Chapter 23. Program initialization and termination for AMODE 64 applications **697**

Initialization overview	697
Termination overview	698
Enclave termination	698
Process termination	698
Putting initialization and termination together	698
Member interfaces for initialization	699
CELQSTRT	699
CELQMAIN	701
CELQFMAN	702
CELBST operation	702
CELQETBL — Language Environment externals table	703
CELQLLST — Language Environment language list	704
Signature CSECT	705
Initialization parameter list	705
Member interfaces for termination	707
CEECOPP — Runtime Option Compiler Service	708

Chapter 24. Storage management for AMODE 64 applications **711**

Vendor heap manager interface for AMODE 64 applications	712
Requirements from the vendor	712
Support provided for the vendor heap manager interface	712
Activating the vendor heap manager	713
__vhm_event()	713
__alcxp() — AMODE 64 DSA extension (alloca)	714
Memory object dump priority	715
Memory object user tokens	715
Saving the stack pointer	715

Chapter 25. Condition representation for AMODE 64 applications **717**

Condition representation model	717
Data objects	718

Condition token data type	718
Feedback code	721
Chapter 26. National language support and message services for AMODE 64 applications	723
National language support	723
Language Environment message services	723
C/C++-specific vendor interfaces	724
Chapter 27. Condition management for AMODE 64 applications	725
Application programming interfaces (APIs)	725
__dsa_prev() — Chain back to previous DSA	725
__ep_find() — returns the address of the entry point of the function owning the dsa_p DSA	728
__far_jump() — Perform far jump	730
Language Environment shunt routine for AMODE 64 applications	732
Establishing a program interrupt shunt service	732
Other Language Environment condition manager topics	733
Language Environment condition information block	733
Errors during condition handling	733
Language Environment-issued abends	734
Chapter 28. Debugging and performance analysis for AMODE 64 applications	735
Language Environment-provided functions for the debug tool	735
__le_debug_set_resume_mch() — set resume machine state	735
__setHookEvents() — specify execute hook events for target process	736
Debug tool-provided event handlers	739
Debug tool event handler	739
Language Environment actions for the interactive debug tool	745
Language Environment interactive debug data areas	745
Execute hook support	745
Performance analysis support	746
Profile tool event handler	746
Language Environment actions for profiler	749
Chapter 29. Anchor support for AMODE 64 applications	751
Chapter 30. Preinitialized Environments for Authorized Programs for AMODE 64 applications	753
Creating Preinitialized Environments for Authorized Programs	753
Creating a user-managed environment	754
Creating a system-managed environment	754

Preinitialized Environments for Authorized Programs tasks	755
Executing a routine in Preinitialized Environments for Authorized Programs	755
Calling a main routine	756
Calling a subroutine	756
Using runtime options	756
Selecting an environment	757
Providing recovery	757
Terminating Preinitialized Environments for Authorized Programs	757
Examples of using Preinitialized Environments for Authorized Programs	757
Using Preinitialized Environments for Authorized Programs in service request block (SRB) mode	758
Using Preinitialized Environments for Authorized Programs in cross-memory mode	758
CELAAUTH macro	759
CELAAUTH environments	759
Syntax for REQUEST=USERINIT	761
Syntax for REQUEST=USERCALL	765
Syntax for REQUEST=USERTERM	770
Syntax for REQUEST=MNGDINIT	772
Syntax for REQUEST=MNGDCALL	778
Syntax for REQUEST=MNGDUPDT	783
Syntax for REQUEST=MNGDTERM	786
CELAAUTH general notes	788
ABEND codes	789
Return and reason codes	789

Part 3. Appendixes 809

Appendix A. Options control block and supplementary options control block 811

Options control block	811
Supplementary options control block	857

Appendix B. CALL linkage argument examples 861

FASTLINK CALL linkage argument examples	861
XPLINK CALL linkage argument examples	866

Appendix C. Accessibility 873

Using assistive technologies	873
Keyboard navigation of the user interface	873
Dotted decimal syntax diagrams	873

Notices 877

Policy for unsupported hardware	878
Minimum supported hardware	879
Permission Notice	879
Programming interface information	879
Trademarks	880

Index 881

Figures

1. Layout entry of Language Environment-conforming routines – standard	7	40. S/370 Argument/parameter list format	101
2. Layout entry of Language Environment-conforming routines – FASTLINK	7	41. Typical FASTLINK stack frame storage map	103
3. Layout entry of C/370-conforming routines	8	42. Argument list passed on a procedure call	105
4. Layout entry of Language Environment-conforming routines – XPLINK	8	43. Stack segment showing FASTLINK frames	107
5. XPLINK stack extension marker	9	44. FASTLINK to FASTLINK linkage code sequence, non-Sleaf routine	111
6. XPLINK end of data marker	10	45. FASTLINK to FASTLINK linkage code sequence, Sleaf routine	113
7. XPLINK stub entry marker	10	46. Language Environment XPLINK stack storage model	115
8. Prolog constants format – level 1 (standard)	12	47. Language Environment XPLINK stack frame layout in a non-64-bit environment	116
9. Prolog constants format – level 2 (FASTLINK)	13	48. XPLINK function layout	129
10. Prolog constants format – level 3 (IEEE floating-point).	14	49. Format of CEEFMAIN	145
11. Compilation flag bits	15	50. Format of CEEMAIN	145
12. Language Environment PPA1 offset X'02'	15	51. Signature CSECT format	148
13. Language Environment PPA1 flag 2 offset X'18'	16	52. CEEBETBL CSECT format	148
14. Language Environment PPA1 flag 3 offset X'1C'	18	53. CEEBLLST format	150
15. Language Environment-enabled language member identifiers	19	54. Format of the initialization parameter list	151
16. Prolog constants format – level 4 (XPLINK), PPA1: entry point block (Version 3)	20	55. Updated format of the initialization parameter list	152
17. PPA1: XPLINK entry point block fixed area (Version 3) details	20	56. Runtime options error table	182
18. Language Environment PPA1 flag 1 offset X'08'	21	57. Language-specific interface validation exit	192
19. Language Environment PPA1 flag 2 offset X'09'	22	58. Preinitialization environment initialization exceprtn flow (task mode)	198
20. Language Environment PPA1 flag 3 offset X'0A'	22	59. Preinitialization environment exception flow (SRB mode)	199
21. Language Environment PPA1 flag 4 offset X'0B'	23	60. Language Environment Condition token (CEETOK)	227
22. Language Environment PPA1 flag word as defined by C++	25	61. Condition token.	228
23. Prolog constants format – level 4 (64-bit XPLINK), PPA2: compile unit block	28	62. Message numbers assigned to the days of the week and months	240
24. Level 4 (XPLINK), PPA2: compile unit block bits	28	63. Access to message insert information	243
25. Timestamp and version information	29	64. Format of a non-XPLINK label variable	262
26. Language Environment Dynamic storage area – non-XPLINK format	38	65. XPLINK extended format label variable	262
27. Language Environment Dynamic storage area – XPLINK format	39	66. XPLINK extended format label variable – resume area	263
28. Member list format	84	67. Library subroutine access table (LIBVEC)	304
29. Format of an entry variable	87	68. Partial LIBPACK CSECT definition	306
30. Example: calling CEETBCK from C	89	69. LIBVEC descriptor	307
31. CALL terminology refresher	91	70. CEETLOC stub for trigger load on call	323
32. Language Environment Non-XPLINK stack storage model.	92	71. CEETLOC stub for trigger load on XPLINK call by name	324
33. DSA allocation, user stack.	94	72. Function descriptor control block (FDCB) format	325
34. DSA extension, user stack.	94	73. CEETLOR stub	328
35. Free a DSA extension using saved NAB value	95	74. Variable descriptor control block (VDCB) format	329
36. DSA allocation in user stack when R13 does not address a Language Environment DSA	96	75. CEETGTFN stub	330
37. DSA allocation, library stack	97	76. Example of using <code>__static_reinit</code>	335
38. DSA return, library stack	98	77. Load list layout	349
39. Get next available byte in user stack	99	78. DFSORT's extended parameter list	366
		79. HLL CWI parameter list format	372
		80. Condition token values for math services	372
		81. Examples of E-format output conversions	389
		82. Examples of F-format output conversions	394

83. Examples of input conversions.	395	122. Prolog constants format – level 4 (XPLINK), PPA1: entry point block (Version 3)	651
84. Examples of input conversions with feedback indicated	397	123. PPA1: XPLINK entry point block fixed area (Version 3) details	652
85. Transferring control between application program, member language library, and Language Environment	400	124. Language Environment PPA1 flag 1 offset 'X'08'	653
86. CEEVDMP output format	417	125. Language Environment PPA1 flag 2 offset 'X'09'	654
87. CEEHDMP output format	419	126. Language Environment PPA1 flag 3 offset 'X'0A'	654
88. CEEBDMP output format	421	127. Language Environment PPA1 flag 4 offset 'X'0B'	655
89. Example: calling the CEEKCTRC CWI from the C RTL.	429	128. Language Environment PPA1 flag word as defined by C/C++	658
90. Language Environment-CICS and Language Environment-batch program model	435	129. Prolog constants format – level 4 (64-bit XPLINK), PPA2: compile unit block	659
91. CICS call argument list	439	130. Level 4 (XPLINK), PPA2: compile unit block bits	659
92. Language Environment-CICS run unit initialization, invocation, and termination	440	131. Timestamp and version information	660
93. Language Environment-CICS run unit termination	441	132. Language Environment dynamic storage area – XPLINK format for AMODE 64 applications	661
94. Structure of interface flags field	443	133. Library anchor area (LAA) field descriptions	664
95. Structure of information supplied to CICS by Language Environment for PGMINFO1	447	134. Library control area (LCA) field descriptions	665
96. Structure of information supplied to CICS by Language Environment for STATUSFLAGS	450	135. Library control area (LCA) field descriptions (cross reference).	666
97. Structure of information supplied to Language Environment by CICS	453	136. Common anchor area (CAA) field descriptions (cross references) AMODE 64	668
98. Structure of standard I/O information provided to Language Environment by CICS	455	137. Enclave data block (EDB) field descriptions (AMODE 64).	672
99. Run information supplied to Language Environment by CICS.	455	138. Enclave data block (EDB) field descriptions (cross reference).	672
100. Termination information supplied from CICS to Language Environment	456	139. Process control block (PCB) field descriptions (AMODE 64).	673
101. Structure of information supplied to Language Environment by CICS	458	140. Process control block (PCB) field descriptions (cross reference).	673
102. Program termination block (PTB) declaration	462	141. Region control block (RCB) field descriptions	674
103. Lang bit definition for CEECICS (60)	469	142. Region control block (RCB) field descriptions (cross reference).	674
104. Lang bit definition for CEECICS (70)	470	143. CALL terminology refresher	677
105. Lang bit definition for GOTOFLGS	471	144. Language Environment AMODE 64 XPLINK stack storage model	679
106. IMS parameter list format	473	145. Language Environment XPLINK stack frame layout for AMODE 64 applications	680
107. Structure of the Language Environment anchor vector	475	146. CELQETBL CSECT format	703
108. Example of code to set and obtain the Language Environment anchor.	478	147. Signature CSECT format	705
109. Syntax by function_code	494	148. Format of the initialization parameter list for AMODE 64 applications	707
110. Syntax by function_code	507	149. Runtime options error table (64-bit)	711
111. Condition qualifying data returned by CEEOKILL CWI	547	150. Language Environment Condition token for AMODE 64 applications	718
112. Link edit information to enable ILBOLLDX in ILBONTR	610	151. Condition token for AMODE 64 applications	719
113. Link edit information to enable ILBOLLDX in ILBOSRV	610	152. Using Preinitialized Environments for Authorized Programs in SRB mode	758
114. Link edit information to enable ILBOLLDX in ILBOSTT	610	153. Using Preinitialized Environments for Authorized Programs in cross-memory mode	759
115. Structure for CALL with name.	612	154. Options control block (OCB) field descriptions (Part 1)	812
116. Structure for CANCEL with name	612	155. Options control block (OCB) field descriptions (Part 2)	813
117. Structure for the block statement	624	156. Options control block (OCB) field descriptions (Part 3)	814
118. Layout entry of Language Environment-conforming routines – XPLINK	648		
119. XPLINK stack extension marker	649		
120. XPLINK end of data marker	649		
121. XPLINK stub entry marker	650		

157. Options control block (OCB) field descriptions (Part 4)	815	178. Options control block (OCB) field descriptions (Part 25)	836
158. Options control block (OCB) field descriptions (Part 5)	816	179. Options control block (OCB) field descriptions (Part 26)	837
159. Options control block (OCB) field descriptions (Part 6)	817	180. Options control block (OCB) field descriptions (Part 27)	838
160. Options control block (OCB) field descriptions (Part 7)	818	181. Options control block (OCB) field descriptions (Part 28)	839
161. Options control block (OCB) field descriptions (Part 8)	819	182. Options control block (OCB) field descriptions (Part 29)	840
162. Options control block (OCB) field descriptions (Part 9)	820	183. Options control block (OCB) field descriptions (cross references 1)	841
163. Options control block (OCB) field descriptions (Part 10)	821	184. Options control block (OCB) field descriptions (cross references 2)	842
164. Options control block (OCB) field descriptions (Part 11)	822	185. Options control block (OCB) field descriptions (cross references 3)	843
165. Options control block (OCB) field descriptions (Part 12)	823	186. Options control block (OCB) field descriptions (cross references 4)	844
166. Options control block (OCB) field descriptions (Part 13)	824	187. Options control block (OCB) field descriptions (cross references 5)	845
167. Options control block (OCB) field descriptions (Part 14)	825	188. Options control block (OCB) field descriptions (cross references 6)	846
168. Options control block (OCB) field descriptions (Part 15)	826	189. Options control block (OCB) field descriptions (cross references 7)	847
169. Options control block (OCB) field descriptions (Part 16)	827	190. Options control block (OCB) field descriptions (cross references 8)	848
170. Options control block (OCB) field descriptions (Part 17)	828	191. Options control block (OCB) field descriptions (cross references 9)	849
171. Options control block (OCB) field descriptions (Part 18)	829	192. Options control block (OCB) field descriptions (cross references 10)	850
172. Options control block (OCB) field descriptions (Part 19)	830	193. Options control block (OCB) field descriptions (cross references 11)	851
173. Options control block (OCB) field descriptions (Part 20)	831	194. Options control block (OCB) field descriptions (cross references 12)	852
174. Options control block (OCB) field descriptions (Part 21)	832	195. Options control block (OCB) field descriptions (cross references 13)	853
175. Options control block (OCB) field descriptions (Part 22)	833	196. Options control block (OCB) field descriptions (cross references 14)	854
176. Options control block (OCB) field descriptions (Part 23)	834	197. Options control block (OCB) field descriptions (cross references 15)	855
177. Options control block (OCB) field descriptions (Part 24)	835	198. Options control block (OCB) field descriptions (cross references 16)	856

Tables

1.	How to use z/OS Language Environment publications.	xviii	45.	Interface validation exit reference entry fields	189
2.	Syntax examples	xx	46.	Heap IDs recognized by Language Environment heap manager.	202
3.	Entry point types and the contents of the PPA2 field	11	47.	Routines using a parameter list interface	202
4.	COBOL V5 32-bit PPA3 layout	29	48.	CAA fields that contain information about abends	278
5.	C/C++ DWARF 32-bit PPA4 layout	29	49.	CEECIB State Variable, Constant values, and associated actions	284
6.	COBOL V5 32-bit PPA4 layout	30	50.	Format of the 31-Bit Language Environment CEEDLLF.	335
7.	Header layout for the base locator table	32	51.	Format of the 64-Bit Language Environment CEEDLLF.	336
8.	Entry layout for the base locator cells array	32	52.	List of CEEDLLF fields	336
9.	Common anchor area (CAA) field descriptions	40	53.	Debugger Language Environment event handler interface	353
10.	Common anchor area (CAA) constants	46	54.	Debugger Language Environment event handler bit mask descriptions	355
11.	Common anchor area (CAA) cross reference	46	55.	CWI CEE3CBTS event handler interface parameters	356
12.	Enclave data block (EDB) field descriptions	61	56.	Profile tool — Language Environment event handler interface	361
13.	Enclave data block (EDB) constants	64	57.	Data types and their abbreviations	370
14.	Enclave data block (EDB) cross reference	64	58.	CWI register interface format	371
15.	EDB field descriptions	66	59.	Result registers for scalar routines (CWI register interface)	371
16.	Process control block (PCB) field descriptions	69	60.	Language Environment Scalar math services	373
17.	Process control block (PCB) constants	70	61.	Degree input/output trigonometry functions	381
18.	Process control block (PCB) cross reference	71	62.	Language Environment Scalar bit manipulation routines.	381
19.	PCB field descriptions	72	63.	Math message_IDs.	383
20.	Region control block (RCB) field descriptions	74	64.	Language Environment Math services - value of inserts	384
21.	Region control block (RCB) constants	75	65.	Output-structure format	618
22.	Region control block (RCB) cross reference	75	66.	Entry format for the procedure table.	620
23.	RCB field descriptions	76	67.	Output-structure format	621
24.	Member list field descriptions	84	68.	Entry layout for statement table	624
25.	Data type definitions for callable services	86	69.	Arguments for __to_xx().	642
26.	Format of save area	103	70.	AMODE 64 entry points	648
27.	Format of linkage area	104	71.	C/C++ DWARF 64-bit PPA4 layout	660
28.	Content of XPLINK stack frame for non-AMODE 64 applications	117	72.	CWIs for AMODE 64 applications	675
29.	Prolog/epilog example: small size stack frame, no backchain, no alloca	118	73.	Content of XPLINK stack frame for AMODE 64 applications	681
30.	Prolog/epilog example: small size stack frame, varag, backchain	118	74.	Prolog/epilog example: small size (AMODE 64) stack frame	682
31.	Prolog/epilog example: intermediate size stack frame, no backchain, no alloca, no varargs.	119	75.	Prolog/epilog example: intermediate size (AMODE 64) stack frame	682
32.	Prolog/epilog example: large size stack frame (4096 ≤ dsasize ≤ 32768), AMODE 31.	119	76.	Prolog/epilog example: large size (AMODE 64) stack frame	682
33.	Prolog/epilog example: huge size stack frame (32768 < dsasize), AMODE 31	120	77.	Prolog/epilog example: XPLINK, no alloca, no storeargs, saves regs 5-9, DSA size=360256 (AMODE 64).	683
34.	Prolog/epilog example: XPLINK, no alloca, no storeargs, saves regs 5-9, DSA size=3712 (AMODE 31).	121	78.	Prolog/epilog example: changes needed to maintain addressability	685
35.	Entry point marker (type 1).	126	79.	Entry point marker (type 1) AMODE64	688
36.	Stack extension marker (type 2)	127	80.	Stack extension marker (type 2) AMODE64	689
37.	Data marker (type 3)	127	81.	Data marker (type 3) AMODE64	689
38.	Stub marker (type 4)	127			
39.	Contents of non-XPLINK CEESTART	141			
40.	Contents of XPLINK CEESTART	143			
41.	Bootstrap behavior.	146			
42.	CEEBETBL field descriptions	148			
43.	Unhandled condition behavior summary	165			
44.	Event codes called for initialization and termination	177			

82. Stub marker (type 4) AMODE64	689	89. Return and reason codes for the CELAAUTH macro	790
83. Contents of the CELQSTRT CSECT	700	90. Options control block (OCB) and supplementary options control block (SOCB) type field definitions	811
84. Bootstrap behavior.	702	91. Options control block (OCB) constants	856
85. CEECIB state variable, constant values, and associated actions for AMODE 64 applications	733	92. Supplementary options control block (SOCB) field descriptions	858
86. Debugger Language Environment event handler interface for AMODE 64 applications .	740	93. Supplementary options control block (SOCB) constants	859
87. Debugger Language Environment event handler bit mask descriptions for AMODE 64 applications	742	94. Supplementary options control block (SOCB) cross reference	859
88. Profile tool — Language Environment event handler interface for AMODE 64 applications .	747		

About this document

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

Language Environment is the prerequisite run-time environment for applications generated with the following IBM compiler products:

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

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

Language Environment supports, but is not required for, an interactive debug tool for debugging applications in your native z/OS environment.

Debug Tool is also available as a standalone product. Debug Tool Utilities and Advanced Functions is also available. For more information, see <http://www.ibm.com/software/awdtools/debugtool/>.

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

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

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

This book documents the set of low-level interfaces, or Compiler-Writer Interfaces (CWIs), that can be used between the common runtime component and C, C++, COBOL, Fortran, PL/I, and other member runtime components of Language Environment.

Note: Throughout this book there are descriptions of Language Environment messages. The text of these messages might not exactly match that produced by Language Environment. You can find the exact text of messages in *z/OS Language Environment Debugging Guide*.

Using your documentation

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

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

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

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

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

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

How to read syntax diagrams

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

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

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

Symbols

The following symbols may be displayed in syntax diagrams:

Symbol	Definition
--------	------------

- | | |
|-----|--|
| ▶— | Indicates the beginning of the syntax diagram. |
| —▶ | Indicates that the syntax diagram is continued to the next line. |
| ▶— | Indicates that the syntax is continued from the previous line. |
| —▶▶ | Indicates the end of the syntax diagram. |

Syntax items

Syntax diagrams contain many different items. Syntax items include:

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

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

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

Item type	Definition
-----------	------------

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

Optional

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

Default

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

Syntax examples

The following table provides syntax examples.

Table 2. Syntax examples

Item	Syntax example
Required item.	
Required items appear on the main path of the horizontal line. You must specify these items.	
Required choice.	
A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	
Optional item.	
Optional items appear below the main path of the horizontal line.	
Optional choice.	
An optional choice (two or more items) appears in a vertical stack below the main path of the horizontal line. You may choose one of the items in the stack.	
Default.	
Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.	
Variable.	
Variables appear in lowercase italics. They represent names or values.	
Repeatable item.	
An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.	
A character within the arrow means you must separate repeated items with that character.	
An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.	

Table 2. Syntax examples (continued)

Item	Syntax example
<p>Fragment.</p> <p>The fragment symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.</p>	

z/OS information

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

When possible, this information uses cross-document links that go directly to the topic in reference using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS[®], see *z/OS Information Roadmap*.

To find the complete z/OS library, including the z/OS Information Center, see z/OS Internet Library (<http://www.ibm.com/systems/z/os/zos/bkserv/>).

How to send your comments to IBM

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

Use one of the following methods to send your comments:

1. Send an email to mhvrcfs@us.ibm.com.
2. Send an email from the Contact z/OS.
3. Mail the comments to the following address:
IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
US
4. Fax the comments to us, as follows:
From the United States and Canada: 1+845+432-9405
From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:
z/OS V2R1.0 Language Environment Vendor Interfaces
SA38-0688-00
- The topic and page number that is related to your comment.
- The text of your comment.

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

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

If you have a technical problem

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

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at [IBM support portal](http://ibm.com/support).

Summary of changes

The following chapters contain new or updated content for COBOL V5:

- Chapter 1: Common interfaces and conventions
- Chapter 3: Program initialization and termination
- Chapter 17: COBOL specific vendor interfaces
- Chapter 20: Common interfaces and conventions for AMODE 64 applications

Part 1. Language Environment vendor interfaces for AMODE 31 / AMODE 24 applications

This part of the book does not apply to AMODE 64. For AMODE 64 information, see Part 2, "Language Environment vendor interfaces for AMODE 64 applications," on page 645.

Chapter 1. Common interfaces and conventions

This section describes the common runtime library components of Language Environment. The description indicates when a convention is mandatory. These conventions form the basis for a well-behaved application and enhance the ability to do interlanguage communication (ILC). Language Environment external names begin with the reserved prefix “CEE”.

Common runtime environment

A thread is represented by a Common Anchor Area (CAA). All thread- and enclave-related resources can be located either directly within the CAA or through the CAA.

An enclave is one or more executable programs that contain one or more separately-compiled bound procedures (also known as compilation units). The executable program that contains the main routine is known as the **root** load module. An enclave can consist of multiple executable programs when a dynamic call is run within the enclave. Fetch mechanisms, such as the C `fetch()` function, introduce a new executable program into the application. However, it typically behaves differently than dynamic calls in today's implementation, in so far as the scope of static external data is concerned. An executable program can exist in a variety of forms. It can be a mixture of an HLL or assembler procedure with Language Environment routines. It can also be a strictly Language Environment library module that does not contain any user-written code.

Situations exist where member subprograms are called from operating environment functions such as SORT, QMF™ or assembler language routines without Language Environment register conventions. Member languages must either disallow this form of specification, or be able to detect this form of access and perform whatever is necessary to re-establish the Language Environment environment.

Library not all linkable

Most Language Environment routines cannot be statically linked. In general, it is not possible to make a complete, self-contained module.

Reentrancy

All Language Environment library code is reentrant. All read/write areas are dynamically acquired from STACK or HEAP. Language Environment provides a reentrant environment for compiled code.

Recursion

All Language Environment-supplied library code can be called recursively. For example, if an interrupt occurs in a Language Environment routine and the exception is signaled to some other code (user, Language Environment, or language-specific), that code could, in turn, during its exception processing, use the function that originally caused the exception. This does not mean that the application itself is recursive.

Language Environment Conventions

Special handling of certain situations, such as short-on-storage conditions, cause recursive entry to be detected and handled appropriately.

AMODE/RMODE

Most Language Environment library routines are AMODE(31) RMODE(ANY). Library routines residing below the 16 MB line are AMODE(ANY) and RMODE(24). These switch to AMODE(24) if necessary and return to the entry AMODE before returning to the caller. HLLs participating in Language Environment and supporting dynamic loading of application programs are responsible for switching and restoring the AMODE between load module calls.

Member code AMODE restrictions

Language Environment can allocate any of its control blocks above the line. Any member code that accesses a Language Environment control block must run in AMODE(31) to have addressability to the control blocks.

External names

Language Environment supports external names such as files, programs, and data structures in the same manner as the host system. External names are limited to eight SBCS characters. No supported host system permits DBCS names.

Some languages permit longer names to be used when referring to externally named objects. In order to conform to the host system requirements, each language can use an algorithm to convert a long internal name to a shorter name that is acceptable to the host system.

Language Environment does not define a common naming convention or name conversion algorithm. Users are responsible for ensuring that names are not ambiguous when long names are converted. External and internal forms of names must match after conversion to a shorter form of the name.

General register usage at entry to callable services

The following registers must have the prescribed contents when control reaches the entry point of a Language Environment callable service. Calls that remain within the same language do not need to adhere to the register conventions described below. ILC calls might or might not adhere to these conventions depending upon the languages involved. A library routine that accommodates the differences in the linkage conventions can be used in some ILC cases.

- R0** Reserved
- R1** Must point to the parameter list or be zero if no parameter list exists
- R2–R11** Not referenced by Language Environment; caller's values are passed through transparently
- R12** Must point to the CAA upon entry to an external routine; R12 does not have to point to the CAA within a routine
- R13** Must point to the caller's DSA
- R14** The return address
- R15** The address of the called entry point

General register usage at exit from callable services

Registers have the following contents when control returns to the caller of the callable service.

R0 Not defined by Language Environment

R1 Not defined by Language Environment

R2–R11

Preserved

R12 Points to the CAA

R13 Points to the caller's DSA

R14 Not preserved

R15 Not preserved

Note:

1. The called procedure must ensure that R2 through R13 have the same values on exit as they had on entry.
2. The called procedure cannot rely upon the values contained in R0, R1, R14, and R15 unless explicitly stated by the interface.

Floating-point register conventions

No conventions have been defined for floating-point registers. The contents are neither saved nor restored by Language Environment, except by the exception handler when exceptions are raised. Intrinsic functions use these registers to return results. For more details, see “CWI conventions for scalar math services” on page 370.

Access register conventions

No conventions have been defined for access registers. Language Environment neither saves nor restores the contents of the access registers. Language Environment does not restrict exploitation of access registers in the future.

Program mask conventions

The maskable program exceptions are enabled for all member languages represented in the root or main load module during Language Environment initialization. Each member language informs Language Environment of its program mask requirements, and Language Environment ORs all of the requirements together and sets the program mask during initialization. During termination, the program mask is reset by Language Environment to its value upon entry to Language Environment initialization.

A language is represented in the load module by providing a load module signature CSECT for each compilation.

The CEE3SPM callable service is provided to query, save, restore, and modify the program mask setting. Users are responsible for managing program mask setting if they alter the program mask while the application is running. Altering the program mask might change some HLL semantics. Use caution when altering the program mask.

Language Environment neither saves nor restores the program mask setting across calls to Language Environment services or calls within the Language Environment environment.

Language Environment Conventions

The runtime option XUFLOW indicates the initial setting of the mask for exponent underflow. You can alter this setting by using the callable service CEE3SPM. (Note, however, that the use of CEE3SPM might alter some HLL semantics.) In summary, the program mask's initial setting is determined by the requirements of the members within the main load module and by the setting of the XUFLOW runtime option.

While the enclave is running, the program mask is influenced by the callable service, CEE3SPM, and by members' requirements that are newly-added as a result of a dynamic call or fetch; this is handled by the CWI service CEE3ADDM.

Routine layout

The following table shows the five types of entry points that Language Environment recognizes as Language Environment-conforming routines. The fifth type is an example of a nonconforming entry point that would be recognized by the member language.

Entry point type is...	If...
Language Environment-conforming	The entry point plus 4 is X'00C3C5C5'. For details, see Figure 1 on page 7.
Language Environment-conforming FASTLINK	The entry point plus 4 is X'01C3C5C5'. FASTLINK linkage conventions are used. For details, see Figure 2 on page 7.
Language Environment-conforming XPLINK	The entry point minus 16 is X'00C300C500C500F1'. XPLINK linkage conventions are used. For details, see Figure 4 on page 8.
C/370	The entry point plus 5 is X'CE'.
CEESTART CSECT	The entry point plus 28 is CL8'CEESTART'.
Nonconforming	The entry point is none of the above. Nonconforming entry points are for routines that follow the linking convention in which the name is at the beginning of the routine. X'47F0Fxxx' is the instruction to branch around the routine name.

FASTLINK supports an optimized linkage convention that reduces the total number of instructions for prolog and epilog sequences. XPLINK provides optimal performance for a certain class of applications. The layout entry for standard routines is shown in Figure 1 on page 7 and the layout entry for FASTLINK routines is shown in Figure 2 on page 7. The layout entry for standard and FASTLINK routines is defined by the field at offset X'04'; X'00' represents standard layouts and X'01' represents FASTLINK layouts.

Language Environment-Conforming Standard Routine Layout Entry

00	B 20(,R15)		Branch around constant areas
04	X'00'	CL3'CEE'	CEE eye catcher
08	Stack frame size for this routine		
0C	Offset to the PPA1 (signed) from routine start		
10	B 01(0, R15)		Disable the +16 entry point
14	Code to acquire a DSA		

Figure 1. Layout entry of Language Environment-conforming routines – standard

Language Environment-Conforming FASTLINK Routine Layout Entry

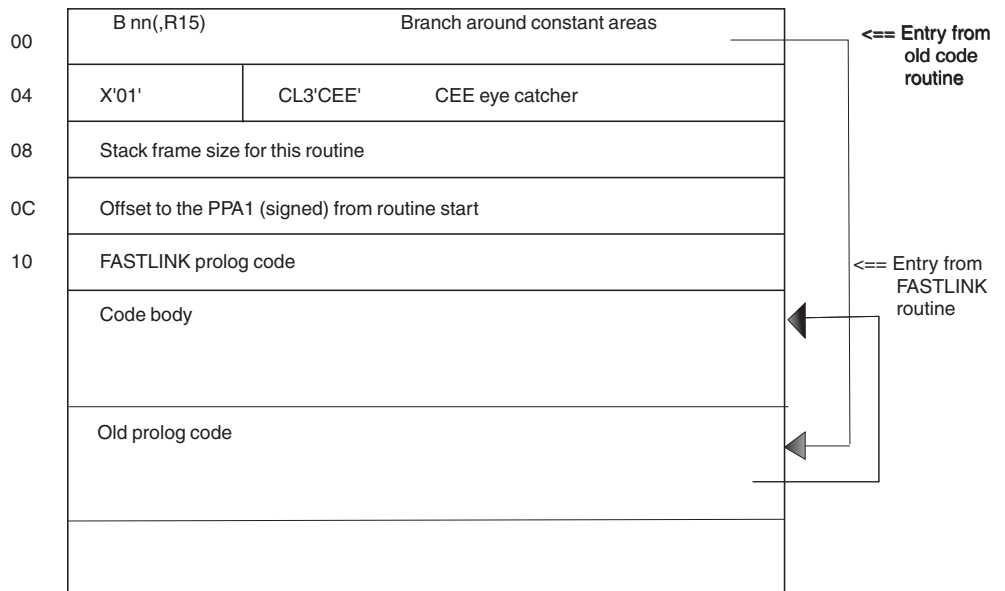


Figure 2. Layout entry of Language Environment-conforming routines – FASTLINK

Figure 3 on page 8 shows the entry point layout and Program Prolog Area-1 (PPA1) for C/370 routines; see “Prolog information blocks” on page 10 for more information about the PPA1 format.

Language Environment Conventions

C Routine Layout Entry and PPA1

00	B xxx(0,15) Branch around prolog data		
04	X'14' Offset to the name	X'CE' (Language Environment signature)	Language Environment Flags
08	A(PPA2)		
0C	A (PPA3) Zero if PPA3 is not available		
10	Stack frame size		
	⋮		
yy	Length of name	Untruncated entry/label name	

Figure 3. Layout entry of C/370-conforming routines

XPLINK data layouts

The layout entry for XPLINK routines is shown in Figure 4. The layout entry for XPLINK routines is defined by the Version field at offset X'00' in the PPA1; see Figure 16 on page 20.

-10	Eyecatcher (XL7'00C300C500C500')	Mark Type C'1'
-08	Offset to PPA1	
-04	DSA Size/32 (27 bits)	Entry Flags (5 bits)
+00	Prolog Code	
	...	

Figure 4. Layout entry of Language Environment-conforming routines – XPLINK

Field	Contents
Eyecatcher	Seven byte field containing the XPLINK eyecatcher, XL7'00C300C500C500'.
Mark Type	Field marking the type of code. Entry code is C'1'.
Offset to PPA1	A signed fullword representing the offset from the start of the entry marker to the start of the PPA1.
DSA Size/32	A 27-bit field representing the size of the routine's DSA in 32-byte increments.

Field	Contents
Entry Flags	A 5-bit field containing flag bits to identify the type of routine. If bit 1 is on, the routine is an XPLEAF routine; these routines save caller's registers in their own stack frame, but do not update the stack pointer. Bit 2 indicates if the routine uses the alloca() service.

The compiler emits an XPLINK stack extension marker in front of the call to Language Environment for the overflow prolog sequence for the +4K DSA scenario. Figure 5 depicts this marker.

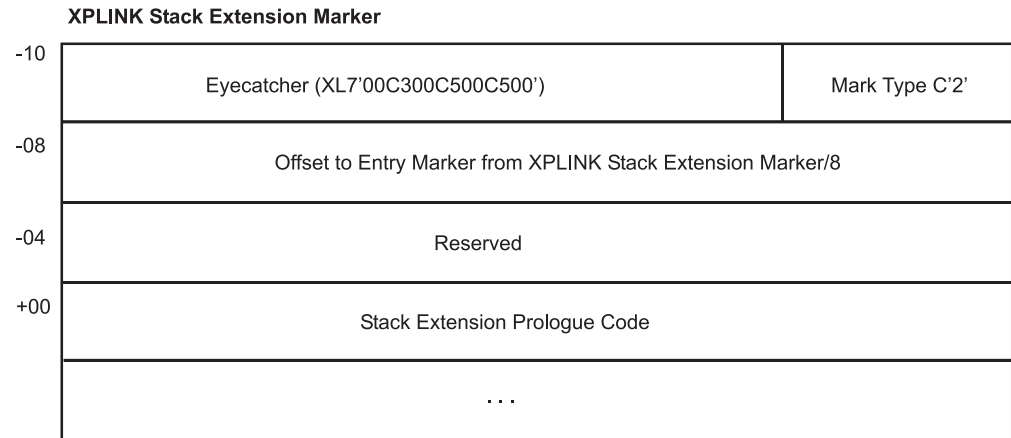


Figure 5. XPLINK stack extension marker

Field	Contents
Eyecatcher	Seven byte field containing the XPLINK eyecatcher, XL7'00C300C500C500'.
Mark type	Field marking the type of code. Entry code is C'2'.
Offset to entry marker from XPLINK stack extension marker/8	The signed offset from the start of the XPLINK stack extension marker to the start of the entry point marker in doublewords.

The XPLINK end of data marker is placed after, or at the end of a section of code, where the compiler may have placed constants. Language Environment's asynchronous signal deliverer uses this in its scan backwards to identify that a signal did not arrive inside a function's prolog. Figure 6 on page 10 depicts this marker.

Language Environment Conventions

XPLINK End of Data Marker

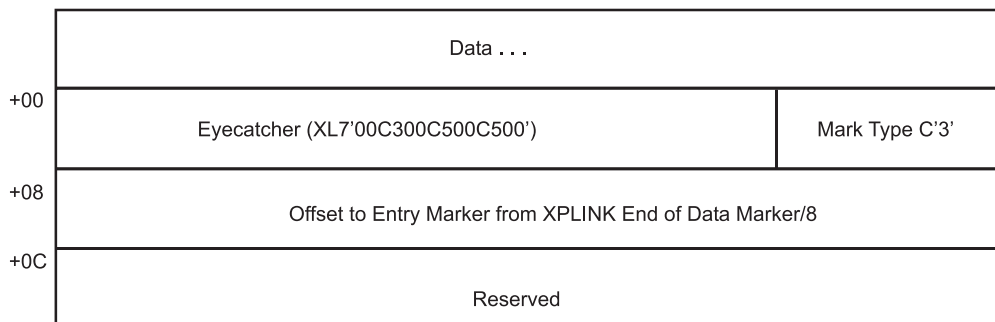


Figure 6. XPLINK end of data marker

Field	Contents
Eyecatcher	Seven byte field containing the XPLINK eyecatcher, XL7'00C300C500C500'.
Mark type	Field marking the type of code. Entry code is C'3'.
Offset to entry marker from XPLINK stack extension marker/8	The signed offset from the start of XPLINK end of data marker to the start of the entry point marker in doublewords.

Language Environment implements an 8-byte XPLINK stub entry marker for Language Environment and C runtime stubs. Figure 7 depicts this marker.

XPLINK Stub Entry Marker

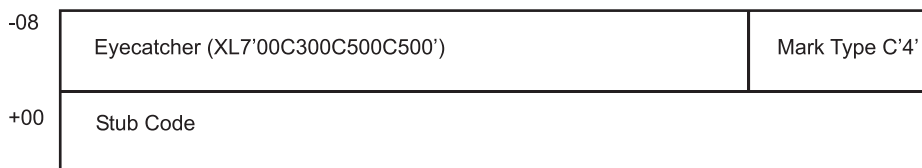


Figure 7. XPLINK stub entry marker

Field	Contents
Eyecatcher	Seven byte field containing the XPLINK eyecatcher, XL7'00C300C500C500'.
Mark type	Field marking the type of code. Entry code is C'4'.
Offset to entry marker from XPLINK stack extension marker/8	The signed offset from the start of XPLINK end of data marker to the start of the entry point marker in doublewords.

Prolog information blocks

The prolog information exists for every block or internal procedure. A block or internal procedure is found by R15 pointing to an area saved in the DSA. Code to allocate stack space is not required in the Language Environment prolog; see Figure 33 on page 94. Several prolog information blocks have been defined:

- the standard layout is defined in Figure 8 on page 12
- the FASTLINK layout is defined in Figure 9 on page 13

- the IEEE floating-point layout is defined in Figure 10 on page 14
- the XPLINK layout is defined in Figure 16 on page 20, Figure 17 on page 20, and Figure 23 on page 28

Program Prolog Area-1 (PPA1) appears for every Language Environment entry point. There is a one-to-one correlation between a PPA1 and a DSA. The length of the name offset field (PPA1 offset 00) ranges from 32 to 255 bytes. Note that for the FASTLINK version, the value in this field is the offset to the name length field, divided by 2; therefore, the value of the field may range from X'10' to X'FF'. An offset zero indicates that an entry name does not exist. A PL/I BEGIN block that does not contain a name is an example of offset zero in the PPA1 length field. The content of the entry/label name field is defined by member languages. The name can be SBCS characters or DBCS characters bracketed by shift-codes. Member-defined information can be placed starting at offset X'20'. Fields described as fullword offsets are treated as signed offsets.

Program Prolog Area-2 (PPA2) appears once for each compile unit and can immediately follow the primary PPA1. The control level field indicates the change level of the prolog. The timestamp and version information normally appears at the end of PPA2 and is optional. The version and release data fields identify the level of the compiler that produced the object code. You can use the PPA2 field at offset X'10' to determine the primary entry point for the compilation unit. It is zero if the compilation unit primary entry point does not exist. Member-defined information can be placed at the end of PPA2.

To establish the member language of a compile unit, use the PPA2 field at offset X'04' in the PPA1 to locate the PPA2. The meaning of the PPA2 field depends on the format of the PPA1. When the PPA1 format is not known, you can use the entry point layout to determine the program model (see "Routine layout" on page 6) and to interpret the content of the PPA1.

Table 3. Entry point types and the contents of the PPA2 field

Entry Point Layout Type	Contents of the PPA2 Field
Standard	Actual address of the PPA2
FASTLINK (includes IEEE floating point)	Signed offset to the PPA2 from the entry point
XPLINK	Signed offset to the PPA2 from the PPA1

When you have located the PPA2, you can find the one byte member language identifier at offset X'00' of the PPA2 (for example, '05' for COBOL, '10' for PL/I, '11' for Enterprise PL/I). For a complete list of identifiers, see Figure 15 on page 19. The PPA2 member identifier may be useful in determining the format of the corresponding PPA1.

Program Prolog Area-3 (PPA3), if available, appears once for every Language Environment entry point. It provides additional information about an entry point, and typically contains information relevant for problem determination tools. There is a one-to-one correlation between a PPA1 and a PPA3. The PPA3 layout may differ among different member languages.

Program Prolog Area-4 (PPA4), if available, appears once for each compilation unit. It provides additional information about a compilation unit, and typically contains information relevant for problem determination tools. There is a one-to-one correlation between a PPA2 and a PPA4. The PPA4 layout may differ among different member languages.

Language Environment Conventions

In the timestamp block, as shown in Figure 25 on page 29, the two characters that indicate the version are to be used at the discretion of the high level language that produces the block; they are not interrogated by Language Environment. In addition, the dump service uses the service level field to add the module service level information to the traceback.

Figure 8 shows the Language Environment-conforming prolog for standard routines.

PPA1: Entry Point Block

X'00'	Offset to the length of name	X'CE' (Lang Env Signature)	Lan Env Flags	Member flags
X'04'	Address of PPA2			
X'08'	Signed offset to PPA3 from the entry point. Zero if PPA3 is not available.			
X'0C'	Reserved			
X'10'	Reserved			
X'14'	Reserved			
X'18'	Reserved			
X'1C'	Language Environment flags (16 bits) - (not present for COBOL)			
	:			
	Length of name	Untruncated entry/label name		

PPA2: Compile Unit Block

X'00'	Member identifier	Member Subid	Member Defined	Control Level (= 1)
X'04'	V(CCEESTART) for load module			
X'08'	Signed offset from PPA2 to PPA4. Zero if PPA4 is not available.			
X'0C'	Signed offset from PPA2 to timestamp/version information. Zero if it is not available.			
X'10'	A(PEP) - address of the compilation unit's Primary Entry Point			
	:			

Figure 8. Prolog constants format – level 1 (standard)

Figure 9 on page 13 shows the Language Environment-supported prolog for FASTLINK routines.

PPA1: Entry Point Block

X'00'	Offset/2 to length of name	X'CE' (Lang Env Signature)	Lan Env flags	Member flags
X'04'	Signed offset to PPA2 from the entry point			
X'08'	Signed offset to PPA3 from the entry point. Zero if PPA3 is not available.			
X'0C'	Reserved			
X'10'	GPR save bit mask	Reserved - must be zero		
X'14'	Member PPA1 word			
X'18'	CEL flag 2	Max space used by nonleaf rtn in caller's DSA/8		
X'1C'	Language Environment flags (16 bits)			
	⋮			
	Length of name	Untruncated entry/label name		

PPA2: Compile Unit Block

X'00'	Member identifier	Member Subid	Member Defined	Control Level (= 2)
X'04'	Signed offset from PPA2 to CEESTART for load module			
X'08'	Signed offset from PPA2 to PPA4. Zero if PPA4 is not available.			
X'0C'	Signed offset from PPA2 to timestamp/version information. Zero if not available.			
X'10'	Signed offset from PPA2 to compilation unit Primary Entry Point			
	⋮			

Figure 9. Prolog constants format – level 2 (FASTLINK)

Figure 10 on page 14 shows the Language Environment-supported prolog for IEEE floating-point routines. The Member Subid (PPA2 offset X'01') is defined by the member language.

Language Environment Conventions

PPA1: Entry Point Block

X'00'	Offset/2 to length of name	X'CE' (Lang Env Signature)	Lang Env flags	Member flags
X'04'	Signed offset to PPA2 from the entry point			
X'08'	Signed offset to PPA3 from the entry point. Zero if PPA3 is not available.			
X'0C'	Pointer to entry point data descriptors			
X'10'	GPR save bit mask	Unsigned offset/16 to DSR of FPR8-15 save area	FPR save bit mask	
X'14'	Member PPA1 word			
X'18'	CEL flag 2	Max space used by nonleaf rtn in caller's DSA/8		
X'1C'	CEL flag 3	Unsigned offset/2 from PPA1 to code descriptor list		
	⋮			
	Length of name	Untruncated entry/label name		

PPA2: Compile Unit Block

+00	Member Identifier	Member Subid	Member Defined	Control Level (= 3)
+04	Signed offset from PPA2 to CEESTART for load module			
+08	Signed offset from PPA2 to PPA4. Zero if PPA4 is not available.			
+0C	Signed offset from PPA2 to timestamp/version information. Zero if it is not available.			
+10	Signed offset from PPA2 to the compilation unit's Primary Entry Point			
+14	Compilation flags			
			

Figure 10. Prolog constants format – level 3 (IEEE floating-point)

Compilation flag bits

Figure 11 on page 15 shows the compilation flag bits.


```
'0.....'B Indicates that program was compiled for hexadecimal floating-point
'1.....'B Indicates that program was compiled for binary floating-point
'.0.....'B Indicates that the code is compiler generated user code
'.1.....'B Indicates that the code is associated with library code
'..0.....'B Program does not contain service information
'..1.....'B Program contains service information
'...000...'B Reserved
'.....0...'B No additional compiler information after service information
'.....1...'B Additional compiler information after service information
'.....0 0...'B Reserved
'.....0.....'B MD5 signature is not located at 16 bytes before the timestamp
'.....1.....'B MD5 signature is located at 16 bytes before the timestamp
'......0.....'B Not compiled with FLOAT(AFP(VOLATILE))
'......1.....'B Compiled with FLOAT(AFP(VOLATILE))
'......00000 00000000 00000000'B Reserved
```

Figure 11. Compilation flag bits

Program flags — PPA1 offset X'02'

Language Environment program flags (PPA1 offset X'02') are shown in Figure 12 and are described following the figure.

```
'0.....'B Internal procedure
'1.....'B External procedure
'.0.....'B Primary entry point
'.1.....'B Secondary entry point
'..0.....'B This procedure/block does not have a DSA
'..1.....'B This procedure/block has a DSA
'...0....'B Compiled object
'...1....'B Library object
'....0...'B Program sampling interrupts are to be attributed
to *LIBRARY
'....1...'B Program sampling interrupts are to be attributed
to this program
'.....0..'B Not an exit DSA - no cleanup needed
'.....1..'B Exit DSA - cleanup processing at exit is needed
'.....0.'B Use own language exception model
'.....1.'B Use caller's exception model (enablement, et.al.)
'.....x'B Reserved
```

Figure 12. Language Environment PPA1 offset X'02'

Program flag	Description
Bit 0	Internal/external procedure 0 Indicates this routine is an internal procedure with a nesting level greater than 0. 1 Indicates this routine is an external procedure with a nesting level of 0.
Bit 1	Primary/secondary entry point 0 Indicates this entry point is a primary entry point. 1 Indicates this entry point is a secondary entry point.
Bit 2	Code with or without a DSA 0 Indicates that this block of code did not allocate its own DSA. 1 Indicates that this block of code did allocate its own DSA.

Language Environment Conventions

Program flag	Description
Bit 3	Library or compiler-generated user code 0 Indicates that the code is compiler-generated user code. 1 Indicates that the code is associated with the library code.
Bit 4	Sampling flag 0 Sampling interrupts that occur in this block of code are attributed to library support code. 1 Sampling interrupts that occur in this block of code are attributed to compiler-generated user code.
Bit 5	Exit DSA marking 0 Indicates that no action is required to be taken on behalf of this routine when abnormally collapsing the associated DSA (nonreturn style). 1 Indicates that this routine requires action to be taken when abnormally collapsing the associated DSA (nonreturn style). The associated DSA is known as an exit DSA. For more information, see "DSA exit routines" on page 276.
Bit 6	Condition management actions 0 Indicates that the HLL of the generated code participates in condition management activities. 1 Indicates that the HLL of the generated code chooses not to participate in condition management activities. All phases of condition management skip the associated DSA. This includes enablement, driving member condition handlers, and user handlers. It is not valid to establish a user handler at this stack frame. Also, stack frames with this flag set are not counted in calls to CEEMRCR.
Bit 7	Reserved and must be zero.

Program flags — PPA1 offset X'18'

Language Environment program flags (PPA1 offset X'18') for FASTLINK are shown in Figure 13 and are described following the figure.

```
'0.....'B CEL Version 1 Release 1 stack frame layout
'1.....'B FASTLINK stack frame layout
'.000....'B CEL version 1 Release 1 calling conventions (Version 2)
'.001....'B Old C C private conventions (+16 Entry Point disabled)
'.101....'B FASTLINK Special conventions (Version 2)
'.110....'B FASTLINK V1R2 conventions (Version 2)
'.111....'B FASTLINK Public conventions (Version 2)
'....00..'B Non Sleaf
'....01..'B Sleaf return/entry address not in save area but in R14 & R15
'....10..'B Sleaf return/entry address in save area
'.....00'B Old code (+0) entry disabled
'.....01'B Old code (+0) entry enabled by member simulation routine
'.....10'B Old code (+0) entry enabled by line code
```

Figure 13. Language Environment PPA1 flag 2 offset X'18'

Program flag	Description
Bit 0	Stack Frame Layout (see note) 0 Indicates the routine uses the Version 1 Release 1 stack frame layout. 1 Indicates the routine uses the Version 1 Release 2 FASTLINK frame layout.

Program flag	Description
Bit 1-3	Calling conventions (see note) 0 Entry point uses R1 non-FASTLINK conventions (Version 2). 1 Entry point uses old C conventions. 6 Entry point uses V1 R2 FASTLINK conventions and is potentially bilingual. 7 Entry point supports FASTLINK conventions and is potentially bilingual.
Bit 4-5	SLEAF only valid for FASTLINK; otherwise, it must be zero. 0 Indicates this entry point is not a SLEAF routine; it allocates its own DSA. 1 Indicates this entry point is a SLEAF routine that keeps its return address in R14 and the entry address in R15 — not in the DSA. 2 Indicates this entry point is a SLEAF routine that keeps its return and entry address in a normal save area location.
Bit 6-7	Old entry enablement only valid for FASTLINK; otherwise, it must be zero. 0 Indicates +0 entry point is disabled. 1 Indicates +0 entry point is enabled but does not obtain its own stack frame; it uses the same stack frame as the primary entry. 2 Indicates +0 entry point is enabled and obtains its own stack frame. Two stack frames are obtained by this routine when it is called from old code: one for old code entry and the other normal one created by the primary entry point.

Note: For Version 1 Release 2, if Bit 0 is 0 (indicates Version 1 Release 1 DSA layout), Bit 1-3 may only have a value of 1, which indicates old C conventions. If Bit 0 is 1 (indicates FASTLINK DSA layout), Bit 1-3 may only have a value of 6, which indicates FASTLINK conventions.

Program flags — PPA1 offset X'1C'

Language Environment program flags (PPA1 offset X'1C') are shown in Figure 14 on page 18 and are described following the figure.

Language Environment Conventions

```
'00.....'B Old code entry performs full save (14,15,2-12)
'01.....'B Old code performs partial save (Version 2)
'10.....'B Old code performs partial save + R12 (Version 2)
'.0.....'B Asynchronous condition processing not deferred
'.1.....'B Asynchronous condition processing deferred (Version 2)
'...0.....'B Word 0 of save area not initialized
'...1.....'B Word 0 of save area initialized
'....0.....'B Code is non-external glue
'....1.....'B Code is external glue
'.....0.....'B Real return address saved in save area at offset 0x0C
'.....1.....'B Real return address saved in linkage area (Version 2)
'.....0.....'B Storage argument area start indeterminate
'.....1.....'B Storage argument area start valid
'.....0.....'B R12 must contain CAA address upon old code entry
'.....1.....'B R12 not defined upon old code entry (Version 2)
'.....0.....'B Not vararg routine
'.....1.....'B Vararg routine
'.....0.....'B Asynchronous interrupts are not supported
'.....1.....'B Asynchronous interrupts are supported
'.....0.....'B No module service level
'.....1.....'B Module service level applied
'.....x.....'B Reserved
'.....x.....'B Reserved
'.....x.....'B Reserved
'.....x.....'B Reserved
'.....x.....'B Reserved
```

Figure 14. Language Environment PPA1 flag 3 offset X'1C'

Bit 0 - 9 are reserved in Prolog Constants Format – Level 1 (Standard). The definition below is for Prolog Constants Format – Level 2 (FASTLINK).

Program flag	Description
Bit 0 - 1 (Entry Point Partial Save Flag)	Only valid for FASTLINK bilingual routines which have Bit 5, return address location, set to one. In other cases this field must be zero. 0 Indicates that the +0 entry point performs full save (GPR14-15 and GPR2 through GPR12). 1 Indicates that the +0 entry point performs partial save, the same as the primary entry point. 2 Indicates that the +0 entry point performs partial save, the same as the primary entry point plus R12 is also saved.
Bit 2	Deferred Asynch Exceptions 0 Indicates allow asynchronous exceptions to take effect. 1 Indicates defer asynchronous exceptions.
Bit 3	Save area Language Word (Offset 0 in Save area). 0 Indicates that the language word is initialized (required for DSAs that are flagged in the save area). 1 Indicates that the language word is uninitialized.
Bit 4	Glue code 0 Indicates that the code is not glue. 1 Indicates that the code is external binder glue or runtime simulated prologue. (Language Environment currently has no operational dependency on this flag.)
Bit 5	Return Address Location 0 Indicates that the return address is in the caller provided save area in the normal R14 slot at offset 12 unless “stolen” by Language Environment to enable CEL to gain control upon return from the routine (for example, by CEEHDLR to provide for automatic de-registration of a user condition handler routine). 1 Indicates that the return address maybe in the linkage area of the callee's DSA.

Program flag	Description
Bit 6	Argument List Valid (FASTLINK only) 0 Indicates that the portion of the argument list corresponding to the parameters passed in registers may not be initialized. 1 Indicates that the portion of the argument list corresponding to the parameters passed in registers is valid. This bit is potentially used by debug or by readers of a dump. In Version 2, all compilers must have an optimization level that produces a prologue in which all parameters passed in registers are stored into the argument list.
Bit 7	CAA Address valid at FASTLINK + 0 entry point 0 Indicates that R12 must contain a valid CAA pointer at entry (preserved). 1 Indicates that R12 contents are undefined at entry and must be preserved.
Bit 8	C vararg routine 0 Indicates that the routine is not a C or C++ varargs. 1 Indicates that the routine is a C or C++ varargs.
Bit 9	Async Interrupt Support 0 Indicates that the routine does not support async interrupts. 1 Indicates that the routine supports async interrupts.
Bit 10	Module Service Level Info 0 Indicates that the function has no service applied. 1 Indicates that the function has service applied.

Member identifiers — PPA2 offsets X'00' and X'01'

The Member Identifier (PPA2 offset X'00') identifies the product origin of the running code by compiler. Figure 15 shows the codes for the various compiler products. The product codes are assigned by IBM and the assignment codes are in decimal. The member list table's implementation size is bound to a maximum of 17 (0 through 16) for Language Environment.

00	Reserved
01	Language Environment (CEL)
02	Reserved
03	OS/390 C/C++, C VM/ESA, XL C/C++
04	COBOL V5
05	COBOL for OS/390 & VM, COBOL for MVS & VM
06	Debug Tool
07	VS FORTRAN
08	Reserved
09	Available
10	PL/I for MVS & VM
11	VisualAge PL/I for OS/390
12	Berkeley Sockets
13	Available
14	Reserved
15	ASSEMBLER
16	Reserved

Figure 15. Language Environment-enabled language member identifiers

PPA1 in support of XPLINK

To optimize the space used for control purposes, the structure and contents of the PPA1 for XPLINK have been redefined. The control block is made up of a fixed part followed by a contiguous optional part, with the presence of optional fields indicated by flag bits. Optional fields, if present, are stored immediately following the fixed part of the PPA1 aligned on fullword boundaries in the order specified,

Language Environment Conventions

as shown in Figure 16.

PPA1: XPLINK Entry Point Block Fixed Area (Version 3)

+00	Version	LE Signature X'CE' (Lan Env Signature)	Saved GPR Mask	
+04	Signed Offset to PPA2 from start of PPA1			
+08	PPA1 Flags 1	PPA1 Flags 2	PPA1 Flags 3	PPA1 Flags 4
+0C	Length/4 of Parm		Length/2 of Prolog	Alloca Reg Offs/2 R4 Chg
+10	Length of Code			

Figure 16. Prolog constants format – level 4 (XPLINK), PPA1: entry point block (Version 3)

The PPA1 is located through an offset field preceding the entry point which provides flexibility to group all PPA1s either by compilation unit or by module. The new PPA1 content is extensible in that a version field identifies the particular table structure.

Program prolog areas are mandatory for languages participating in XPLINK. Each entry point must have a corresponding PPA1 associated with it.

PPA1 fixed area fields:

+0	Version	CEL Signature X'CE' (Lang Env Signature)	Saved GPR Mask	
+4	Signed offset to PPA2 from start of PPA1			
+8	PPA1 Flag 1 0 DSA Format 0: 32 bit 1: 64 bit 1 0: Short form PPA1 1: Reserved 2 Exception Model 0: Own 1: Caller's 3 PPA3 type flags 0: tiny PPA3 1: full PPA3 4 Invoke member for DSA exit event 5 XPLink Exit DSA 6 Special Linkage 7 Vararg function	PPA1 Flag 2 0 Procedure 0: Internal 1: External 1 Reserved, 0 2 Reserved, 0 3 Reserved, 0 4 Reserved, 0 5 Reserved, 0 6 Reserved, 0 7 Reserved, 0	PPA1 Flag 3 0 State Variable Locator 1 Argument Area Length 2 FPR Mask 3 AR Mask 4 Member PPA1 Word 5 Block Debug Info 6 Interface Mapping Flags 7 Java Method Locator Table Indicating fields in optional area	PPA1 Flag 4 0 Reserved, 0 1 Reserved, 0 2 Reserved, 0 3 Reserved, 0 4 Reserved, 0 5 Reserved, 0 6 Reserved, 0 7 Name Length and Name Indicating fields in optional area
+12 0x0c	Length/4 of Parm		Length/2 of Prolog	Alloca Reg Offset/2 to StackPointer Update
+16 0x10	Length of Code			

Figure 17. PPA1: XPLINK entry point block fixed area (Version 3) details

Version

An 8-bit field that is set to X'02' to identify this PPA1 as having the Level 4, XPLINK (Version 3) layout.

Note: No Version 1 or Version 2 layouts of the XPLINK PPA1 exist.

Language Environment Signature

An 8-bit field that must be set to X'CE'.

Saved GPR Mask

A 16 bit mask, indicating which registers are saved and restored by the associated routine. Bit 0 indicates register 0, followed by bits for registers 1 to 15 in order.

Signed offset to PPA2 from the start of PPA1

The offset of the PPA2 block belonging to the compilation unit containing the function described by this PPA1.

PPA1 Flag 1: Program flags (PPA1 offset X'08') are shown in Figure 17 on page 20 and are described in Figure 18.

```
'0.....'B GPR Save area is 32 bit.
'1.....'B GPR Save area is 64 bit.
'.0.....'B Reserved.
'.0.....'B Own exception model.
'.1.....'B Inherited exception model.
'...0....'B tiny PPA3.
'...1....'B full PPA3.
'....0...'B Do Not call member for Exit DSA event.
'....1...'B Call member for Exit DSA event.
'.....0..'B Do Not treat as PL/I style exit DSA.
'.....1..'B Treat as PL/I style exit DSA.
'.....0.'B This is not a Special linkage routine.
'.....1.'B This is a Special linkage routine.
'.....0'B Not a Vararg routine.
'.....1'B Vararg routine.
```

Figure 18. Language Environment PPA1 flag 1 offset X'08'

Program flag	Description
Bit 0	Format of General Purpose Registers (GPR) save area 0 Indicates that GPRs are saved as 32-bit quantities. 1 Indicates that GPRs are saved as 64-bit quantities.
Bit 1	Format of PPA1 0 Indicates that this is a short form of the PPA1. 1 Reserved.
Bit 2	Exception Model Flag 0 Indicates that this routine uses it's own exception model. 1 Indicates that this routine inherited the exception model from its caller.
Bit 3	PPA3 Type Flag 0 Indicates a tiny PPA3. 1 Indicates a full PPA3.
Bit 4	Call Member for DSA Exit flag 0 Indicates that the owning member of the DSA should not be called for Exit DSA processing. 1 Indicates that the owning member of the DSA should be called for Exit DSA processing.

Language Environment Conventions

Program flag	Description
Bit 5	XPLINK Exit DSA Flag 0 Indicates that the associated stack frame is not an XPLINK Exit DSA. 1 Indicates that the associated stack frame is an XPLINK Exit DSA and its R7 (return addr) should be given control during stack collapse.
Bit 6	Special linkage Flag 0 Indicates that this is not a special linkage routine. 1 Indicates that this is a special linkage routine used to handle calls between XPLINK and non-XPLINK routines or to handle calls that cause a stack segment extension.
Bit 7	Vararg Flag 0 Indicates that this is not a variable argument (Vararg) routine. 1 Indicates that this is a Vararg routine.

PPA1 Flag 2: Program flags (PPA1 offset X'09') are shown in Figure 17 on page 20 and are described in Figure 19.

'0.....'B	Internal procedure
'1.....'B	External procedure
'.0000000'B	Reserved for future use (must all be zero).

Figure 19. Language Environment PPA1 flag 2 offset X'09'

Program flag	Description
Bit 0	Internal/External procedure 0 Indicates that this procedure is an internal procedure with a nesting level greater than zero. 1 Indicates that this procedure is an external procedure with a nesting level of zero.
Bit 1 - 7	Reserved for future use.

PPA1 Flag 3: Program flags (PPA1 offset X'0A') are shown in Figure 17 on page 20 and are described in Figure 20.

'0.....'B	State Variable locator field is not in optional area.
'1.....'B	State Variable locator field is in the optional area.
'.0.....'B	Argument Area Length is not in the optional area.
'1.....'B	Argument Area Length is in the optional area.
'..0....'B	FP Register Mask is not in the optional area.
'..1....'B	FP Register Mask is in the optional area.
'...0....'B	No ARs are saved. AR mask not in optional area.
'...1....'B	ARs are saved. AR mask in optional area.
'...0... 'B	Member PPA1 word is not present in optional area.
'...1... 'B	Member PPA1 word is present in the optional area.
'....0..'B	Offset to PPA3 is not present in optional area.
'....1..'B	Offset to PPA3 is present in the optional area.
'.....0'B	Interface mapping flags not in the optional area.
'.....1'B	Interface mapping flags in the optional area.
'.....0'B	Java Method Locator Table not in the optional area.
'.....1'B	Java Method Locator Table in the optional area.

Figure 20. Language Environment PPA1 flag 3 offset X'0A'

Program flag	Description
Bit 0	State Variable Locator Flag 0 Indicates that this field is not present in the optional part of the PPA1. 1 Indicates that this field is present in the optional part of the PPA1.
Bit 1	Argument Area Length 0 Indicates that this field is not present in the optional part of the PPA1. 1 Indicates that this field is present in the optional part of the PPA1.
Bit 2	Floating-Point Registers Flag 0 Indicates that the Floating-Point registers are not saved in the DSA. 1 Indicates that the Floating-Point registers are saved in the DSA and that the FPR mask and Offset to FPR savearea is present in the optional PPA1 area. If this field is present, the entire word containing FPR Mask and AR Mask is present in the optional area.
Bit 3	Access Registers Flag 0 Indicates that the Access Registers are not saved in the DSA. 1 Indicates that the Access Registers (as indicated by the Saved AR Bit Mask field) are saved in the DSA and the AR mask in the optional area. If this field is present, the entire word containing FPR Mask, Alloca Reg, and AR Mask is present in the optional area.
Bit 4	Member PPA1 Word Flag 0 Indicates that this field is not present in the optional part of the PPA1. 1 Indicates that this field is present in the optional part of the PPA1.
Bit 5	Offset to PPA3 Flag 0 Indicates that this field is not present in the optional part of the PPA1. 1 Indicates that this field is present in the optional part of the PPA1.
Bit 6	Interface Mapping Flag 0 Indicates that this field is not present in the optional part of the PPA1. 1 Indicates that this field is present in the optional part of the PPA1.
Bit 7	Java™ Method Locator Table 0 Indicates that this field is not present in the optional part of the PPA1. 1 Indicates that this field is present in the optional part of the PPA1.

PPA1 Flag 4: Program flags (PPA1 offset X'0B') are shown in Figure 17 on page 20 and are described in Figure 21.

```
'0.....'B Offset to Entry Point Marker not in the optional area.
'1.....'B Offset to Entry Point Marker in the optional area.
'.0.....'B Upper GPR mask and save area locator not in the optional area.
'.1.....'B Upper GPR mask and save area locator in the optional area.
'..00000.'B Reserved for future optional fields (must all be zero).
```

Figure 21. Language Environment PPA1 flag 4 offset X'0B'

Program flag	Description
Bit 0	Offset To Entry Point Marker 0 Indicates that the offset to the entry pointer marker is not present in the optional part of the PPA1. 1 Indicates that the offset to the entry pointer marker is present in the optional part of the PPA1.

Language Environment Conventions

Program flag	Description
Bit 1	Upper GPR mask and save area locator 0 Indicates that the upper GPR mask and save area locator are not present in the optional part of the PPA1. 1 Indicates that the upper GPR mask and save area locator are present in the optional part of the PPA1.
Bit 2-6	Reserved for future optional fields Reserved for future optional fields
Bit 7	Procedure/Label Name Flag 0 Indicates that the length of name field and the entry/label name field are not present in the optional part of the PPA1. 1 Indicates that the length of name field and the entry/label name field are present in the optional part of the PPA1.

Length/4 of Parm: Length of expected parameter area for this function in fullwords (for vararg functions, the length of the fixed portion of the parameter list). This is used for copying parameters on stack extension. For vararg functions the entire caller's argument area must be copied on stack extension.

Length/2 of Prolog: Length of prolog instruction sequence in halfwords starting from the entry point. The prolog is complete when all conditions described in this architecture are satisfied. This includes: saving the non-volatile registers used by the function, including FPRs and ARs; updating the stack pointer; and loading the `alloca()` register. Other instructions from the function body, including setting up various base registers, may be moved into the prolog, so no component can assume anything about the state of registers within the prolog without scanning the prolog code.

alloca() Register: The register used to point to automatic storage (and other parts of the originally-allocated stack frame) in functions that use `alloca()`. This must be zero if `alloca()` is not used.

Offset/2 to Stack Pointer Update: The offset in halfwords from the Entry Point to the beginning of the instruction that updates the stack pointer (register 4). For XPLLeaf routines, this field will be set to zero.

Length of Code: The length of the code for this function, starting from the entry point marker associated with this PPA1 to the last instruction in the function, in bytes. This does not necessarily include instructions which are the target of "execute," which may be in other parts of the code section, the stack frame, or writable static.

State Variable Locator: Defines the location of the state variable. Bits 0-3 contain the number of a GPR whose contents are added to the unsigned offset in bits 4-31 to calculate the address of the state variable. The register used to address the State Variable, typically the stack register or the `alloca()` register, must be set in the prolog and retain its value throughout the function. This field is optional; its presence is indicated by PPA1 Flag 3, Bit 0.

Argument Area Length: Length of argument area allocated by this function on the stack. If present, this field contains the size of the largest argument list used by this function. This field is optional; its presence is indicated by PPA1 Flag 3, Bit 1. However, this field is required for every function that contains a call with an argument list longer than 128 bytes.

FPR Mask: A 16-bit mask indicating which of FPRs are saved and restored by this routine. Bit 0 indicates FPR0, followed by bits for FPR1 to FPR 15. Space is reserved in the function's local storage for those FPRs actually saved by the function. This field is optional; its presence is indicated by PPA1 Flags3, bit 2. The word containing this field, if present, has either PPA1 Flags3 bits 2 or 3 on.

Access Register Mask: Reserved for future use.

Floating Point Register Save Area Locator: Defines the location of the Floating Point Register Save Area. Bits 0-3 contain the number of a GPR whose contents are added to the unsigned offset in Bits 4-31 to calculate the address of this save area. The register used to address this save area, typically the stack register or the `alloca()` register, must be set in the prolog and retain its value throughout the function. This field is optional; its presence is indicated by PPA1 Flag 3, Bit 2.

Access Register Save Area Locator: Defines the location of the Access Register Save Area. Bits 0-3 contain the number of a GPR whose contents are added to the unsigned offset in bits 4-31 to calculate the address of this save area. The register used to address this save area, typically the stack register or the `alloca()` register, must be set in the prolog and retain its value throughout the function. This field is optional; its presence is indicated by PPA1 Flag 3, Bit 3.

Member PPA1 word: This word contains the information shown Figure 22 for C and C++ (previously part of the Member Flags) when present.

```
'00000000000000000000000000000000.....'B Reserved (must be zero)
'.....0.....'B Argparse
'.....1.....'B No argparse
'.....0.....'B Redirection
'.....1.....'B No redirection
'.....0.....'B Execops
'.....1.....'B No execops
'.....00000'B Reserved (must be zero)
```

Figure 22. Language Environment PPA1 flag word as defined by C++

For C++, this word is used for flags as shown in the preceding figure and are described as follows:

Program flag	Description
Bit 0 - 23	Reserved (must be zero)
Bit 24	Noargparse 0 Indicates argparse. 1 Indicates no argparse.
Bit 25	Noredirection 0 Indicates redirection. 1 Indicates no redirection.
Bit 26	Noexecops 0 Indicates execops. 1 Indicates no execops.
Bit 27 - 31	Reserved (must be zero)

Offset to PPA3: Signed offset to PPA3 from the start of PPA1. This field is optional; its presence is indicated by PPA1 Flag 3, Bit 5.

Language Environment Conventions

Interface mapping flags: This field is provided to allow interface mapping by a glue routine when an XPLINK routine is called from non-XPLINK. It describes the linkage type, the floating-point parameters expected by this routine, and the format of the function return value. This field is optional; its presence is indicated by PPA1 Flag 3, Bit 6.

Java Method Locator Table: Used to locate meta-information for Java classes. This field is optional; its presence is indicated by PPA1 Flag 3, Bit 7.

Offset to entry point marker: Signed offset to entry point marker from the start of PPA1. This field is optional; its presence is indicated by PPA1 Flag 4, Bit 0.

Upper GPR mask and save area locator: Identifies the 64-bit general purpose registers for which the upper halves (bits 0-31) are saved and restored by this routine, and defines the location of the save area. Bits 0-15 are a mask indicating the GPRs for which the upper halves are saved and restored by this routine. Bit 0 indicates GPR0, followed by bits for GPR1 to GPR15. Space is reserved in the function's local storage for those GPRs actually saved by the routine. Bits 16-31 are reserved and must be zero. Bits 32-35 contain the number of a GPR whose contents are added to the unsigned offset in bits 36-63 to calculate the address of this save area. The register used to address this save area, typically the stack register or the alloca() register, must be set in the prolog and retain its value throughout the function. This field is optional; its presence is indicated by PPA1 Flag 4, Bit 1.

PPA1 Optional Area Fields: There are several optional PPA1 Fields; each one's presence indicated by a flag bit in PPA1 Flags 3 or PPA1 Flags 4. Where an optional field is less than 4 bytes in length, the entire word is present if any of the fields in that word are present. Unused parts of the word are filled with zeroes. The optional fields are fullword aligned and appear in the order listed here. The field name and length are given:

Field description	Field length
State Variable Locator (PPA1 Flag 3, Bit 0)	4

Field description	Field length
Argument Area Length (PPA1 Flag 3, Bit 1)	4

Field description	Field length	
FPR mask (PPA1 Flag 3, Bit 2)	AR mask (PPA1 Flag 3, Bit 3)	4

Note: If either Bit 2 or Bit 3 of Flag 3 is on, the fullword variable representing FPR mask and AR mask is present.

Field description	Field length
Floating Point Register Save Area Locator (PPA1 Flag 3, Bit 2)	4

Field description	Field length
Access Register Save Area Locator (PPA1 Flag 3, Bit 3)	4

Field description	Field length
PPA1 Member Word (PPA1 Flag 3, Bit 4)	4

Field description	Field length
Offset to PPA3 (PPA1 Flag 3, Bit 5)	4

Field description	Field length
Interface Mapping Flags (PPA1 Flag 3, Bit 6)	4

Field description	Field length
Java Method Locator Table (MLT) (PPA1 Flag 3, Bit 7)	8

Field description		Field length
Length of Name (PPA1 Flag 4, Bit 7)	Name of Function	variable length
Name of Function (continued)		

Note: Zero to three bytes of zeroes may be needed after the name to ensure that the next optional field starts on a word boundary.

Field description	Field length
Offset To Entry Point Marker (PPA1 Flag 4, Bit 0)	4

Field description	Field length
Upper GPR mask and save area locator (PPA1 Flag 4, Bit 1)	8

PPA2 in support of XPLINK

The following sections describe the structure of the PPA2 format that supports XPLINK. Figure 23 on page 28 shows the format of the prolog constants.

Language Environment Conventions

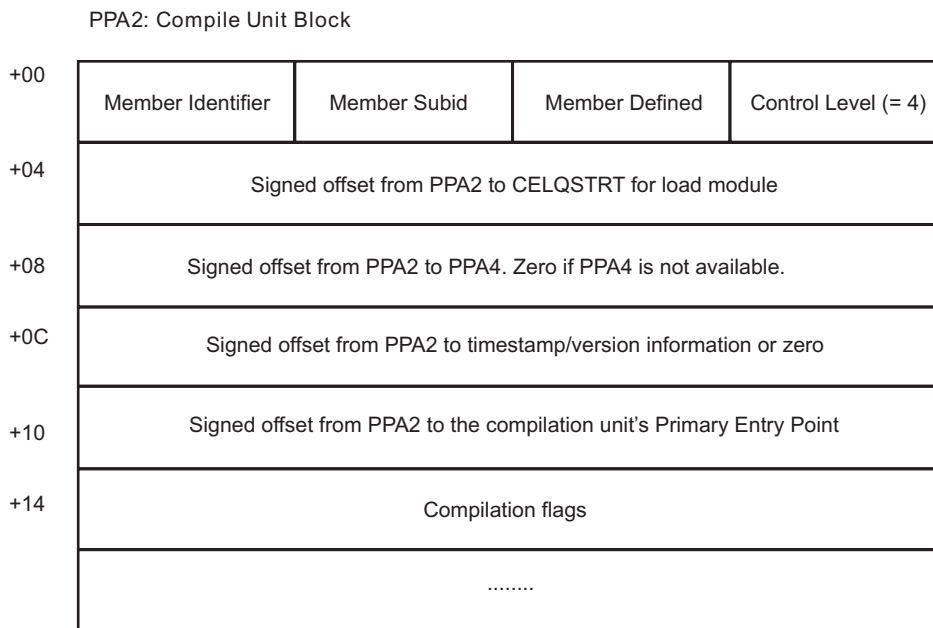


Figure 23. Prolog constants format – level 4 (64-bit XPLINK), PPA2: compile unit block

The level 4 (XPLINK), PPA2: compile unit block bits are described in Figure 24. The XPLINK(STOREARGS) and XPLINK flags were added in PPA2 Level 4.

```
'0.....'B Indicates that program was compiled for hexadecimal floating-point
'1.....'B Indicates that program was compiled for binary floating-point
'.0.....'B Indicates that the code is compiler generated user code
'.1.....'B Indicates that the code is associated with library code
'.0.....'B Program does not contain service information
'.1.....'B Program contains service information
'.0.....'B Not compiled with XPLINK(STOREARGS)
'.1.....'B Compiled with XPLINK(STOREARGS)
'.0.....'B Reserved
'.0.....'B Compiled unit is EBCDIC
'.1.....'B Compiled unit is ASCII
'.0.....'B No additional compiler information after service information
'.1.....'B Additional compiler information after service information
'.0.....'B Not compiled with XPLINK
'.1.....'B Compiled with XPLINK
'.0.....'B Reserved
'.0.....'B MD5 signature is not located at 16 bytes before the timestamp
'.1.....'B MD5 signature is located at 16 bytes before the timestamp
'.0.....'B Not compiled with FLOAT(AFP(VOLATILE))
'.1.....'B Compiled with FLOAT(AFP(VOLATILE))
'.00000 00000000 00000000' B Reserved
```

Figure 24. Level 4 (XPLINK), PPA2: compile unit block bits

Timestamp and Version: Figure 25 on page 29 shows the format of the information in the timestamp and version.

00	CL4'yyyy' Year of compilation	
04	CL4'mmdd' Date of compilation	
08	CL4'hhmm' Time of compilation	
0C	CL2'ss' Time of compilation	CL2 'vv' Version
10	CL4'rrmm' Release/Modification	
14	Service level string length	Untruncated service level string

Figure 25. Timestamp and version information

COBOL V5 32-bit PPA3 layout

PPA3 conforms to this layout under these conditions:

- Member identifier (PPA2 offset X'00') is 4
- PPA4 version in PPA4 program flags is 1
- PPA4 program flags indicates 31-bit compile

Table 4. COBOL V5 32-bit PPA3 layout

Offset	Length	Description
X'00'	4	Reserved
X'04'	4	Signed offset from PPA3 to base locator table. Zero if not available.

C/C++ DWARF 32-bit PPA4 layout

PPA4 conforms to this layout under these conditions:

- Member identifier (PPA2 offset X'00') is 3
- PPA4 version in PPA4 program flags is 2
- PPA4 program flags indicates 31-bit compile

Table 5. C/C++ DWARF 32-bit PPA4 layout

Offset	Length	Description
X'00'	4	PPA4 debug flags for PPA4 version 2
X'04'	4	PPA4 program flags
X'08'	4	Signed offset from CEESTART address to NORENT static
X'0C'	4	Signed offset from WSA to RENT static
X'10'	4	Signed offset from PPA4 to symbol offset table
X'14'	4	Signed offset from PPA4 to code csect

Language Environment Conventions

Table 5. C/C++ DWARF 32-bit PPA4 layout (continued)

Offset	Length	Description
X'18'	4	Length of code csect (in bytes)
X'1C'	4	Signed offset from PPA4 to DWARF line number table embedded in C_CDA class [optional field, check PPA4 debug flags]

COBOL V5 32-bit PPA4 layout

PPA4 conforms to this layout under these conditions:

- Member identifier (PPA2 offset X'00') is 4
- PPA4 version in PPA4 program flags is 1
- PPA4 program flags indicates 31-bit compile

Table 6. COBOL V5 32-bit PPA4 layout

Offset	Length	Description
X'00'	4	PPA4 debug flags for PPA4 version 1
X'04'	4	PPA4 program flags
X'08'	4	Address of NORENT static
X'0C'	4	Signed offset from WSA to 32-bit RENT static
X'10'	4	Signed offset from 32-bit RENT static to 24-bit RENT static address cell. Note: You need to dereference the address cell to get the address of 24-bit RENT static.
X'14'	4	Signed offset from PPA4 to code csect
X'18'	4	Length of code csect (in bytes)
X'1C'	4	Length of NORENT static (in bytes)
X'20'	4	Length of 32-bit RENT static (in bytes)
X'24'	4	Length of 24-bit RENT static (in bytes)
X'28'	2	Signed offset from PPA4 to code csect name (prefixed with 2 bytes string length). Zero if code csect name is not available.

PPA4 debug flags

PPA4 debug flags for PPA4 version 1 - PPA4 offset X'00' are shown in the following code sample:


```
'00.....'B Reserved
'..0.....'B DWARF is not embedded in NOLOAD D_* class
'..1.....'B DWARF is embedded in NOLOAD D_* class
'...0.....'B DWARF is not embedded in LOAD D_* class
'...1.....'B DWARF is embedded in LOAD D_* class
'....0.....'B Compilation unit is compiled with TEST
'....1.....'B Compilation unit is not compiled with TEST
'.....000 00000000 00000000 00000000'B Reserved
```

PPA4 debug flags for PPA4 version 2 - PPA4 offset X'00' are shown in the following code sample:

```
'0.....'B DWARF line number table is not in C_CDA class.
'1.....'B DWARF line number table is in C_CDA class.
'.0.....'B Primary source file name is not available.
'.1.....'B Primary source file name follows DWARF sidefile name.
          (prefixed with 2 bytes string length)
'..0.....'B DWARF is not embedded in NOLOAD D_* class
'..1.....'B DWARF is embedded in NOLOAD D_* class
'...0.....'B DWARF is not embedded in LOAD D_* class
'...1.....'B DWARF is embedded in LOAD D_* class
'....0.....'B Compilation unit is compiled with DEBUG
'....1.....'B Compilation unit is not compiled with DEBUG
'.....000 00000000 00000000 00000000'B Reserved
```

PPA4 program flags

PPA4 program flags - PPA4 offset X'04' are shown in the following code example:

```
'00000000 00000... 'B Reserved
'.....0.. 'B 31-bit compile
'.....1.. 'B 64-bit compile
'.....00 'B Reserved
'..... xxxxxxxx 'B PPA4 version
          0: DWARF information not present
          1: COBOL V5 PPA4
          2: C/C++ DEBUG(FORMAT(DWARF)) PPA4
'..... xxxxxxxx 'B Offset to file name (zero if not applicable)
          file name is prefixed with 4 bytes string length
          PPA4 version is 0: unsigned offset from PPA4 to source file name
          PPA4 version is 2: unsigned offset from PPA4 to DWARF sidefile name
```

Epilog code

Explicit code to free stack space is not required in the Language Environment epilog.

Base locator table

In COBOL V5, there is one BL<n> cell location table for each program or subprogram in a compile unit (that is, entry points). Given an entry point, you can follow a relative offset chain that leads to the base locator table, as follows:

entry point -> PPA1 -> PPA3 -> base locator table

The COBOL base locator table consists of the following:

- COBOL base locator table header
- 0 or more base locator cells array entry
- 2 NULL bytes to signal end of list

Each base locator cells array entry is variable length and contains information to locate the base locator cells array. There can be more than one cells array entry in a table for a particular cell type. Header layout for the base locator table:

Language Environment Conventions

Table 7. Header layout for the base locator table

Offset bytes (bits)	Length bytes (bits)	Field name
0	1	base locator table version (currently 1)
1	1	reserved
2	2	header length (the number of bytes from the beginning of the header to the first byte of the base locator cells array entry)
4	4	length of base locator cells arrays (size of all base locator cells array entries plus the 2 end-of-list NULL bytes)

Entry layout for the base locator cells array:

Table 8. Entry layout for the base locator cells array

Offset bytes (bits)	Length bytes (bits)	Field name
0	0 (5)	base locator cells type 0: end of list 1: BLF cells 2: BLL cells 3: BLX cells 4: BLO cells 5: BLT cells 6: BLV cells
0 (5)	0 (3)	Access method 0: Stack 1: NORENT static 2: 32-bit RENT static 3: 24-bit RENT static
1	0 (2)	[*] byte size of base locator cells array count specified value + 1 (that is, 0 means BL cells array size is 1 byte)
1 (2)	0 (3)	[**] unsigned byte offset to next entry from the 'future expansion' field address
1 (5)	0 (3)	reserved
2	4	unsigned offset to base locator cells array

Table 8. Entry layout for the base locator cells array (continued)

Offset bytes (bits)	Length bytes (bits)	Field name
		This field is used to calculate the starting address of the base locator cells array, each array entry occupies 4 bytes, and contains the address of a base locator cell. The unsigned offset is from: Access method==0: top of stack address Access method==1: address of NORENT static Access method==2: address of 32-bit RENT static Access method==3: address of 24-bit RENT static
6	see [*]	array count for base locator cells
6 + [*]	see [**]	future expansion

CEEYEPAF — locates an XPLINK or non-XPLINK entry point PPA1 and PPA2 from a passed DSA

CEEYEPAF locates an XPLINK or non-XPLINK entry point, PPA1, and PPA2 from a passed DSA.

Syntax

```
void CEEYEPAF (dsa_ptr, dsa_fmt, epa_ptr, (ppa1_ptr, ppa2_ptr),(fc))
```

```
POINTER *dsa_ptr;
INT4 *dsa_fmt;
POINTER *epa_ptr;
POINTER *ppa1_ptr;
POINTER *ppa2_ptr;
FEED_BACK *fc;
```

CEEYEPAF

From a non-XPLINK routine, call this CWI interface as follows:

```
L R15,CEECAALEOV-CEECAA(,R12) Address of CAA in R12
L R15,4(,R15)
BALR R14,R15
```

dsa_ptr (input)

Pointer to the DSA to be examined.

dsa_fmt (input/optionally output)

Format of the stackframe.

```
0 Upward-growing stack
1 Downward-growing stack
-1 CWI determines and returns
```

epa_ptr (output)

Address of Entry Point. If unable to identify entry, returns zero.

ppa1_ptr (output/optional)

Optional PPA1 address to be returned. If unable to identify PPA1 address, returns zero.

ppa2_ptr (output/optional)

Optional PPA2 address to be returned. If unable to identify PPA2 address, returns zero.

fc (output/optional)

Optional feedback code. If omitted and the CWI will end in other than a CEE000, the CWI raises the feedback code as an error condition. The following conditions may result from this CWI:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3EM	Severity	
	Msg_No	
	Message	Unable to find a valid entry point or PPA1 or PPA2 for this DSA.

Note: It is recommended, for performance reasons, that whenever possible, this service is passed the DSA format instead of determining it dynamically. When used in conjunction with CWI CEEYDSAF, to find the previous DSA, the DSA format derived from CEEYDSAF can be passed directly into CEEYEPAF to identify the owner of an XPLINK or non-XPLINK DSA.

__ep_find () — returns the address of the entry point of the function owning the dsa_p DSA

The `__ep_find()` function returns the address of the entry point of the function owning the `dsa_p` DSA. `__ep_find()` can be used when the passed-in DSA is not in the current address space. To access storage outside the current address space, the user must provide the `callback_p` parameter, which is a pointer to a user-written function that fetches all data required by `__ep_find()`. Generally, the `(*callback_p)` function would obtain the data using some application-dependent method (like BPX1PTR) and move it into the current address space, where `__ep_find()` can access it directly. If the passed-in DSA is in the same address space and is directly accessible to `__ep_find()`, `callback_p` can be NULL.

Syntax

```
#include <edcwccwi.h>
```

```
void *__ep_find (const void * dsa_p, int dsa_fmt, void (*callback_p)(void * data_p, size_t data_l))
```

```
const void * dsa_p
```

Pointer to the DSA. `dsa_p` may point to a DSA in another address space or in some other place not directly accessible by `__ep_find()`. If this address is not directly accessible, the `callback_p` parameter must be non-NULL. The callback function will be used to access `dsa_p` indirectly.

```
int dsa_fmt
```

The format of the DSA pointed to by `dsa_p`. The allowed values for `dsa_fmt` are:

__EDCWCWI_UP

This value indicates that *dsa_p* points to a non-XPLINK DSA.

__EDCWCWI_DOWN

This value indicates that *dsa_p* points to an XPLINK DSA.

void * (*callback_p) ()

Pointer to a user-provided function that fetches data not normally accessible by `__ep_find()`. If *callback_p* is NULL, `__ep_find()` accesses *dsa_p* and any other required Language Environment data areas directly in the current address space. All required data must be directly accessible to `__ep_find()` in this case. The user-provided *(*callback_p)()* function is passed the address and length of data to access. It must fetch the data in some application-dependent manner, and make the data available in the current address space in a place accessible to `__ep_find()`. *(*callback_p)()* must return a pointer to the copied data. This data must remain available to `__ep_find()` until the next call to *(*callback_p)()*, or until `__ep_find()` returns to its caller, whichever happens first. On subsequent calls, *(*callback_p)()* is allowed to reuse the same data passback area. There is no provision for *(*callback_p)()* to pass back an error return code, indicating that the requested data could not be obtained. If *(*callback_p)()* cannot return the requested data, it must not return to `__ep_find()`. When an error occurs, *(*callback_p)()* may:

- `longjmp()` back to some error return point in the user code that called `__ep_find()`
- `abend` or otherwise terminate abnormally
- `exit()`, `pthread_exit()`
- Raise a caught signal where the catcher does `longjmp()` so as not to return to `__ep_find()`
- Use Language Environment condition management to bypass `__ep_find()` after the error and resume in user code
- Recover in some other way that does not involve returning to `__ep_find()`.

`__ep_find()` calls *(*callback_p)()* with two parameters:

void * data_p

Pointer to the start of the required data. This address might not be in the current address space.

size_t data_l

The number of bytes of data required. *data_l* will never exceed 16 bytes. If *(*callback_p)()* cannot pass back the complete data requested, it must not return to `__ep_find()`.

`__ep_find()` can return the following values:

- If successful, `__ep_find()` returns the entry point address of the function owning the *dsa_p* DSA.
- If unsuccessful, `__ep_find()` returns a NULL pointer, and sets `errno`. to one of the following values:

ESRCH

This error indicates that the entry point could not be located for the passed-in DSA. This error also occurs if *dsa_p* is NULL when `__ep_find()` is called.

EINVAL

This error occurs if *dsa_fmt* is not `__EDCWCWI_UP` or `__EDCWCWI_DOWN`.

__ep_find ()

Usage Notes:

1. __ep_find() may cause program checks if it accesses invalid addresses. This is especially likely to happen if *callback_p* is NULL and the DSA being looked at is not valid. For this reason, the caller should consider having a signal catcher set up to handle SIGSEGV with appropriate error recovery.
2. The Vendor Interfaces header file, <edcwccwi.h>, is located in member EDCWCCWI of the SCEESAMP data set. To include <edcwccwi.h> in an application, the header file must be copied into a PDS or into a directory in the UNIX file system where the z/OS XL C/C++ compiler will find it.

CEEYPPAF — locates a field in the PPA1 optional area based on a passed pointer to the PPA1

CEEYPPAF locates a field in the PPA1 optional area based on a passed pointer to the PPA1 and an indicator for which field is requested.

Syntax

```
void CEEYPPAF (ppa1_ptr, opt_nam, opt_ptr, opt_ptr2, fc)
```

```
POINTER    *ppa1_ptr;  
INT4       *opt_nam;  
POINTER    *opt_ptr;  
POINTER    *opt_ptr2;  
FEED_BACK  *fc;
```

CEEYPPAF

From a non-XPLINK routine, call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)   Address of CAA in R12  
L    R15,8(,R15)  
BALR R14,R15
```

ppa1_ptr (input)

Pointer to the PPA1.

opt_nam (input)

An integer indicating the requested PPA1 optional field.

- 1 = State variable locator
- 2 = Argument area length
- 3 = Floating point register mask and Offset to FPR savearea
- 4 = Access register mask and Offset to AR savearea
- 5 = Member PPA1 word
- 6 = Block debug info offset
- 7 = Interface mapping flags
- 8 = Java method locator table
- 9 = Name length/name

opt_ptr (output)

Address of the requested optional field in passed PPA1. If unable to identify field, returns zero.

opt_ptr2 (output/optional)

Optional address of the offset to the FPR or AR savearea if FPR (3) or AR (4) requested. If unable to identify or not applicable, returns zero.

fc (output/optional)

Optional feedback code. If omitted and the CWI will end in other than a CEE000, the CWI raises the feedback code as an error condition. The following conditions may result from this CWI:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3EN	Severity	
	Msg_No	
	Message	Requested optional field not found in the passed PPA1.
CEE3EO	Severity	
	Msg_No	3544
	Message	Optional field requested is not valid (1-9).
CEE3EP	Severity	
	Msg_No	3545
	Message	Unable to verify the passed PPA1 as valid for XPLINK.

Language Environment dynamic storage area – non-XPLINK

A DSA (dynamic storage area) is an extension to the save area described in the OS Type I linkage convention. The DSA is described in Figure 26 on page 38. Note that DSAs are sometimes referred to as *stack frames*. This DSA is used by exception handling and Debug Tool services. A macro is provided for assembler language programs.

00	'0000'X	Note 1	Member-defined
04	CEEDSABACK - standard save area back chain		Note 2
08	CEEDSAFWD - standard save area forward chain		Note 3
0C	CEEDSASAVE - GPRs 14, 15, 0-12		Note 4
48	CEEDSALWS - PL/I LWS		Note 8
4C	CEEDSANAB - Current Next Available Byte (NAB) in stack		Note 2
50	CEEDSAPNAB - End of Prolog NAB		
54	Member-defined		Note 5
58	Member-defined		
5C	Member-defined		
60	Member-defined		
64	Reserved		Note 6
68	Member-defined		
6C	CEEDSAMODE - Return address of the module that caused the last mode switch.		Note 7
70	Member-defined		
74	Member-defined		
78	Reserved for future exception handling		
7C	Reserved for future use		

Figure 26. Language Environment Dynamic storage area – non-XPLINK format

R13 addresses the currently active DSA or standard system save area. The DSA is required for all callers of Language Environment services. A DSA is allocated every time a block is entered and might be extended for member use. For the code sequences to allocate or extend a DSA, see “Allocate/extend/return storage in user stack” on page 93 and “Allocate/return storage in library stack” on page 96.

All DSAs and save areas are backward-chained. A stopping DSA, known as the *dummy DSA*, the *zeroth stack frame*, or the *zeroth DSA*, indicates the first DSA on the stack. The DSA layout includes all fields used/accessed by Language Environment and language-specific components.

Notes on DSA Format:

1. IBM language products use these two bytes. All other products must set these two bytes to X'0000'.
2. This field must be initialized.
3. This field is not used by Language Environment but is reserved for compatibility. If it is used, it is the standard forward chain of save areas.

4. This area should only be used to save the caller's general registers. General registers R14 through R12 are saved during prolog and restored during epilog. Bit 0 of R14 indicates the AMODE of the caller.
5. The member-defined fields are established by the caller.
6. This field is reserved for Debug Tool use. It is currently used by the compiled code EXecute hook mechanism.
7. This field is used by the Language Environment library routines.
8. If any vendor package calls a PL/I application, the caller's DSA must have the address of the PL/I LWS. Before calling PL/I user or library routines, the application must pick up the address of the LWS from the CAA (CEECAALWS) and store it into the DSA at CEEDSALWS.

The following is the minimum set of DSA requirements:

- CEEDSANAB must contain a valid NAB (Next Available Byte).
- CEEDSAMODE does not need to be initialized.
- CEEDSABACK must be properly set.
- R14 and R15, used as the linkage registers, must be saved in the appropriate offsets within the DSA.

Non-Language Environment DSAs can be in the save area chain. Routines that scan the stack should be aware that the length of the save area and the saved register contents might not conform to Language Environment conventions.

Language Environment dynamic storage area – XPLINK

An XPLINK DSA (Dynamic Storage Area) differs significantly from the non-XPLINK DSA based on the OS Type I linkage convention that is described in Figure 27.

000	CEEDSAHP_BIAS - Stack Bias, DO NOT USE	Note 1
800	CEEDSAHP4TO15 - Save area for GPRs 4-15	Note 2
830	Reserved for use by run-time	
838	CEEDSAHPTRAN - Debug Area	Note 3
83C	CEEDSAHP_ARG_PRE - Argument prefix area	Note 4
840	CEEDSAHP_ARGLIST - Start of variable length argument list	Note 5

Figure 27. Language Environment Dynamic storage area – XPLINK format

Note:

1. This is the size of the bias between the actual value in the XPLINK stack register (R4) and the start of the DSA. This area is not usable by the current function. It will contain the DSAs of any called XPLINK functions.
2. A called XPLINK function will only save the registers that might be altered during its execution.
3. Used by Debug Tool.
4. Used by stack switching glue code for compatibility with non-XPLINK functions.

- Area where argument list for called functions will be built. Only parameters that are not passed in registers will be stored into the argument area.

In an XPLINK function, the currently active DSA is located by R4. However, R4 is "biased" by x'800' (2048) bytes. This bias needs to be added to the contents of R4 to get the actual start of the XPLINK register save area.

XPLINK DSAs can be back-chained using the value of GPR4 in the register save area. However, GRP4 is only optionally saved. The correct way to find the caller's DSA is to add the size of the current DSA to its location.

Language Environment common anchor area

Each thread is represented by a Common Anchor Area (CAA). It is the central communication area for the environment. All thread- and enclave-related resources are anchored, provided for, or can be obtained through the CAA. The CAA is generated during thread initialization and deleted during thread termination. The CAA points to the encompassing Enclave Data Block (EDB).

The CAA is addressed by R12 when calling Language Environment-participating external routines. This requirement is relaxed when calling internal routines within a given HLL.

Fields in the CAA should be used as described in other sections of this document. In particular, fields should not be modified and routine addresses should not be used as entry points, except as specified. Fields marked reserved exist for migration of specific languages, or internal use by Language Environment. Their location in the CAA is defined by Language Environment, but their use is not. They should be neither used nor referenced except as specified by the language that defines them.

The following tables show the format of the CAA:

- Table 9 shows the CAA field descriptions.
- Table 10 on page 46 shows the CAA constants.
- Table 11 on page 46 shows the CAA cross reference information.

Table 9. Common anchor area (CAA) field descriptions

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
0	(0)	STRUCTURE	976	CEECAA	CAA mapping
0	(0)	CHARACTER	976	CEECAA_EXTERNAL	External portion
0	(0)	BITSTRING	1	CEECAAFLAG0	CAA Flags
		1111 11..		*	Reserved
	1.		CEECAAXHDL	Bypass exception handling
	1		*	Reserved
1	(1)	BITSTRING	1	*	Reserved
		11..		*	Reserved
		..1.		CEECAADBGINIT	Debugger is init'd
		...1 1111		*	Reserved
2	(2)	BITSTRING	1	CEECAALANGP	PL/I Compatibility flags
		1111		*	Reserved
	 1...		CEECAATHFN	If set, NO PL/I FINISH on-unit is active

Table 9. Common anchor area (CAA) field descriptions (continued)

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
	111		*	Reserved
3	(3)	CHARACTER	5	*	Reserved
8	(8)	ADDRESS	4	CEECAABOS	Start of current storage seg
12	(C)	ADDRESS	4	CEECAAEOS	End of current storage seg
16	(10)	CHARACTER	52	*	Reserved
68	(44)	SIGNED	4	CEECAATORC	Thread level ret code
68	(44)	SIGNED	2	*	
70	(46)	SIGNED	2	CEECAATURC	
72	(48)	CHARACTER	44	*	Reserved
116	(74)	ADDRESS	4	CEECAATOVF	Stack overflow rtn
120	(78)	CHARACTER	168	*	Reserved
288	(120)	ADDRESS	4	CEECAAATTN	Addr of CEL attention handler
292	(124)	CHARACTER	56	*	Reserved
348	(15C)	ADDRESS	4	CEECAAHLEXIT	Set by CEEBINT
Debugger controls					
352	(160)	CHARACTER	56	*	Reserved
408	(198)	BITSTRING	12	CEECAAHOOK	Code to pass control to the debugger
420	(1A4)	ADDRESS	4	CEECAADIMA	A(debugger entry)
Each of the following hook switches will contain: DS X'0700', S(CEECAAUDHOOK) When the hook switch is activated, the X'0700' is changed to X'45C0', so it becomes: BAL 12,CEECAAUDHOOK					
424	(1A8)	CHARACTER	72	CEECAAHOOKS	Hook control words for debug
424	(1A8)	CHARACTER	4	CEECAALLOC	ALLOCATE descr. built
428	(1AC)	CHARACTER	4	CEECAASTATE	New statement begins
432	(1B0)	CHARACTER	4	CEECAAENTRY	Block entry
436	(1B4)	CHARACTER	4	CEECAAEXIT	Block exit
440	(1B8)	CHARACTER	4	CEECAAMEXIT	Multiple block exit
444	(1BC)	CHARACTER	32	CEECAAPATHS	PATH hooks
444	(1BC)	CHARACTER	4	CEECAALABEL	At a label constant
448	(1C0)	CHARACTER	4	CEECAABCALL	Before CALL
452	(1C4)	CHARACTER	4	CEECAAACALL	After CALL
456	(1C8)	CHARACTER	4	CEECAADO	DO block starting
460	(1CC)	CHARACTER	4	CEECAAIIFTRUE	True part of IF
464	(1D0)	CHARACTER	4	CEECAAIIFFALSE	False part of IF
468	(1D4)	CHARACTER	4	CEECAAWHEN	WHEN group starting
472	(1D8)	CHARACTER	4	CEECAAOTHER	OTHERWISE group
476	(1DC)	CHARACTER	4	CEECAACGOTO	GOTO hook for C
480	(1E0)	CHARACTER	4	CEECAARSVDH1	Reserved hook
484	(1E4)	CHARACTER	4	CEECAARSVDH2	Reserved hook
488	(1E8)	CHARACTER	4	CEECAAMULTEVT	Multiple event hook
492	(1EC)	CHARACTER	4	CEECAAMEVMASK	Multiple event hook mask
496	(1F0)	CHARACTER	80	CEECAAMEMBER_AREA	
496	(1F0)	CHARACTER	4	CEECAACGENE	C/370 CGENE
500	(1F4)	ADDRESS	4	CEECAACRENT	C or C++ writable static
504	(1F8)	CHARACTER	8	CEECAACFLTINIT	Convert fixed to float cfltnit is used by compiled code
512	(200)	ADDRESS	4	CEECAACPRMS	Parameters passed to IBMBLIIA cprms is reference by user's code offset: 4*128
516	(204)	SIGNED	4	CEECAAC_RTL	Combination of 24 unique C/370 trc types & 8 common trc types
520	(208)	ADDRESS	4	CEECAACTHD	C/370 CTHD

Common Anchor Area (CAA)

Table 9. Common anchor area (CAA) field descriptions (continued)

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
524	(208)	ADDRESS	4	CEECAACURRFECB	
528	(210)	ADDRESS	4	CEECAAEDCV	C/C++ runtime library vector table
532	(214)	ADDRESS	4	CEECAACPCB	Reserved
536	(218)	ADDRESS	4	CEECAACEDB	C/370 CEDB
540	(21C)	CHARACTER	3	*	Reserved
543	(21F)	CHARACTER	1	CEECAASPCFLAG3	Used for SPC
544	(220)	ADDRESS	4	CEECAACIO	Address of cio
548	(224)	CHARACTER	4	CEECAAFDSETFD	Used by FD_* macros
552	(228)	CHARACTER	2	CEECAAFCBMUTEXOK	
554	(22A)	CHARACTER	2	*	Reserved
556	(22C)	CHARACTER	4	CEECAATC16	
560	(230)	SIGNED	4	CEECAATC17	
564	(234)	ADDRESS	4	CEECAAEDCOV	C/370 Open Libvec
568	(238)	SIGNED	4	CEECAACTOFSV	
572	(23C)	ADDRESS	4	CEECAATRTSPACE	C/370 Open Libvec
576	(240)	CHARACTER	24	*	Reserved
600	(258)	CHARACTER	36	CEECAA_TCASRV	TCA Service Rtn Vctr
600	(258)	ADDRESS	4	CEECAA_TCASRV_USERWORD	
604	(25C)	ADDRESS	4	CEECAA_TCASRV_WORKAREA	
608	(260)	ADDRESS	4	CEECAA_TCASRV_GETMAIN	
612	(264)	ADDRESS	4	CEECAA_TCASRV_FREEMAIN	
616	(268)	ADDRESS	4	CEECAA_TCASRV_LOAD	
620	(26C)	ADDRESS	4	CEECAA_TCASRV_DELETE	
624	(270)	ADDRESS	4	CEECAA_TCASRV_EXCEPTION	
628	(274)	ADDRESS	4	CEECAA_TCASRV_ATTENTION	
632	(278)	ADDRESS	4	CEECAA_TCASRV_MESSAGE	
636	(27C)	CHARACTER	4	*	Reserved
640	(280)	ADDRESS	4	CEECAALWS	Addr of PL/I LWS
644	(284)	ADDRESS	4	CEECAASAVR	Register save
648	(288)	CHARACTER	36	*	Reserved
684	(2AC)	BITSTRING	1	CEECAASYSYM	Underlying Op Sys
685	(2AD)	BITSTRING	1	CEECAAHDRWR	Underlying Hardware
686	(2AE)	BITSTRING	1	CEECAASBSYS	Underlying Subsystem
687	(2AF)	BITSTRING	1	CEECAAFLAG2	
		1...		CEECAABIMODAL	Bimodal addressing
		.1..		CEECAA_VECTOR	Vector hardware avail
		..1.		CEECAATIP	Thread terminating
		...1		CEECAA_THREAD_INITIAL	Initial thread
	 1...		CEECAA_TRACE_ACTIVE	Library trace is active (the TRACE runtime option was set)
	1..		CEECAA_ALTSTK_ACTIVE	Alternate stack active
	1.		CEECAA_ENQ_WAIT_INTERRUPTABLE	
	1		CEECAA_USRSTK_ACTIVE	C-RTL context switching user stack active
688	(2B0)	UNSIGNED	1	CEECAALEVEL	CEL level identifier
689	(2B1)	BITSTRING	1	CEECAA_PM	Image of current program mask
690	(2B2)	BIT (16)	2	CEECAA_INVAR	Field that is at the same fixed offset in 31-bit and 64-bit CAAs
691	(2B3)	BIT		*	Reserved.

Table 9. Common anchor area (CAA) field descriptions (continued)

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
692	(2B4)	ADDRESS	4	CEECAAGETLS	Addr of CEL library stack mgr
696	(2B8)	ADDRESS	4	CEECAACELV	Addr of CEL LIBVEC
700	(2BC)	ADDRESS	4	CEECAAGETS	Addr of CEL get stack stg rtn
704	(2C0)	ADDRESS	4	CEECAALBOS	Start of library stack stg seg
708	(2C4)	ADDRESS	4	CEECAALEOS	End of library stack stg seg
712	(2C8)	ADDRESS	4	CEECAALNAB	Next available byte of lib stg
716	(2CC)	ADDRESS	4	CEECAADMC	Addr ESPIE Devil-May-Care rtn
720	(2D0)	SIGNED	4	CEECAACD	Most recent ABEND completion code
720	(2D0)	SIGNED	4	CEECAAABCODE	
724	(2D4)	SIGNED	4	CEECAARS	Most recent ABEND reason code
724	(2D4)	SIGNED	4	CEECAARSNCODE	
728	(2D8)	ADDRESS	4	CEECAAERR	Addr of the current CIB
732	(2DC)	ADDRESS	4	CEECAAGETSX	Addr of CEL stack stg extender
736	(2E0)	ADDRESS	4	CEECAADDSA	Addr of the dummy DSA
740	(2E4)	SIGNED	4	CEECAASECTSIZ	Vector Section Size
744	(2E8)	SIGNED	4	CEECAAPARTSUM	Vector Partial Sum Number
748	(2EC)	SIGNED	4	CEECAASSEXPNT	Log of Vector Section Size
752	(2F0)	ADDRESS	4	CEECAAEDB	A(EDB)
756	(2F4)	ADDRESS	4	CEECAAPCB	A(PCB)
The following two fields are used for validation of the CAA.					
760	(2F8)	ADDRESS	4	CEECAAEYEPTR	Addr of CAA eyecatcher
764	(2FC)	ADDRESS	4	CEECAAPTR	Addr of this CAA
768	(300)	ADDRESS	4	CEECAAGETS1	DSA alloc - R13 not DSA addr
772	(304)	ADDRESS	4	CEECAASHAB	ABEND shunt routine address
776	(308)	ADDRESS	4	CEECAAPRGCK	Pgm interrupt code for CAADMC
780	(30C)	BITSTRING	1	CEECAAFLAG1	CAA Flags 1
		1... ..		CEECAASORT	Call to DF/SORT is active
		.1.. ..		CEECAA_USE_OLD_STK	Use old stack
		..1.		CEECAACICS_EXT_REG	ERTLI CICS extended register interface in effect
		...1		CEECAASHAB_RECOVER_IN_ESTAE_MODE	When ON, Language Environment will set up for retry to the abend shunt routine only if the PSW key that was in effect at the time the Language Environment ESTAE or user-provided error recovery routine was established matches the IPK result stored in CEECAASHAB_KEY.
	 1...		*	Reserved
	1..		CEECAA_FETCH_RELES_IN_PROGRESS	CEEFETCH or CEERELES in progress on this thread.
	11		*	Reserved
781	(30D)	CHARACTER	1	CEECAASHAB_KEY	IPK result when CEECAASHAB is set.
782	(30E)	CHARACTER	2	*	Reserved
784	(310)	SIGNED	4	CEECAAURC	Thread level return code
The following four fields are for FASTLINK capability.					
788	(314)	ADDRESS	4	CEECAAESS	End of current user stack
792	(318)	ADDRESS	4	CEECAALESS	End of current library stack
796	(31C)	ADDRESS	4	CEECAAOGETS	Overflow user seg from FASTLINK
800	(320)	ADDRESS	4	CEECAAOGETLS	Overflow lib seg from FASTLINK
The following field contains the Pre-Init Compatibility Control Block address.					
804	(324)	ADDRESS	4	CEECAAPICIB	Addr of pre-init compat cb

Common Anchor Area (CAA)

Table 9. Common anchor area (CAA) field descriptions (continued)

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
The following field is for FASTLINK capability.					
808	(328)	ADDRESS	4	CEECAAOGETSX	User DSA ext from FASTLINK
812	(32C)	SIGNED	4	*	Fields used by GOTO and CEEHTRAV
812	(32C)	SIGNED	2	CEECAAGOSMR	When set will be used to indicate additional frames to skip
814	(32E)	SIGNED	2	*	Indicate additional frames to skip.
816	(330)	ADDRESS	4	CEECAALEOV	Addr of Lang Env/z/OS UNIX LIBVEC
820	(334)	SIGNED	4	CEECAA_SIGSCTR	SIGSAFE counter
824	(338)	BITSTRING	4	CEECAA_SIGSFLG	SIGSAFE flags
		1...		CEECAA_SIGPUTBACK	Signal putback
		.1..		CEECAA_SA_RESTART	SA_RESTART loopback is required this time
		..1.		*	Reserved
		...1		CEECAA_SIGSAFE	It is safe to unconditionally accept delivery of a synchronous signal
	 1...		CEECAA_CANCELSAFE	It is safe to unconditionally accept delivery of a synchronous cancel
	1..		CEECAA_SIGRESYNCH	One or more synchronous signals may have been recently put back the last time a signal was resolicited while returning from library to user code
	1.		CEECAA_FRZ_UNSAFE	It is unsafe to freeze the thread
	1		CEECAA_NOAPPREGS	User application registers may be saved in a nonstandard place
825	(339)	1...		CEECAA_EINTR_RSOL	Secondary signal resolicit in progress after EINTR from inner function
		.1..		CEECAA_EINTR_PUTB	Secondary re-solicited signal has been put back
		..1.		CEECAA_EINTR_REST	User catcher returned after catching secondary re-solicited signal with SA_RESTART in effect
		...1		CEECAA_EINTR_SIGG	"Stray" signal interrupted CEEOSIGG while secondary signal re-solicitation was in progress
	 1111			Reserved.
826	(33A)	BIT (16)	2	*	Reserved.
828	(33C)	CHARACTER	8	CEECAATHDID	Thread ID
836	(344)	ADDRESS	4	CEECAA_DCARENT	Read/write static external anchor
840	(348)	ADDRESS	4	CEECAA_DANCHOR	Per-thread anchor
844	(34C)	ADDRESS	4	CEECAA_CTOC	TOC anchor for CRENT
848	(350)	ADDRESS	4	CEECAARCB	A(RCB)
852	(354)	SIGNED	4	CEECAACICRSN	CICS reason code from member language
856	(358)	ADDRESS	4	CEECAAMEMBR	Address of thread-level
860	(35C)	ADDRESS	4	CEECAA_SIGNAL_STATUS	Signal status of the terminating thread member list
864	(360)	ADDRESS	4	CEECAA_HCOM_REG7	HCOM saved R7
864	(360)	ADDRESS	4	CEECAA_HCOM_REG14	HCOM saved R14
868	(364)	ADDRESS	4	CEECAA_STACKFLOOR	Lowest usable addr in XP stack
872	(368)	ADDRESS	4	CEECAHPGETS	XP stack extension rtn
876	(36C)	ADDRESS	4	CEECAEDCHPXV	C/C++ XPLINK libvec
880	(370)	ADDRESS	4	CEECAAFOR1	Reserved for FORTRAN
884	(374)	ADDRESS	4	CEECAAFOR2	Reserved for FORTRAN
888	(378)	ADDRESS	4	CEECAATHREADHEAPID	Thread heapid
892	(37C)	CHARACTER	4	CEECAA_SYS_RTNCODE	System (kernel) return code

Table 9. Common anchor area (CAA) field descriptions (continued)

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
896	(380)	CHARACTER	4	CEECAA_SYS_RSNCODE	System (kernel) reason code
900	(384)	ADDRESS	4	CEECAAGETFN	Address of the WSA swap routine
904	(388)	CHARACTER	8	CEECAA_LER4	Reserved
912	(390)	ADDRESS	4	CEECAASIGNPTR	Pointer to 'signam' external variable in a C application
916	(394)	SIGNED	4	CEECAASIGNG	Value of sign of lgamma() -1 - negative sign 0 - zero +1 - positive sign
920	(398)	ADDRESS	4	CEECAA_FORDBG	Ptr to AFHDBHIM - FORTRAN hook interface
924	(39C)	BITSTRING	1	CEECAAAB_STATUS	Validity flags
		1...		CEECAAAB_GR0_VALID	CEECAAAB_GR0 is valid
		.1..		CEECAAAB_ICD1_VALID	CEECAAAB_ICD1 is valid
		..1.		CEECAAAB_ABCC_VALID	CEECAAAB_ABCC is valid
		...1		CEECAAAB_CRC_VALID	CEECAAAB_CRC is valid
	 1...		CEECAAAB_GR15_VALID	CEECAAAB_GR15 is valid
	111		*	Reserved
925	(39D)	UNSIGNED	1	CEECAA_STACKDIRECTION	Stack direction
926	(39E)	BITSTRING	2	*	Reserved
928	(3A0)	SIGNED	4	CEECAAAB_GR0	Reg 0 at the time of abend
932	(3A4)	SIGNED	4	CEECAAAB_ICD1	SDWAICD1
936	(3A8)	SIGNED	4	CEECAAAB_ABCC	SDWAABCC
940	(3AC)	SIGNED	4	CEECAAAB_CRC	SDWACRC
944	(3B0)	ADDRESS	4	CEECAAAGTS	Entry point of CEEVAGTS routine
948	(3B4)	ADDRESS	4	CEECAA_LER5N1	Reserved
952	(3B8)	ADDRESS	4	CEECAAHERP	Address of CEEHERP routine
956	(3BC)	ADDRESS	4	CEECAAUSTKBOS	Start of user stack segment
960	(3C0)	ADDRESS	4	CEECAAUSTKEOS	End of user stack segment
964	(3C4)	ADDRESS	4	CEECAUSERRTN@	Address of thread start routine. Undefined on IPT or prior to thread init event.
968	(3C8)	CHARACTER	8	CEECAAUDHOOK	Hook swapping XPLINK
976	(3D0)	ADDRESS	4	CEECAACEL_HP XV_B	Address of XPLINK compat vector for Base library
980	(3D4)	ADDRESS	4	CEECAACEL_HP XV_M	Address of XPLINK compat vector for Math library
984	(3D8)	ADDRESS	4	CEECAACEL_HP XV_L	Address of XPLINK compat vector for Locale library
988	(3DC)	ADDRESS	4	CEECAACEL_HP XV_O	Address of XPLINK compat vector for Open library
992	(3E0)	ADDRESS	4	CEECAACEL4VEC3	Address of 3rd C-RTL library vector
996	(3E4)	ADDRESS	4	CEECAA_CEEDLLF	Address of the newest CEEDLLF control block
1000	(3E8)	ADDRESS	4	CEECAA_SAVSTACK	Saved Stack Pointer when OS_NOSTACK linkage routine is called.
1004	(3EC)	CHARACTER	8	*	Reserved
1008	(3F0)	CHARACTER	4	CEECAA_USER_WORD	4-byte user field available for application use
1012	(3F4)	ADDRESS	4	CEECAA_SAVSTACK_ASYNC	When the value is not zero, CEECAA_SAVSTACK_ASYNC contains the address of a 4-byte field provided by the application that holds the Saved Stack Pointer when the register for the stack pointer is being used for other purposes. When the value is zero, CEECAA_SAVSTACK_ASYNC does not contain that address.

Common Anchor Area (CAA)

Table 10. Common anchor area (CAA) constants

Len	Type	Value	Name	Description
Declare constants for operating system, hardware, and subsystem CEECAASYSTM, CEECAAHRDWR, CEECAASBSYS				
1	DECIMAL	0	CEECAASYUND	Undefined
1	DECIMAL	1	CEECAASYUNS	Unsupported
1	DECIMAL	2	CEECAASYVM	VM
1	DECIMAL	3	CEECAASYMVS	z/OS Underlying Hardware
1	DECIMAL	0	CEECAAHWUND	Undefined
1	DECIMAL	1	CEECAAHWUNS	Unsupported
1	DECIMAL	2	CEECAAHW370	System/370 non-XA
1	DECIMAL	3	CEECAAHWXA	System/370 XA
1	DECIMAL	4	CEECAAHWESA	System/370 ESA Underlying Subsystem
1	DECIMAL	0	CEECAASSUND	Undefined
1	DECIMAL	1	CEECAASSUNS	Unsupported
1	DECIMAL	2	CEECAASSNON	No subsystem
1	DECIMAL	3	CEECAASSTSO	TSO
1	DECIMAL	5	CEECAASSCIC	CICS
Declare constants for stack direction CEECAA_STACKDIRECTION				
1	DECIMAL	0	CEECAASTACK_UP	UP
1	DECIMAL	1	CEECAASTACK_DOWN	DOWN

Table 11. Common anchor area (CAA) cross reference

Name	Hex Offset	Hex Value	Level
CEECAA	0		1
CEECAA_CANCELSAFE	338	08	4
CEECAA_CTOC	34C		3
CEECAA_DANCHOR	348		3
CEECAA_DCARENT	344		3
CEECAA_ENQ_WAIT_INTERRUPTABLE	2AF	02	4
CEECAA_EXTERNAL	0		2
CEECAA_FORDBG	398		3
CEECAA_FRZ_UNSAFE	338	02	4
CEECAA_HCOM_REG14	360		3
CEECAA_HCOM_REG7	360		4
CEECAA_INVAR	2B2		3
CEECAA_LER4	388		3
CEECAA_LER5	3BC		3
CEECAA_LER5N1	3B4		3
CEECAA_NOAPPREGS	338	01	4
CEECAA_PM	2B1		3
CEECAA_SA_RESTART	338	40	4
CEECAA_SAVSTACK	3E8		4
CEECAA_SAVSTACK_ASYNC	3F4		4
CEECAA_SIGNAL_STATUS	35C		3
CEECAA_SIGPUTBACK	338	80	4
CEECAA_SIGRESYNCH	338	04	4
CEECAA_SIGSAFE	338	10	4

Table 11. Common anchor area (CAA) cross reference (continued)

Name	Hex Offset	Hex Value	Level
CEECOA_SIGSCTR	334		3
CEECOA_SIGSFLG	338		3
CEECOA_STACKDIRECTION	39D		3
CEECOA_STACKFLOOR	364		3
CEECOA_TCASRV	258		3
CEECOA_TCASRV_ATTENTION	274		4
CEECOA_TCASRV_DELETE	26C		4
CEECOA_TCASRV_EXCEPTION	270		4
CEECOA_TCASRV_FREEMAIN	264		4
CEECOA_TCASRV_GETMAIN	260		4
CEECOA_TCASRV_LOAD	268		4
CEECOA_TCASRV_MESSAGE	278		4
CEECOA_TCASRV_USERWORD	258		4
CEECOA_TCASRV_WORKAREA	25C		4
CEECOA_THREAD_INITIAL	2AF	10	4
CEECOA_TRACE_ACTIVE	2AF	08	4
CEECOA_USE_OLD_STK			
CEECOA_USER_WORD	3F0		3
CEECOA_VECTOR	2AF	40	4
CEECOAAB_ABCC	3A8		3
CEECOAAB_ABCC_VALID	39C	20	4
CEECOAAB_CRC	3AC		3
CEECOAAB_CRC_VALID	39C	10	4
CEECOAAB_GR0	3A0		3
CEECOAAB_GR0_VALID	39C	80	4
CEECOAAB_GR15_VALID	39C	08	4
CEECOAAB_ICD1	3A4		3
CEECOAAB_ICD1_VALID	39C	40	4
CEECOAAB_STATUS	39C		3
CEECOAABCODE	2D0		4
CEECOAACALL	1C4		5
CEECOAAGTS	3B0		3
CEECOAALLOC	1A8		4
CEECOAATTN	120		3
CEECAABCALL	1C0		5
CEECAABIMODAL	2AF	80	4
CEECAABOS	8		3
CEECAACD	2D0		3
CEECAACEDB	218		4
CEECOA_CEEDLLF	3E4		3
CEECAACEL4VEC3	3E0		3
CEECAACEL_HP XV_B	3D0		3
CEECAACEL_HP XV_M	3D4		3
CEECAACEL_HP XV_L	3D8		3
CEECAACEL_HP XV_O	3DC		3

Common Anchor Area (CAA)

Table 11. Common anchor area (CAA) cross reference (continued)

Name	Hex Offset	Hex Value	Level
CEECAACELV	2B8		3
CEECAACFLTINIT	1F8		4
CEECAACGENE	1F0		4
CEECAACGOTO	1DC		4
CEECAACICSRSN	354		3
CEECAACIO	220		4
CEECAACPCB	214		4
CEECAACPRMS	200		4
CEECAACRENT	1F4		4
CEECAACTHD	208		4
CEECAACURRFECB	20C		4
CEECAADBGINIT	1	20	4
CEECAADDSA	2E0		3
CEECAADIMA	1A4		3
CEECAADMC	2CC		3
CEECAADO	1C8		5
CEECAAEDB	2F0		3
CEECAAEDCHPXV	36C		3
CEECAAEDCOV	234		4
CEECAAEDCV	210		4
CEECAAENTRY	1B0		4
CEECAAEOS	C		3
CEECAAERR	2D8		3
CEECAAESS	314		3
CEECAAEXIT	1B4		4
CEECAAIEPTR	2F8		3
CEECAAFCBMUTEXOK	228		4
CEECAAFDSETFD	224		4
CEECAAFLAG0	0		3
CEECAAFLAG1	30C		3
CEECAAFLAG2	2AF		3
CEECAAFOR1	370		3
CEECAAFOR2	374		3
CEECAAGETFN	384		3
CEECAAGETLS	2B4		3
CEECAAGETS	2BC		3
CEECAAGETSX	2DC		3
CEECAAGETS1	300		3
CEECAAGOSMR	32C		4
CEECAAHERP	3B8		3
CEECAAHLEXIT	15C		3
CEECAAHOOK	198		3
CEECAAHOOKS	1A8		3
CEECAAHPGETS	368		3
CEECAAHRDWR	2AD		3

Table 11. Common anchor area (CAA) cross reference (continued)

Name	Hex Offset	Hex Value	Level
CEECAAIFFALSE	1D0		5
CEECAAIFTRUE	1CC		5
CEECAALABEL	1BC		5
CEECAALANGP	2		3
CEECAALBOS	2C0		3
CEECAALEOS	2C4		3
CEECAALEOV	330		3
CEECAALESS	318		3
CEECAALEVEL	2B0		3
CEECAALNAB	2C8		3
CEECAALWS	280		3
CEECAAMEMBER_AREA	1F0		3
CEECAAMEMBR	358		3
CEECAAMEXIT	1B8		4
CEECAAOGETLS	320		3
CEECAAOGETS	31C		3
CEECAAOGETSX	328		3
CEECAAOTHER	1D8		5
CEECAAPARTSUM	2E8		3
CEECAAPATHS	1BC		4
CEECAAPCB	2F4		3
CEECAAPICCB	324		3
CEECAAPRGCK	308		3
CEECAAPTR	2FC		3
CEECAARCB	350		3
CEECAARS	2D4		3
CEECAARSNCODE	2D4		4
CEECAARSVDH1	1E0		4
CEECAARSVDH2	1E4		4
CEECAARSVDH3	1E8		4
CEECAARSVDH4	1EC		4
CEECAASAVR	284		3
CEECAASBSYS	2AE		3
CEECAASECTSIZ	2E4		3
CEECAASHAB	304		3
CEECAASHAB_KEY	3D0		2
CEECAASIGNG	394		3
CEECAASIGNGPTR	390		3
CEECAASORT	30C	80	4
CEECAASPCFLAG3	21F		4
CEECAASSEXPNT	2EC		3
CEECAASTATE	1AC		4
CEECAASYSTM	2AC		3
CEECAATC16	22C		4
CEECAATC17	230		4

Common Anchor Area (CAA)

Table 11. Common anchor area (CAA) cross reference (continued)

Name	Hex Offset	Hex Value	Level
CEECAATHDID	33C		3
CEECAATHFN	2	08	4
CEECAATHREADHEAPID	378		3
CEECAATIP	2AF	20	4
CEECAATORC	44		3
CEECAATOVF	74		3
CEECAATURC	46		4
CEECAAUDHOOK	3C8		3
CEECAATRTSPACE	23C		4
CEECAAURC	310		3
CEECAAUSTKBOS	3BC		3
CEECAAUSTKEOS	3C0		3
CEECAAWHEN	1D4		5
CEECAAXHDL	0	02	4

The fields are defined as follows:

CEECAAFLAG0

CAA flag bits; the bits are defined as follows:

- 0-5 Reserved
- 6 CEECAAXHDL: a flag used by the exception handler. If the flag is set to 1, the application requires immediate return/percolation to the system on any interrupt or exception handler event.
- 7 Reserved

CEECAALANGP

PL/I language compatibility flags external to Language Environment; the bits are defined as follows:

- 0-3 Reserved
- 4 CEECAATHFN : A flag set by PL/I to indicate a PL/I FINISH ON UNIT is active. If flag is set to 1, then NO PL/I FINISH ON UNIT is active.
- 5-7 Reserved

CEECAABOS

Start of the current storage segment. This field is initially set during thread initialization. It indicates the start of the current stack storage segment. It is altered when the current stack storage segment is changed.

CEECAAEOS

This field is used to determine if a stack overflow routine must be called when allocating storage from the user stack. Normally, the value of this field will represent the end of the current user stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the user stack. This field is used by function prologs that do not use FASTLINK linkage conventions.

CEECAATORC

Thread level return code. The thread level return code set by CEESRC callable service.

CEECAATOVF

Address of stack overflow routine. This routine is called when there is no space available in the current stack extension to allocate a new stack frame. The routine allocates a new stack extension, updates the CEECAABOS and CEECAAEOS fields in the CAA, and returns the DSA address in the stack extension.

CEECAAATTN

Address of the Language Environment attention handling routine, which supports Language Environment's polling code convention for attention processing.

CEECA AHLLEXIT

Exit list control block address. Exit list control block address as passed back from the HLL user exit in the *A_exit* parameter. For more information, see *z/OS Language Environment Programming Guide*.

CEECAAHOOK

Hook code sequence. CEECAAHOOK contains the following code sequence:

```
ST 12,CEEDSARENT    Put return addr into DSA
BALR 12,0           Get addressability
L 12,CEECAADIMA-*(,12) Get A(CEECAADIMADDR)
BALR 12,12          Go with 12 the base reg.
```

CEECAADIMA

DIM address. Address of the Debugger Interface Module (DIM)

CEECAAHOOKS

Hook area. This is the start of 18 fullword execute hooks. Language Environment initializes each fullword to X'0700',S(CEECAAUDHOOK). The hooks can be altered to support various debugger hook mechanisms such as the Debug Tool's EXecute hooks.

CEECAAALLOC

ALLOCATE description built hook.

CEECAASTATE

New statement begins hook.

CEECAAENTRY

Block entry hook.

CEECAAEXIT

Block exit hook.

CEECAAMEXIT

Multiple block exit hook.

CEECAAPATHS

PATH hook.

CEECAALABEL

At a label constant hook.

CEECAABCALL

Before CALL hook.

CEECAACALL

After CALL hook.

CEECAADO

DO block starting hook.

Common Anchor Area (CAA)

CEECAIFTRUE	True part of IF hook.
CEECAIFFALSE	False part of IF hook.
CEECAAWHEN	WHEN group starting hook.
CEECAAOTHER	OTHERWISE group hook.
CEECAAGOTO	GOTO hook for C hook.
CEECAARSVDH1	Reserved hook.
CEECAARSVDH2	Reserved hook.
CEECAAMULTEVT	Multiple event hook
CEECAAMEVMASK	Multiple event hook mask
CEECAACGENE	C/370 CGENE
CEECAACRENT	C or C++ writable static.
CEECAACFLTINIT	Convert fixed to float cfltinit is used by compiled code
CEECAACPRMS	Parameters passed to IBMBLIIA cprms is reference by user's code offset: 4*128
CEECAAC_RTL	Combination of 24 unique C/370 trc types & 8 common trc types
CEECAACTHD	C/370 CTHD
CEECAACURRFECB	
CEECAAEDCV	Pointer to the C/370 vector table.
CEECAACPCB	Reserved
CEECAACEDB	C/370 CEDB
CEECAASPCFLAG3	Used for SPC
CEECAACIO	Address of cio
CEECAAFDSETFD	Used by FD_* macros
CEECAAFCBMUTEXOK	

CEECAATC16**CEECAATC17****CEECAAEDCOV**

C/370 Open Libvec

CEECAACTOFSV**CEECAARTSPACE**

C/370 Open Libvec

CEECAA_TCASRV

TCA service routine vector, which contains the following fullword address pointers:

- CEECAA_TCASRV_USERWORD
- CEECAA_TCASRV_WORKAREA
- CEECAA_TCASRV_GETMAIN
- CEECAA_TCASRV_FREEMAIN
- CEECAA_TCASRV_LOAD
- CEECAA_TCASRV_DELETE
- CEECAA_TCASRV_EXCEPTION
- CEECAA_TCASRV_ATTENTION
- CEECAA_TCASRV_MESSAGE

CEECAALWS

Address of PL/I Language Working Space.

CEECAASAVR

Register save area.

CEECAASYSTM

Underlying operating system. The value indicates the operating system supporting the active program. The values are defined as follows:

- | | |
|---|---|
| 0 | Undefined—this value should never occur after initializing Language Environment |
| 1 | Unsupported |
| 2 | VM/ESA |
| 3 | z/OS |

CEECAAHRDWR

Underlying hardware. The value indicates the type of hardware on which the program is executing; the values are defined as follows:

- | | |
|---|---|
| 0 | Undefined—this value should never occur after initializing Language Environment |
| 1 | Unsupported |
| 2 | System/370, non-XA |
| 3 | System/370, XA |
| 4 | System/370, ESA |

CEECAASBSYS

Underlying subsystem. The value indicates the subsystem, if any, on which the program is executing; the values are defined as follows:

- | | |
|---|---|
| 0 | Undefined—this value should never occur after initializing Language Environment |
| 1 | Unsupported |
| 2 | None—the program is not executing under a subsystem according to Language Environment |
| 3 | TSO |
| 4 | IMS™ |
| 5 | CICS |

Common Anchor Area (CAA)

CEECAAFLAG2

CAA Flag 2. The bits are defined as follows:

- 0 Set if bimodal addressing
- 1 Set if vector hardware
- 2 Thread terminating
- 3 Initial thread
- 4 Library trace is active; the TRACE runtime option was set
- 5 Reserved
- 6 Thread is in an enqueue wait
- 7 Reserved

CEECAALEVEL

Language Environment level identifier. This contains a unique value that identifies each release of Language Environment. This number is incremented for each new release of Language Environment. Beginning with 10, the version and release numbers are the same as OS/390[®] version and release numbers. The values are defined as follows:

- 1 IBM SAA AD/Cycle LE/370 V1 R1
- 2 IBM SAA AD/Cycle LE/370 V1 R2
- 3 IBM SAA AD/Cycle LE/370 V1 R3
- 4 IBM Language Environment for MVS & VM V1 R4
- 5 OS/390 Language Environment V1 R5
- 6 OS/390 Language Environment V1 R6
- 7 OS/390 Language Environment V1 R7
- 8 OS/390 Language Environment V1 R8
- 9 OS/390 Language Environment V1 R9
- 10 OS/390 Language Environment V2 R7
- 11 OS/390 Language Environment V2 R8
- 12 OS/390 Language Environment V2 R9
- 13 OS/390 Language Environment V2 R10
- 14 z/OS Language Environment V1 R2
- 15 z/OS Language Environment V1 R3
- 16 z/OS Language Environment V1 R4
- 17 z/OS Language Environment V1 R5
- 18 z/OS Language Environment V1 R6
- 19 z/OS Language Environment V1 R7
- 20 z/OS Language Environment V1 R8
- 21 z/OS Language Environment V1 R9
- 22 z/OS Language Environment V1 R10
- 23 z/OS Language Environment V1 R11
- 24 z/OS Language Environment V1 R12
- 25 z/OS Language Environment V1 R13

CEECAA_PM

Program mask.

CEECAA_INVAR

Field that is at the same fixed offset in 31-bit and 64-bit CAAs

CEECAAGETLS

Address of stack overflow for library routines.

CEECAACELV

Address of the Language Environment library vector. This field is used to locate dynamically loaded Language Environment routines.

CEECAAGETS

Address of the Language Environment prolog stack overflow routine. The address of the Language Environment get stack storage routine is included for fast reference in prolog code.

CEECAALBOS

Start of the library stack storage segment. This field is initially set during thread initialization. It indicates the start of the library stack storage segment. It is altered when the library stack storage segment is changed.

CEECAALEOS

This field is used to determine if a stack overflow routine must be called when allocating storage from the library stack. Normally, the value of this field will represent the end of the current library stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the library stack. This field is used by function prologs that do not use FASTLINK linkage conventions.

CEECAALNAB

Next available library stack storage byte. This contains the address of the next available byte of storage on the library stack. It is modified when library stack storage is obtained or released.

CEECAADMC

Language Environment shunt routine address. Its value is initially set to zero during thread initialization. If it is nonzero, this is the address of a routine used in specialized exception processing. For more information, see *z/OS Language Environment Programming Guide*.

CEECAAACD

Most recent CAASHAB abend code.

CEECAAABCODE

Most recent abend completion CDE.

CEECAAARS

Most recent CAASHAB reason code.

CEECAAARSNCODE

Most recent abend reason code.

CEECAAERR

Address of the current CEECIB. After completion of initialization, this always points to a CEECIB. During exception processing, the current CEECIB contains information about the current exception being processed. Otherwise, it indicates no exception being processed.

CEECAAGETSEX

Address of the user stack extender routine. This routine is called to extend the current DSA in the user stack. Its address is in the CEECAA for performance reasons.

CEECAADDSA

Address of the Language Environment dummy DSA. This address determines if a DSA is the dummy DSA, also known as the zeroth DSA.

CEECAASECTSIZ

Vector section size.

CEECAAPARTSUM

Vector partial sum number.

Common Anchor Area (CAA)

CEECAASSEXPNT

Log of the vector section size.

CEECAAEDB

Address of the Language Environment enclave data block. This field points to the encompassing EDB.

CEECAAPCB

Address of the Language Environment Process Control Block. This field points to the encompassing PCB.

CEECAAIEYEPTR

Address of the CAA eye catcher. This field can be used for validation of the CAA.

CEECAAPTR

Address of the CAA. This field points to the CAA itself and can be used in validation of the CAA.

CEECAAGETS1

Non-DSA Stack overflow. This field is the address of a stack overflow routine which cannot guarantee that the current R13 is pointing at a DSA. R13 must point, at a minimum, point to a save area. For additional details, see "Obtain a DSA in user stack with R13 pointing to save area" on page 95.

CEECAASHAB

ABEND shunt routine. Its value is initially set to zero during thread initialization. If it is nonzero, this is the address of a routine used in specialized exception processing for ABENDs that are intercepted in the ESTAE exit. For more information, see *z/OS Language Environment Programming Guide*.

CEECAAPRGCK

Program interrupt code for CEECAADMC. If CEECAADMC is nonzero, and a program interrupt occurs, this field is set to the program interrupt code and control is passed to the address in CEECAAMDC. For more information, see *z/OS Language Environment Programming Guide*.

CEECAAFLAG1

CAA flag bits; the bits are defined as follows:

- 0 CEECAASORT: a call to DFSORT is active.
- 1 CEECAA_USE_OLD_STK: Use old stack
- 2 CEECAACICS_EXT_REG: ERTLI CICS extended register interface is in effect.
- 3 CEECAASHAB_RECOVER_IN_ESTAE_MODE: instructs Language Environment to set up for retry to the abend shunt routine, only if the PSW key that was in effect at the time the Language Environment ESTAE or user-provided error recovery routine was established matches the IPK result stored in CEECAASHAB_KEY.
- 4 Reserved.
- 5 CEECAA_FETCH_RELES_IN_PROGRESS: CEEFETCH or CEERELES is in progress on this thread.
- 6-7 Reserved.

CEECAASHAB_KEY

IPK result when CEECAASHAB is set.

CEECAAURC

Thread level return code. This is the common place for members to set the return codes for sub-to-sub return code processing.

CEECAAESS

This field is used to determine if a stack overflow routine must be called when allocating storage from the user stack. Normally, the value of this field will represent the end of the current user stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the user stack. This field is used by function prologs that use FASTLINK linkage conventions.

CEECAALESS

This field is used to determine if a stack overflow routine must be called when allocating storage from the library stack. Normally, the value of this field will represent the end of the current library stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the library stack. This field is used by function prologs that use FASTLINK linkage conventions.

CEECAAOGETS

Pointer to overflow user segment from FASTLINK.

CEECAAOGETLS

Pointer to overflow library segment from FASTLINK.

CEECAAPICICB

Address of preinit compatibility control block. This is provided in support of the PL/I preinitialization compatibility support.

CEECAAOGETSX

Pointer to user DSA exit from FASTLINK.

CEECAAGOSMR

Go Some More—Used CEEHTRAV multiple.

CEECAALEOV

Address of the Language Environment—z/OS UNIX System Services (z/OS UNIX) LIBVEC.

CEECAA_SIGSCTR

Signal Safe counter. When 0, an interrupt is allowed; when greater than 0, interrupts are temporarily inactive. Four types of interrupts can be blocked or allowed: signal interrupts, cancel interrupts, quiesce-terminate interrupts, and quiesce-freeze interrupts.

CEECAA_SIGSFLG

Signal Safe flags.

- 0 CEECAA_SIGPUTBACK — A signal was put back.
- 1 CEECAA_SA_RESTART — indicates that a signal registered with the SA_RESTART flag interrupted the last kernel call and the signal catcher returned (that is, loopback is required to re-issue the kernel call).
- 2 Reserved.
- 3 CEECAA_SIGSAFE: Indicates that synchronous signals are safe to be delivered, regardless of where the interrupt occurred.
- 4 CEECAA_CANCELSAFE: Indicates that it is safe to unconditionally accept delivery of a synchronous cancel.
- 5 CEECAA_SIGRESYNC: Indicates that one or more synchronous signals may have been recently put back the last time a signal was resolicited while returning from library to user code.

Common Anchor Area (CAA)

- 6 CEECAA_FRZ_UNSAFE: Indicates that the thread is unsafe to be frozen.
- 7 CEECAA_NOAPPREGS: Indicates that user application registers may be saved in a nonstandard place.
- 8 CEECAA_EINTR_RSOL: Secondary Signal re-solicitation is in progress, after EINTR errno from inner function.
- 9 CEECAA_EINTR_PUTB: Secondary re-solicited signal has been put back.
- 10 CEECAA_EINTR_REST: User signal catcher returned after catching secondary re-solicited signal with SA_RESTART in effect.
- 11 CEECAA_EINTR_SIGG: Stray signal interrupted CEEOSIGG while secondary signal resolicitation was in progress.

CEECAATHDID

This CAA's POSIX thread identifier (8 bytes).

CEECAA_DCRENT

Read/write static external anchor.

CEECAA_DANCHOR

Per-thread anchor.

CEECAA_CTOC

TOC anchor for CRENT.

CEECAARCB

Address of RCB.

CEECAACICRSRN

CICS reason code for member language.

CEECAAMEMBR

Address of thread-level member list. An entry is reserved for each member known to Language Environment. There is one member list per thread. For details, see "Language Environment member list and event handler" on page 83.

CEECAA_SIGNAL_STATUS

Signal status for terminating thread.

CEECAA_HCOM_REG7

The original register 7 value overlaid by a pointer to CEEOSIGX when the latest signal was put back.

CEECAA_HCOM_REG14

The original register 14 value overlaid by a pointer to CEEOSIGR when the latest signal was put back.

CEECAA_STACKFLOOR

Lowest usable address in the XPLINK stack.

CEECAAHPGETS

XPLINK stack extension routine.

CEECAAEDCHPXV

C++ XPLINK libvec.

CEECAAFOR1

Reserved for Fortran.

CEECAAFOR2

Reserved for Fortran.

CEECAATHREADHEAPID

Pointer to thread heap ID.

CEECAA_SYS_RTNCODE

System (kernel) return code.

CEECAA_SYS_RSNCODE

System (kernel) reason code.

CEECAAGETFN

Address of the WSA swap routine.

CEECAASIGNPTR

Pointer to the "signam" external variable.

CEECAASIGNG

Value of the sign of lgamma() function.

-1 Negative sign

0 Zero

+1 Positive sign

CEECAA_FORDBG

Pointer to AFHDBHIM — FORTRAN hook interface.

CEECAAAB_STATUS

Contains the following validity flags:

CEECAAAB_GR0_VALID

Indicates if the CEECAAAB_GR0 field contains valid data about the last abend.

CEECAAAB_ICD1_VALID

Indicates if the CEECAAAB_ICD1 field contains valid data about the last abend.

CEECAAAB_ABCC_VALID

Indicates if the CEECAAAB_ABCC field contains valid data about the last abend.

CEECAAAB_CRC_VALID

Indicates if the CEECAAAB_CRC field contains valid data about the last abend.

CEECAAAB_GR15_VALID

Indicates if the CEECAAAB_GR15 field contains valid data about the last abend.

CEECAA_STACKDIRECTION

Stack direction.

CEECAAAB_GR0

Register 0 contents at the time of the ABEND. This is only valid if the CEECAAAB_GR0_VALID bit is on.

CEECAAAB_ICD1

The eight bit interrupt code from SDWAICD1 field of the SDWA for the abend. This is only valid if the CEECAAAB_ICD1_VALID bit is on.

Common Anchor Area (CAA)

CEECAAB_ABCC

The abend completion code, taken from SDWAABCC field of the SDWA for the shunted abend. This is only valid if the CEECAAAB_ABCC_VALID bit is on.

CEECAAAB_CRC

Component reason code, or return code associated with the abend, taken from the SDWACRC field of the SDWA for the shunted abend. This is only valid if the CEECAAAB_CRC_VALID bit is on.

CEECAAAGTS

A 4-byte pointer that contains the address of the entry point of the CEEVAGTS routine. CEEVAGTS supports the code that the C compiler generates in module prologs for DSA allocation.

CEECAA_LER5N1

Reserved.

CEECAAHERP

Address of the CEEHERP routine.

CEECAAUSTKBOS

Start of user stack segment.

CEECAAUSTKEOS

End of user stack segment.

CEECAAUSERRTN@

Address of thread start routine.

CEECAAUDHOOK

Hook swapping XPLINK.

CEECAACEL_HP XV_B

Address of XPLINK vector for Base library.

CEECAACEL_HP XV_M

Address of XPLINK vector for Math library.

CEECAACEL_HP XV_L

Address of XPLINK vector for Locale library.

CEECAACEL_HP XV_O

Address of XPLINK vector for Open library.

CEECAACEL4VEC3

Address of 3rd C-RTL library vector.

CEECAA_CEEDLLF

Address of the newest CEEDLLF control block.

CEECAA_SAVSTACK

Saved Stack Pointer when the OS_NOSTACK linkage routine is called. After the call returns, the CEECAA_SAVSTACK field must be set back to zero. When the value in CEECAA_SAVSTACK is not zero, condition management and signal processing use this value as the current stack pointer. The format of the stack is determined by the value in the CEECAA_STACKDIRECTION field. Asynchronous signals are put back if the interrupt occurs outside the bounds of the routine that owns the stack frame.

CEECAA_SAVSTACK_ASYNC

When the value is not zero, CEECAA_SAVSTACK_ASYNC contains the address of a 4-byte field provided by the application that holds the Saved

Stack Pointer when the register for the stack pointer is being used for other purposes. When the value is zero, CEECAA_SAVSTACK_ASYNC does not contain that address. When the field exists and is not zero, condition management and signal processing use this value as the current stack pointer. The format of the stack is determined by the value in the CEECAA_STACKDIRECTION field. Asynchronous signals are processed even if the interrupt occurs outside the bounds of the routine that owns the stack frame.

Language Environment enclave data block

Each enclave is represented by an enclave data block (EDB), which supports the program model. All enclave-related resources are provided in the EDB; it is generated during enclave initialization and deleted during enclave termination. Fields in the EDB should be used as described in other sections of this document. In particular, fields should not be modified and routine addresses should not be used as entry points, except as specified.

The following tables show the format of the EDB.

- Table 12 shows the EDB fields and Table 15 on page 66 describes their contents.
- Table 13 on page 64 shows the EDB constants.
- Table 14 on page 64 shows the EDB cross reference information.

Table 12. Enclave data block (EDB) field descriptions

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
0	(0)	STRUCTURE	164	CEEEDB	EDB mapping
0	(0)	CHARACTER	164	CEEEDB_EXTERNAL	External portion
0	(0)	CHARACTER	8	CEEEDBEYE	Eyecatcher 'CEEEDB '
8	(8)	BITSTRING	4	CEEEDBFLAGS	Enclave information
8	(8)	BITSTRING	1	CEEEDBFLAG1	EDB Flags
		1...		CEEEDBMAINI	Main program initialized
		.1..		CEEEDB_INITIAL_AMODE	
		..1.		CEEEDBACTIV	Environment is now active
		...1		CEEEDBTIP	Termination In Progress
	 1..		CEEEDBPICI	Pre-Init Compat. is active
	1..		CEEEDB_POSIX	z/OS UNIX is active and runtime option POSIX(ON) is active
	1.		CEEEDBMULTITHREAD	Multithreading environment
	1		CEEEDB_OMVS_DUBBED	z/OS UNIX is dubbed
9	(9)	BITSTRING	1	CEEEDBIPM	Initial Program Mask
10	(A)	BITSTRING	1	CEEEDBPM	Current Program Mask
11	(B)	UNSIGNED	1	CEEEDB_CREATOR_ID	Enclave creator ID
12	(C)	ADDRESS	4	CEEEDBMEMBR	A(member list body)
16	(10)	ADDRESS	4	CEEEDBOPTCB	A(options control block)
20	(14)	SIGNED	4	CEEEDBURC	User Return Code
24	(18)	SIGNED	4	CEEEDBRSNCD	CEL Reason Code
28	(1C)	ADDRESS	4	CEEEDBDBGHEH	Addr of debugger event handler
32	(20)	SIGNED	4	CEEEDBANHP	CEL Anywhere Heap ID
36	(24)	SIGNED	4	CEEEDBBEHP	CEL Below Heap ID

Enclave Data Block (EDB)

Table 12. Enclave data block (EDB) field descriptions (continued)

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
40	(28)	ADDRESS	4	CEEEDBCELV	Addr of CEL LIBVEC
44	(2C)	ADDRESS	4	CEEEDBPCB	A(PCB)
48	(30)	ADDRESS	4	CEEEDBELIST	Exit list from HLL user exit
52	(34)	ADDRESS	4	CEEEDB_PL_ASTRPTR	A(appl parm str)
56	(38)	ADDRESS	4	CEEEDBDEFPLPTR	A(main parm list)
60	(3C)	SIGNED	4	CEEEDBCXIT_PAGE	Cxit_page value for user exit
64	(40)	CHARACTER	4	CEEEDB_DEBUG_TERMID	Debugger terminal ID
68	(44)	ADDRESS	4	CEEEDBPARENT	Addr of the parent enclave CAA
<p>When the enclave is created, its creator (or parent) needs to provide:</p> <ol style="list-style-type: none"> 1. Enclave termination routine (CEEEDB_TERM). 2. Information where to return to when the enclave terminates along with the environment that is to be restored. CEEEDB_R13_PARENT is a convenient way to provide the return information. It is a pointer to the DSA that contains all the registers of the enclave's parent. 					
72	(48)	ADDRESS	4	CEEEDB_R13_PARENT	A(DSA of enclave creator)
76	(4C)	CHARACTER	64	CEEEDB_LER3	Lang Env V1R3M0 externals
76	(4C)	CHARACTER	8	*	Reserved from Lang Env V1R2M0
84	(54)	ADDRESS	4	CEEEDBLEOV	Addr of z/OS UNIX LIBVEC
88	(58)	ADDRESS	4	CEEEDBENVAR	Address of environment variable array. This is the case only when a POSIX-C prog is not part of the application. WARNING: this field should not be updated by other than CEL or C initialization.
92	(5C)	ADDRESS	4	CEEEDBENVIRON	Address of environment variable anchor. In POSIX-C, it is the environ variable, otherwise it points to the CEEEDBENVAR.
96	(60)	ADDRESS	4	CEEEDB_CEEOSIGR@	CEEOSIGR address
100	(64)	ADDRESS	4	CEEEDBOTRB	Pointer to trace table
<p>The following five fields are used by the CEEXGPES (get permanent enclave storage) macro. This macro allows member languages to quickly allocate storage that is freed by CEL only after member enclave termination.</p>					
104	(68)	ADDRESS	4	CEEEDBPSA31	Address and length of ...
108	(6C)	SIGNED	4	CEEEDBPSL31	... preallocated 31 storage
112	(70)	ADDRESS	4	CEEEDBPSA24	Address and length of ...
116	(74)	SIGNED	4	CEEEDBPSL24	... preallocated 24 storage
120	(78)	ADDRESS	4	CEEEDBPSRA	Addr of overflow routine
124	(7C)	ADDRESS	4	CEEEDB_CAACHAIN@	Pointer to IPT's CAA
128	(80)	BITSTRING	4	CEEEDBFLAGS1	Additional external
128	(80)	BITSTRING	1	CEEEDBFLAG1A	Flags
		1...		CEEEDB_SIGENABLED	Signals enabled
		.1..		CEEEDB_MVS_BATCH	Running z/OS batch
		..1.		CEEEDB_TERM_DNFR	Do not free heap or delete programs during termination of the enclave.
		...1		CEEEDB_TERM_NOEDSA	No scan for exit DSAs at enclave termination.

Table 12. Enclave data block (EDB) field descriptions (continued)

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
	 1...		CEEEDB_CICS_OPEN_PROGRAM	1 Program runs only on an OTE TCB and can use Open C functions 0 Program may run on OTE or QR TCB
	1..		CEEEDB_MAIN_HP	Main uses XP linkage
	1.		CEEEDB_HPLINK	XPLINK is being used
	1		CEEEDB_EVNTDEST	Running destructors
129	(81)	BITSTRING	1	CEEEDBFLAG1B	Flags
		1...		CEEEDB_2_ENV_TABLES	1 Lang Env maintains two identical tables of environment variables: one in EBCDIC and one in ASCII 0 Only an EBCDIC table is maintained
		.1..		CEEEDB_CICS_REUSE_ENCLAVE	1 Program is part of a reusable enclave 0 Program is not part of a reusable enclave
		..1.		CEEEDB_CICS_RE_DIRTY	1 Reusable enclave has been corrupted and is no longer reusable by Lang Env. CICS requested to terminate enclave 0 Enclave is clean and still reusable
		...1		CEEEDB_EXEC_EXIT	1 User exit routine for exec() processing is running 0 User exit routine is not running
	 1111		*	Reserved
130	(82)	CHARACTER	2	*	Reserved
132	(84)	ADDRESS	4	CEEEDB_CEEOSGR1@	CEEOSIGR end address
136	(88)	ADDRESS	4	CEEEDB_XPL_NODLL_FDS	Pointer to chain of XPLINK compat descriptors representing NODLL func pointers
140	(8C)	CHARACTER	8	CEEEDB_LER4	
140	(8C)	BITSTRING	4	CEEEDBMEMBERCOMPAT	
Member compatibility flags					
140	(8C)	BITSTRING	1	CEEEDBMEMBERCOMPAT1	
		1...		CEEEDBPLITASKING	PL/I tasking
		.111 1111		*	Reserved
141	(8D)	BITSTRING	1	CEEEDBMEMBERCOMPAT2	Reserved
142	(8E)	BITSTRING	1	CEEEDBMEMBERCOMPAT3	Reserved
143	(8F)	BITSTRING	1	CEEEDBMEMBERCOMPAT4	Reserved
144	(90)	SIGNED	4	CEEEDBTHREADSACTIVE	Threads active
148	(94)	CHARACTER	8	CEEEDB_LER5	
148	(94)	SIGNED	4	CEEEDBCURMSGFILEDCBPTR	DCB ptr
152	(98)	ADDRESS	4	CEEEDB_CEEINT_INPUT_R1	

Enclave Data Block (EDB)

Table 12. Enclave data block (EDB) field descriptions (continued)

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
When the request block boundary is crossed, a new enclave is created and request block info must be maintained. This is to maintain compatibility with the VS COBOL II definition of a run unit. The following two fields allow support for implicit enclave create.					
156	(9C)	ADDRESS	4	CEEEDB_LAST_RBADDR	A(Last request block)
160	(A0)	SIGNED	4	CEEEDB_LAST_RBCNT	Index of last request blk
164	(A4)	SIGNED	4	CEEEDB_ENVLENGTH	Length of envvar array of pointers
168	(A8)	ADDRESS	4	CEEEDBENVAR_A	Address of alternate environment variable array
172	(AC)	ADDRESS	4	CEEEDBENVIRON_A	Address of alternate environment variable anchor

Table 13. Enclave data block (EDB) constants

Len	Type	Value	Name	Description
Declare constants to identify creator of an enclave				
1	DECIMAL	1	CEEEDB_CREATOR_BINIT	batch (BINIT)
1	DECIMAL	2	CEEEDB_CREATOR_RINI	CICS (RINI)
1	DECIMAL	3	CEEEDB_CREATOR_BCREN	cr_enc(BCREN)
1	DECIMAL	4	CEEEDB_CREATOR_PIP1_MAIN	preinit main
1	DECIMAL	5	CEEEDB_CREATOR_PIP1_SUBR	preinit subr
1	DECIMAL	6	CEEEDB_CREATOR_IMPLICIT	LINK SVC
1	DECIMAL	7	CEEEDB_CREATOR_EXEC	POSIX exec()
1	DECIMAL	0	CEEEDBTRMRSN_NORMAL_RETURN	
1	DECIMAL	1	CEEEDBTRMRSN_CEETREN_EXIT	
1	DECIMAL	2	CEEEDBTRMRSN_CEETREC_EXIT	
1	DECIMAL	3	CEEEDBTRMRSN_CEEEXIT_EXIT	_exit()
1	DECIMAL	4	CEEEDBTRMRSN_UNHANDLED_COND	
1	DECIMAL	5	CEEEDBTRMRSN_PTHREAD_EXIT	
1	DECIMAL	6	CEEEDBTRMRSN_QUIESCE	
1	DECIMAL	7	CEEEDBTRMRSN_CEEEXIT_EXEC	exec
1	DECIMAL	1	CEEEDB_PIN_UNSET	
1	DECIMAL	2	CEEEDB_PIN_UNAVAIL	
1	DECIMAL	3	CEEEDB_PIN_SET	
Maximum member ID and maximum member number both relate to the number of CEL members currently supported. The range of member ID values is from 0 to max_member_id.				
4	DECIMAL	17	CEEEDB_MAXMEMID	max member ID
4	DECIMAL	18	CEEEDB_MAXMEMNUM	max member number

Table 14. Enclave data block (EDB) cross reference

Name	Hex Offset	Hex Value	Level
CEEEDB	0		1
CEEEDB_CAACHAIN@	7C		4
CEEEDB_CEEINT_INPUT_R1	98		4
CEEEDB_CEEOSIGR@	60		4
CEEEDB_CEEOSGR1@	84		4
CEEEDB_CICS_OPEN_PROGRAM	80		6

Table 14. Enclave data block (EDB) cross reference (continued)

Name	Hex Offset	Hex Value	Level
CEEEDB_CREATOR_ID	B		4
CEEEDB_DEBUG_TERMID	40		3
CEEEDB_ENVLENGTH	A4		3
CEEEDB_EVNTDEST	80	01	6
CEEEDB_EXTERNAL	0		2
CEEEDB_HPLINK	80		6
CEEEDB_INITIAL_AMODE	8	40	5
CEEEDB_LAST_RBADDR	9C		3
CEEEDB_LAST_RBCNT	A0		3
CEEEDB_LER3	4C		3
CEEEDB_LER4	8C		3
CEEEDB_LER5	94		3
CEEEDB_MAIN_HP	80		6
CEEEDB_MVS_BATCH	80	40	6
CEEEDB_OMVS_DUBBED	8	01	5
CEEEDB_PL_ASTRPTR	34		3
CEEEDB_POSIX	8	04	5
CEEEDB_R13_PARENT	48		3
CEEEDB_SIGENABLED	80	80	6
CEEEDB_TERM_DNFR	80	20	6
CEEEDB_XPL_NODLL_FDS	88		4
CEEEDBACTIV	8	20	5
CEEEDBANHP	20		3
CEEEDBBEHP	24		3
CEEEDBCELV	28		3
CEEEDBCURMSGFILEDCBPTR	94		4
CEEEDBCXIT_PAGE	3C		3
CEEEDBDBGEH	1C		3
CEEEDBDEFPLPTR	38		3
CEEEDBELIST	30		3
CEEEDBENVAR	58		4
CEEEDBENVAR_A	168		3
CEEEDBENVIRON	5C		4
CEEEDBENVIRON_A	172		3
CEEEDBEYE	0		3
CEEEDBFLAGS	8		3
CEEEDBFLAGS1	80		4
CEEEDBFLAG1	8		4
CEEEDBFLAG1A	80		5
CEEEDBIPM	9		4
CEEEDBLEOV	54		4
CEEEDBMAINI	8	80	5
CEEEDBMEMBERCOMPAT	8C		4
CEEEDBMEMBERCOMPAT1	8C		5
CEEEDBMEMBERCOMPAT2	8D		5

Enclave Data Block (EDB)

Table 14. Enclave data block (EDB) cross reference (continued)

Name	Hex Offset	Hex Value	Level
CEEEDBMEMBERCOMPAT3	8E		5
CEEEDBMEMBERCOMPAT4	8F		5
CEEEDBMEMBR	C		3
CEEEDBMULTITHREAD	8	02	5
CEEEDBOPTCB	10		3
CEEEDBOTRB	64		4
CEEEDBPARENT	44		3
CEEEDBPCB	2C		3
CEEEDBPICI	8	08	5
CEEEDBPLITASKING	8C	80	6
CEEEDBPM	A		4
CEEEDBPSA24	70		4
CEEEDBPSA31	68		4
CEEEDBPSL24	74		4
CEEEDBPSL31	6C		4
CEEEDBPSRA	78		4
CEEEDBRSNCD	18		3
CEEEDBTHREADSACTIVE	90		4
CEEEDBTIP	8	10	5
CEEEDBURC	14		3

Table 15 describes the EDB fields in more detail.

Table 15. EDB field descriptions

Field	Contents
CEEEDBFLAG1	CEEEDB flags. The bits in this flag byte are defined as follows: <ul style="list-style-type: none"> 0 CEEEDBMAINI: Indicates that a main program has been initialized within the current enclave. Each member language must ensure that a main program written in that language sets this bit when it is initialized. 1 CEEEDB_INITIAL_AMODE: Indicates the amode upon entry into the Language Environment initialization routine. ON indicates a 31-bit entry; OFF indicates a 24-bit entry. 2 CEEEDBACTIV: Indicates the environment is currently active. A preinitialized environment has this bit initially set to zero. 3 CEEEDBTIP: Indicates termination is in progress. 4 CEEEDBPICI: Preinitialization compatibility is active. 5 CEEEDB_POSIX: POSIX(ON) was specified and z/OS UNIX is available. 6 CEEEDBMULTITHREAD: Multithread environment is active. 7 CEEEDB_OMVS_DUBBED: z/OS UNIX is dubbed.
CEEEDBIPM	The initial program mask. This is the result of ORing all of the member language's program mask requirements. Language Environment sets the program mask to this value during initialization.
CEEEDBPM	The current program mask setting.

Table 15. EDB field descriptions (continued)

Field	Contents
CEEEDB_CREATOR_ID	<p>ID of enclave creator. The values defined in this byte are as follows:</p> <ol style="list-style-type: none"> 1 CEEEDB_CREATOR_BINIT: Indicates this is the first enclave in the process created under batch. 2 CEEEDB_CREATOR_RINI: Indicates the enclave was created under CICS. 3 CEEEDB_CREATOR_BCREN: Indicates the enclave was created with the callable service to create enclaves. 4 CEEEDB_CREATOR_PIP1_MAIN: Indicates the enclave was created with preinitialization services for the main routine. 5 CEEEDB_CREATOR_PIP1_SUBR: Indicates the enclave was created with preinitialization services for subroutines. 6 CEEEDB_CREATOR_IMPLICIT: Indicates the enclave was created implicitly with host system services, such as the LINK SVC. 7 CEEEDB_CREATOR_EXEC: Indicates the enclave was created and invoked from the kernel as a result of an exec().
CEEEDBMEMBR	Address of a list of member entries. An entry is reserved for each member known to Language Environment. There is one member list per enclave. For details, see “Language Environment member list and event handler” on page 83.
CEEEDBOPTCB	Address of the options control block. Enclave initialization processes the runtime options and generates the options control block, CEEOCB. There is one CEEOCB per enclave. This pointer makes the runtime options easily available to all members.
CEEEDBURC	User return code. This field contains the return code generated and stored here by the user program. It is augmented by the Language Environment reason code and returned at enclave termination.
CEEEDBRSNCD	Language Environment reason code. The value indicates the reason for Language Environment termination. It augments the return code, and is returned separately at enclave termination.
CEEEDBDBGEH	Debugger event handler. This field holds the address of the debugger event handler, which is loaded by Language Environment. For more information, see Chapter 9, “Debugging and performance analysis,” on page 339.
CEEEDBANHP	Language Environment Anywhere heap ID. This field holds the identification for Language Environment’s defined heap storage that is typically allocated above the 16M line. For more information, see “Dynamic storage (heap) services” on page 201 for more information.
CEEEDBBEHP	Language Environment below heap ID. This field holds the identification for Language Environment’s defined heap storage that is always allocated below the 16M line; see “Dynamic storage (heap) services” on page 201 for more information.
CEEEDBCELV	Address of Language Environment LIBVEC. This field holds the address of Language Environment’s library vector table (LIBVEC). Access to Language Environment routines is through this vector table.
CEEEDBPCB	Address of the process control block. This field holds the address of Language Environment’s process control block (PCB). This allows access to process-level resources and information.
CEEEDBELIST	Address of exit list from the HLL user exit. The address of a list of user exits provided by the user with the HLL user exit. Language Environment copies the value to the EDB.

Enclave Data Block (EDB)

Table 15. EDB field descriptions (continued)

Field	Contents										
CEEEDB_PL_ASTRPTR	Address of the user parameter list varying string pointer.										
CEEEDBDEFPLPTR	The default pointer that is the inbound parameter list.										
CEEEDBCEXIT_PAGE	<i>Cxit_page</i> value for user exit parameter list.										
CEEEDB_DEBUG_TERMID	Debugger terminal ID under CICS.										
CEEEDBPARENT	Address of parent enclave CAA. When the enclave is created, its creator (or parent) needs to provide: <ol style="list-style-type: none"> 1. Enclave termination routine (CEEEDB_TERM). 2. Information where to return to when the enclave terminates along with the environment which is to be restored. 										
CEEEDB_R13_PARENT	Address of DSA enclave creator. CEEEDB_R13_PARENT is a convenient way to provide return information. It is a pointer to the DSA which contains all the registers of the enclave's parent.										
CEEEDB_LER3	External section.										
CEEEDBLEOV	Address of the LIBVEC for z/OS UNIX support.										
CEEEDBENVAR	Address of the environment variable array.										
CEEEDBENVAR_A	Address of the alternate environment variable array.										
CEEEDBENVIRON	Address of the environment variable anchor.										
CEEEDBENVIRON_A	Address of the alternate environment variable anchor.										
CEEEDB_CEEOSIGR@	Address of the CEEOSIGR routine.										
CEEEDBOTRB	Address of the in-core wrapping trace table										
CEEEDBPSA31	Address and preallocated 31 storage.										
CEEEDBPSL31	Length of preallocated 31 storage.										
CEEEDBPSA24	Address of preallocated 24 storage.										
CEEEDBPSL24	Length of preallocated 24 storage.										
CEEEDBPSRA	Address of overflow routine.										
CEEEDBFLAG1A	Additional EDB flags, as follows: <table border="0" style="margin-left: 2em;"> <tr> <td>0</td> <td>CEEEDB_SIGENABLED: Signal processing enabled.</td> </tr> <tr> <td>1</td> <td>CEEEDB_MVS_BATCH: Running z/OS batch first enclave.</td> </tr> <tr> <td>2</td> <td>CEEEDB_TERM_DNFR: Do not free heap or delete programs at enclave termination.</td> </tr> <tr> <td>3</td> <td>CEEEDB_ENVTDEST: Running destructors.</td> </tr> <tr> <td>4-7</td> <td>Reserved</td> </tr> </table>	0	CEEEDB_SIGENABLED: Signal processing enabled.	1	CEEEDB_MVS_BATCH: Running z/OS batch first enclave.	2	CEEEDB_TERM_DNFR: Do not free heap or delete programs at enclave termination.	3	CEEEDB_ENVTDEST: Running destructors.	4-7	Reserved
0	CEEEDB_SIGENABLED: Signal processing enabled.										
1	CEEEDB_MVS_BATCH: Running z/OS batch first enclave.										
2	CEEEDB_TERM_DNFR: Do not free heap or delete programs at enclave termination.										
3	CEEEDB_ENVTDEST: Running destructors.										
4-7	Reserved										
CEEEDB_CICS_OPEN_PROGRAM	<table border="0" style="margin-left: 2em;"> <tr> <td>0</td> <td>Program may run on OTE or QR TCB.</td> </tr> <tr> <td>1</td> <td>Program runs only on an OTE TCB and can use Open C functions.</td> </tr> </table>	0	Program may run on OTE or QR TCB.	1	Program runs only on an OTE TCB and can use Open C functions.						
0	Program may run on OTE or QR TCB.										
1	Program runs only on an OTE TCB and can use Open C functions.										
CEEEDB_MAIN_HP	Main uses XPLINK linkage.										
CEEEDB_HPLINK	XPLINK is being used.										
CEEEDB_CEEOSGR1@	CEEOSIGR and address										
CEEEDB_XPL_NODLL_FDS	Pointer to a chain of XPLINK compatibility descriptors representing NODLL function pointers.										
CEEEDB_ENVLENGTH	Length to envar array of pointers.										

Language Environment process control block

Each process is represented by a Process Control Block (PCB). All process resources are anchored, provided for, or can be obtained through the PCB. The PCB is generated during process initialization and deleted during process termination. Fields in the PCB should be used as described in other sections of this document

The following tables show the format of the PCB.

- Table 16 shows the PCB fields and Table 19 on page 72 describes their contents.
- Table 17 on page 70 shows the PCB constants.
- Table 18 on page 71 shows the PCB cross reference information.

Table 16. Process control block (PCB) field descriptions

Offsets		Type	Len	Name (* = Reserved)	Description
Dec	Hex				
0	(0)	STRUCTURE	76	CEEPCB	PCB mapping
0	(0)	CHARACTER	76	CEEPCB_EXTERNAL	External portion
0	(0)	CHARACTER	8	CEEPCBEYE	Eyecatcher 'CEEPCB '
8	(8)	BITSTRING	1	CEEPCBSYSTEM	Underlying Operating System
9	(9)	BITSTRING	1	CEEPCBHRDWR	Underlying Hardware
10	(A)	BITSTRING	1	CEEPCBSBSYS	Underlying Subsystem
11	(B)	BITSTRING	1	CEEPCBFLAG2	
		1...		CEEPCBBIMODAL	Bimodal addressing is avail.
		.1..		CEEPCB_LVFORM	LIBVEC format 1=stat./0=dynam
		..1.		CEEPCB_VECTOR	Vector hardware available
		...1		CEEPCB_CL24	CEL Libvec AMODE24 is built
	 1..		CEEPCB_OMVS	z/OS UNIX is up and available
	1..		*	RESERVED
	1.		CEEPCB_PICI	PICI environment
	1		CEEPCB_REUSE	This CCIS process contains a reusable enclave environment
12	(C)	ADDRESS	4	CEEPCBDBGEH	A(debug event handler)
16	(10)	CHARACTER	8	CEEPCBDBGSRVD	Reserved for debugger
24	(18)	ADDRESS	4	CEEPCBDMEMBR	A(process member list)
28	(1C)	ADDRESS	4	CEEPCB_ZLOD	A(process load routine)
32	(20)	ADDRESS	4	CEEPCB_ZDEL	A(process delete routine)
36	(24)	ADDRESS	4	CEEPCB_ZGETST	A(process get storage rtn)
40	(28)	ADDRESS	4	CEEPCB_ZFREEST	A(process free storage rtn)
44	(2C)	ADDRESS	4	CEEPCB_LVTL	Address of a Lang Env library table that contains info about Lang Env libvecs, to determine which transfer vector should be used to access a library routine and be signal safed.
48	(30)	ADDRESS	4	CEEPCBRCB	Address of the RCB
52	(34)	ADDRESS	4	CEEPCB_SYSEIB	A(CICS System EIB)
The following three fields are used by the CEEYGPPS (get permanent process storage) macro. This macro allows the member languages to quickly allocate storage at the process level that is freed only by CEL after member process termination.					
56	(38)	SIGNED	4	CEEPCBPSL	Length of perm process stg
60	(3C)	ADDRESS	4	CEEPCBPSA	Addr of perm process stg
64	(40)	ADDRESS	4	CEEPCBPSRA	Perm process stg overflow routine address

Process Control Block (PCB)

Table 16. Process control block (PCB) field descriptions (continued)

Offsets		Type	Len	Name (* = Reserved)	Description
Dec	Hex				
68	(44)	BITSTRING	4	CEEPCH_OMVS_LEVEL	z/OS UNIX release level (Multiple bits may be set)
		1...		*	Reserved
		.1..		CEEPCH_OMVS_1120	HOM1120 functions are present.
		..1.		CEEPCH_OMVS_1130	HOM1130 functions are present.
68	(44)	BITSTRING	3	*	Reserved
72	(48)	ADDRESS	4	CEEPCH_CHAIN	Pointer to next PCB on PICI environment chain
76	(4C)	ADDRESS	4	CEEPCH_VSSFE	Address of the stack segment free routine
80	(50)	ADDRESS	4	CEEPCHPRFEH	Address of profile event handler
84	(54)	BITSTRING	1	CEEPCHFLAG6	Additional PCB flags
		1...	4	CEEPCH_ESAME	ESAME supported
		.111 1111		*	Reserved
85	(55)	CHARACTER	3	CEEPCH_RSRVED	Reserved
88	(58)	ADDRESS	4	*	Reserved
92	(5C)	ADDRESS	4	CEEPCH_DBGINFO	Address of debugger Info block

Table 17. Process control block (PCB) constants

Len	Type	Value	Name	Description
Constants				
4	DECIMAL	16384	CEEPCH_IS_SIZE	Init dummy stk size
4	DECIMAL	2048	CEEPCH_LIS_SIZE	Init dummy lib size
CAUTION: CEEPCH_IS_SIZE and CEEPCH_LIS_SIZE must be multiple of doubleword size.				
4	DECIMAL	8	CEEPCH_MAXLVTNUM	Maximum library transfer vector tables in Lang Env
Declare constants for operating system, hardware, and subsystem CEEPCHSYSTEM, CEEPCHHRDWR, CEEPCHSYS				
1	DECIMAL	0	CEEPCHSYUND	Undefined
1	DECIMAL	1	CEEPCHSYUNS	Unsupported
1	DECIMAL	2	CEEPCHSYVM	VM
1	DECIMAL	3	CEEPCHSYMVS	z/OS Underlying Hardware
1	DECIMAL	0	CEEPCHHWUND	Undefined
1	DECIMAL	1	CEEPCHHWUNS	Unsupported
1	DECIMAL	2	CEEPCHHW370	System/370 non-X
1	DECIMAL	3	CEEPCHHWXA	System/370 XA
1	DECIMAL	4	CEEPCHHWESA	System/370 ESA Underlying Subsystem
1	DECIMAL	0	CEEPCHSSUND	Undefined
1	DECIMAL	1	CEEPCHSSUNS	Unsupported
1	DECIMAL	2	CEEPCHSSNON	No subsystem
1	DECIMAL	3	CEEPCHSSTSO	TSO
1	DECIMAL	5	CEEPCHSSCIC	CICS
Declare constants describing state of process				
1	DECIMAL	0	CEEPCHSTATE_INIT	Process init
1	DECIMAL	1	CEEPCHSTATE_TERM	Process term

Table 17. Process control block (PCB) constants (continued)

Len	Type	Value	Name	Description
1	DECIMAL	2	CEEPCBSTATE_ACTIVE	Process active

Table 18. Process control block (PCB) cross reference

Name	Hex Offset	Hex Value	Level
CEEPCB	0		1
CEEPCB_CHAIN	48		3
CEEPCB_CL24	B	10	4
CEEPCB_DBGINFO	5C		3
CEEPCB_ESAME	54	80	4
CEEPCB_EXTERNAL	0		2
CEEPCB_LVFORM	B	40	4
CEEPCB_LVTL	2C		3
CEEPCB_OMVS	B	08	4
CEEPCB_OMVS_LEVEL	44		3
CEEPCB_OMVS_1120	44	40	4
CEEPCB_OMVS_1130	44	20	4
CEEPCB_PICI	B	02	4
CEEPCB_REUSE	B	01	4
CEEPCB_SYSEIB	34		3
CEEPCB_VECTOR	B	20	4
CEEPCB_VSSFE	4C		3
CEEPCB_ZDEL	20		3
CEEPCB_ZFREEST	28		3
CEEPCB_ZGETST	24		3
CEEPCB_ZLOD	1C		3
CEEPCBBIMODAL	B	80	4
CEEPCBDBGEGH	C		3
CEEPCBDBGRSVD	10		3
CEEPCBDMEMBR	18		3
CEEPCBEYE	0		3
CEEPCBFLAG2	B		3
CEEPCBFLAG6	54		3
CEEPCBHRDWR	9		3
CEEPCBPRFEH	50		3
CEEPCBPSA	3C		3
CEEPCBPSL	38		3
CEEPCBPSRA	40		3
CEEPCBRCB	30		3
CEEPCBSBSYS	A		3
CEEPCBSYSTEM	8		3
RESERVED	B	04	4

Table 19 on page 72 describes the PCB fields in more detail.

Process Control Block (PCB)

Table 19. PCB field descriptions

Field	Contents
CEEPCBEYE	8-character eyecatcher 'CEEPCB'.
CEEPCBSYSTEM	Underlying operating system. The value indicates the operating system supporting the active program. The values are defined as follows: 0 Undefined — this value should never occur after initializing Language Environment 1 Unsupported 2 VM/ESA 3 z/OS
CEEPCBHRDWR	Underlying hardware The value indicates the type of hardware on which the program is executing; the values are defined as follows: 0 Undefined — this value should never occur after initializing Language Environment 1 Unsupported 2 System/370, non-XA 3 System/370, XA 4 System/370, ESA
CEEPCBSBSYS	Underlying subsystem The value indicates the subsystem, if any, on which the program is executing; the values are defined as follows: 0 Undefined — this value should never occur after initializing Language Environment 1 Unsupported 2 None — the program is not executing under a subsystem according to Language Environment 3 TSO 4 Reserved 5 CICS 6 - 7 Reserved
CEEPCBFLAG2	PCB flag bits; the bits are defined as follows: 0 CEEPCBBIMODAL – When 1, this indicates the hardware is capable of bimodal addressing 1 CEEPCB_LVFORM – Reserved 2 CEEPCB_VECTOR – When 1, the vector facility is available on the hardware 3 CEEPCB_CL24 – LIBVEC for AMODE24 is available 4 CEEPCB_OMVS – z/OS UNIX is up and available 5 Reserved 6 CEEPCB_PICI – PICI environment is in effect 7 CEEPCB_REUSE – When 1, the CICS process contains a reusable enclave environment. This flag is required to indicate how Language Environment will getmain, freemain, load, or delete resources upon requests in a reusable enclave environment. These resources must be freed explicitly during transaction termination.
CEEPCBDBGEH	Address of the debug tool event handler. This field holds the address of the debug tool event handler. When this field is zero, a debug tool has not been initialized.
CEEPCBDBGRSVD	Reserved for the debug tool's use. A doubleword that is reserved for the debug tool's use. It is zeroed by Language Environment process initialization.
CEEPCBMEMBR	Address of the process level member list. An entry is reserved for each member known to Language Environment. There is one member list per process. The process level member list has the same format as the enclave level member list. For details, see "Language Environment member list and event handler" on page 83.
CEEPCB_ZLOD	Process level LOAD service. This is the address of a LOAD service. Routines loaded using this service persist across enclaves within this process. For details, see "Loading and deleting programs in different environments" on page 289.

Table 19. PCB field descriptions (continued)

Field	Contents
CEEPB_ZDEL	Process level DELETE service. This is the address of a DELETE service. Routines loaded using CEEPB_ZLOD must be deleted using this service. For details, see "Loading and deleting programs in different environments" on page 289.
CEEPB_ZGETST	Process level GETMAIN service. This is the address of a GETMAIN service. Storage obtained using this service persist across enclaves within this process.
CEEPB_ZFREEST	Process level FREEMAIN service. This is the address of a FREEMAIN service. Storage obtained using CEEPB_ZGETST must be freed using this service.
CEEPB_LVTL	Address of a Language Environment library vector.
CEEPBRCB	Address of the RCB.
CEEPB_SYSEIB	Address of CICS system EIB.
CEEPBPSL	Length of permanent process storage. This field is used by the CEEXGPPS (get permanent process storage) macro. This macro allows the member languages to quickly allocate storage at the process level that is freed only by Language Environment after member process termination.
CEEPBPSA	Address of permanent process storage. This field is used by the CEEXGPPS (get permanent process storage) macro. This macro allows the member languages to quickly allocate storage at the process level that is freed only by Language Environment after member process termination.
CEEPBPSRA	Permanent process storage overflow routine address table which contains information for all Language Environment LIBVECs that allow signal safing of Language Environment library for asynchronous signals. This field is used by the CEEXGPPS (get permanent process storage) macro. This macro allows the member languages to quickly allocate storage at the process level that is freed only by Language Environment after member process termination.
CEEPB_OMVS_LEVEL	z/OS UNIX release level. The flags are as follows: 0 Reserved 1 HOM1120 functions are present 2 HOM1130 functions are present
CEEPB_CHAIN	Used to run the PICI environment chain; it will be NULL when there is no next environment in the chain.
CEEPB_VSSFE	Address of the stack segment free routine.
CEEPBPRFEH	Address of the profile event handler
CEEPBFLAG6	Additional PCB flag bits. The bits are defined as follows: 0 CEEPB_ESAME 1 Level 1 tracing on 2 Level 2 tracing on 3 Debugger was HFS loaded 4 - 7 Reserved
CEEPB_DBGINFO	Address of the debugger info block.

Language Environment region control block

Regions are defined to effectively manage the resources for multiple processes, allowing, for instance, for the reuse of resources. Regions are:

- Internally defined
- Initialized once for each environment
- Not part of the program model
- Not visible to the HLL programmer

Region Control Block (RCB)

For example, under CICS, each CICS thread corresponds to a CEE process. Resources that are common to multiple CICS threads (or CEE processes) are managed at the region level. The region control block (RCB) provides access to region-level resources.

Although CICS is the only environment that has multiple processes in a single region, regions exist for all environments and region initialization/termination events are called in all environments, not just for CICS. For this reason, you should write all event handlers so that resources created during region initialization can be shared across multiple processes. Region and process initialization/termination events should be designed for the possibility of having multiple processes sharing a single Language Environment region in environments other than CICS.

There is one RCB per instance of a Language Environment environment and there is no link between RCB in separate Language Environment environments. The following tables show the format of the RCB.

- Table 20 shows the RCB fields and Table 23 on page 76 describes their contents.
- Table 21 on page 75 shows the RCB constants.
- Table 22 on page 75 shows the RCB cross reference information.

Table 20. Region control block (RCB) field descriptions

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
0	(0)	STRUCTURE	48	CEERCB	RCB mapping
0	(0)	CHARACTER	48	CEERCB_EXTERNAL	External portion
0	(0)	CHARACTER	8	CEERCBEYE	Eyecatcher 'CEERCB '
8	(8)	BITSTRING	1	CEERCBSYSTEM	Underlying Operating System
9	(9)	BITSTRING	1	CEERCBHRDWR	Underlying Hardware
10	(A)	BITSTRING	1	CEERCBSBSYS	Underlying Subsystem
11	(B)	BITSTRING	1	CEERCBFLAGS	
		1...		CEERCBBIMODAL	Bimodal addressing is avail.
		.1..		CEERCBLRR	ON= Lib Routine Retention is in effect
		..1.		CEERCBLRRTR	ON= Lib Routine Retention is being terminated
		...1 1111		*	Reserved
12	(C)	ADDRESS	4	CEERCB_PMUSER	Address of pattern-match work area
16	(10)	SIGNED	4	*	Reserved
20	(14)	ADDRESS	4	CEERCBDMEMBR	A(region member list)
24	(18)	ADDRESS	4	CEERCB_ZLOD	A(region load routine)
28	(1C)	ADDRESS	4	CEERCB_ZDEL	A(region delete routine)
32	(20)	ADDRESS	4	CEERCB_ZGETST	A(region get storage rtn)
36	(24)	ADDRESS	4	CEERCB_ZFREEST	A(region free storage rtn)
40	(28)	SIGNED	4	CEERCB_VERSION_ID	Ver., Rel., and Mod. of Language Environment
44	(2C)	ADDRESS	4	CEERCB_PCBCHAIN	Head of the PICI environment chain
48	(30)	SIGNED	4	CEERCB_REUSE_STATE	Runtime reuse state
52	(34)	BITSTRING	4	CEERCB_CICS_FLAGS	CICS flags from Partition Initialization Call
		1...		CEERCB_CICS_POK_OK	CICS indicated Program Objects are supported

Table 20. Region control block (RCB) field descriptions (continued)

Offsets		Type	Len	Name (Dim) (* = Reserved)	Description
Dec	Hex				
		.1..		*	Reserved
		..1.		CEERCB_CICS_OTE	CICS OTE is supported
		...1		CEERCB_CICS_RRWA_OK	CICS indicated Reusable Rununit Work Areas are available
	 1...		CEERCB_CICS_OTE2_OK	CICS OTE II is supported
	11.		*	Reserved
	1		CEERCB_CICS_TRANS_OK	CICS dump data set is supported
56	(38)	ADDRESS	4	CEERCB_CICS_QR_TCB	CICS QR TCB address
60	(3C)	ADDRESS	4	CEERCB_PMADDR	Address of a pattern-match function

Table 21. Region control block (RCB) constants

Len	Type	Value	Name	Description
Declare constants for operating system, hardware, and subsystem CEERCBSYSTEM, CEERCBHRDWR, CEERCBSBSYS				
1	DECIMAL	0	CEERCBSYUND	Undefined
1	DECIMAL	1	CEERCBSYUNS	Unsupported
1	DECIMAL	2	CEERCBSYVM	VM
1	DECIMAL	3	CEERCBSYMVS	z/OS Underlying Hardware
1	DECIMAL	0	CEERCBHWUND	Undefined
1	DECIMAL	1	CEERCBHWUNS	Unsupported
1	DECIMAL	2	CEERCBHW370	System/370, non-XA
1	DECIMAL	3	CEERCBHWXA	System/370 XA
1	DECIMAL	4	CEERCBHWESA	System/370 ESA Underlying Subsystem
1	DECIMAL	0	CEERCBSSUND	Undefined
1	DECIMAL	1	CEERCBSSUNS	Unsupported
1	DECIMAL	2	CEERCBSSNON	No subsystem
1	DECIMAL	3	CEERCBSSTSO	TSO
1	DECIMAL	5	CEERCBSSCIC	CICS
1	DECIMAL	0	CEERCB_REUSE_NONE	Not a reuse environment
1	DECIMAL	1	CEERCB_REUSE_FULL	Reuse, full init is needed
1	DECIMAL	2	CEERCB_REUSE_PART	Reuse, partial init is needed
1	DECIMAL	3	CEERCB_REUSE_TERM	Terminate the reuse environment

Table 22. Region control block (RCB) cross reference

Name	Hex Offset	Hex Value	Level
CEERCB	0		1
CEERCB_CICS_QR_TCB	38		3
CEERCB_EXTERNAL	0		2
CEERCB_PCBCCHAIN	2C		3
CEERCB_PMADDR	3C		3
CEERCB_PMUSER	C		2
CEERCB_REUSE_STATE	30		3
CEERCB_VERSION_ID	28		3

Region Control Block (RCB)

Table 22. Region control block (RCB) cross reference (continued)

Name	Hex Offset	Hex Value	Level
CEERCB_ZDEL	1C		3
CEERCB_ZFREEST	24		3
CEERCB_ZGETST	20		3
CEERCB_ZLOD	18		3
CEERCBBIMODAL	B	80	4
CEERCBDMEMBR	14		3
CEERCBEYE	0		3
CEERCBFLAGS	B		3
CEERCBHRDWR	9		3
CEERCBLRR	B	40	4
CEERCBLRRTR	B	20	4
CEERCBSBSYS	A		3
CEERCBSYSTEM	8		3

Table 23 describes the RCB fields in more detail.

Table 23. RCB field descriptions

Field	Contents
CEERCBSYSTEM	Underlying operating system. The value indicates the operating system supporting the active program and are defined as follows: 0 Undefined; this value should never occur after initializing Language Environment 1 Unsupported 2 VM/ESA 3 z/OS
CEERCBHRDWR	Underlying hardware. The value indicates the type of hardware on which the program is executing and are defined as follows: 0 Undefined; this value should never occur after initializing Language Environment 1 Unsupported 2 System/370, non-XA 3 System/370, XA 4 System/370, ESA
CEERCBSBSYS	Underlying subsystem. The value indicates the subsystem, if any, on which the program is executing; they are defined as follows: 0 Undefined — this value should never occur after initializing Language Environment 1 Unsupported 2 None — the program is not executing under a subsystem according to Language Environment 3 TSO 4 Undefined 5 CICS
CEERCBFLAGS	A byte containing various flags. The flags are defined in the bits of the byte, from high order to low order, as follows: 0 CEERCBBIMODAL; bimodal addressing is available 1 CEERCBLRR; ON= Lib Routine Retention is in effect 2 CEERCBLRRTR; ON= Lib Routine Retention is is being terminated 3–7 Reserved
CEERCBDMEMBR	Address of the region member list.

Table 23. RCB field descriptions (continued)

Field	Contents
CEERCB_PMUSER	Address of work area to be given to pattern match routine when called.
CEERCB_ZLOD	Address of region-level load routine. The parameters to this routine are the same as for the process-level load routine. The modules loaded by this routine remain loaded until explicitly deleted by the region-level delete routine.
CEERCB_ZDEL	Address of the region-level delete routine. The parameters to this routine are the same as for the process-level delete routine.
CEERCB_ZGETST	Address of the region-level GETMAIN routine. The parameters to this routine are the same as for the process-level GETMAIN routine. Storage obtained by this routine remains allocated until explicitly freed by the region-level FREEMAIN routine. The <i>user_word</i> parameter should not be filled in.
CEERCB_ZFREEST	Address of the region-level FREEMAIN routine. The parameters to this routine are the same as for the process-level FREEMAIN routine. The <i>user_word</i> parameter should not be filled in.
CEERCB_VERSION_ID	A fullword integer that contains the Language Environment Product Number, Version, Release, and Modification levels; the levels are presented as hexadecimal values. This field is useful when debugging a problem using a static dump. The field's structure and an example are shown below: byte 0 Product number in hex byte 1 Version in hex byte 2 Release in hex byte 3 Modification level in hex
CEERCB_PCBCHAIN	Points to the first PCB on the chain. The PCB may belong to either a batch environment or PICI environment. The field may be NULL, which would be the case under CICS and for PIP1 processes.
CEERCB_REUSE_STATE	Indicates the runtime reuse state; the following values are defined: 0 This is not a reuse environment 1 Reuse environment, full initialization is needed 2 Reuse environment, partial initialization is needed 3 Terminate the reuse environment
CEERCB_CICS_FLAGS	A byte containing various flags. The flags are defined in the bits of the byte, from high order to low order, as follows: 0 CEERCB_CICS_POK_OK; CICS indicated Program Objects are supported 2 CEERCB_CICS_OTE; CICS OTE is supported 3 CEERCB_CICS_RRWA_OK; CICS indicated Reusable Rununit Work Areas are available 4 CEERCB_CICS_OTE2_OK; CICS OTE II is supported 7 CEERCB_CICS_TRANS_OK; CICS Dump data set is supported 1, 5, 6 Reserved
CEERCB_CICS_QR_TCB	Address of CICS QR TCB.
CEERCB_PMADDR	Address of a 31-bit pattern-match routine.

Note:

1. CICS SPF: The RCB can be in key 8 storage under CICS. Member languages should assume the storage is write-protected when running in key 9. Also, storage allocated by the region-level GETMAIN service is always in key 8 under CICS. It is write protected when running in key 9.
2. Storage allocated at the region level should be released at the region level and storage allocated at the process level should be released at the process level.

Example of a condition information block

The code example (below) shows a sample condition information block for AMODE 31 applications.

OFFSETS					
DEC	HEX	TYPE	LEN	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	268	CEEICB	

Condition Information Block - Prefix area. Area 0					

0	(0)	CHARACTER	4	CIB_EYE	Eye catcher.
4	(4)	ADDRESS	4	CIB_BACK	Previous CIB.
8	(8)	ADDRESS	4	CIB_FWRD	Next CIB.
12	(C)	SIGNED	2	CIB_SIZ	Size of ceexeb
14	(E)	SIGNED	2	CIB_VER	Version code of ceexeb
16	(10)	SIGNED	4	CIB_PLAT_ID	Action Code.
20	(14)	SIGNED	4	*	Reserved. Do not use.

CIB Area provides for CSC Information. Area 1					

24	(18)	CHARACTER	120	CIB_AREA1	
24	(18)	BITSTRING	12	CIB_COND	Current Lang Env Condition
24	(18)	BITSTRING	8	CIB_COND_64	Condition Ident.
32	(20)	ADDRESS	4	CIB_MIB	Pointer to the msg insert area.
36	(24)	ADDRESS	4	CIB_MACHINE	Address of associated machine the Exception. I_Pgm_loc
40	(28)	BITSTRING	12	CIB_OLD_COND	Initial Lang Env Condition
40	(28)	BITSTRING	8	CIB_OLD_COND_64	Condition Ident.
48	(30)	ADDRESS	4	CIB_OLD_MIB	Pointer to the msg insert area.
52	(34)	BITSTRING	4	CIB_CSC_FLG	
52	(34)	BITSTRING	1	CIB_FLG_1	
53	(35)	BITSTRING	1	CIB_FLG_2	
54	(36)	BITSTRING	1	CIB_FLG_3	
55	(37)	BITSTRING	1	CIB_FLG_4	
		1...		*	Reserved.
		.1..		*	Reserved.
		..1.		*	Reserved.
		...1		*	Reserved.
	 1...		CIB_RSM_MVE	Resume cursor moved explicit
	1..		CIB_MSG_OUT	Message service processed condition.
	1.		CIB_RSM_MVR	Resume cursor moved relative.
	1		*	Reserved.
56	(38)	CHARACTER	12	CIB_HDL	the exception The HandleCursor.
56	(38)	ADDRESS	4	CIB_HDL_SF	Pointer to Stack Frame.
60	(3C)	CHARACTER	8	CIB_HDL_ENTRY	Pointer to Current Handler
60	(3C)	ADDRESS	4	CIB_HDL_EPT	Pointer to entry
64	(40)	ADDRESS	4	CIB_HDL_RST	Pointer to Language Specific data
68	(44)	CHARACTER	12	CIB_RSM	The Resume Cursor.
68	(44)	ADDRESS	4	CIB_RSM_SF	Save area part.
72	(48)	ADDRESS	4	CIB_RSM_POINT	Instruction address part
76	(4C)	ADDRESS	4	CIB_RSM_MACHINE	Address of associated machine State.
80	(50)	SIGNED	4	CIB_COND_DEFAULT	Default condition handler.
84	(54)	ADDRESS	4	CIB_PH_CALLEE_SF	Physical callee DSA ptr
88	(58)	CHARACTER	1	CIB_HDL_SF_FMT	Stack format for CIB_HDL_SF (0 = up, 1 = down)
89	(59)	CHARACTER	1	CIB_PH_CALLEE_SF_FMT	Stack format for CIB_PH_CALLEE_SF (0 = up, 1 = down)
90	(5A)	CHARACTER	54	*	Reserved.

Vector hardware and math routines support Area 3					

144	(90)	CHARACTER	32	CIB_VMA	Vector and math support
144	(90)	CHARACTER	8	CIB_VSR	Vector status register save area
152	(98)	ADDRESS	4	CIB_VSTOR	A(of vector envir. save areas)
156	(9C)	ADDRESS	4	CIB_VRPSA	A(1st vector reg pair save area)
160	(A0)	ADDRESS	4	CIB_MCB	A(MCB at time of interrupt)
164	(A4)	CHARACTER	8	CIB_MRN	Math routine name
172	(AC)	BITSTRING	1	CIB_MFLAG	Math flag
		1...		CIB_MDSF1B0	ON for callable service and CWI, else OFF
		.1..		CIB_MDSF1B1	ON for callable service, else OFF
		..11 1111		*	Method of math invocation
173	(AD)	CHARACTER	3	*	Reserved

Language Environment Exception Manager Flags. Area 4

176	(B0)	BITSTRING	4	CIB_BIT	Status Flags.
176	(B0)	BITSTRING	1	CIB_FLG_5	Language Environment Event
		1... ..		CIB_ABF	ABEND Caused.
		.1.. ..		CIB_PCF	Program Check Caused.
		..1.		CIB_KILL	Signal via CEEOKILL
		...1		*	Empty
	 1..		CIB_TIU	Condition management raised TIU
	1..		CIB_PROMO	New condition result from a promote.
	1.		CIB_SGL	Signaled condition.
	1		CIB_EXT	Attention Interrupt Caused.
177	(B1)	BITSTRING	1	CIB_FLG_6	Language Environment Actions
		1... ..		CIB_ARCV	Abend reason code valid.
		.1.. ..		CIB_MRC	Math routine condition.
		..1.		CIB_ALW_RSM	Allow resume operation.
		...1		CIB_MRC_TYP	MRC type 1.
	 1..		CIB_ENABLE_ONLY	Enable only pass (no cond. pass)
	1..		CIB_OWNING_SF	Hcursor pointing to owning SF
	1.		CIB_SF0	Doing post SF0 scan.
	1		CIB_TC_DONE	Members informed of condition.
178	(B2)	BITSTRING	1	CIB_FLG_7	Named Conditions.
		1... ..		CIB_STG	Storage Condition.
		.1.. ..		CIB_SDWA_SET	Indicates an SDWA is associated with the condition
		..1.		*	Empty.
		...1		*	Empty.
	 1..		*	Empty.
	1..		*	Empty.
	1.		*	Empty.
	1		CIB_NOREC	Do not allow recursion cond
179	(B3)	BITSTRING	1	CIB_FLG_8	Flags used to ask for dump.

Language Environment Extras. Area 5

180	(B4)	CHARACTER	88	CIB_AREA5	
ABEND Codes copied from the SDWA					
180	(B4)	SIGNED	4	CIB_ABCD	Abend code word.
184	(B8)	SIGNED	4	CIB_ABRC	Abend Reason Word.
188	(BC)	CHARACTER	8	CIB_ABNAME	Abend. load module name in sdwa

Information relating to the most significant Save area.

196	(C4)	ADDRESS	4	CIB_PL	Pointer to the prolog see cib_ppav for version of PAA we are pointing at
200	(C8)	ADDRESS	4	CIB_SV2	Save area of first significant Language Environment Program
204	(CC)	ADDRESS	4	CIB_SV1	Address of save area at time of the exception.
208	(D0)	ADDRESS	4	CIB_INT	Address of instruction causing

Miscellaneous information.

212	(D4)	BITSTRING	4	CIB_Q_DATA_TOKEN	Token passed by CICS Routine.
216	(D8)	ADDRESS	4	CIB_FDBK	Address of feedback token for signaled conditions.
220	(DC)	SIGNED	4	CIB_FUN	Member list function code
224	(E0)	CHARACTER	4	CIB_TOKE	Token from CEEHDL routines or Ptr to SF that cased the poll to invoke Mbr Ex Handler
228	(E4)	CHARACTER	4	CIB_MID	ID Code at time of interrupt.
232	(E8)	SIGNED	4	CIB_STATE	Codes used to identify activity associated with this event.
236	(EC)	SIGNED	4	CIB_RTCC	Action Code.
240	(F0)	SIGNED	4	CIB_PPAV	Version of PPA in Cib_pl -1 = C/370 Version 1 1 = Language Environment 1.1.0
244	(F4)	CHARACTER	8	CIB_AB_TERM_EXIT	Name of the abnorm term exit in control.
252	(FC)	ADDRESS	4	CIB_SDWA_PTR	Address of SDWA associated with the condition.
256	(100)	UNSIGNED	4	CIB_SIGNO	Signal number (This field will be zero if the associated condition has not been mapped to

260	(104)	ADDRESS	4	CIB_PPSD	a signal)
					Pointer to Lang Env's copy of the PPSD (For a description of the PPSD, see z/OS UNIX's BPXYPPSD or Language Environment's CEEOSIGD)
264	(108)	CHARACTER	4	*	Reserved.

The code example (below) shows the cross reference summary of the condition information block for AMODE 31 applications.

NAME	HEX OFFSET	HEX VALUE	LEVEL
CEEICIB	0		1
CIB_AB_TERM_EXIT	F4		3
CIB_ABCD	B4		3
CIB_ABF	B0	80	4
CIB_ABNAME	BC		3
CIB_ABRC	B8		3
CIB_ALW_RSM	B1	20	4
CIB_ARCV	B1	80	4
CIB_AREA1	18		2
CIB_AREA5	B4		2
CIB_BACK	4		2
CIB_BIT	B0		2
CIB_COND	18		3
CIB_COND_DEFAULT	50		3
CIB_COND_64	18		4
CIB_CSC_FLG	34		3
CIB_ENABLE_ONLY	B1	08	4
CIB_EXT	B0	01	4
CIB_EYE	0		2
CIB_FDBK	D8		3
CIB_FLG_1	34		4
CIB_FLG_2	35		4
CIB_FLG_3	36		4
CIB_FLG_4	37		4
CIB_FLG_5	B0		3
CIB_FLG_6	B1		3
CIB_FLG_7	B2		3
CIB_FLG_8	B3		3
CIB_FUN	DC		3
CIB_FWRD	8		2
CIB_HDL	38		3
CIB_HDL_ENTRY	3C		4
CIB_HDL_EPT	3C		5
CIB_HDL_RST	40		5
CIB_HDL_SF	38		4
CIB_HDL_SF_FMT	58		3
CIB_INT	D0		3
CIB_KILL	B0	20	4
CIB_MACHINE	24		3
CIB_MCB	A0		3
CIB_MDSF1B0	AC	80	4
CIB_MDSF1B1	AC	40	4
CIB_MFLAG	AC		3
CIB_MIB	20		4
CIB_MID	E4		3
CIB_MRC	B1	40	4
CIB_MRC_TYP	B1	10	4
CIB_MRN	A4		3
CIB_MSG_OUT	37	04	5
CIB_NOREC	B2	01	4
CIB_OLD_COND	28		3
CIB_OLD_COND_64	28		4
CIB_OLD_MIB	30		4
CIB_OWNING_SF	B1	04	4
CIB_PCF	B0	40	4
CIB_PH_CALLEE_SF	54		3
CIB_PH_CALLEE_SF_FMT	59		3
CIB_PL	C4		3
CIB_PLAT_ID	10		2
CIB_PPAV	F0		3
CIB_PPSD	104		3
CIB_PROMO	B0	04	4
CIB_Q_DATA_TOKEN	D4		3

CIB_RSM	44		3
CIB_RSM_MACHINE	4C		4
CIB_RSM_MVE	37	08	5
CIB_RSM_MVR	37	02	5
CIB_RSM_POINT	48		4
CIB_RSM_SF	44		4
CIB_RTCC	EC		3
CIB_SDWA_PTR	FC		3
CIB_SDWA_SET	B2	40	4
CIB_SF0	B1	02	4
CIB_SGL	B0	02	4
CIB_SIGNO	100		3
CIB_SIZ	C		2
CIB_STATE	E8		3
CIB_STG	B2	80	4
CIB_SV1	CC		3
CIB_SV2	C8		3
CIB_TC_DONE	B1	01	4
CIB_TIU	B0	08	4
CIB_TOKE	E0		3
CIB_VER	E		2
CIB_VMA	90		2
CIB_VRPSA	9C		3
CIB_VSR	90		3
CIB_VSTOR	98		3

Example of a machine state block

The example below shows a sample of the machine state block.

Note:

1. After program checks, if the TRAP(ON,NOSPIE) and ALL31(OFF) runtime options are in effect, the HR_VALID flag bit in the Machine State FLAGS field will be off; this indicates that the saved high registers are not valid. After ABENDs, if the ALL31(OFF) runtime option is in effect, the HR_VALID flag bit in the Machine State FLAGS field will be off; this indicates that the saved high registers are not valid.
2. If a nested enclave ends because of an unhandled condition and a 4094-40 ABEND is declared, the high registers may not be valid in the Machine State that contains information about the 4094-40 ABEND.
3. If an ABEND occurs or a program check occurs with the TRAP(ON,NOPSPIE) runtime option in effect, and the SDWA registers at the time of interrupt (in the SDWAGRSV field) are not appropriate or recognizable, and Language Environment instead saves the registers from the SDWASRSV field in the Machine State, the high registers may not be valid in the Machine State.

OFFSETS					
DEC	HEX	TYPE	LEN	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	512	MCH	Lang Env Machine State
0	(0)	CHARACTER	4	MCH_EYE	Eye Catcher
4	(4)	SIGNED	2	MCH_SIZE	Size of area
6	(6)	SIGNED	2	MCH_LEVEL	Level of generation
8	(8)	CHARACTER	64	REG	GPR at interrupt
8	(8)	SIGNED	4	GPR (0:15)	Individual regs
72	(48)	CHARACTER	8	PSW	Basic or extended PSW at time of interrupt
80	(50)	SIGNED	4	INTI	EPIE Fields - ILC & code
80	(50)	SIGNED	2	ILC	Extended PSW ILC
82	(52)	SIGNED	2	IC	Extended PSW interrupt
82	(52)	UNSIGNED	1	IC1	1st byte of Ext PSW Int code
83	(53)	UNSIGNED	1	IC2	2nd byte of Ext PSW Int code
84	(54)	ADDRESS	4	PFT	Page fault location
88	(58)	CHARACTER	32	FLT	Float regs
88	(58)	CHARACTER	8	FLT_0	Floating point reg 0
96	(60)	CHARACTER	8	FLT_2	Floating point reg 2
104	(68)	CHARACTER	8	FLT_4	Floating point reg 4
112	(70)	CHARACTER	8	FLT_6	Floating point reg 6
120	(78)	BITSTRING	44	*	(reserved)

CEEMCH

164	(A4)	ADDRESS	4	INT_SF	Interrupt stack frame
168	(A8)	BITSTRING	11	*	(reserved)
179	(B3)	BITSTRING	1	FLAGS	MCH flags
		.1... ..		HR_VALID	HI regs saved in MCH
		..1.		INT_SF_VALID	"X'20" Interrupt stackframe valid in INT_SF field
		...1		SAVSTACK	"X'10" CEECAA_SAVSTACK field was set to the value in INT_SF field at interrupt time
	 1...		SAVSTACK_ASYNC	"X'08" CEECAA_SAVSTACK_ASYNC field pointed to a field that was set to the value in INT_SF field at interrupt time
	1..		AR_VALID	Access registers saved in MCH
		1... ..11		*	Internal flags
180	(B4)	BITSTRING	4	*	(reserved)
184	(B8)	ADDRESS	4	MCH_EXT	Ptr to language MCH extension
188	(BC)	BITSTRING	4	MCH_BEA	Copy of SDWA_BEA
192	(C0)	ADDRESS	4	SAVSTACK_ASYNC_PTR	Value in CEECAA_SAVSTACK_ASYNC field at time of interrupt (for debugging purposes only)
					(reserved)
196	(C4)	BITSTRING	12	*	(reserved)
208	(D0)	CHARACTER	104	AFP	Additional FP regs
208	(D0)	CHARACTER	8	FLT_1	Floating point reg 1
216	(D8)	CHARACTER	8	FLT_3	Floating point reg 3
224	(E0)	CHARACTER	8	FLT_5	Floating point reg 5
232	(E8)	CHARACTER	8	FLT_7	Floating point reg 7
240	(F0)	CHARACTER	8	FLT_8	Floating point reg 8
248	(F8)	CHARACTER	8	FLT_9	Floating point reg 9
256	(100)	CHARACTER	8	FLT_10	Floating point reg 10
264	(108)	CHARACTER	8	FLT_11	Floating point reg 11
272	(110)	CHARACTER	8	FLT_12	Floating point reg 12
280	(118)	CHARACTER	8	FLT_13	Floating point reg 13
288	(120)	CHARACTER	8	FLT_14	Floating point reg 14
296	(128)	CHARACTER	8	FLT_15	Floating point reg 15
304	(130)	CHARACTER	4	FPC	FP control register
		1... ..		FPC_IMI	IEEE Invalid operation mask
		.1.. ..		FPC_IMZ	IEEE Divide by zero mask
		..1.		FPC_IMO	IEEE Overflow mask
		...1		FPC_IMU	IEEE Underflow mask
	 1...		FPC_IMX	IEEE Inexact mask
	111		FPC_RS0	Byte 0 reserved bits
305	(131)	1... ..		FPC_SFI	IEEE Invalid operation flag
		.1.. ..		FPC_SFZ	IEEE Divide by zero flag
		..1.		FPC_SFO	IEEE Overflow flag
		...1		FPC_SFU	IEEE Underflow flag
	 1...		FPC_SFX	IEEE Inexact flag
	111		FPC_RS1	Byte 1 reserved bits
306	(132)	BITSTRING	1	FPC_DXC	Data Exception Code
307	(133)	1111 11..		FPC_RS3	Byte 3 reserved bits
	11		FPC_RM	Rounding Mode
308	(134)	BITSTRING	1	_AFP_FLAGS	AFP flag byte
		1... ..		AFP_SAVED	FPRs 1,3,5,7,8-15 were saved in MCH
309	(135)	CHARACTER	11	RSV2	reserved
320	(140)	CHARACTER	64	REG_H	GPR-hi at interrupt
320	(140)	SIGNED	4	GPR_H (0:15)	Individual regs
384	(180)	CHARACTER	64	AREG	Access registers
384	(180)	SIGNED	4	AR(0:15)	Individual access registers
448	(1C0)	CHARACTER	64	RSV3	reserved

The code example (below) shows the cross reference summary of the machine state block.

NAME	HEX OFFSET	HEX VALUE	LEVEL
MCH	0		1
_AFP_FLAGS	134		2
AFP_SAVED	134	80	3
APF	D0		2
AR_VALID	B3	04	3
AR(0:15)	180		3
AREG	180		2
FLAGS	B3	00	2
FLT	58		2

FLT_0	58		3
FLT_1	D0		3
FLT_10	100		3
FLT_11	108		3
FLT_12	110		3
FLT_13	118		3
FLT_14	120		3
FLT_15	128		3
FLT_2	60		3
FLT_3	D8		3
FLT_4	68		3
FLT_5	E0		3
FLT_6	70		3
FLT_7	E8		3
FLT_8	F0		3
FLT_9	F8		3
FPC	130		3
FPC_DXC	132		4
FPC_IMI	130	80	4
FPC_IMO	130	20	4
FPC_IMU	130	10	4
FPC_IMX	130	08	4
FPC_IMZ	130	40	4
FPC_RM	133	0X	4
FPC_RS0	130	0X	4
FPC_RS1	131	0X	4
FPC_RS3	133	XX	4
FPC_SFI	131	80	4
FPC_SFO	131	20	4
FPC_SFU	131	10	4
FPC_SFX	131	08	4
FPC_SFZ	131	40	4
GPR(0:15)	8		3
GPR_H(0:15)	140		3
HR_VALID	B3	40	3
IC	52		3
IC1	52		4
IC2	53		4
ILC	50		3
INTI	50		2
INT_SF	A4		2
INT_SF_VALID	B3	20	3
MCH_BEÄ	BC	00000000	2
MCH_EXT	B8		2
MCH_EYE	0		2
MCH_LEVEL	6		2
MCH_SIZE	4		2
PFT	54		4
PSW	48		2
REG	8		2
REG_H	140		2
RSV2	135		2
RSV3	1C0		2
SAVSTACK	B3	10	3
SAVSTACK_ASYNC	B3	8	3
SAVSTACK_ASYNC_PTR	C0		2

Language Environment member list and event handler

A Language Environment member list is created during region, process, enclave, and thread initialization. Each is a table of member-specific data. The length of the member list is determined by the highest assigned member number, which is 18 for Language Environment. Thus, indices range from 0 to 17 inclusive. An entry is reserved for each member that is known to Language Environment. The offset into the list is determined by using the member code defined in the program prolog (PPA2) as an index. The contents of the member list are shown in Figure 28 on page 84.

Note:

1. A member language should write only into its CEEMEMLDEF field in the member list.

Member list

- CICS SPF: The RCB can be in key 8 storage under CICS. Member languages should assume the storage is write protected when running in key 9. Member languages should insure that they are running in key 8 before writing into the member list.

-0018	CEEMEMLEYE CL8'CEEMEML'	
-0010	CEEMEMLVER - Version	CEEMEMLLEN - length
-000C	Reserved	
-0008	Reserved	
-0004	CEEMEMLSIZE - Number of entries in member list	
000000	CEEMEMLDEF - Member 0 defined (8 bytes)	
000008	CEEMEMLEXIT(0) - Address of member 0 event handler for region, process, or enclave. Reserved for thread level	
00000C	Reserved	
/	/	
/	/	
16*n+0	CEEMEMLDEF(n) - Member n defined (8 bytes)	
16*n+8	CEEMEMLEXIT(n) - Address of member n event handler for region, process, or enclave. Reserved for thread level	
16*n+C	Reserved	

Figure 28. Member list format

Table 24 describes the member list fields in more detail.

Table 24. Member list field descriptions

Field	Contents
CEEMEMLEYE	Eyecatcher for the member list.
CEEMEMLVER	Version number for the control block; set to 1.
CEEMEMLLEN	Total length of the control block in bytes.
CEEMEMLSIZE	Number of entries in the control block; set to 18.
CEEMEMLDEF	Eight bytes, member-defined; initially set to zero.
CEEMEMLEXIT	Address of the member language event handler. If the member event handler does not exist, this field is set to point to a dummy event handler, which always returns -4 in R15. This field is set during region, process, or enclave initialization. For thread-level processing, this field is reserved.

When certain events occur, Language Environment calls a member-provided event handler, and passes it a parameter list consisting of an event code and other event-specific information. This allows the member to process its own resources within the context of the event.

Language Environment can allocate any of its control blocks above the line. Any member code that accesses a Language Environment control block must run in AMODE(31) to have addressability to the control blocks.

Language Environment gets a member's event handler address from the member list entry. Each language running in a Language Environment environment that provides a signature CSECT or that appears in the dependent member list of a signature CSECT is required to have an event handler routine. For a description of signature CSECTs, see "Signature CSECT" on page 147. The load name of the event handler routine must be CEEEV*nnn*, where *nnn* is a decimal member number. The values for *nnn* are in Figure 15 on page 19.

During enclave initialization, CEEEV*nnn* is dynamically loaded. Its address is saved in the member list as the event handler entry point. All other entries in the member list are initialized to the address of a Language Environment-provided default event handler. Event handlers are required to set a return code in R15. If an event handler does not process the event being called, the return code should be set to -4. For more information about events, see Chapter 15, "Member language information," on page 479.

Language Environment callable services calling conventions

Language Environment callable services supports the following argument passing styles; language semantics usually determine when data are passed by value and when they are passed by reference:

- Indirect/by value
- Indirect/by reference

Calls that occur within the same HLL, or between the compiled code and its associated HLL library support routine are free to choose the manner in which arguments are passed.

In Language Environment, the following calling conventions are followed.

- All Language Environment languages must support the indirect access mode for passing arguments for external calls.
- The last argument pointer in the argument list body has the high-order bit ON. Thus, the length of the argument list can be determined through the argument list itself.
- When no argument is provided, R1 must be zero.
- All addresses are considered to be 31-bit addresses. Explicit ESA support is not provided.
- All stack frames on the call path must be back-chained, even if they are not explicitly on the Language Environment-managed stack.
- When a stack frame is present on the Language Environment-managed stack, it must be in the format of a DSA containing a valid NAB. Except for those exceptions noted in "Language Environment dynamic storage area – non-XPLINK" on page 37, the first word of the DSA must contain zero.
- Language Environment callable services, at times, impose their own restrictions.

Callable services syntax declarations

Throughout this document, the callable service syntax is shown as a C function prototype; a function declaration for the routine which is called. Data structures are described by C **struct** definitions.

Callable Services Syntax Declarations

By using the C function prototype, the argument list as well as the data type of each argument can be shown accurately and in one place. In addition, the prototype makes clear if a parameter is passed by value or by reference. The caller then matches parameters to the argument descriptions.

The application writer's interface is described from the callee's point of view. Usually, when the call is described from the caller's point of view, the data type of each parameter is not clear unless it is explained later in the document. It is also often not clear which parameters are required for calling by reference rather than by value.

Some basic properties of the callable services are:

- C function declarations have a return type of void since they are procedures. No value is returned by the function.
- All parameters are passed by reference.
- Each argument is a pointer.
- Brackets '[']' surround parameters that are optional.

Optional parameter support

Optional parameters are represented by a zero address in the parameter address list. Not all HLL compilers are capable of generating this form of optional parameter. Thus, the syntax examples are misleading for some HLLs.

Language Environment tolerates the high-order bit on in the parameter address list for an optional parameter. The high-order bit is used to indicate the “end-of-list”.

Data type definitions

To insure a consistent interpretation of the arguments, the data type definitions listed in Table 25 are used in the callable service descriptions. When declaring fixed length strings in C or C++ of size n , specify a length of $n+1$ so the NULL can be placed in the $n+1$ position. Language Environment neither sets nor interrogates the $n+1$ st position. A *stack frame* in the next section is equivalent to a DSA.

Table 25. Data type definitions for callable services

Data type	Definition
CEE_COND	A condition variable, as defined by the type <code>pthread_cond_t</code>
CEE_CONDATTR	A condition variable attributes object, as defined by the type <code>pthread_condattr_t</code>
CEE_ENTRY	Entry point address of a Language Environment-conforming function to be run on a new thread
CEE_LOCKATTR	A mutex attributes object, as defined by the type <code>pthread_mutexattr_t</code> , or a read-write lock attributes object, as defined by the type <code>pthread_rwlockattr_t</code>
CEE_MUTEX	A mutex object, as defined by the type <code>pthread_mutex_t</code>
CEE_PTAT	A thread attributes object, as defined by the type <code>pthread_attr_t</code>
CEE_RWLOCK	A read/write lock, as defined by the type <code>pthread_rwlock_t</code>
CEE_THDID	An identifier representing a pthread-crafted thread, as defined by the type <code>pthread_t</code>
CEE_THDKEY	A key identifier, as defined by the type <code>pthread_key_t</code> , that is used to associate thread-specific data with a given thread

Table 25. Data type definitions for callable services (continued)

Data type	Definition
CEE_TOKEN	A miscellaneous identifier, used in specific instances where more general data types do not apply
CHAR n	A string (character array) of length n
CONST INT	A fullword constant numeric value
ENTRY	Language-dependent entry constant and/or entry variable
FEED_BACK	A mapping of the feedback (condition) code; see Chapter 5, "Condition representation," on page 225.
FLOAT4	A 4-byte single-precision floating-point number
FLOAT8	A 8-byte double-precision floating-point number
INT2	A 2-byte signed integer
INT4	A 4-byte signed integer
LABEL	Language-dependent label variable
LABEL370	370 extensions to CSC label variable; see "LABEL variable" on page 88
POINTER	A fullword address pointer
VSTRING	A Language Environment string of arbitrary length, which is used for polymorphic string parameter declarations. The string may be any one of a fixed-length string, a null-terminated varying string (known as an ASCIIZ string), or a length-prefixed string. Language Environment assumes the following defaults for VSTRING: <ul style="list-style-type: none"> • For input parameters, assume a halfword length-prefixed character string. • For output parameters, assume a fixed-length 80-character string padded right with blanks.

Strong alignment is assumed in all data structures. Each item is aligned on the proper boundary for its type with padding, if necessary.

ENTRY variable

Language Environment defines an entry variable as a doubleword, and is shown in Figure 29.

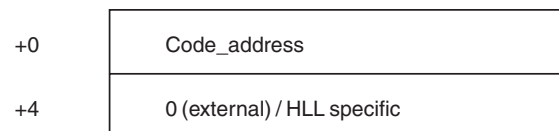


Figure 29. Format of an entry variable

Language Environment's support of entry variables has the following characteristics:

- An entry variable can be either an external routine (nesting level zero), or an internal routine (nesting level greater than zero). The nesting level can be determined statically at compile time.
- Language Environment's use of an entry variable is restricted to an external variable, for example, a nesting level zero.

ENTRY variable

- An entry variable consists of a doubleword. The first word contains the entry point address of the routine, and for external routines the second word contains a zero.
- An HLL can use the second word of the entry variable for internal routines to enforce block scoping rules.
- An entry variable for an internal (nested) routine can only be created and used by the same HLL. It is the responsibility of the called routine to establish its proper block scope as presented by the entry variable.

LABEL variable

Language Environment defines a label variable to have a fixed portion and a language-dependent portion that is pointed to by an extension field. The label variable is used in the service CEEGOTO. For more information, see “CEEGOTO — restart execution at specified label” on page 261.

Callable service example

An example of a callable service declaration is shown below for the fictitious service CEESERV. A list explaining each argument follows the syntax description. The information given for each argument is:

- Whether it is an input only, input/output, or output only argument
- Any values that have special meaning
- A description of invalid parameters when necessary

Usage notes generally follow each description. They contain information about error conditions and any clarifications needed to completely specify the behavior of the service.

Syntax

```
void CEESERV(heap_id, size, address, [fc])
```

```
INT4      *heap_id;  
INT4      *size;  
POINTER   *address;  
FEED_BACK *fc;
```

In C, a variable that has an asterisk preceding it is a pointer. To be more precise, **heap_id* is a pointer to the variable *heap_id* in the statement `x = *heap_id`. It has the value of the pointer *heap_id*.

Invoking a callable service from C/C++

Many of the Compiler-Writer Interfaces (CWIs) do not exist as a STUB and there is no C interface. However, there are two methods for invoking this service from C or C++; in either case, a C prototype for the callable service must be defined. Note that most of the Callable Services use OS linkage.

1. Build a STUB in assembler and link-edit that stub with the application.
2. Construct C mappings of the Common Anchor Area (CAA) and the library vector where the address of the Callable Service is stored, and use C declarations to access the routine. This is identical to how the C Run Time Library accesses Vendor Interface functions.

Figure 30 on page 89 shows an example of calling the CEETBCK callable service from C. In the example, the function `caa()` is a macro that addresses the CAA using the `_gtca` builtin. The typedefs for some parameters (used with Language

Environment[®] interfaces) are declared in <leawi.h>. The CAA contains the offsets where the Language Environment library vectors (CEECAACELV and CEECAALEOV) are located. It also documents the offset from the start of the specific library vector (in this case CEECAALEOV) to the address of the CEETBCK callable service. For information about the CAA, see “Language Environment common anchor area” on page 40. For information about CEETBCK, see *z/OS Language Environment Programming Reference*.

```

#ifdef __gtca
#define __gtca() _gtca()
#ifdef __cplusplus
extern "builtin"
#else
#pragma linkage(_gtca,builtin)
#endif
const void* _gtca(void);
#endif

#ifdef caa
#define caa() ( (struct caa *)__gtca() )
#endif
struct caa
{
char foo[816];
void *ceecaaleov;
};

struct ceeleov
{
char foo[304];
void *CEELEOVTBCK;
};

typedef struct ceeleov CEELEOV;
typedef void ceetbck_cwi_func
( void **, int *, void **, int *, char *, int *,
int *, int *, char *, int *, int *, int *,
void **, int *, int *, char *, int *, void **,
int *, _FEEDBACK *);

#define CEETBCK_CWI
((ceetbck_cwi_func *)(((CEELEOV *)
(caa()->ceecaaleov))->CEELEOVTBCK))

ceektbck_cwi_func *tbkfn_ptr =
(ceektbck_cwi_func *)CEETBCK_CWI;

(*tbkfn_ptr)(arg1, arg2, arg3, ...); /* call CEETBCK */

```

Figure 30. Example: calling CEETBCK from C

If you want to call a Language Environment callable service from DLL-compiled C code (or from C++ code), you need to compile with the `CALLBACKANY` suboption of the DLL compiler option. This is because the library vector is an array of addresses, and DLL-compiled code needs to make calls through function descriptors. Additionally, you must turn off the high-order bit of the address in the library vector. If the high-order bit is on, then the code that makes the `CALLBACKANY` call acts as though there are no passed parameters. To do this, the call looks like:

```
ceektbck_cwi_func *tbkfn_ptr =  
(ceektbck_cwi_func *)((long)CEETBCK_CWI & 0x7FFFFFFF);  
  
(*tbkfn_ptr)(arg1, arg2, arg3, ...); /* call CEETBCK */
```

Chapter 2. CALL linkage conventions

This chapter describes the program call linkage conventions supported by Language Environment:

- The standard Language Environment linkage, used by all Language Environment-conforming languages
- FASTLINK linkage, used by default with z/OS C++ and High Performance Compiled Java (HPCJ)
- Extra Performance Linkage (XPLINK) produced by the z/OS XL C and C++ compilers to provide optimal performance for a certain class of applications

Terminology

The terminology around the call or function invocation is not exactly the same in all HLLs. Figure 31 summarizes the terminology in this section.

TERMINOLOGY REFRESHER

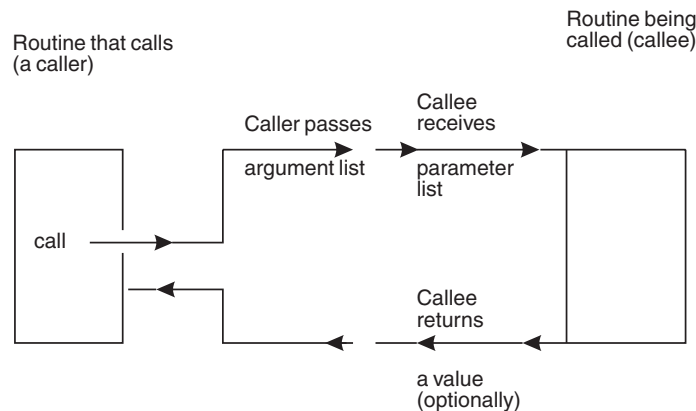


Figure 31. CALL terminology refresher

The formats of a Language Environment “argument list” and “parameter list” are identical. Rather than refer to both formats in this section, the term “argument list” only is used. However, everything that applies to an argument list format also applies to a parameter list format.

There are two access modes for arguments:

Direct The value of the argument is placed directly in the argument list body.

Indirect

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

Programming languages have two basic argument passing semantics:

By value

The value of the object is passed. No change made by the callee to the argument value is reflected in the calling routine.

CALL linkage conventions

By reference

Changes made by the callee to the argument value are reflected in the calling routine.

Standard CALL linkage conventions

The prime purpose of a call or a function invocation is to transfer control to a target routine and optionally pass/receive data to/from the called routine. The transfer of control and communication must be as efficient as possible. Language Environment assumes that:

1. Caller's arguments match the callee's parameters.
2. The only supported way to pass arguments on the ILC call is **indirectly**, either by **reference** or by **value**.
3. Pointers longer than 31 bits are **NOT** supported.
4. Pointers are assumed to be 31-bit capable.

This section describes the standard Language Environment protocols for passing arguments to external routines. These protocols do not apply to internal routines or to compiled code calling its own library routine. Each HLL is free to decide the method for transferring control as well as passing arguments between internal routines.

Register usage

The following list shows the register usage and linkage.

- GPR1 => a list of argument addresses, terminated with an address containing a 1 in its high order bit
- GPR2-12 => Preserved
- GPR13 => an 18-word save area
- GPR14 => the return point in the caller's routine
- GPR15 => the entry point in the called routine
- FPRs => not preserved, value undefined

Stack format

Figure 32 shows the standard Language Environment stack storage model.

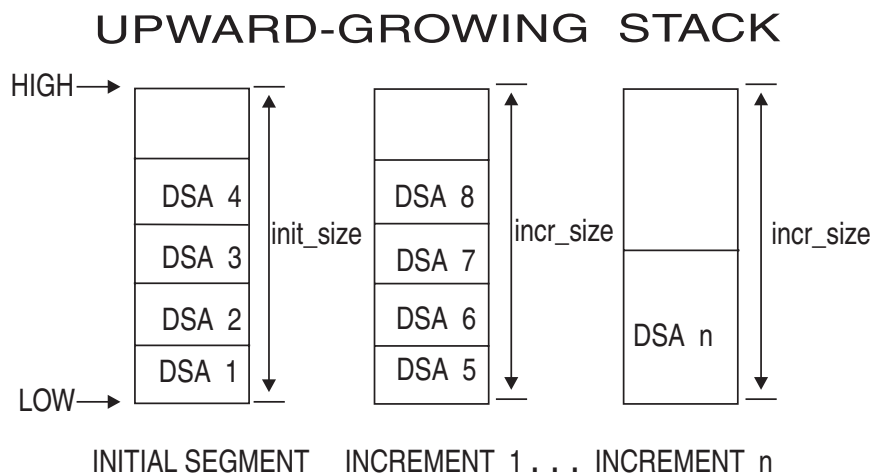


Figure 32. Language Environment Non-XPLINK stack storage model

Allocate/extend/return storage in user stack

DSA allocation code sequences must be used whenever a DSA is required and/or needs to be extended. The generated code and the Language Environment and member libraries use code sequences such as the following examples. Rules which must be followed are:

- A DSA allocated in the Language Environment user stack must be in the Language Environment format. Storage management expects the CEEDSANAB field to be located at offset X'4C' from the start of the DSA. Figure 26 on page 38 shows the format of a Language Environment DSA.
- Stack storage must be requested in doubleword increments, and is obtained from the stack in doubleword increments. This ensures that a DSA begins on a doubleword.

Figure 33 on page 94 shows an example of the code sequence to allocate a DSA in the user stack when the calling routine is known to pass the Language Environment Anchor Area address in R12. R13 points to a valid DSA containing the current next available byte address, CEEDSANAB. Note that R13 is set to point to the new DSA only after the new DSA has been completely constructed.

Figure 34 on page 94 shows an example of a code sequence that gets additional storage appended to the DSA in the user stack. Note that the DSA extension is not guaranteed to be contiguous with the DSA.

Figure 35 on page 95 shows an example of a code sequence freeing a DSA extension. The code sequence depends upon the proper initialization of CEEDSAPNAB (prolog NAB) before the DSA is extended. Member languages can save the *prolog NAB* in CEEDSAPNAB or elsewhere. The NAB value is the DSA address as it has been acquired by prolog code and before it is extended. There is no Language Environment requirement to use the CEEDSAPNAB field for this purpose when allocating user stack extensions. However, there *is* a Language Environment requirement to use CEEDSAPNAB when allocating from the Library stack.

Note: If the STACK(,,,FREE) runtime option is in effect, empty user stack segments are returned to the operating system at the next stack overflow request, or at termination.

The DSA stack storage in the current user stack segment is automatically freed when R13 is updated by the **L 13,4,(13)** instruction at procedure or block termination. Freeing stack storage occurs because the current NAB is always accessed from the DSA pointed to by R13 when a routine is entered.

The registers do not need to be those shown in the examples as long as the interface to the stack overflow routine, whose address is contained in CEECAAGETS, adheres to the following conditions:

- The stack overflow routine is called with a BALR instruction. R15 contains the entry point address of the stack overflow routine and R14 the address of the next sequential instruction in the caller routine.
- R0 contains the newly generated NAB address from the BALR instruction. That is, it would have been the NAB address if the segment were long enough. This value and the information in the DSA allows the stack overflow routine to determine the minimum amount of storage to obtain for the next stack segment.
- R13 contains the address of the last DSA in the stack and this DSA contains a valid NAB value.

Standard CALL linkage

Examples of Managing the User Stack: The examples in this section illustrate some user stack management techniques. Figure 33 shows how to manage a DSA allocation.

```
ENTRYPT B    **+20                SKIP OVER CONSTANT AREA
          DC   AL4(X'00C3C5C5')    EYECATCHER '.CEE'
          DC   AL4(length)         DSA LENGTH
          DC   AL4(CEEPPA1-ENTRYPT) OFFSET TO PPA1
          B    1(,15)              WRONG ENTRY POINT, CAUSE EXCEPTION
CL..0     STM  14,12,12(13)        SAVE CALLER'S REGISTERS
          L    Ra,CEEDSANAB-CEEDSA(,13) LOAD NEW DSA ADDRESS
          L    0,length            LOAD DSA LENGTH
          ALR  0,Ra                GENERATE NEW NAB ADDRESS
          CL   0,CEECAAEOS-CEECA(,12) EXCEED CURRENT STORAGE SEGMENT?
          BNH  CL..1               NO - WE GOT IT
          L    15,CEECAAGETS-CEECA(,R12) ADDRESS OF STACK SEGMENT MGR
*-- Input to stack overflow routine
*-- 1) R0  calculated required next available byte
*-- 2) R12 address of CEECAA
*-- 3) R13 caller's save area address
*-- 4) R14 return address
*-- 5) R15 stack overflow routine entry point address
          BALR 14,15              GET ANOTHER STACK SEGMENT
*-- Upon return from the stack segment manager:
*-- 1) R15 has the new DSA address
*-- 2) R0  has the new NAB address
          LR   Ra,15              PUT DSA ADDRESS INTO WORK REGISTER
CL..1     ST   13,4(,Ra)           BACK CHAIN NEW DSA TO CALLER
          ST   0,CEEDSANAB-CEEDSA(,Ra) STORE NEW NAB ADDRESS
*-- The following instruction is required to set the
          Language Environment architecture.
*-- first word of the DSA to zero (some exceptions noted).
          XC   0(2,Ra),0(Ra)       ZERO FIRST HALF WORD
*-- The following instruction is optional. It is used to store the
*-- prolog NAB address for later use. For example, to free a DSA
*-- extension, we just copy CEEDSAPNAB back to CEEDSANAB.
          ST   0,CEEDSAPNAB-CEEDSA(,Ra) STORE END OF PROLOG NAB ADDRESS
          LR   13,Ra              SET DSA POINTER REGISTER
```

Figure 33. DSA allocation, user stack

Figure 34 shows how to manage a DSA extension.

```
          L    Ra,CEEDSANAB-CEEDSA(,13) LOAD DSA EXTENSION POINTER
          L    0,length            LOAD EXTENSION LENGTH
          ALR  0,Ra                GET STACK NAB ADDRESS
          CL   0,CEECAAEOS-CEECA(,12) EXCEED CURRENT STORAGE SEGMENT?
          BNH  CL..1               NO - WE GOT IT
          L    15,CEECAAGETSX-CEECA(,R12) ADDR OF STACK EXTENSION RTN
*-- Input to DSA extension overflow routine
*-- 1) R0  calculated required next available byte
*-- 2) R12 address of CEECAA
*-- 3) R13 caller's Language Environment-managed DSA
*-- 4) R14 return address
*-- 5) R15 DSA extension overflow routine entry point address
          BALR 14,15              GET ANOTHER STACK SEGMENT
*-- Upon return from the stack segment manager:
*-- 1) R15 has the new DSA extension address
*-- 2) R0  has the new NAB address
          LR   Ra,15              SET DSA EXTENSION POINTER REGISTER
CL..1     ST   0,CEEDSANAB-CEEDSA(,13) STORE NEW NAB ADDRESS
```

Figure 34. DSA extension, user stack

In Figure 35, the NAB value was previously saved in field CEEDSAPNAB. Languages that save the prolog NAB at a different location should replace CEEDSAPNAB-CEEDSA(13) with the appropriate storage location. Do not use this code sequence unless CEEDSAPNAB was initialized as shown at the bottom of Figure 33 on page 94.

```
MVC  CEEDSANAB-CEEDSA(4,13),CEEDSAPNAB-CEEDSA(13) FREE EXTENSION USING
                                           SAVED NAB VALUE
```

Figure 35. Free a DSA extension using saved NAB value

Obtain a DSA in user stack with R13 pointing to save area

Figure 36 on page 96 shows an example of the code sequence to allocate a DSA in the user stack when the calling routine is known to be passing the Language Environment Anchor Area address in R12 and R13 points to a O/S save area that might or might not be a DSA. The DSA stack storage in the current user stack segment is automatically freed when R13 is updated by the **L 13,4,(13)** instruction at procedure or block termination.

When a DSA cannot be contained within the current stack, a stack overflow routine that does not depend upon R13 pointing to a DSA is called. The address of this overflow routine is held in the CAA at CEECAAGETS1. Typically, this overflow routine is called after a call to CEEVGTUN, described in “CEEVGTUN — next available byte locator service” on page 98. The interface is as follows.

- The stack overflow routine is called as shown below using a BALR instruction. R15 is the entry point address of overflow routine and R14 the return address.
- R1 contains the current NAB.
- R0 contains the result of the BALR instruction. That is, it is what would have been NAB value if the segment were long enough. This value and the information in R1 allows the stack overflow routine to determine the minimum amount of storage to obtain for the next stack segment. The requested amount of storage for a DSA must be in doubleword increments.
- R13 contains the address of a standard OS save area, which can be a DSA. Note that the NAB value is **not** obtained from the save area, and the contents of the save area are not changed by either the CEEVGTUN routine or the overflow routine whose address is in CEECAAGETS1.

Standard CALL linkage

```
ENTRYPT B    **+20                Skip over constant area
DC      AL4(X'00C3C5C5')          Eyecatcher 'CEE'
DC      AL4(length)              DSA Length rounded to a dword
DC      AL4(CEEPPA1-ENTRYPT)     Offset to PPA1
B       1(,15)                   Disable this entry point
CL..0   STM   14,12,12(13)        Save caller's regs
        L     15,CEECAACELV-CEECAA(,12)  Get libvec
        L     15,CEECELVVGTUN-CEECELV(,15)  Get A(Get User NAB)
BALR    14,15                    Find the user NAB
LR      1,15                     Save NAB into R1
L       15,16(,13)              Reset 15 to ENTRYPT
L       0,length                Get new DSA len rounded to a dword
ALR     0,1                     Calc a new NAB
CL      0,CEECAAEOS-CEECAA(,12) Exceed current stack segment?
BNH     CL..1                    No
L       15,CEECAAGETS1-CEECAA(,12)  A(overflow routine)
*-- Input to stack overflow routine
*-- 1) R0  calculated required next available byte
*-- 2) R1  current NAB
*-- 2) R12 address of CEECAA
*-- 3) R13 caller's save area address
*-- 4) R14 return address
*-- 5) R15 stack overflow routine entry point address
*--
        BALR  14,15                GET ANOTHER STACK SEGMENT
*-- Upon return from the stack segment manager:
*-- 1) R15 has the new DSA address
*-- 2) R0  has the new NAB address
*--
        LR    R1,15                Put DSA addr into work reg
CL..1   ST    13,4(,R1)            Back chain the DSA
        ST    0,CEEDSANAB-CEEDSA(,Ra)  Save the new NAB address
*-- The following instruction is required to set the
        Language Environment architecture.
*-- first word of the DSA to zero (exceptions noted).
        XC    0(2,Ra),0(Ra)        Zero the first half word
        LR    13,R1                Make the new DSA official
```

Figure 36. DSA allocation in user stack when R13 does not address a Language Environment DSA

Allocate/return storage in library stack

DSA allocation code sequences can be used whenever a DSA is required from the Language Environment library stack. It is always below the 16M line. Obtaining storage from the library stack is illustrated in the following examples. Several coding rules must be followed:

- A DSA allocated in the Language Environment library stack must be in the Language Environment format. Figure 26 on page 38 shows the format of a Language Environment DSA. For example, storage management expects the CEEDSANAB field to be located at offset X'4C' from the start of the DSA.
- Stack storage must be requested in doubleword increments, and is obtained from the stack in doubleword increments. This ensures that a DSA begins on a doubleword.
- User stack NAB must be carried forward in the CEEDSANAB field.
- Library stack NAB (CEECAALNAB) is saved in CEEDSAPNAB before being updated by the current routine.
- Due to this special use of CEEDSAPNAB, library stack extensions cannot be extended like user stack frames.

Figure 37 on page 97 shows a coding example that allocates a DSA in the library stack. The maintenance of the user stack CEEDSANAB value is required to allow a

routine using the library stack to call a routine expecting to use the user stack. This example passes the caller's CEEDSANAB address through unchanged. The library stack NAB address is maintained in the CAA field CEECAALNAB. The library beginning of stack and end of stack addresses are also maintained in the CAA fields, CEECAALBOS and CEECAALEOS, respectively. Each routine using a library stack must save the CEECAALNAB address in the CEEDSAPNAB field at the time of entry. Special processing by the **go to out of block** function interrupts the normal flow of control to restore the CEECAALNAB value from the CEEDSAPNAB field in all DSAs in the library stack.

Figure 38 on page 98 shows a coding example to return from a routine which has allocated a DSA in the library stack.

Note: Empty library-stack segments are returned to the operating system at the next invocation of CEECAAGETLS, or at termination.

Examples to Manage Library Stack: This section contains examples of how to manage the library stack. Figure 37 shows how to manage a DSA allocation.

```

ENTRYPT B    **20                SKIP OVER CONSTANT AREA
        DC    AL4(X'00C3C5C5')    EYECATCHER '.CEE'
        DC    AL4(length)         DSA LENGTH
        DC    AL4(CEEPPA1-ENTRYPT) OFFSET TO PPA1
        B     1(,15)             WRONG ENTRY POINT, CAUSE EXCEPTION
CL..0   STM   14,12,12(13)        SAVE CALLER'S REGISTERS
        L     Ra,CEECAALNAB-CEECAA(,12) LOCATE LIBRARY STACK NAB
        L     0,length           LOAD DSA LENGTH
        ALR   0,Ra               GENERATE NEW NAB ADDRESS
        CL   0,CEECAALEOS-CEECAA(,12) EXCEED CURRENT STORAGE SEGMENT?
        BNH  CL..1              NO - WE GOT IT
        L     15,CEECAAGETLS-CEECAA(,R12) A(LIBRARY STACK SEG MGR)
*-- Input to library stack overflow
*-- 1) R0  calculated required next available byte
*-- 2) R12 address of CEECAA
*-- 3) R13 caller's save area address
*-- 4) R14 return address
*-- 5) R15 library stack routine entry point address
        BALR  14,15             GET ANOTHER STACK SEGMENT
*-- Upon return from the stack segment manager:
*-- 1) R0  has the new NAB address
*-- 2) R15 has the new DSA address
*--
        LR   Ra,15              PUT DSA ADDRESS INTO WORK REGISTER
CL..1   ST   13,4(,Ra)          BACK CHAIN NEW DSA TO CALLER
        MVC  CEEDSANAB-CEEDSA(4,Ra),CEEDSANAB-CEEDSA(13) SAVE USER
        NAB ADDR
        MVC  CEEDSAPNAB-CEEDSA(4,Ra),CEECAALNAB-CEECAA(12) SAVE LIB
        NAB ADDR
        ST   0,CEECAALNAB-CEECAA(,12) STORE NEW LIBRARY NAB ADDRESS
        XC  0(2,Ra),0(Ra)       ZERO FIRST HALFWORD
        LR   13,Ra             SET DSA POINTER REGISTER

```

Figure 37. DSA allocation, library stack

Figure 38 on page 98 shows how to manage a DSA return.

```

MVC  CEECAALNAB-CEECAA(4,12),CEEDSAPNAB-CEEDSA(13) RESET LIB
      NAB ADDR
L     13,4(,13)          LOAD CALLER'S DSA ADDRESS
LM    14,12,12(13)      LOAD CALLER'S REGISTERS
BR    14                 RETURN TO CALLER

```

Figure 38. DSA return, library stack

CEEVGTUN — next available byte locator service

The Language Environment storage manager provides a service that returns the next available byte address for the user stack to the caller. CEEVGTUN is a S/370-specific CWI (compiler writer interface) that performs this service. CEEVGTUN isolates the user from Language Environment internals. This prevents the problem of having generated code use any of the Language Environment storage management internal control blocks and structures. Only a low-level interface is provided with the following conventions.

Register type	Register number	Register description
Input Registers	R0–R11	Not used.
	R12	Address of CAA.
	R13	Save area address of the CEEVGTUN caller's caller. Note that this save area is not modified by CEEVGTUN.
	R14	Return address to the caller.
	R15	Address of CEEVGTUN.
Output Registers	R0–R14	Unchanged.
	R15	Next available byte in the user stack.

CEEVGTUN

Call this CWI interface as follows:

```

L     R15,CEECAACELV-CEECAA(,R12)
L     R15,148(,R15)
BALR  R14,R15

```

If CEEVGTUN encounters any errors, it abends with code 4088. The reason code associated with abend 4088 indicates the cause of the failure:

- 99** An exception occurred while trying to locate the NAB, or a zero back chain pointer was found before finding the Language Environment dummy DSA.

Use the code sequence shown in Figure 39 on page 99 only in a library routine, not in compiler-generated code.

```

CL..0  STM  14,12,12(13)      SAVE CALLER'S REGISTERS
        L   15,CEECAACELV    GET ADDRESS OF LIBVEC
        L   15,CEECELVGTUN-CEECELV(15)  LOAD ADDR OF GET USER NAB
                                           SERVICE
        BALR 14,15           CALL THE SERVICE
        LR  Ra,15           LOAD NEW DSA ADDRESS
        L   15,16(,13)      RESTORE ADDRESSABILITY
        L   0,length        LOAD DSA LENGTH
        ALR 0,Ra           GENERATE NEW NAB ADDRESS
        ...

```

Figure 39. Get next available byte in user stack

CEEVSSEG — return the stack segment bounds

The Stack Segment Bounds CWI returns the beginning point and the ending point of a Language Environment stack segment given an address within the bounds of that segment.

Syntax

```
void (*CEECELVVSSEG) (ss_ptr, ss_type, ss_start, ss_end, ss_chain, [fc])
```

```

POINTER    *ss_ptr;
INT4       *ss_type;
POINTER    *ss_start;
INT4       *ss_end;
POINTER    *ss_chain;
FEED_BACK  *fc;

```

CEECELVVSSEG

A field in Language Environment LIBVEC that points to the CEEVSSEG CWI. Call this CWI interface as follows:

```

L   R12,A(CAA)           Get the address of CAA in R12
L   R15,CEECAACELV-CEECAA(,R12)
L   R15,3372(,R15)
BALR R14,R15

```

ss_ptr (input)

An address within the bounds of a user or library stack segment.

ss_type (output)

A fullword binary integer representing the type of Language Environment stack segment containing *ss_ptr*. If *fc* is CEE3MO, the *ss_type* value is undefined. The possible values for *ss_ptr* are:

- 1 User stack
- 2 Library stack
- 3 Downward-growing stack

ss_start (output)

A pointer to the beginning of the stack segment, containing *ss_ptr*. If *fc* is CEE3MO, the *ss_start* value is undefined.

ss_end (output)

A pointer to the end of usable stack segment, containing *ss_ptr*. If *fc* is CEE3MO, the *ss_end* value is undefined.

ss_chain (output)

A pointer to the next stack segment. If *ss_ptr* points to the last stack segment, *ss_chain* is set to 0. If *fc* is CEE3MO, the *ss_chain* value is undefined.

fc (output/optional)

The resulting feedback code. The following conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3MO	Severity	3
	Msg_No	3800
	Message	The address passed to the stack segment service is not within any Language Environment stack segment.
	Explanation	The address passed to the stack segment bounds service is not within any currently allocated Language Environment stack segment.
	Programmer Response	This is an internal problem. Contact your service representative.
	System Action	The bounds, segment type, and chain are undefined.

Usage Notes:

1. This service is intended for members to use to access stack extensions when the NAB field in a DSA indicates a stack frame has been extended beyond the current stack segment boundary.
2. The *ss_ptr* value is usually a DSA or NAB address. CEEVSSEG searches both the library and user stack for the segment containing this address.
3. The sequence of stack segments that follow the segment that contains *ss_ptr* can be located by repeatedly passing the value of *ss_chain* returned by the previous call to CEEVSSEG into another call to CEEVSSEG and obtaining new values of *ss_type*, *ss_start*, *ss_end*, and *ss_chain*.

Standard save area

The save area is 128 bytes (X'80') in length. The first 72 bytes of the save area matches the format of a traditional OS save area and is provided to called routines for the purpose of saving general registers. Certain fields are critical to this description and are included here as well as documenting existing HPCJ usage of some reserved fields. This existing usage occurs in OS/390 V1R1 and older code.

Field location	Field description
X'00'	STKLANG - Language word
X'04'	CEEDSABACK - Back chain pointer to previous save area
X'08'	CEEDSAFWD - Forward chain pointer to next save area
X'0C'	CEEDSASAVE - GPR save area (registers 14 through 12)
X'4C'	CEEDSANAB - Next Available Byte
X'78'	Used by C to save the parm list address (r1)
X'7C'	Reserved

Argument list format

An argument list is located by an argument list pointer. In S/370, the argument list pointer is held in general purpose R1. An argument list has an architected way to access individual arguments and their data descriptors. It is sometimes known as a **Type-I parameter list**. The format of this argument list is seen in Figure 40.

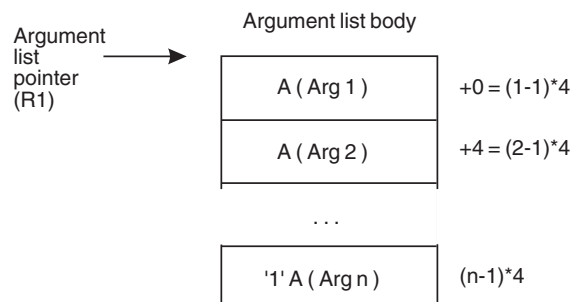


Figure 40. S/370 Argument/parameter list format

Argument passing - C linkage

When a C linkage routine is called from another C linkage routine, GPR1 contains the address of the caller's argument list. The argument list may contain addresses of arguments passed indirectly (by reference) or values of arguments passed directly by value. The end of the parameter list is not marked by the high order bit being turned on. Since the end of the argument list is not identified the programmer is responsible to ensure that the callee only accesses as many parameters as the caller had arguments.

In C linkage, the caller of a function whose return value is not passed in registers must provide storage where this value may be placed. The address of such storage is passed as a hidden first argument at the beginning of the argument list.

C linkage use a logical argument list. At a +0 Entry Point the argument list is located by means of GPR1 and may be placed anywhere in storage at the discretion of the calling routine. C linkage supports both direct and indirect arguments for calls between cooperating routines and thus the argument list may contain a mixture of values and addresses.

When using C conventions, floating point parameters and structure return values are placed in storage whose address is passed as the first parameter, other types are returned in GPR15. The +0 Entry Point prolog must relocate the return value into register 15 or in some cases into storage provided by the caller. In C the arguments are all passed in storage, thus the logical and physical argument list are identical.

All addresses in the argument list are of a consistent width of 4 bytes. Each parameter takes up a multiple of 4 bytes.

Pointers to indirect arguments in the list are aligned on fullword boundaries. Direct by value scalar arguments are right-aligned within one or more 4 byte slots in the argument list. With this alignment, they may be simply loaded into an appropriate register. In particular:

- fullword integers and addresses are aligned on a fullword boundary.
- halfword integers are placed in the 2 low order bytes of a fullword-aligned field.

Argument Passing

- single byte integers are placed in the low order byte of a fullword aligned field.

Note: The high order bytes are sign extended in the case of a signed argument or are zero for an unsigned argument.

- a Boolean scalar is placed in the low order bit of a fullword-aligned field, whose high order 31 bits are zero.
- real or complex floating point numbers are fullword-aligned and may occupy one or more 4-byte slots in the argument list.
- structures begin in the high order byte of a fullword and occupy an integral number of fullwords. Any padding bytes on the right end of the last full word are unused and their value is undefined.

FASTLINK CALL linkage conventions

FASTLINK is essentially an extension of the OS linkage convention, which has been in use since the inception of System/360. FASTLINK linkage is used today as the default linkage for the C++ and High Performance Compiled Java (HPCJ) compilers.

Register usage

The following list shows register usage and linkage.

GPR0	=> writable Static Area (WSA)
GPR1-3	=> arguments (depending upon type)
GPR4-12	=> preserved
GPR12	=> CAA, the key Language Environment control block
GPR13	=> the caller's stack frame in the Language Environment stack. Each such stack frame begins with a 36-word save area.
GPR14	=> the return point in the caller's routine
GPR15	=> the entry point in the called routine
FPRs	=> arguments (depending upon type)

Some of the caller's arguments are placed in registers and the remainder in a portion of what will be the callee's stack frame. With FASTLINK the caller enters the called routine at an offset of 16 bytes from the called routine's entry point.

Note:

1. The module is assumed to be readonly and never changed during execution, in particular the eyecatcher, frame size or offset to PPA1 do not change during execution.
2. The frame size field in the prolog above is owned by the compiled module and the value it contains is whatever is required by the prolog of the routine. It does not necessarily contain the precise value of the dsa size. For example, in C++ vararg routines, it contains the size of the fixed portions of the stack frame. Since the frame size may change from one call to the next and the size of the argument area is passed from the caller to the callee, a runtime calculation of the actual dsa size is required.
3. The eyecatcher is changed slightly to signify that this procedure uses the FASTLINK dsa layout and is thus prepared for future support of extended addresses.

FASTLINK is designed to operate in conjunction with the Language Environment-provided execution stack. The current stack pointer is maintained in GPR13. The prolog of a Language Environment-enabled routine may allocate space (referred to as a "frame", "stack frame" or "dsa") in this stack for its own purposes

and to support subsequent calls to other routines. The stack frame is an architected area that contains the following subsections:

- A save area to be used by any routines called by the executing routine for saving registers and other values as architected. This save area, which is the first sub-section in a Language Environment stack frame, is pointed to by GPR13, and begins with a 36-word OS save area.
- A link area reserved for Language Environment defined use. This area contains a number of architected fields used by languages and glue code.
- The argument area where the caller of this routine places arguments when more arguments exist than can be passed in registers.
- The near auto area used to guarantee a register spill area within 4K of the stack pointer.
- The work area where scratch and or automatic variables are located.

Stack frame mapping

Figure 41 shows the storage map of a typical FASTLINK stack frame. The stack frame is double-word-aligned, in terms of where the stack frame pointer (R13) points.

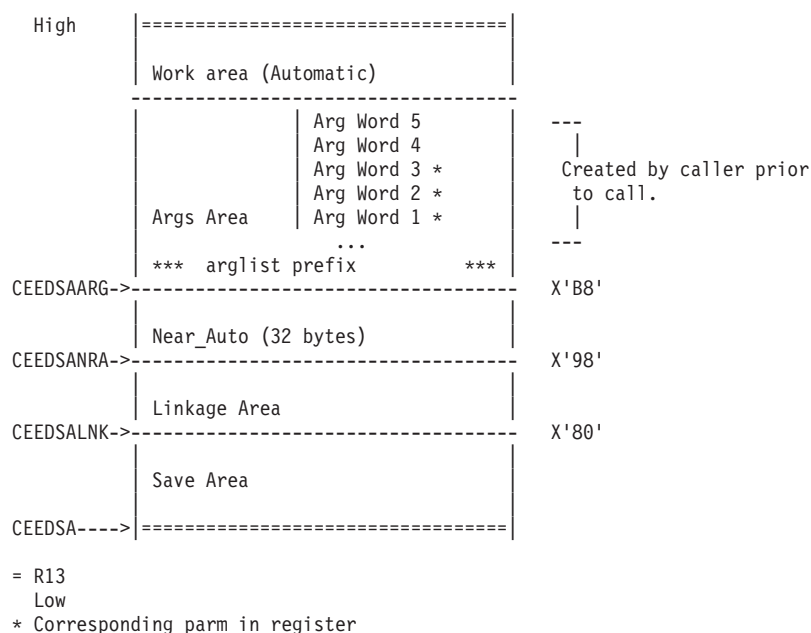


Figure 41. Typical FASTLINK stack frame storage map

The **Save Area** in FASTLINK is 128 bytes (X'80') in length. In detail, the save area appears as shown in Table 26.

Table 26. Format of save area

Field location	Field description
X'00'	STKLANG - Language word Note 1
X'04'	CEEDSABACK - Back chain pointer to previous save area (see note 2 on page 104)
X'08'	CEEDSAFWD - Forward chain pointer to next save area (see note 3 on page 104)
X'0C'	CEER14DSASAVE - GPR Link save area (register 14) (see note 4 on page 104)
X'10'	CEER15DSASAVE - GPR Link save area (register 15) A (see note 4 on page 104)

FASTLINK CALL linkage

Table 26. Format of save area (continued)

Field location	Field description
X'14'	CEEDSAGSAVE - GPR save area register 0 through 12
X'48'	(future AR save area register 1 through 12) (see note 5)

Notes:

1. The PPA1 indicates validity of fields in the language word.
2. The back chain pointer to the previous save area must be set by any routine that allocates a stack frame.
3. The forward chain pointer is not required to be valid and is reserved.
4. The return address must be saved at offset X'0C' and the entry point address at X'10'.
5. Shown only for illustration purposes. The Language Environment routines use some locations between X'0C' and X'7C'. In particular, Language Environment continues to use Save Area words (X'4C' and X'6C') for the same purposes as in R1. This does not cause a problem in R1 FASTLINK because
 - a. Language Environment does not support greater than 32 bit addressing in FASTLINK-compiled code or in library code, thus there is no requirement to save or restore ARs from this area.
 - b. FASTLINK generated code does not read or write from the this area except
 - 1) in the code prolog, and then only to retrieve the NAB from the caller's stack frame.
 - 2) possibly to set the NAB at X'4C' in the current stack frame, for example just prior to a call to a Language Environment facility.
6. The only part of the caller's DSA that a callee may update is the portion of the caller's Save Area into which registers are saved (X'0C' through X'47'). In particular, the STKLANG, CEEDSABACK, and CEEDSAFWD fields may not be changed by a callee. Words X'48' through X'7C' of the save area in the caller's DSA are never changed by any FASTLINK callee.
7. FASTLINK programs containing calls must be compiled assuming that the current Save Area addressed by R13 offsets X'0C' through X'7C' are overwritten across calls.
8. For stack unwinding and exception processing purposes, the PPA1 specifies which GPR registers must be restored from their slots in the save area.

The **linkage area**, described in Table 27, is used to store the Next Available Byte (NAB) in CEEODSANAB. CEEODSARET contains information used in support of the +0 Entry Point entry point. It contains a logical flag rather than the real return address in some instances. Neither CEEODSARET nor the following words are initialized if they are not in use. Use of the Link Area is only as described in this document; it must not be used for any other purposes than shown.

Table 27. Format of linkage area

Field location	Field description
X'00'	CEEODSANAB - Next Available Byte (see note 1 on page 105)
X'04'	CEEODSARET - Real Return or epilog flag (see note 2 on page 105)
X'08'	Amode switching (reserved)
X'0C'	reserved contents unspecified

Table 27. Format of linkage area (continued)

Field location	Field description
X'10'	reserved contents unspecified
X'14'	reserved contents unspecified
Note: <ol style="list-style-type: none"> 1. The NAB field points to the first free byte on the stack (double word aligned) following this stack frame. Programs can always assume that the NAB field is double word aligned when they receive control. 2. This word is available for use by the members to control execution of the +0 Entry Point as contrasted to the +16 Entry Point code epilg. 	

With a large argument area, it is possible that none of the Work area is addressable within a 4K displacement of R13. The **Near_Auto** area is provided to guarantee the compiler some work space in the first 4K block of the DSA. Logically Near_Auto is part of the Work area.

The **Argument Area** is at fixed DSA offset X'B8'; it contains the argument list passed from caller to callee on a procedure call. Figure 42 shows the format of the argument list. C and FASTLINK use argument lists of almost identical format. The FASTLINK argument list is always prefixed with space for a pointer to the descriptors.

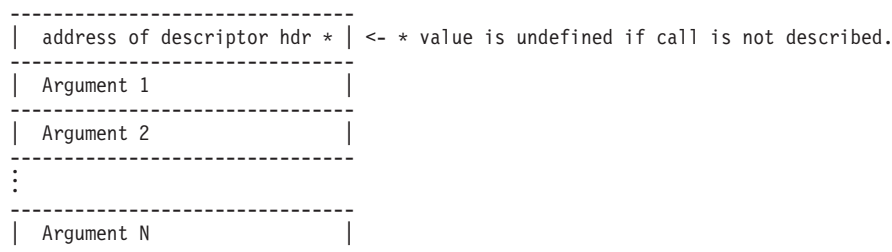


Figure 42. Argument list passed on a procedure call

Upon return the callee's argument area may have been modified regardless of format. FASTLINK programs must assume that callees may update their parameters and rebuild the argument area prior to each call.

The **Work area** is the space (work_size) owned by the executing procedure (which allocated the stack frame) and may be used at its discretion for local variables and temporaries. The executing routine has total control of the work area.

The **Total stack frame size** is best described as the difference between the NAB and stack pointer, R13, assuming that both are in the same stack segment. The frame_size is rounded up to a double word boundary. Most frequently it will be:

$$\text{frame_size} = \text{save_size} + \text{arg_area_size} + \text{work_size} + \text{link_area} + \text{near_auto}$$

The **Stack Segment Pad** for FASTLINK is a 256 byte pad that is added at the high end of the stack segment and is used to allow calling programs to build their argument lists in the callee's stack frame with minimal code. Thus if the caller's argument list is smaller than 256 - save_size - near_auto - link_size (72 bytes), the parameter list can be constructed without checking for stack segment overflow or including logic to support stack frame segment overflow. FASTLINK uses a new CAA field (CEECAAESS) for its stack segment limit. The stack segment is actually

FASTLINK CALL linkage

256 bytes larger than indicated in CEECAAESS. CEECAAESS = MAX(CEECAAEOS - 256,0). The STACK runtime option is reflected in the stack size value in CEECAAEO, thus the FASTLINK stack appears to be 256 bytes smaller.

Few procedures create argument lists larger than this size and thus code to handle large argument lists will not be common. Callers which create argument lists greater than allowed above will have to ensure that the current stack segment has sufficient space (check against CEECAAESS) or, if not, obtain a free segment from CEL. The beginning address of this additional segment must be placed in the NAB field of the current frame and arguments must be stored at the appropriate offset in the stack segment just obtained.

When stack segment overflow is detected in the prolog, the run time is called to obtain a new segment. As well as allocating a new segment, this code also copies the argument area from the old stack segment to the new stack segment. No language allows addresses of parameters to be passed as a parameter and thus such a copy preserves address values in the argument list. The stack segment overflow logic is the same as for non-FASTLINK except that CEECAAEO must also be set to mimic the setting of CEECAAESS. While the overflow stack segment is being used CEECAAESS, like CEECAAEO, has a value of zero.

The **Stack Segment**, as shown in Figure 43 on page 107, contains multiple stack frames. The stack pointer register (R13) grows from numerically lower storage addresses to numerically higher ones.

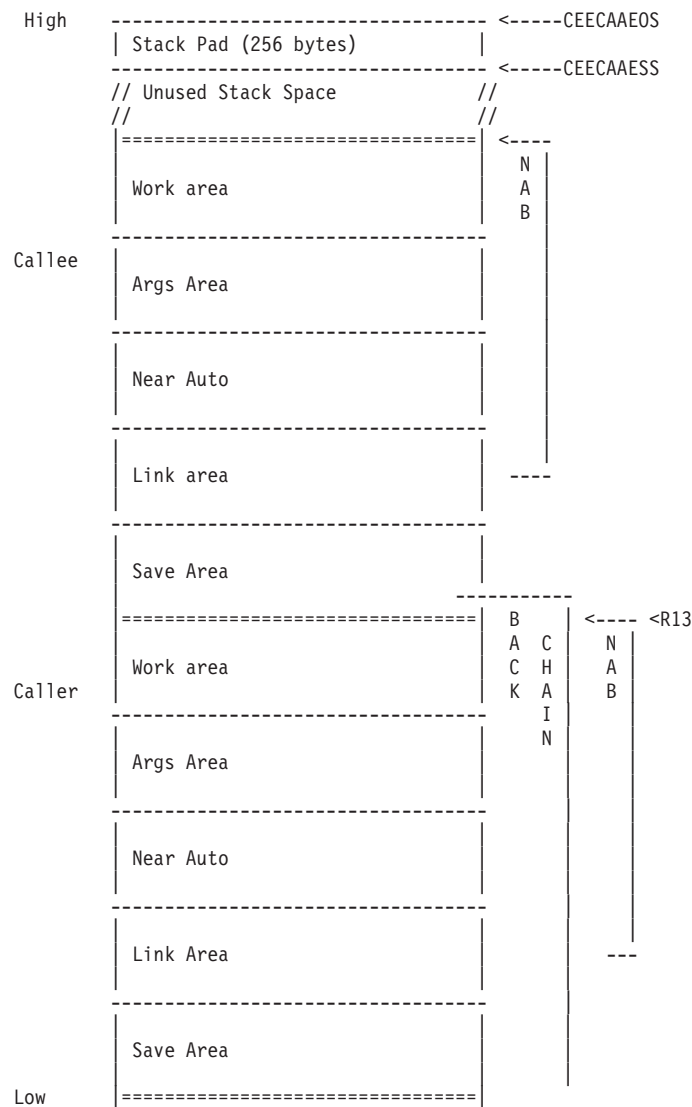


Figure 43. Stack segment showing FASTLINK frames

Argument list format

FASTLINK utilizes a logical argument list. Upon entry to the FASTLINK entry point at +16, the argument list is located in the argument area which is at a fixed location in what will be the callee's stack frame. At the +16 Entry Point, some of the argument values are passed in registers and some in storage. The physical argument list in storage has space for the arguments which are passed in registers. The logical argument list contains all of the arguments. The logical argument list consists of the physical argument list plus the contents of those registers used to pass arguments. Depending upon the type of the parameters, some arguments are loaded into either the GPRs or the FPRs.

FASTLINK linkage supports both direct and indirect arguments for calls between cooperating routines and thus the argument list may contain a mixture of values and addresses. Because the argument list may contain values, it has no explicit termination bit and the length of the FASTLINK argument list is specified elsewhere.

FASTLINK CALL linkage

Note: The width of argument list elements is 4 bytes or a multiple thereof when direct values are passed.

Argument passing

The logical argument list used with FASTLINK linkage is of the same format as the C linkage argument list, however, GPR1 does not point to the argument list. Instead, the arguments are placed into the argument area of the callee's stack frame and/or certain general purpose or floating point registers.

In FASTLINK the first 3 words of the virtual argument list are loaded into GPR1-3 if they represent indirect arguments or direct value arguments of data types other than floating point (real or complex). If a direct value floating point argument (real or complex) begins in the first 3 argument words, it is loaded into an appropriate number of floating point registers FPR0 through FPR6. Note that only one such floating point value is loaded into a floating point register. If a second floating point value begins in the first 3 virtual argument words, it is located in storage. When a floating point argument is loaded in FPRs, the contents of the GPRs corresponding to those argument words are unpredictable and are not preserved over the call.

Arguments that are not loaded into a GPR or FPR are located in the physical argument list in storage. The argument slots in the physical argument list corresponding to the arguments loaded into registers are reserved and their contents at the time of call are undefined; these slots in the argument area may be used by the callee.

The unused, reserved slots in the argument list may be used to store the arguments passed in registers. This is useful if the callee takes the address of an argument that is passed in a register or in a code produced by a compiler which has fixed register usage assignments which overlap with registers 1 through 3.

C allows arithmetic to be performed on pointers and the address of a parameter may be taken. Although it is not ANSI C conforming, some programmers use address arithmetic to locate and reference any of the parameters. Since some arguments are located in registers this practice may access uninitialized storage. Hence if the address of a parameter is taken then the callee's prolog code must store all of the parameters passed in registers into the physical argument list (potentially any of the arguments may be referenced without the compiler being able to detect such references).

The argument list can be modified by the called routine. However, such updates are not reflected to the calling programs HLL variables, for example, when compiling code for the caller the compiler assumes that the argument list is destroyed across a call.

Considerations for FASTLINK routines with variable number of parameters

When a C++ caller has a prototype visible which ends with an ellipsis then the first three words of the argument list are loaded in GPRs 1-3 regardless of their type, and no values are loaded into the floating point registers. From the rules given earlier we observe that for FASTLINK callers without prototypes, GPRs 1-3 are always loaded with the first three words of the virtual argument list. Thus when a procedure whose prototype contains ellipses is invoked at the +16 entry point the location of the first words of the argument list is always in the GPRs. The +16 Entry Point prolog stores GPRs1-3 into the physical argument list. The +0 Entry Point prolog does not copy the argument list into the argument area. Prologs

for both the +0 Entry Point and the +16 Entry Point must pass GPR1 containing the address of the physical argument list to the body of the code and therefore a varargs code body always addresses its parameters based upon GPR1. GPRs 2-3 must be preserved by a vararg routine when entered at the +16 Entry Point.

Register conventions

The FASTLINK register linkage conventions at the +16 Entry Point follow. The caller is responsible to ensure that registers are set up as indicated. The callee is responsible to preserve or restore certain registers as noted. The stack pointer (R13) must be kept valid at all times during execution.

Register	Description			
GPR0	Undefined, Not preserved.			
GPR1-3	First argument words, or undefined: 1. If no arguments exist 2. If the corresponding arguments are floating point scalars or floating point scalar complex values.			
GPR1-3	When are GPR Registers 1-3 preserved?			
GPR1-3	Logical Argument Word 1	Logical Argument Word 2	Logical Argument Word 3	Registers Preserved
GPR1-3	<i>empty</i>	<i>empty</i>	<i>empty</i>	GPR1-3
GPR1-3	<i>empty</i>	<i>empty</i>	<i>empty</i>	GPR2-3
GPR1-3	argument	<i>empty</i>	<i>empty</i>	GPR3
GPR1-3	argument	argument	argument	none
Note: When specified, <i>empty</i> means that there is no corresponding parameter value. Thus, a call with no parameters preserves the GPRs 1-3. A call with one floating point extended parameter uses the FPRs to contain the floating value and, except for a very special case, GPRs 1-3 have an undefined value and are not preserved over the call.				

Register	Description
GPR4-11	Undefined, preserved.
GPR12	CAA address. Must be valid on entry to any Language Environment routine. Need not be valid during execution of a routine. Preserved.
GPR13	Stack frame address. Must be valid at all times.
GPR14	Return address. Preserved.
GPR15	The +0 Entry Point entry point address. Must be valid on entry. Value contains return code on return. Note: The return code referred to here is not to be confused with the return value for functions. Some languages use a return code to facilitate multiple return points, and others pass a status code between the caller and the callee using this return code.
FPR0-6	Value of first floating point value if one of the first three argument words represents a direct floating point argument, otherwise undefined. For functions with floating point result, contains result on exit, otherwise not preserved.
Condition register	Undefined. Not preserved
Program mask	As documented in this book
Vector Registers	Undefined Not preserved
ARs	Undefined Preserved

Leaf routines

Leaf routines are routines that do not call any other routines (with the possible exception of routines that are inlined). Leaf routines that have the following characteristics do not need to allocate a stack frame. Such routines are called Sleaf routines and they may use truncated prologs and epilogs:

- whose work area requirements may be obtained from the stack segment pad
- which are not vararg
- which do not perform stack frame extension
- which are not bilingual

Note: When entered at the +16 Entry Point, the value of the NAB in the caller's DSA provides 256 bytes of pad area that can be used by the current routine to store data into.

Code sequences

This section contains annotated code sequences for FASTLINK linkages (calls, prologs and epilog). These code sequences will, in general, be used by FASTLINK C++ and HPCJ but they are not necessarily exhaustive. Thus the compiler may have to supplement it to meet compiler-specific requirements like dealing with long argument lists. C-to-FASTLINK is shown as an illustrative example — there currently is no support in either C++ or HPCJ for old-to-new linkage.

FASTLINK, non-Sleaf routine

Figure 44 on page 111 shows an example of FASTLINK to FASTLINK linkage code sequence with a non-Sleaf routine.

FASTLINK to FASTLINK, Non-Sleaf Routine

```

R0      Undefined, not preserved
R1-R3   Args
R4-R11  Undefined, must be preserved
R12 =>  CAA
R13 =>  Language Environment stack frame (DSA)
R14,R15 Linkage registers

Caller:
00- 58F0 ****      L   15,=V(routine)
04- 4DE0 F010      BAS 14,16(15)      Call to FASTLINK entry pt
08- 4700 ****      NOP N

Callee:
00      routine DS   0D      C-style entry point...
00- 47F0 F001      B    1(,r15)      ...is invalid for FASTLINK
                                DC   X'01'      Language Environment
                                                eyecatcher

04- 01C3C5C5      DC   CL3'CEE'
08- *****      DC   A(SIZE)      DSA size
0C- *****      DC   A(PPA1-routine) Offset to PPA1
                                DS   0D      FASTLINK entry point
10- 90Ex D00C      STM  r14,rXX,12(r13) Save caller's regs
14- 58E0 D04C      L    r14,76(,r13) Get NAB
18- 4100 Exxx      LA   r0,Size(,r14) Move NAB forward by Size
1C- 5500 C314      CL   r0,CEECAAESS-CAA(,r12)
                                Check for stack end
20- 4140 F04C      LA   r4,76(,r15) Set up basereg
24- 47D0 F03A      BNH  58(,r15) Branch if no stack overflow
28- 58F0 C31C      L    r15,CEECAAOGETS-CAA(,r12)
                                FASTLINK overflow routine
2C- 184E          LR   r4,r14 Copy requested NAB into r4
2E- 05EF          BALR r14,r15 Branch to overflow rtn
30- 00000000      =F'0'
34- 0540          BALR r4,r0 Establish addressability...
36- 4140 4016      LA   r4,22(,r4) ...and set up basereg
3A- 5000 E04C      ST   r0,76(,r14) Save new NAB in DSA
3E- 9210 E000      MVI  0(r14),16 Initialize member word
42- 50D0 E004      ST   r13,4(,r14) Backchain to caller's DSA
46- 5800 D014      L    r0,20(,r13) Reload reg 0 (WSA address)
4A- 18DE          LR   r13,r14 Set new DSA addr in stack reg

                                End of Prolog
4C-          CODE DS   0H
                                ...
                                Start of Epilog
60- 58D0 D004      L    r13,4(,r13) Get caller's DSA addr
64- 58E0 D00C      L    r14,12(,r13) Restore return reg
68- 98yz Doff      LM   rYY,rZZ,OFF(r13) Restore other regs
6C- 47F0 E004      B    4(,r14) FASTLINK return

```

Figure 44. FASTLINK to FASTLINK linkage code sequence, non-Sleaf routine

Notes:

1. Instructions at offsets X'00' and X'04' in the caller.
2. Offset X'04' in caller: FASTLINK callers enter the called routine at +16. C linkage callers enter at +0, but for FASTLINK routines this entry point is invalid and will cause an abend.
3. Offset X'08' in caller: The instruction used to pass control from a FASTLINK-enabled routine must be followed by a NOP instruction, which contains the length of the argument list. The index field is reserved and must be zero. The base and displacement fields (indicated by "N") are treated as a half word signed binary quantity. The value is positive and represents the number of bytes in the argument list. Negative values are reserved.
4. Offset X'00' in callee: This entry point is not valid.

FASTLINK CALL linkage

5. Offset X'04' in callee: Bit 7 in the first byte of the eyecatcher indicates that this routine uses the new FASTLINK dsa layout and that the PPA2 is located by a relative offset. This bit must only be non zero if the new dsa layout and relative offsets are used, use a mask to test this bit and not the whole byte for an exact match to X'01' since some other bits may be assigned for special purposes in the future.
6. Offset X'08' in callee: In cases where the stack frame size is not known at compile time such as variable length argument lists (for example, C++ varargs) then DSASIZ represents only the fixed portion of the stack frame. The actual stack frame size is calculated from DSASIZ plus the size of the argument list contained in the NOP.
7. Instructions at offsets X'10' and X'68' in callee: Line is only required to save/restore the registers actually required by this routine thus the instruction could be a ST/L or even entirely missing. Note that two registers are used in the stack extension logic and thus RXX must be set correspondingly. In routines with few parameters, it is possible that no registers beyond 14 and 15 would need to be saved and restored. The offset into the save area is based upon the first GPR saved.
8. Offset X'18' in callee: Line may be replaced by the following lines when the stack frame size is larger than 4K.

L	DS0,DSASZE-OLD(,15)	Get DSA size
ALR	0,14	Move Nab forward by size
9. Offset X'3E' in callee: Stack frame is marked as FASTLINK for ILC calls with PL/I, COBOL,FORTRAN, or an OS linkage routine, or if this routine needs an exit DSA.
10. Offset X'60' in callee: For details of the handling of function return values, see Function Results.
11. Special considerations, required to handle variable length parameter lists, are documented in Routines with a Variable Number of Parameters.

Considerations for large argument lists

When one routine calls another routine with an argument list greater than 72 bytes, the calling routine must ensure that the current stack segment is large enough to contain the large argument list. The calling routine may elect to accomplish this in two ways:

1. As part of the code to actually generate the call the current stack segment size may be checked and a new segment obtained if necessary. If a new stack segment is required then the current NAB must be updated appropriately prior to the call and just following the return.
2. The calling routines prolog may ensure that sufficient space exists both for the calling routines own DSA requirements plus space for the largest argument list that the calling routine builds.

FASTLINK, Sleaf routine

Figure 45 on page 113 shows an example of FASTLINK to FASTLINK linkage code sequence with a Sleaf routine. Instructions at offsets 10 and 34 in callee are only required to save/restore the registers actually required by this routine. The STM/LM may be replaced by a ST/L if only one register needs to be saved, or may be deleted if no registers need be saved/restored.

FASTLINK to FASTLINK, Sleaf Routine

```

R0      Undefined, not preserved
R1-R3   Args
R4-R11  Undefined, must be preserved
R12 =>  CAA
R13 =>  CEL stack frame (DSA)
R14,R15 Linkage registers

Caller:

00- 58F0 ****          L    r15,=V(leaftrtn)
04- 4DE0 F010          BAS  r14,16(,r15)    Call to FASTLINK entry pt
08- 4700 ****          NOP  N

Callee:
00-          leaftrtn DS    0D              C-style entry point...
00- 47F0 F001          B    1(,r15)        ...is invalid for FASTLINK
          DC    X'01'                    Language Environment
          DC    CL3'CEE'                  eyecatcher

04- 01C3C5C5          DC    A(SIZE)        DSA size
08- *****          DC    A(PPA1-leaftrtn) Offset to PPA1
0C- *****          DS    0D              FASTLINK entry point

10- 9016 D018          STM  r1,r6,24(r13)
14- 58x0 D04C          L    rX,76(,r13)

          End of Prolog
18-          CODE   DS    0H
          ...
          Start of Epilog
34- 9816 D018          LM   r1,r6,24(r13)    Restore regs
38- 47F0 E004          B    4(,r14)        FASTLINK return

```

Figure 45. FASTLINK to FASTLINK linkage code sequence, Sleaf routine

CEECAOGETS get new stack segment routine

CEECAOGETS is similar in function to CEECAAGETS except that the linkage is different and it is intended for use by FASTLINK enabled procedures. When called, the registers should contain the following data:

Register	Contents
R0	calculated required next available byte
R4	caller's next available byte
R12	address of CAA
R13	caller's save area address which must contain a valid NAB field. The save area addressable by R13 is not useable by CEECAOGETS.
R14	address of a fullword containing the length of the argument list. Return to the code is made to R14+4.
R15	address of CEECAOGETS routine

Upon return, the registers have the following contents.

Register	Contents
R14	contains the new DSA address
R0	contains the new NAB
R15	undefined

FASTLINK CALL linkage

Register	Contents
R1-R13	preserved

If the supplied length of the argument list is non zero then the arguments are copied from the old stack segment to the new one. If the Storage option `dsa_alloc_value` indicates that the stack frame is to be initialized then this routine also initializes the `dsa` work area as required. The Link area is copied unconditionally from the old stack segment to the new one.

Note: Functions which are `var_arg/sleaf` do not have their `dsa` frames initialized by this option.

The condition manager and this code must cooperate for the short on stack storage condition. After stack segment overflow has occurred then this routine must ensure that the stack address returned in R0 allows for the 256 byte stack segment pad, for example, the request size behaves as if it were 256 bytes larger than the input R0 would indicate.

Extra Performance Linkage (XPLINK) CALL linkage conventions

This section describes the Language Environment XPLINK protocols for passing arguments to external routines. XPLINK is a linkage convention which differs substantially from the standard Language Environment linkage and FASTLINK linkage protocols. The Language Environment XPLINK protocols are compatible with the 64-bit environment.

The primary goal of XPLINK is to make subroutine calls as fast and efficient as possible by removing all nonessential instructions from the main path. This is achieved by introducing the following:

- growing the stack from higher to lower addresses ("negative-" or "downward-growing")
 - to eliminate overhead in stack frame allocation
 - to eliminate need for inline stack overflow check
 - to allow for an improved epilog
 - to allow addressability to information (such as parameters) in the caller's stack frame
- biasing the stack pointer (by 2048 bytes), so that small functions can save registers in their own stack frame before updating the stack pointer, avoiding address generation interlocks
- reassignment of registers (see "Register Conventions" on page 17) to support more efficient saving and restoring of registers in function prologs and epilogs
- parameter passing in registers, accepting return values in registers
- elimination of Inter-language Call (ILC) overhead (marking of stack frame) for non-ILC calls
- faster call sequences for inter-module calls
- passing the address of the data area associated with a function, its "environment", to the function on entry
- no branching around CEL words
- use of relative branching for function calls where possible

- unification of the various (RENT and NORENT, DLL and NODLL) function pointer implementations, reducing the costs of all operations involving function pointers

An important additional goal is the reduction in size of the function in memory. This is accomplished by eliminating unused information in function control blocks.

XPLINK applications are supported under IMS and LRR (Language Routine Retention).

Register usage

The following list shows register usage and linkage.

- GPR1-3 => arguments (depending upon type)
- GPR4 => the caller's stack frame in the downward-growing stack. This is biased and actually points to 2048 bytes before the real start of the stack frame.
- GPR5 => the called routine's environment pointer
- GPR6 => the entry point in the called routine if the call was made by a BASR instruction
- GPR7 => the return point in the caller's routine. The return point also contains information to determine if the call was made via BASR or branch relative.
- GPR8-15 => preserved
- GPR12 => CAA, the key Language Environment control block (non-64-bit environment)
- FPRs => arguments (depending upon type)

Stack frame mapping

Figure 46 shows the Language Environment XPLINK stack storage model. The prolog of a function usually allocates space (referred to as a "frame", "Stack Frame", or "DSA" - dynamic storage area) in the Language Environment-provided stack segment for its own purposes and to support calls to other routines.

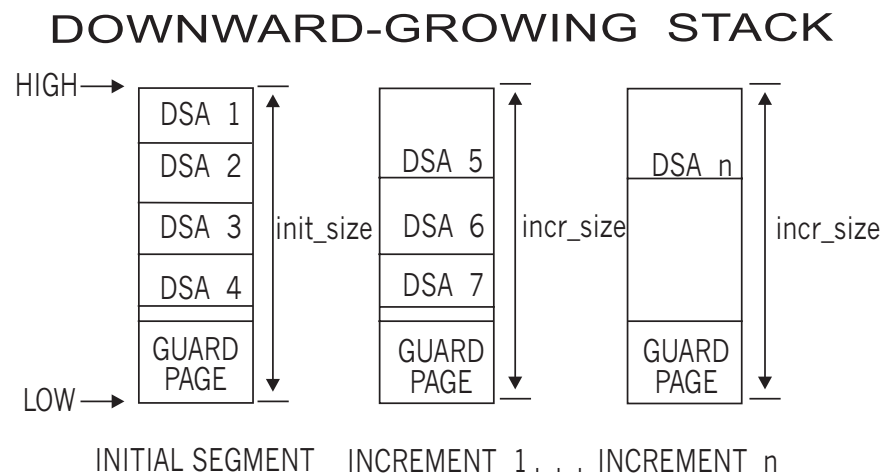


Figure 46. Language Environment XPLINK stack storage model

Stack layout

Figure 47 on page 116 shows the stack frame layout (Figure 145 on page 680 shows the stack frame layout for AMODE64). The stack register points to a location 2048 bytes before the stack frame for the currently active routine. It grows from numerically higher storage addresses to numerically lower ones, that is the stack frame for a called function is normally at a lower address than the calling

Stack Frame Mapping

function. The stack frame is quadword-aligned.

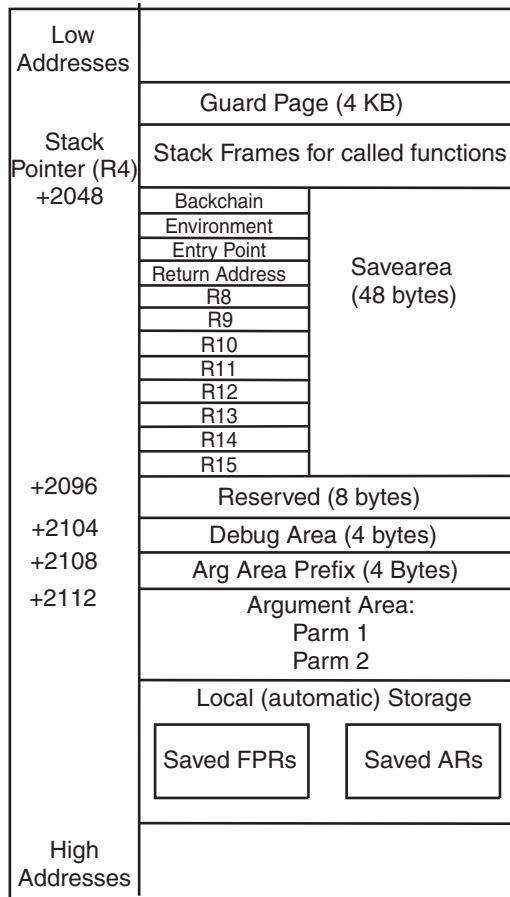


Figure 47. Language Environment XPLINK stack frame layout in a non-64-bit environment

Table 28 on page 117 describes the contents of each area within the stack frame shown in Figure 47.

Table 28. Content of XPLINK stack frame for non-AMODE 64 applications

Stack frame area	Content
Save area	<p>This area is always present when a stack frame is required. It holds up to 12 registers. The first two words hold, optionally, GPRs 4 and 5, the registers containing the address of the previous stack frame and the environment address passed into the function. This is followed by the two words containing GPR6, which may or may not hold the actual entry point address depending on the type of call, and GPR7, the return address. As many of the 8 non-volatile registers as are used by the called function are saved in the following 32 bytes.</p> <p>Except when registers are saved in the prolog, this area may not be altered by compiled code. The PPA1 GPR Save Mask indicates which GPRs are saved in this area by the prolog.</p> <p>Stack overflow is detected by the STM or STMY instruction used to save registers in this save area.</p> <p>Storage of the Backchain field in the save area is triggered by the optional XPLINK(BACKCHAIN) compiler option (or at the convenience of the compiler). The Environment Address is stored when the TEST compiler option or the optional XPLINK(STOREARGS) compiler option is specified, or at the convenience of the compiler.</p> <p>The third slot in the save area contains the value in GPR6 on entry to the routine. If the routine was called with a BASR instruction, the address is that of the function entry point. The fourth slot contains the return address. The return point can be examined to determine how the function was called:</p> <ul style="list-style-type: none"> • If the function was called with a BASR instruction, the entry point address can be found in the third slot of the save area • If the function was called with a relative branch, the entry point can be computed from the return address and the branch offset contained in the relative branch instruction
Reserved	This area is always present and is for the exclusive use of the runtime. It is uninitialized by compiled code.
Debugger area	This area is always present and is for the exclusive use of the debugger. It is uninitialized by compiled code.
Argument area prefix	This area is used for parameter mapping (hidden parameter) to accommodate calls between new and old code. It is uninitialized by compiled code.
Argument area	This area is at the fixed DSA offset of 64 bytes into the caller's stack frame. It contains the argument lists passed on function calls made by the function associated with this stack frame. The called function finds its parameters in the caller's stack frame. A minimum of 4 words (16 bytes) must be always be allocated.
Local storage	This is the space owned by the executing procedure and may be used for its local variables and temporaries.

Stack overflow

To maximize function call performance, XPLINK replaces the explicit inline check for overflow with a storage protect mechanism that detects stores past the end of the stack segment.

The stack floor is the lowest usable address of the current stack segment. In the lower storage addresses, it is preceded by a store-protected guard page used to detect stack overflows.

Availability of space for a stack frame is ensured in the function prolog usually by storing into the start of the called function's frame. In case of overflow, this triggers an exception which in turn causes a discontinuous extension of the stack by Language Environment. Functions with a DSA larger than the guard page use the

Stack Frame Mapping

stack floor address in the CAA to verify space availability. Allocation and deallocation of extensions is transparent to the application.

To make the stack appear contiguous to the application, a small stack frame containing all fields up to and including the Argument area will be allocated in the new stack segment for use by the called function and the contents of the caller's stack up to the end of the argument area should be copied into the new stack segment. The length of the argument list expected is available in the called function's PPA1 except for vararg functions, where the entire argument area in the calling function must be copied.

Stores into the guard page done outside the prolog and done outside "alloca" built-in processing should be treated as invalid and cause the application to be terminated.

Prolog/epilog examples

This section contains typical prolog and epilog code sequences for XPLINK. These are examples, not definitive code sequences that must be generated by conforming compilers.

Table 29 is an example of a small size stack frame (the dsasize is less than or equal to 2048 bytes); there is no backchain or alloca.

Table 29. Prolog/epilog example: small size stack frame, no backchain, no alloca

	DC	0D'0',XL8'00C300C500C500F1'.C.E.E.1	
	DC	A(*-8-PPA1),AL.27(dsasize/32),AL.5(flags)	
EP	STM	6,lastused,2048-dsasize+8(4)	
	STM	1,Rx,2112(4)	if XPLINK(STOREARGS) or TEST, or varargs
	AHI	4,-dsasize	update stack pointer
	...		
		<i>function body</i>	
	...		
	LM	7,lastused,2048+12(4)	restore registers
	LA	4,dsasize(,4)	restore stack pointer
	B	4(,7)	return to caller

Table 30 is an example of a small size stack frame (dsasize is less than or equal to 2048 bytes) with a backchain and vararg.

Table 30. Prolog/epilog example: small size stack frame, vararg, backchain

	DC	0D'0',XL8'00C300C500C500F1'.C.E.E.1	
	DC	A(*-8-PPA1),AL.27(dsasize/32),AL.5(flags)	
EP	STM	4,lastused,2048-dsasize(4)	
	STM	2,3,2112+4(4)	save varargs if any in first 3
	AHI	4,-dsasize	update stack pointer
	...		
		<i>function body</i>	
	...		
	LM	7,lastused,2048+12(4)	restore registers
	LA	4,dsasize(,4)	restore stack pointer

Table 30. Prolog/epilog example: small size stack frame, vararg, backchain (continued)

B	4(,7)	return to caller
---	-------	------------------

Table 31 is an example of an intermediate size stack frame ($2048 < dsasize < 4096$); there is no backchain, alloca, or vararg.

Table 31. Prolog/epilog example: intermediate size stack frame, no backchain, no alloca, no varargs

	DC	0D'0',XL8'00C300C500C500F1'.C.E.E.1	
	DC	A(*-8-PPA1),AL.27(dsasize/32),AL.5(flags)	
EP	AHI	4,-dsasize	update stack pointer
	STM	6,lastused,2048+8(4)	
	...		
		<i>function body</i>	
	...		
	LM	7,lastused,2048+12(4)	restore registers
	LA	4,dsasize(,4)	restore stack pointer
	B	4(,7)	return to caller

Table 32 is an example of a large size stack frame ($4096 \leq dsasize \leq 32768$) in AMODE 31.

Table 32. Prolog/epilog example: large size stack frame ($4096 \leq dsasize \leq 32768$), AMODE 31

	DC	0D'0',XL8'00C300C500C500F1'.C.E.E.1	
	DC	A(*-8-PPA1),AL.27(dsasize/32),AL.5(flags)	
EP	DS	0D	
*		Combine any of the following 3 instructions into STM	
	ST	1,2112+0(,4)	if XPLINK(STOREARGS)
	ST	2,2112+4(,4)	if XPLINK(STOREARGS) or 2nd parameter is vararg
	ST	3,2112+8(,4)	if XPLINK(STOREARGS) or more than 2 parameters
	AHI	4,-dsasize	update stack pointer
	C	4,CEECAA_STACKFLOOR-CEECAA(,12)	check bottom of stack
	JM	EXT	
STK	DS	0H	
	STM	6,lastused,2048+8(4)	
	...		
		<i>function body</i>	
	...		
	LM	6,lastused,2048(4)	restore registers
	AHI	4,dsasize	
	B	4(,7)	return to caller
	DC	0D'0',XL8'00C300C500C500F2'.C.E.E.2	
	DC	A(this marker - entry point marker)/8	
EXT	DS	0D	

Prolog/Epilog examples

Table 32. Prolog/epilog example: large size stack frame ($4096 \leq dsasize \leq 32768$), AMODE 31 (continued)

	LR	0,3	
	L	3,CEECAHPGETS-CEECAA(,12)	
	BASR	3,3	call Language Environment stack extender
	NOP		
	LR	3,0	
	J	STK	

Table 33 is an example of a huge size stack frame, where the dsasize is greater than 32768; this is also an AMODE 31 example.

Table 33. Prolog/epilog example: huge size stack frame ($32768 < dsasize$), AMODE 31

	DC	0D'0',XL8'00C300C500C500F1'.C.E.E.1	
	DC	A(*-8-PPA1),AL.27(dsasize/32),AL.5(flags)	
EP	DS	0D	
	ST	3,2120(,4)	save in XPLINK(STOREARGS) slot
	LR	0,4	
*		There will be one or more AHI instructions of size -32768 until the	
*		remainder of the dsasize is less than 32768	
	AHI	4,H'-32768'	
	AHI	4,H'-(dsasize%32768)'	
	C	4,CEECAA_STACKFLOOR-CEECAA(,12)	check bottom of stack
	JM	EXT	
STK	DS	0H	
	STM	6,9,2048+8(4)	
	ST	0,2048(,4)	save backchain, possibly updated by the runtime if there was a stack extension
	...		
	<i>function body</i>		
	...		
	LM	4,lastused,2048(4)	restore registers
	B	4(,7)	return to caller
	...		
	DC	0D'0',XL8'00C300C500C500F2'.C.E.E.2	
	DC	A(this marker - entry point marker)/8	
EXT	DS	0D	
	L	3,CEECAHPGETS-CEECAA(,12)	
	BASR	3,3	call Language Environment stack extender
	NOP		
	LR	3,0	
	L	3,2120(,3)	
	J	STK	

Finally, Table 34 on page 121 shows an XPLINK example in AMODE 31.

Table 34. Prolog/epilog example: XPLINK, no alloca, no storeargs, saves regs 5-9, DSA size=3712 (AMODE 31)

@1L0	DS	0D	
		=F'12779717'	
		=F'12910833'	
		=F'152'	
		=F'3712'	
main	DS	0D	
	STMY	r5,r9,-1660(r4)	save caller's regs
	AHI	r4,H'-3712'	R4 = new DSA address
		<i>function body</i>	
	L	r7,2060(,r4)	restore return address
	LM	r8,r9,2064(r4)	restore caller's registers
	LA	r4,3712(,r4)	point to caller's DSA
	SR	r3,r3	R/C = 0
	B	4(,r7)	return to caller

Stack extension

When the stack frame size is greater than the guard page size, the new stack pointer value must be compared to the CEECAA_STACKFLOOR field. When the stack pointer is less, then a stack expansion routine must be called explicitly to create the new stack increment.

DSA Extension -- alloca(): Sometimes a program's automatic (stack) storage requirements are not known until runtime, DSA extension allows a program to dynamically allocate additional automatic (stack) storage. (The z/OS XL C/C++ compiler built-in function `alloca()` is the C/C++ implementation of DSA extension.) For XPLINK, allocating additional stack storage will also require moving the register save area at the beginning of the stack frame (for example, the Register 4 value will change). This storage is automatically freed when the function in which it was acquired returns.

When DSA extension causes a stack extension, the processing performed will be very different from normal stack extension in terms of what gets copied to the new stack increment and the mechanism to free the stack increment.

The following discussion explains the rules to be observed in handling `alloca()` in XPLINK:

- The stack pointer (R4) must always point to a location 2048 bytes before the current function's stack frame. This may or may not be within the Guard page.
- Functions that use "alloca" must use a different register (called the "alloca() register") to address their automatic storage and their parameters. This register must be set to point to automatic storage (computed from GPR4) in the prolog; it must keep this value throughout the function (until register contents are restored in the epilog).
- A function that uses "alloca" must acquire a stack frame and its prolog must store GPRs 4, 6 and 7 in its stack frame. Such a function cannot be considered a XPLleaf routine and may not be marked as such in the PPA.
- The argument area used to construct argument lists for called function must be addressed using the top of the stack pointer (R4).

DSA Extension

- All live values from the beginning of the stack frame up to and including the entire argument area must be copied to the new start of the stack frame. This includes all saved registers, but not slots for registers that were not saved. It does not include the Debug Area or the Reserved field. It does not include the Arg Area Prefix field. If an argument list is under construction when `alloca()` is called then it includes those arguments already constructed, otherwise not. When an external call is made to the runtime for `alloca()` the generated code must ensure that any live values in the argument area are copied; the runtime is responsible for copying the entire 48-byte savearea.
- `alloca` must round all requested storage amounts to a multiple of 16 bytes (a quadword) to maintain stack frame alignment
- it is intended that `alloca` may, in future, be inlined. An inline `alloca` will trigger a guard page exception if stack extension is required. The design for this is not part of this document.

Functions that use "alloca" require changes to their prologs and epilogs to maintain addressability to their automatic variables and parameter list. Also, fields in the entry mask and PPA1 must correctly indicate that the routine uses a DSA extension. For more information, see "XPLINK DSA extension services" on page 216.

	DC	0D'0',XL8'00C300C500C500F1'	.C.E.E.1
	DC	A(*-8-PPA1),AL.27	(dsasize/32),AL.5(flags)
EP	STM	4,lastused,2048-dsasize(4)	
	STM	1,Rx,2112(4)	if XPLINK(STOREARGS), TEST, or varargs
	AHI	4,-dsasize	update stack pointer
	LA	Ry,64+argsize(,4)	set alloca register
	:		
	:		
		<i>function body (addresses auto storage using the alloca() register)</i>	
	:		
	:		
	L	7,2048+12(,4)	restore return address
	LM	8,lastused,2048+16(4)	restore remaining registers
	L	4,2048(,4)	restore stack pointer
	BR	7	return to caller

Obtain an XPLINK Downward-Growing Stack Extension: This CWI is invoked when there is not enough room in the current XPLINK downward-growing stack segment to hold the caller's stack frame. It will be used by z/OS XL C/C++ compiler-generated code when the stack frame size is greater than the size of the guard page (4K).

Obtain an XPLINK Downward-Growing Stack Extension

Input/Output	Register	Used for
Input Registers	R0	Previous stack pointer value (if PPA1 indicates that routine stores the backchain)
	R1 - R2	Not used
	R3	Return Address
	R4	Calculate stack pointer
	R5	Not used
	R6	Value to be saved at offset 2056 of new DSA
	R7	Return value to be saved at offset 2060 of new DSA
	R8 - R11	Not used
	R12	CAA address
	R13- R15	Not used
Output Registers	R0	Modified previous stack pointer (or unchanged)
	R1 - R3	Unchanged
	R4	New stack pointer
	R5	Unchanged
	R6	Modified entry point
	R7	Modified return address
	R8 - R15	Unchanged

The following is an example of an invocation of this CWI:

```
L      3,CEECAHPGETS-CEECAA(,12)
BASR   3,3
DC     X'4707'
DC     Y(call offset)
```

Where call offset is a signed offset in doublewords from the doubleword at or preceding the return point of the BASR instruction.

- If the value is negative, it is the signed offset to the entry point marker.
- If the value is positive, it is the offset to the call descriptor for this call.

A call descriptor is required when the signed offset to the entry marker is not negative or cannot be represented by a 2-byte signed field. See Call Descriptor for the format of the call descriptor.

This CWI returns control to its invoker at the return address:

```
BR     3
```

Exceptions

The following sections describes some rules and exceptions that should be considered. In these rules, “pointing to stack frame” means “pointing to 2048 bytes before the stack frame”.

Rules Applicable to Prologs:

- The prolog must be contiguous (except for the out-of-line call to the stack extender) and less than or equal to 128 bytes in length.
- When a procedure requires a stack frame, it must check the stack segment for space availability in the prolog and it must save GPRs 6 and 7 in the Save Area. GPR6 must be saved by the instruction that checks for stack space availability.

Exceptions

- Saved GPRs must always be saved in their canonical location which is as if a STM 4,15,2048(4) had been executed.
- When a routine does not require a stack frame, it must maintain the contents of GPR7 (return address) and GPR6 received at entry at all times (not just during prolog execution) for exception handling purposes.
- GPRs 6 and 7 may not be changed in the prolog.
- Any instruction that is part of the window ranging from the entry point up to and including the instruction updating GPR4, may not introduce any potential exceptions other than as might be caused by an invalid GPR4.
- Except for a NOP, a prolog may not start with a Branch on Condition instruction (opcode 0x47). (Many non-XPLINK functions start with a branch instruction; this rule minimises the possibility of tools that examine prologs mistaking an XPLINK prolog for an older-style prolog.)
- If the stack pointer (GPR4) is updated before the registers are saved, GPR0 must be set to the value in GPR4 at function entry before GPR4 is updated. GPR0 is updated by Language Environment during stack extension; the updated value should be stored in the backchain field of the stack frame.
- R4 points to the caller's stack frame, the new stack frame, or the proposed new stack frame location (possibly in the guard page) throughout the prolog. No other value is allowed.
- Registers 5-15 may not be modified in the prolog until after GPR4 is updated to point to the new stack frame.
- If an explicit check for stack overflow is not done in the prolog using the "End of Stack" field in the CAA, the first instruction that touches the new stack frame must be STM 4,x,nnn(4), STM 5,x,nnn(4), STM 6,x,nnn(4), STMY 4,x,nnn(4), STMY 5,x,nnn(4), or STMY 6,x,nnn(4).

Rules Applicable to Epilogs:

- The epilog must be contiguous and less than or equal to 128 bytes in length.
- Except for XPLleaf routines, epilog code must extract the return address from the savearea, and it must do this before updating GPR4 to point to the caller's stack frame. In XPLleaf routines, the return address must be taken from GPR7, which remains unaltered by compiled code throughout the life of the function. This allows the runtime to steal the return address for its own purposes.
- GPR4 must point to the current function's stack frame on entry to the epilog; when it's updated it must point to the caller's stack frame; no other value is allowed.
- The epilog contains no call, including alloca().
- Compiled code may not refer to its own stack frame after updating GPR4 .

XPLleaf Routines: XPLleaf routines are functions that make no function calls (including alloca()). They do not contain try, catch, or throw statements nor do they acquire their own stack frame. GPRs 4, 6 and 7 must not be altered by the routine.

Stack Overflow Exception: In XPLINK, stack frame allocation is designed to trigger a protection exception when insufficient storage remains in the current stack segment. This exception requires proper handling in the Language Environment interrupt exit. A valid request for stack extension can be recognized by Language Environment as follows:

- The exception is caused by STM 4,x,nnnn(4), STM 5,x,nnnn(4), STM 6,x,nnnn(4), STMY 4,x,nnnn(4), STMY 5,x,nnnn(4), or STMY 6,x,nnnn(4).

- The target address in nnn(4) is within the guard page of the current stack segment.
- The exception address is within the prolog defined by the PPA1 of the function experiencing the exception.

Exception processing may need to distinguish between a request made in the function prolog and through "alloca". For example, set up and initialization of an extension may be different in the two cases (e.g., copying of parameters). The prolog length field in the PPA1 is provided for this purpose.

For requests in the prolog, the required stack frame size is available in the entry point marker while for requests in alloca it must be taken from R0.

When a stack overflow occurs, the caller's arguments must be made available in the newly created stack segment.

It is expected that Language Environment will update the stack floor field in the CAA when the application traverses a stack segment and will handle stack segment deallocation. For calls, this could be done by inserting a stack frame for a special library function in the new stack segment such that the function becomes part of the return flow of the application. When a stack segment extension is caused by alloca, the special linkage routine needs to be inserted in the return path of the function issuing alloca. It should be noted that one function could cause multiple segments to be allocated. The active stack segment could be pointed to by a fullword in the CAA.

Stack Unwinding: Because XPLINK does not always provide a back chain, a new method for unwinding the stack must be followed:

- Determine if the current instruction address is in a function prolog (see below):
- If the current point of execution is in a prolog, determine if GPR4 has been updated (the offset of the beginning of the instruction updating GPR4 is in the PPA1). If GPR4 has been updated, reverse this by adding the DSA size (found in the entry point marker for the function) to GPR4. This is the address of the previous stack frame.
- At this point, GPR4 points to a 2048 bytes before a valid stack frame (the caller's in the case on an incomplete prolog).
- Using the current GPR4 value, locate the entry point of the function associated with the stack frame:

Locate the return address of the function in the 4th slot of the current stack frame at 2060(4). At the return address find the call type, to determine the instruction making the call. If it's a relative branch, compute the target offset from the branch instruction contents and its address to determine the entry point. If it is a BASR instruction, the entry point to the function is the value passed into the function in GPR6 and stored in the 3rd slot of the current stack frame at 2056(4).

- The current entry point can be used to locate the PPA1 for this function, but this is not required for stack unwinding:
 - Subtract 16 from the entry point address to get the address of the entry point marker.
 - Add the word at 8 bytes past this address (the PPA1 offset) to this value.
- "Special linkage" stack frames contain identifying markers. Language Environment architecture specifies how to use information in this stack frame to get to the previous (possibly non-XPLINK) stack frame.

Exceptions

- The entry point marker contains a flag to indicate if `alloca()` is used in the function. If it is not, the entry point marker contains the `dsasize` of the function associated with the current stack frame; add this value to the current stack frame address to get the address of the previous stack frame.
- If `alloca()` is used in the function, the previous value of GPR4 (2048 bytes before the previous stack frame) is stored at 2048(4).
- Continue, as required.

Determining if an Execution Point is in a Prolog: From a point of execution:

- Scan backwards for up to 16 doublewords looking for a doubleword-aligned marker as described in "Code Markers" below.
- If not found the current point of execution is not in a prolog.
- If found and the marker is not an entry point marker, the current point of execution is not in a prolog.
- In the entry point marker, the word at offset +8 contains the offset, from the marker, of the associated PPA1.
- The PPA1 contains the length of the prolog. If the current point of execution is not within this range (from the entry point, the doubleword following the entry point marker), the current point of execution is not in a prolog.

Finding the Entry Point of the Current Function:

- Determine if the current point of execution is in a prolog. If it is, the entry point is at the beginning of the prolog.
- Locate the return address of the function in the 4th word of the current stack frame at 2060(4). At the return address find the call type, to determine the instruction making the call. If it's a relative branch, compute the target offset from the branch instruction contents and its address, to determine the entry point. If it's a BASR instruction, the entry point to the function is the value passed into the function in GPR6 and stored in the 3rd word of the current stack frame at 2056(4).

Code markers

The following sequences identify points in code that are significant to Language Environment. Each of these is doubleword-aligned and has the same initial 7-byte sequence. Markers that could be found in the body of compiled code (types 2 and 3) contain the offset of the associated entry point marker at offset +8.

- Entry point marker (type 1)
- Stack extension marker (type 2)
- Data marker (type 3)
- Stub marker (type 4)

Table 35 shows the format of entry point marker type 1.

Table 35. Entry point marker (type 1)

+0	0x00	'C'	0x00	'E'	0x00	'E'	0x00	'1'
+8	offset of PPA1 from entry point marker				dsasize/32			EP flags

In an entry point marker, the word at offset +8 is at offset from the beginning of the Entry Point marker to the PPA1 associated with the entry point. EP flags has the following format.

1	...	Function is an XPLleaf routine, saving registers in its own stack frame but not updating the stack pointer	
		.	..
0	0	0	Must be zero

The stack extension marker (type 2), shown in Table 36, identifies stack extension code that is logically part of the function's prolog but not within the range of instructions defined to be part of the prolog by the PPA1 "(length of prolog)/2" field.

Table 36. Stack extension marker (type 2)

+0	0x00	'C'	0x00	'E'	0x00	'E'	0x00	'2'
+8	offset to entry point marker from this Marker/8				Reserved			

The data marker (type 3), shown in Table 37, follows any data in the code section that might be confused for a "real" marker because it contains the values in the first seven bytes of any marker style:

Table 37. Data marker (type 3)

+0	0x00	'C'	0x00	'E'	0x00	'E'	0x00	'3'
+8	offset to entry point marker from this Marker/8				Reserved			

The stub marker (type 4), shown in Table 38, marks the beginning of runtime stubs.

Table 38. Stub marker (type 4)

+0	0x00	'C'	0x00	'E'	0x00	'E'	0x00	'4'
----	------	-----	------	-----	------	-----	------	-----

Argument list format

The following sections describe the format of the argument list in detail.

Function Calls: In XPLINK, each function has a data area associated with it, its environment, whose address is passed by a caller in general purpose register 5. For C and C++ programs, this environment will in most cases be the compiler defined area @STATIC. @STATIC is a structure existing once for each compilation unit and residing in the WSA. Callers therefore need two pieces of information for each function they call:

- the address of the called function's environment area

Argument list format

- the address of the called entry point

This information, organized in two consecutive long integers (fullwords) on a doubleword boundary, is referred to as a Function Descriptor.

Resolution of function linkage is done at the stage in the compile/link/execute process where enough information is available to make the proper choice with respect to performance and flexibility. In some cases, calls can be resolved at compile time. For calls outside a compilation unit the resolution is postponed to the binder for best results, and when DLLs are used, to the runtime environment.

Excluding parameter handling, the **Calling Scheme** is made up of a sequence of instructions (CALL) that load the called function's Environment area address, load the called function's entry point address, and invoke the called function. Details of the generated sequences for different types of calls are described in separate sections below. Calls to routines in Dynamic Link Libraries (DLLs) are supported naturally without special compiler options. At every call site, Register 12 must contain the address of the CAA.

With XPLINK, the function entry point address is not always passed to the called function. To allow Language Environment and other tools to find the entry point of the currently executing routine, every call site, located by the "return address" field of the current stack frame, contains information necessary to locate the entry points of both the calling and called functions and, if required, information about floating-point parameters passed and return value adjustment required to allow interface mapping when mixing XPLINK and non-XPLINK code. This is done by encoding information in a NOP instruction at the return point.

	CALL		
* *	NOP	0(call type)	Shown as <i>NOP type,<offset></i> in subsequent sequences
	ORG	*-2	Back up to last two bytes of NOP
	DC	AL2(call offset)	A signed offset in doublewords from the doubleword at or preceding the return point (NOP). If negative, offset to entry point marker; if positive, offset to the Call descriptor for this signature

"Call type" is a 4-bit field describing the type of call. The call is not required to pass the function entry point address; the NOP following the call, which can be found via the return address (in GPR7), provides the information required to compute the entry point address in cases where it is not passed in register.

Call Type	
0000	BASR
0001	BRAS
0010-0110	Reserved
0111	Special linkage (for example, 3,3 for explicit stackextension)
1000-1111	Reserved

Call offset is a 16-bit field containing the offset in doublewords from the call site to, if negative the entry point marker for the function or, if positive, a call descriptor, described below, which contains both the offset to the entry point marker and information about parameter and return types. This definition requires both entry point markers and call descriptors to be on doubleword boundaries, but imposes no alignment requirement on the call itself. The entry point marker is located by taking the address of the call information field, setting the last 3 bits to zero, and adding $8 * (\text{the call offset})$. Figure 48 shows the resulting XPLINK function layout.

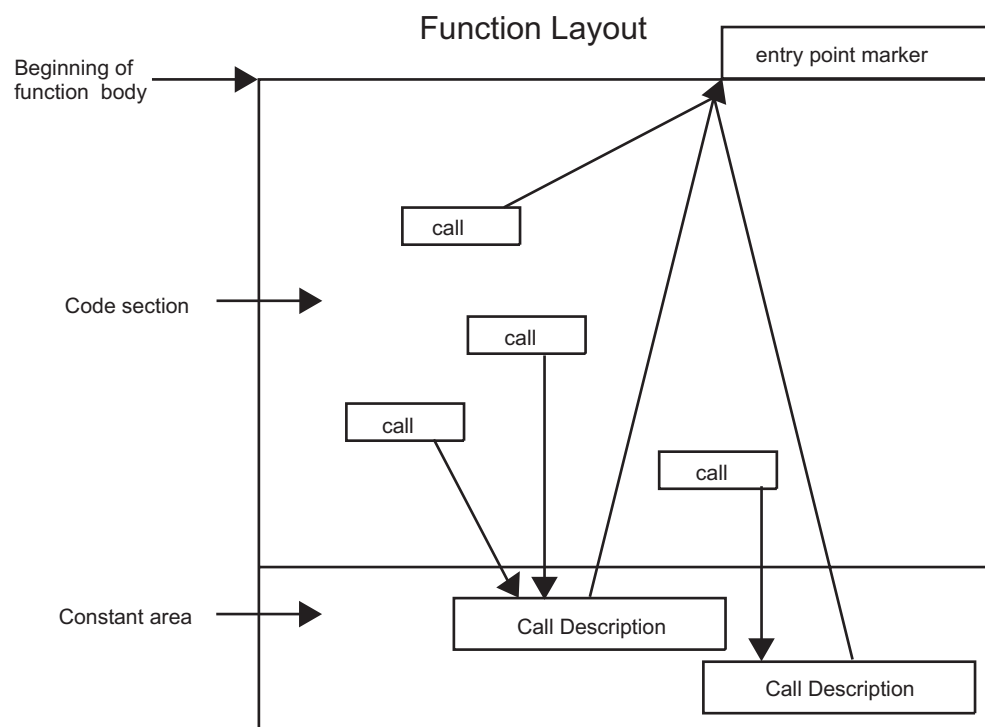


Figure 48. XPLINK function layout

A **Call Descriptor** is created if one of the following occurs:

- The call site is so far removed from the entry point marker of the function that its offset cannot be contained in the 16 bits available in the call information field (the NOP) following the call site.
- The call contains a return value or parameters that are passed in registers or in ways incompatible with non-XPLINK code, that is when the second word of the call descriptor would have a non-zero value

A call descriptor is doubleword aligned with the following format:

Location	Content		
+0	Signed offset, in bytes, to entry point marker		
+4	Linkage	Return Value Adjust	Parameter Adjust

The meaning and content of the second word of the Call Descriptor are described in "Argument Passing" on page 132, in "Function Return Values" on page 134, and in "Call descriptor - linkage type" on page 136.

Argument list format

Calls by Name: The following sections describe how calls are made by name.

Calling Name: The following code sequence is used to call a function by name when that function exists outside the compilation unit, (that is, the function reference is resolved at link-edit time, either statically or dynamically). Calls relative branch:

LM	5,6,...	load environment and function addresses
:	:	
BASR	7,6	call the function
NOP	type,<offset>	

Function Descriptor (space reserved by compiler):

DC	A(environment)	address of function's environment
DC	A(func)	address of function

Intra-module calls: When functions are bound within the same program object as the caller, the address constants to the function's environment and entry point are resolved directly by the binder and loader.

Calling Imported Functions: For calls to imported functions, the compiler will generate the same instruction sequence as for intra-module calls. The function descriptors for all calls to imported functions should be initialized by the binder as required for delayed DLL loading.

Function descriptor, unresolved:

DC	A(function ID)	function ID
DC	A(CEETHLOC)	address of CEETHLOC

Function Descriptor, resolved:

DC	A(environment)	address of function's environment
DC	A(func)	address of function

Function Pointers: A function pointer is a data type whose values range over procedure names. Variables of this type are usually used in procedure call contexts where the particular procedure to be called cannot be determined at compile time. They can also be passed as arguments of a call or used in comparison expressions.

Function pointers are a fullword quantity that is the address of a function descriptor. With some exceptions, there is only one "call-by-pointer" function descriptor per entry point for calls via function pointer. The exceptions are:

- pointers to internal (nested) functions
- pointers to fetched functions and function pointers created by fetched function, because the same function can be fetched more than once.

Note: If an imported function is also called by name, additional function descriptors, as specified in "Calls by Name" on page 130 will also exist.

This is different from NOXPLINK DLL linkage where more than one function descriptor - and hence different function pointer values - may exist for one function, each created in the WSA of the routine that takes the address of (or calls) the function. With a unique function pointer value, *int* to pointer casting works as expected when used with DLLs, providing the same result as with S/390[®] non-DLL and on most other platforms. Also, function pointer comparisons will be significantly faster.

Language Environment will create function descriptors for functions whose address is taken in a separate dynamically acquired storage area (not loaded as part of a module's WSA image) based on information added to a module by the binder. The compiler will flag taking the address of a function differently if it is for a function pointer than if it is for a call by name.

Calling Sequence:

L	Rx,fp	load address of descriptor from function pointer
:	:	
LM	5,6,16(Rx)	load environment and function 1 addresses
:	:	
BASR	7,6	call the function
NOP	type,<offset>	

Function Descriptor:

DC	A(environment)	address of function's environment
DC	A(func)	address of function

Reentrancy: Reentrant programs are structured to allow more than one user to share a single copy of a program object. Users create reentrant programs by writing code that does not modify data in the executable. This is referred to as a naturally-reentrant program. In many languages, users can also request that the compiler create reentrant programs on their behalf by allocating external data in

Argument list format

the writable static area; this is referred to as constructed reentrancy. If a function refers to data in the writable static, its environment must also reside in writable static.

When a program is naturally reentrant it may be desirable to bypass constructed reentrancy to avoid allocation and initialization of a writable static area.

Argument Passing Register Conventions: The following tables describe the XPLINK register conventions used for passing arguments.

Register	Conventions on function entry	Volatility
	exit	
GPR 0	undefined	not preserved
GPR 1	1st word of argument list or undefined	n/a
	part of return value or undefined	
GPR 2	2nd word of argument list or undefined	n/a
	part of return value or undefined	
GPR 3	3rd word of argument list or undefined	n/a
	part of return value or undefined	
GPR 4	Pointer to caller's stack frame - 2048	preserved
GPR 5	Address of environment	not preserved
GPR 6	undefined	not preserved
GPR 7	Return address	not preserved
GPR 8-11	Undefined	preserved
GPR 12	The CAA address	preserved
GPR 13-15	Undefined	preserved

Register	Conventions on function entry	Volatility
	exit	
FPR 0	FP parameter 1 or undefined	not preserved
	part of return value or undefined	
FPR 2	FP parameter 2 or part of FP parameter 1 in register pair 0,2 (for long double) or undefined	not preserved
	part of return value or undefined	
FPR 4	FP parameter or undefined	not preserved
	part of return value or undefined	
FPR 6	FP parameter or part of an FP parameter in register pair 4,6 (for long double) or undefined	not preserved
	part of return value or undefined	
FPR 1, 3, 5 and 7	undefined	not preserved
FPR 8-15	undefined	preserved

Argument Passing: XPLINK uses a logical argument list consisting of contiguous 32-bit words where some arguments are passed in registers and some in storage. This is similar to FASTLINK (see References and Related Documents on page 7) but with some important differences outlined below.

The argument list is located in the caller's stack frame at a fixed offset (+2112) from the stack register (GPR4). It provides space for all arguments, including those passed in registers. It also includes an extra unused word (4 bytes), which may be required in compatibility situations, at the end of the argument area. Its size is sufficient to contain all the arguments, plus the extra unused word, passed on any call statement from a procedure associated with the stack frame.

Since support of stack extensions may require copying of argument lists to different storage locations, the argument list must not include arguments that are pointers to locations in the argument list. The rules for argument passing in registers are as follows:

- The first 3 (4-byte) words of the argument area, regardless of their composition or source, are passed in GPRs 1, 2, and 3, and not in the argument area (although space for these words is reserved in the argument area), except for floating point values, including the real or imaginary constituents of complex types.

Not every language supports complex types. For the purposes of argument passing and function return values, in every language, every aggregate that is (a) not a union, and (b) contains exactly two floating-point types of the same size (4, 8, or 16 bytes) is treated as a complex type.

- Except for arguments in the variable part of a vararg parameter list, up to four floating-point value arguments (the first four) are loaded into floating-point register(s) FPR0, FPR2, FPR4, FPR6 and not passed in the argument area, although space is set aside for these arguments in the argument area. In this fashion, up to four floating-point arguments can be passed depending on their precision (single, double, extended), provided each of these:
 - can be fully (considering the constituent parts of complex arguments separately) contained in the remaining available FPRs, and
 - can be represented in the parameter descriptor flags (that is, they are within 15 words of the previous floating point argument in the argument list).

An extended precision floating point parameter (long double) is always passed in FPR0/2 or FPR4/6. If, for example, the first floating point parameter is double (passed in FPR0) and the second floating point parameter is long double FPR2 will be unused in the parameter list.

If a floating point argument occupies one of the first three words in the argument area, a prototype for the function is visible, and the argument is not part of the vararg portion of a parameter list, the corresponding GPR's value is undefined on entry to the called function.

- Normally, arguments passed in registers are not stored in the argument list although a slot in the argument list is reserved for them.

There is an exception to this rule: if it is required that part of a floating point value be stored in the argument area then the entire floating value is stored in the argument area. This situation arises in calls to unprototyped functions or in the vararg portion of a parameter list when part of the floating point parameter is in the first three words of the argument area. For more information, see examples f13, f18, and f20 in Appendix B, "CALL linkage argument examples," on page 861.

For calls to unprototyped functions, where the caller cannot know if the called function contains a variable (vararg) portion, the argument list must be constructed to allow a call to either a vararg or non-vararg function. In this situation: floating-point arguments in the first 3 words of the parameter list are passed in both GPRs and FPRs; other floating point arguments passed in FPRs are also passed in the argument list.

Argument Passing

To support varargs functions, calls to unprototyped functions, and compatibility with older linkages, the minimum argument area length must be 16 bytes. This allows the compiler to map the first three arguments in storage as well as registers and provides for compatibility with linkages that have a hidden last parameter.

Call Descriptor - Parameter Descriptions: If any floating point argument is passed in a register, the call requires a Call Descriptor, which is pointed to from the call site as described in "Calling Sequence" on page 131. Functions which receive a floating point parameter in a register require Interface Mapping Flags in their PPA1 control blocks as described in "PPA1 in support of XPLINK" on page 19; this takes the same format as the second half of the call descriptor used for calls to the same function. There is a 6-bit field in the call descriptor for each parameter passed in a floating point register.

Location	Content					
+0	Signed offset, in bytes, to entry point marker					
+4	Linkage	Return Value Adjust	FPR0	FPR2	FPR4	FPR6

Each of these parameter descriptor fields (FPRx) takes the following form:

Value	Meaning
001000	For the FPR0 field only, this indicates an unprototyped call. Floating point arguments are passed both in registers and in the argument area.
000000	floating point register is not part of the argument list
01....	This floating point register occupies 4 bytes in the argument list. It may be a single short floating point parameter or, if followed by 110000, the first half of a short floating point complex argument.
10....	This floating point register occupies 8 bytes in the argument list. It may be a single long floating point argument or, if followed by 110000 or 110001, the first 8 bytes of a longer floating point (including complex) type.
..nnnn	For the bit patterns above, the number of words (0 - 15) between the slot for this argument in the argument list and the slot for the previous (used) floating point register or, for FPR0, the beginning of the argument list.
110000	This floating point register occupies the same number of bytes as the previous register, immediately follows the slot associated with the previous register, and is the next part of a complex type.
110001	This floating point register occupies the same number of bytes (8) as the previous register, immediately follows the slot associated with the previous register, and is the second half of an extended precision floating point argument.

It is the compiler's responsibility to pass the maximum number of parameters that fit this encoding scheme so that the parameters in registers will match between caller and called function. Note, when calling a vararg routine, no argument in the variable portion of the argument is passed in a Floating Point Register. When calling unprototyped functions floating point parameters are passed in FPRs matching this encoding scheme and are also shadowed, by the caller, in GPRs or memory. Call descriptors are not required for calls to unprototyped functions whose return value is not examined by the caller.

Function Return Values: Functions return their values according to type:

1. Integral and pointer data types that are less than or equal to 32 (≤ 32) bits in length are widened to 32 bits and returned in GPR3.
2. Integral data types greater than 32 bits and less than or equal to 64 (≤ 64) bits in length are widened to 64 bits and returned in GPR2 (the leftmost 32 bits) and GPR3 (the rightmost).
3. Floating point types, including complex types, are returned FPR0, FPR2, FPR4 and FPR6, using as many registers as required.
Some languages do not support complex types. For the purposes of argument passing and function return values, in every language every aggregate that is (a) not a union, and (b) contains exactly two floating-point types of the same size (4, 8, or 16 bytes) is treated as a complex type.
4. Aggregates or packed decimal types 1-4 bytes in length are returned left adjusted in GPR1.
5. Aggregates or packed decimal types 5-8 bytes in length are returned left adjusted in GPRs 1 and 2.
6. Aggregates or packed decimal types 9-12 bytes in length are returned left adjusted in GPRs 1, 2, and 3.
7. Any other type is *always completely* returned in a buffer allocated by the caller. The address of this buffer is passed as a hidden first argument. For example struct {double,long double} is returned entirely in a buffer, with no part of the aggregate returned in registers.
8. Functions returning a return value and a reason code will pass the return value in GPR3 and the reason code in GPR2. In this case, both the return value and the reason code must be integral types that are less than or equal to 32 (≤ 32) bits in length; or, aggregates consisting of a single integral type that are less than or equal to 32 (≤ 32) in length.

Call Descriptor - Return Values: Calls to functions which return aggregates mapped to registers require a Call Descriptor, which is pointed to from the call site as described in "Calling Sequence" on page 131. Functions which return such aggregates require Interface Mapping Flags in their PPA1 control blocks as described in "PPA1 in support of XPLINK" on page 19; this takes the same format as the second half of the call descriptor used for calls to the same function. The call descriptor takes the following form:

Location	Content					
+0	Signed offset, in bytes, to entry point marker					
+4	Linkage	Return Value Adjust	FPR0	FPR2	FPR4	FPR6

The Return Value Adjust field takes the following form:

Value	Meaning
00000	Default return adjust. Function returns: <ol style="list-style-type: none"> 1. nothing, or 2. (Call Descriptor only) an integral or floating point type that is not examined by the caller. There is no need for compatibility code to copy the return value.
00	...
	001
	An Integral type ≤ 32 bits, returned in GPR3

Call Descriptor - Return Values

Value		Meaning
	010	An integral type >32 bits, returned in GPR2/GPR3
010	..	A floating point type
	00	A single precision floating point type (4 bytes) returned in FPR0
	01	A double precision floating point type (8 bytes) returned in FPR0
	10	An extended precision floating point type (16 bytes) returned in FPR0/2
011	..	A complex floating point type, including any aggregate containing exactly two floating point values of the same size that is not a union
	00	A single precision complex type (8 bytes) returned in FPR0/2
	01	A double precision complex type (16 bytes) returned in FPR0/2
	10	An extended precision complex type (32 bytes) returned in FPR0/2/4/6
1	An aggregate
	0000	An aggregate returned in an area provide by the caller
	<i>nnnn</i>	An aggregate of length <i>nnnn</i> (1-12) bytes, left adjusted in GPRs 1/2/3 as required

Call descriptor - linkage type: Calls using non-XPLINK parameter lists are indicated by the *linkage* field in a call descriptor. Possible values are:

- 0 XPLINK linkage
- 1 reserved
- 2 PL/I: arguments are passed by reference, the last (indicated by its high-order bit) being the address of a return buffer allocated by the caller.
- 3 Fortran
- 4 reserved
- 5 OS: arguments are passed by reference, the last having its high-order bit on. This value is used for COBOL calls if the return type is char, short, or int.
- 6 reserved
- 7 COBOL: arguments are passed by reference, the first being the address of a return buffer allocated by the caller and the last having its high-order bit on. This value is not used if the return type is char, short, or int.

Chapter 3. Program initialization and termination

Initialization and termination establishes the state of the components of the Language Environment program model supporting multi-language applications. Specifically, this section discusses the initialization and termination of a process, an enclave, and a thread.

Initialization overview

The program model describes three major constructs of a program structure. The constructs are:

Process

A collection of resources (code and data)

Enclave

A collection of program units consisting of at least one main and zero or more subroutines

Thread

The basic unit of execution and owner of a condition handler, a stack, and the machine state

Initialization provides services which support the construction of the entities described in this model. Brief descriptions of process, enclave, and thread initialization follow.

Process Initialization

Process initialization sets up the framework to manage enclaves and initializes those resources that can be shared among enclaves. It is during process initialization that the **anchor vector** is obtained and initialized. For more information, see Chapter 14, “Anchor support,” on page 475.

Enclave Initialization

Enclave initialization creates the framework to manage enclave-related resources and the threads that run within the enclave. For more information about enclaves, see *z/OS Language Environment Programming Guide*.

Thread Initialization

Thread initialization consists of the acquisition of a stack and the enablement of the condition manager for the thread.

Language Environment provides an interface under batch that establishes the three levels of the Language Environment program model. This interface is CEEINT. For the complete interface description of CEEINT, see “CEEINT interface” on page 153.

The first user routine to gain control within the enclave is designated as the **main** routine. If user parameters are passed from the host system/subsystem, the user parameters are made available to the main routine. By the time the main routine receives control, the following resources are available:

- Stack storage
- Heap storage
- Condition handling
- Message services
- Math library

Termination overview

The following section covers enclave and process termination.

Enclave termination

An enclave terminates when one of the following events occurs:

- The last thread in the enclave terminates.
- The **main** routine in the enclave returns to its caller. That is, an implicit STOP or return is done.
- An HLL construct issues a request for the termination of an enclave. For example:
 - C's `abort()`, `raise(SIGTERM)`, or `exit()` functions
 - COBOL's STOP RUN statement
 - COBOL's GOBACK statement in a main program
 - Fortran's STOP statement
 - Fortran's END or RETURN statements in a main program
 - Fortran's CALL SYSRCX, CALL EXIT, CALL DUMP, or CALL CDUMP statement
 - PL/I's STOP function
 - PL/I's EXIT function

When a severity 2 or greater condition remains unhandled at stack frame zero, the thread terminates. Because Language Environment supports only a single thread within an enclave, when the thread terminates due to an unhandled condition, the enclave also terminates.

To support the HLL constructs that terminate the enclave, such as STOP RUN, as well as an implicit STOP, two CWIs, CEETREC, and CEETREN:

- Save the Language Environment termination modifier, and the user's return code
- Raise the Termination Imminent due to Stop (T_I_S) condition (CEETREC only)
- Set the enclave condition token to zero
- Terminate all enclave level member exits and user exits
- Terminate the enclave

Details on how HLLs and Language Environment use the termination facilities appear later. When an enclave terminates, Language Environment releases resources allocated on behalf of the enclave and performs various other activities such as the following:

- Member-specific termination routines for those members that were active during the execution of the program are called.
- Language Environment exception handlers are canceled.
- All modules loaded by Language Environment are deleted.
- All storage obtained by way of Language Environment services is freed.
- The assembler user exit is called for enclave termination.
- All Language Environment control blocks for the enclave are freed.
- Return code and reason code are set in R15 and R0, respectively.
- The program mask and registers are restored to their values at the call to enclave initialization.
- Control is returned to the enclave creator.

In addition to the CWIs CEETREC and CEETREN, Language Environment provides a callable service that issues an abend. This service is a Language

Environment-specific callable service known as CEE3ABD. For more information, see *z/OS Language Environment Programming Guide*.

Process termination

Process termination occurs when the last enclave in the process terminates. Process termination dissolves the structure that kept track of the enclaves within the process and returns to the creator of the process. The PCB and associated resources are released. Language Environment explicitly relinquishes all resources that were obtained by Language Environment. Routines that obtain resources directly from the host system (such as opening a DCB) need to explicitly relinquish the resource because Language Environment does not have any knowledge of its acquisition.

Putting initialization/termination together

Presented here is an overview of running an application. Many details are omitted, but it demonstrates how all of the pieces fit together. For simplicity, compatibility is not described here. Also, the CICS initialization does not follow the steps provided below; for information on CICS, see Chapter 13, “Subsystem considerations,” on page 431.

- The operating system passes control to the application providing a save area, which we term the O/S Save Area.
- Regardless of which code receives control (compiler-generated code or runtime library), an STM into the O/S Save Area is performed preserving the operating system's registers.
- The application (probably an HLL library routine) calls CEEINT with R13 pointing to the O/S Save Area (and some other parameters as well).
- While running CEEINT, Language Environment determines the HLLs that are included in the application. For those HLLs present, a language-specific routine (known as an EVENT handler) is loaded and called once for process initialization, and once for enclave initialization. This allows for language-specific initialization activities to occur.
- Upon return from CEEINT, R13 points to the Dummy (or zeroth) DSA, R12 contains the address of the CAA, and R1 contains a pointer to any parameters or a pointer to a list of addresses that point to any parameters that are to be passed to the main routine.
- The HLL library routine allocates a DSA of its own and call the main routine.
- If the user code completes through a HLL construct such as STOP RUN, or if the main routine returns to its caller, the HLL library routine calls the Language Environment service CEETREC or CEETREN which terminate the enclave.
- The return code and reason code are set into R15 and R0 and returned.
- Control is returned through the save area that was passed to CEEINT during Language Environment initialization. That is, the registers are restored from the O/S Save Area, including R14. Then control is returned using R14. In this example, control is returned to the operating system.

Member interfaces for initialization

The following section covers enclave initialization. CEEINT is the Language Environment initialization routine that establishes a Language Environment environment (the process and the first enclave within the process) in which an application can run. The interface to CEEINT is described in “CEEINT interface” on page 153. CEEINT relies on a number of components to be link-edited with the

Initialization

application. Language Environment uses these components to describe the contents of the application, and to locate other elements contained in the application. A description of these components follows.

CEESTART

The CEESTART CSECT is a required part of each application; it identifies an application. The CEESTART CSECT must be accessible by Language Environment throughout the duration of the Language Environment environment. It cannot be link-edited with a module that is deleted during program execution. Language Environment produces a default version of CEESTART, but it can also be generated by the member languages. All member languages must have an external reference for CEESTART; this requirement is satisfied if a PPA2 is generated.

Language Environment provides a common CEESTART. Essentially, CEESTART can be nominated as the entry point for any other language that provides a CEEMAIN main or fetchable subroutines (and any other language that provides a CEEFMAIN). Entry into CEESTART causes the Language Environment environment to be initialized and execution to be passed to the main routine as specified in CEEMAIN. Entry into CEESTART causes control to be passed to a routine specified in CEEFMAIN given the Language Environment environment is already initialized, and CEEMAIN is not resolved.

CEESTART physical layout

CEESTART is logically divided into five sections. It is intended that the section structure and fields currently defined in CEESTART remain constant over time. It is also intended that necessary changes to CEESTART will be made in an upwardly compatible manner, so as to preserve the structure and fields as currently defined.

Two new formats of CEESTART are provided. One format supports non-XPLINK linkage protocols. The code sample below shows the format of the non-XPLINK CEESTART; its fields are described in Table 39 on page 141. The other format supports XPLINK linkage protocols; the XPLINK CEESTART format is shown here.

SECTION 1

CEESTART	CSECT		
CEESTART	AMODE	ANY	
CEESTART	RMODE	ANY	
	EXTRN	CEEBETBL	
	EXTRN	CEERootA	
		or	
	WXTRN	CEERootA	
	WXTRN	CEEMAIN	
	WXTRN	CEEFMAIN	Library copy

SECTION 2

000000		NOP	0
000004		NOP	2
000008		STM	14,12,12(13)
00000C		BALR	3,0
		USING	*,3
00000E		B	AROUND
	SIGNATUR	EQU	*
000012	SIG_LEN	DC	XL2(14)
000014	SIG_CEE	DC	X'CE'
000015	SIG_ID	DC	X'mm'
000016	SIG_VER	DC	X'vv'
000017	SIG_REL	DC	X'rr'
000018	SIG_PL	DC	A(PLIST)
00001C	SIGN_EYE	DC	CL8'CEESTART'
000024		DC	H'0'
	AROUND	EQU	*

SECTION 3

L R15,AROOT_A
BALR R0,R15

SECTION 4

	PLIST	DS	0F	
x+00	ACEEMAIN	DC	A(CEEMAIN)	or 0
x+04		DC	A(0)	Reserved
x+08		DC	A(0)	Reserved
x+0C		DC	A(0)	Reserved
x+10		CXD		
x+14	VMARKER	DC	H'-1'	
x+16	PLISTLEN	DC	AL2(PLIST_LEN)	
x+18		DC	A(0)	Reserved
x+1C		DC	A(0)	Reserved
x+20		DC	A(0)	Reserved
x+24		DC	A(0)	Reserved
x+28		DC	A(0)	Reserved
x+2C		DC	A(0)	Reserved
x+30		DC	A(0)	Reserved
x+34	SIG_ADDR	DC	A(SIGNATUR)	Reserved
x+38		DC	A(0)	or 0
x+3C	FMAIN	DC	A(CEEFMAIN)	Reserved
x+40		DC	A(0)	Reserved
x+44		DC	A(0)	Length of
x+48	BETBL	EQU	A(CEEBETBL)	parameter list
	PLIST_LEN		*-PLIST	

SECTION 5

AROOT_A DC A(CEEROOTA)
END CEESTART

Table 39. Contents of non-XPLINK CEESTART

Section	Content
Section 1	Declarations for the entry points and external routines.
Section 2	Additional entry points and signature. The signature is used for identification and provides access to the parameter list found in Section 4.
	<p>mm Member identifier of the creator. The HLL compilers should set this value to their corresponding member identifier.</p> <p>vv Member-defined version level; Language Environment has no dependencies on it.</p> <p>rr Member-defined release level; Language Environment has no dependencies on it.</p>
Section 3	Executable code that invokes the bootstrap routine CEEROOTA. Control is not returned to CEESTART once the bootstrap routine is invoked. It is intended that minimal logic is contained within CEESTART and that the structure and content of CEESTART remains constant over time.

Table 39. Contents of non-XPLINK CEESTART (continued)

Section	Content
Section 4	<p>Parameter list that is passed to the bootstrap routine. This parameter list is also intended to remain unchanged in future releases.</p> <p>ACEEMAIN Points to the CEEMAIN CSECT that contains the address of the main routine. This spot was used for the address of PLIMAIN in PLISTART.</p> <p>PRV_LEN Length of the pseudo register vector. This field is retained for compatibility. Language Environment does not allocate the PRV during initialization.</p> <p>VMARKER This is an identifying characteristic for the CEESTART PLIST.</p> <p>PLISTLEN Indicates the number of bytes contained within this PLIST.</p> <p>SIG_ADDR Points to the CEESTART signature contained in Section 2.</p> <p>FMAIN Points to the CEEFMAIN CSECT that is used during fetch or dynamic load.</p> <p>BETBL Points to the Language Environment owned externals table. It is through the externals table that Language Environment passes load module information into initialization.</p> <p>Language Environment does not interrogate unidentified fields; they are considered to be language-specific.</p>
Section 5	<p>Bootstrap routine addresses. This provides the routine address to the initialization bootstrap routine.</p> <p>AROOT_A Address of the bootstrap routine which corresponds to a CEESTART entry. The Language Environment library copy of CEESTART has a WXTRN to CEERootA and requires that CEERootA be INCLUDED during link-editing of the application. CEERootA can be excluded from applications where CEESTART is not the entry point.</p>

The code example below shows the format of the XPLINK CEESTART. Table 40 on page 143 describes the contents of each section.

SECTION 1

CEESTART	CSECT		
CEESTART	AMODE	ANY	
CEESTART	RMODE	ANY	
	EXTRN	CEEBETBL	Compiler
	EXTRN	CEERootD	Compiler
		or	
	WXTRN	CEERootA	Non-library copy
	WXTRN	CEERootD	Non-library copy
	WXTRN	CEEMAIN	Non-library copy
	WXTRN	CEEFMAIN	Non-library copy

SECTION 2

000000	NOP	0
000004	NOP	2

000008		STM	14,12,12(13)	
00000C		BALR	3,0	
		USING	*,3	
00000E		B	AROUND	
	SIGNATUR	EQU	*	
000012	SIG_LEN	DC	XL2(14)	
000014	SIG_CEE	DC	X'CE'	
000015	SIG_ID	DC	X'mm'	
000016	SIG_VER	DC	X'vv'	
000017	SIG_REL	DC	X'rr'	
000018	SIG_PL	DC	A(PLIST)	
00001C	SIGN_EYE	DC	CL8'CEESTART'	
000024		DC	H'0'	
	AROUND	EQU	*	
SECTION 3				
		L	R15,AROOT_D	Compiler
		BALR	R0,R15	Compiler
			or	
		L	15,AROOTA	Library Copy
		LTR	15,15	Library Copy
		BNZ	BALR	Library Copy
		ABEND	4093,REASON=112	Library Copy
		BALR	BALR 0,15	Library Copy
SECTION 4				
	PLIST	DS	0F	
x+00	ACEEMAIN	DC	A(CEEMAIN)	or 0
x+04		DC	A(0)	Reserved
x+08		DC	A(0)	Reserved
x+0C		DC	A(0)	Reserved
x+10		CXD		
x+14	VMARKER	DC	H'-2'	
x+16	PLISTLEN	DC	AL2(PLIST_LEN)	
x+18		DC	A(0)	Reserved
x+1C		DC	A(0)	Reserved
x+20		DC	A(0)	Reserved
x+24		DC	A(0)	Reserved
x+28		DC	A(0)	Reserved
x+2C		DC	A(0)	Reserved
x+30		DC	A(0)	Reserved
x+34	SIG_ADDR	DC	A(SIGNATUR)	
x+38		DC	A(0)	Reserved
x+3C	FMAIN	DC	A(CEEFMAIN)	or 0
x+40		DC	A(0)	Reserved
x+44		DC	A(0)	Reserved
x+48	BETBL	DC	A(CEEBETBL)	Length of
	PLIST_LEN	EQU	*-PLIST	parameter list
SECTION 5				
	AROOT_A	DC	A(CEERootA)	Compiler
			or	
	AROOT_D	DC	A(CEERootD)	Library Copy
		END	CEESTART	

Table 40. Contents of XPLINK CEESTART

Section	Contents
Section 1	Declarations for the entry points and external routines.

Table 40. Contents of XPLINK CEESTART (continued)

Section	Contents
Section 2	<p>Additional entry points and signature. The signature is used for identification and provides access to the parameter list found in Section 4.</p> <p>mm Member identifier of the creator. The HLL compilers should set this value to their corresponding member identifier.</p> <p>vv Member-defined version level; Language Environment has no dependencies on it. This contains a version level corresponding to the CEESTART defined by Language Environment or the compiler.</p> <p>rr Member-defined release level; Language Environment has no dependencies on it. This contains a release level corresponding to the CEESTART defined by Language Environment or the compiler.</p>
Section 3	<p>Executable code that invokes the bootstrap routine CEEROOTA. Control is not returned to CEESTART once the bootstrap routine is invoked. Minimal logic is contained within this section of CEESTART.</p>
Section 4	<p>Parameter list that is passed to the bootstrap routine. This parameter list is also intended to remain unchanged in future releases.</p> <p>ACEEMAIN This parameter list will typically not change. It is intended that any necessary changes will be made in an upwardly compatible manner, preserving the position and meaning of the current fields. This spot was used for the address of PLIMAIN in PLISTART.</p> <p>PRV_LEN Length of the pseudo register vector. This field is retained for compatibility. Language Environment does not allocate the PRV during initialization.</p> <p>VMARKER This is an identifying characteristic for the CEESTART PLIST.</p> <p>PLISTLEN Indicates the number of bytes contained within this PLIST.</p> <p>SIG_ADDR Points to the CEESTART signature contained in Section 2.</p> <p>FMAIN Points to the CEEFMAIN CSECT which is used during fetch or dynamic load.</p> <p>BETBL Points to the Language Environment owned externals table. It is through the externals table that Language Environment passes load module information into initialization.</p> <p>Language Environment does not interrogate unidentified fields; they are considered to be language-specific.</p>
Section 5	<p>Bootstrap routine addresses. This provides the routine address to the initialization bootstrap routine.</p> <p>AROOT_A Address of the bootstrap routine which corresponds to a CEESTART entry. The Language Environment library copy of CEESTART has a WXTRN to CEEROOTA and requires that CEEROOTA be INCLUDED during link-editing of the application. CEEROOTA can be excluded from applications where CEESTART is not the entry point.</p>

CEEFMAIN

CEEFMAIN, as shown in Figure 49, contains the address of fetchable routines that gain control from CEESTART if the Language Environment environment is already initialized and CEEMAIN is not resolved.

Address (fetchable procedure entry point)
0

Figure 49. Format of CEEFMAIN

+0	0x02	0x00	0x00	0x01
+4	A(fetchable entry point)			
+8	Q(environment), or -1 if no environment			

+0	0x03	0x00	0x00	0x01
+4	A(fetchable entry point)			
+8	A(environment)			

CEEMAIN

CEEMAIN has been extended; see Figure 50 for the format of the old CEEMAIN and the extensions. Bits 30 and 31 are used to differentiate between the two. The 'ctl' field will be one for the extended CEEMAIN.

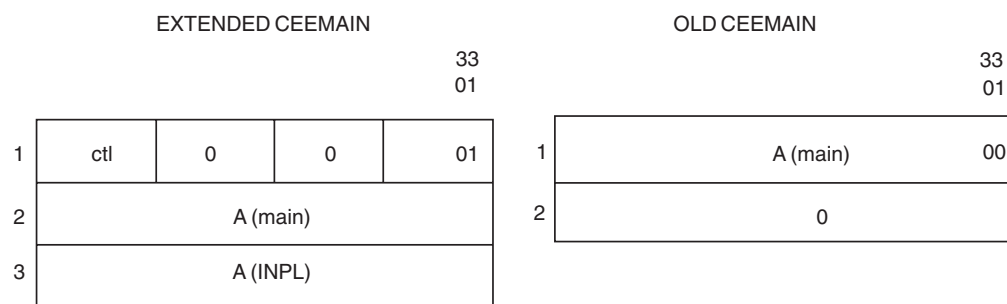


Figure 50. Format of CEEMAIN

+0	0x02	0x00	0x00	0x01
+4	A(main entry point)			
+8	A(EDCINPL)			
+12 0x0c	Q(environment), or -1 if no environment			

+0	0x03	0x00	0x00	0x01
+4	A(main entry point)			
+8	A(EDCINPL)			
+12 0x0c	A(environment)			

CEESTART operation

The Language Environment bootstrap routine takes the actions as described in Table 41.

Table 41. Bootstrap behavior

Enclave Initialized?	MAIN?	FMAIN?	Comments
No	Yes	No	Initialize the enclave and execute MAIN
No	Yes	Yes	Initialize the enclave and execute MAIN
No	No	Yes	Abend 4093-112
No	No	No	Abend 4093-112
Yes	Yes	No	Raise the condition CEE393
Yes	Yes	Yes	Raise the condition CEE396
Yes	No	Yes	Call the FMAIN subprogram
Yes	No	No	Raise the condition CEE392
Notes: <ol style="list-style-type: none"> 1. The enclave can either be the initial enclave or a nested enclave. Enclave Initialized is no if CEEINT has not yet been called for that enclave. 2. MAIN refers to the address of the main routine contained in the PLIMAIN or CEEMAIN CSECTs. 3. FMAIN refers to the address of the fetchable entry contained in the CEEFMAIN CSECT. 			

When CEESTART is invoked from within a Language Environment environment, and CEEMAIN or PLIMAIN is resolved, an error is raised. The bootstrap behavior should also be reflected in FETCH limitations.

Note: The address of the main routine can potentially be found in two places: CEEMAIN and within the Initialization Parameter List (INPL). Language Environment honors the address found in the INPL.

Main routine invocation event

When the environment is initialized by CEESTART, a new event allows the CEESTART owning member to invoke the main routine. The new event is provided for compatibility support. The interface to the Main Routine Invocation Event is shown in “Event code 14 — main routine invocation event” on page 500.

Language Environment allocates a DSA in order to call the main routine. The handler for event 14 must handle any AMODE switching required to invoke the MAIN routine.

After control returns from the main program and optional FINISH processing has completed, event 14 invokes CEETREN to terminate the enclave.

CEESIOP — set interrupt option service

The CEESIOP CWI is invoked to set the PL/I options INTERRUPT or NOINTERRUPT during enclave initialization.

Syntax

void (*CEECELVBSIOP) (on, [fc])

```
INT4      *on
FEED_BACK *fc;
```

CEECELVBSIOP

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)
L      R15,3384(,R15)
BALR   R14,R15
```

on (input)

Is not equal to 0 to set the INTERRUPT option.

fc (output/optional)

A feedback code to indicate the result of this call; possible values are:

Condition		
CEE000	Message	Success.
CEE3HV	Message	The region default for the runtime option <i>option</i> could not be overridden.
CEE3LO	Message	The system default for the runtime option <i>option</i> could not be overridden.

Usage Notes:

1. Unless the INTERRUPT option is marked nonoverrideable, this service can override the IBM-supplied, system-level or region-level default setting and it can be overridden by any other source of options.
2. The source of this option is marked programmer default in the Language Environment options report.
3. This routine only has affect in a single-language application. If this routine is called in a multiple-language application, it has no effect and CEE000 is returned.

Signature CSECT

Each language called by Language Environment for member-specific initialization and termination must generate a CEESG $_{mmn}$ signature CSECT. The signature CSECT denotes the presence of a member in the application. In addition, the signature CSECT provides a mechanism for the member to convey user load module information to the dynamically loaded member event handler. The mmn value is the decimal member number for each language.

In addition, the signature CSECT can contain a list of member identifiers upon which this current member is dependent. Language Environment orders these dependencies and calls the member-specific initializations in the dependent order. Termination is performed in the reverse order. Language Environment assumes that circular dependencies do not occur.

The format of the signature CSECT is shown in Figure 51 on page 148. The fields SG_MBR1 and SG_MBR2 are optional and provide a vehicle for the member to pass member-specific load module related information to the member-specific handler during initialization (or any other time). During enclave initialization, the signature CSECT can be accessed indirectly through the initialization parameter list. Language Environment does not interrogate, alter, or check for the presence of

Signature CSECT

SG_MBR1 or SG_MBR2. It is the member's responsibility to allocate SG_MBR1 and SG_MBR2, and to access these fields based upon their presence.

```

CEESGnnn CSECT
CEESGnnn AMODE ANY
CEESGnnn RMODE ANY
          DC CL4'Snnn'      Eye catcher
          DC H'20'         Length of CSECT
          DC H'1'         Version id
          DC H'0'         Number of dependent member IDs
          DC H'0'         Offset from the start of the CSECT...
*
SG_MBR1 DC A             Reserved for member's use
SG_MBR2 DC A             Reserved for member's use

```

Figure 51. Signature CSECT format

CEEBETBL — Language Environment externals table

The CEEBETBL CSECT, shown in Figure 52, is linked with any Language Environment-enabled application program. The CSECT is defined by the CEEBETBL module. The externals table contains various external references to entities in the executable program, which allows Language Environment to locate entities if they exist in the executable program. For compatibility with old load modules, a HLL can construct its own CEEBETBL.

```

CEEBETBL CSECT ,
CEEBETBL AMODE ANY
CEEBETBL RMODE ANY
          WXTRN CEEUOPT
          WXTRN CEEBXITA
          WXTRN IEWBLIT
ETBL_A_ENTRIES DC F'10'      Number of fullwords in this table
ETBL_A_CEEBXITA DC V(CEEBXITA) Assembler User Exit
ETBL_A_CEEBINT DC V(CEEBINT) HLL User Exit
ETBL_A_CEEBLLST DC V(CEEBLLST) Language List
ETBL_A_CEEUOPT DC V(CEEUOPT) User declared runtime option table
ETBL_A_CEEBTRM DC V(CEEBTRM) Termination stub routine address
ETBL_A_IBMXXITA DC A(0)      Holds address of PL/I or C user exit
ETBL_A_CEEBPUBT DC V(CEEBPUBT) Unique binding table
ETBL_A_I EWBLIT DC V(IEWBLIT) Loader information table (or 0)
          DC A(0)           Reserved
          END

```

Figure 52. CEEBETBL CSECT format

Table 42 describes the contents of each field in the CEEBETBL.

Table 42. CEEBETBL field descriptions

Field	Contents
ETBL_A_ENTRIES	Fullword number containing the number of fullwords in CEEBETBL, including this word
ETBL_A_CEEBXITA	Address of the assembler user exit (CEEBXITA) or zero. If zero, the installation-wide assembler user exit, which is linked with the Language Environment dynamically loaded routines, is called.
ETBL_A_CEEBINT	Address of the HLL User Exit (CEEBINT) or zero. If zero, the HLL user exit is not called.

Table 42. CEEETBL field descriptions (continued)

Field	Contents
ETBL_A_CEEBLLST	Address of the language list (CEEBLLST). This is a vector of weak external references for the signature CSECTs. When an entry in the vector is nonzero, the corresponding HLL is present in the executable program and its language-specific initialization is performed. (This is provided by Language Environment.)
ETBL_A_CEEUOPT	Address of the user declared option table or zero. If zero, user-defined runtime options are not available (for example, link-edited with the application).
ETBL_A_CEEBTRM	Address of the termination stub that releases the resources obtained in CEEINT. Essentially, the termination stub deletes the routine loaded by CEEINT and returns using R14 found in the save area provided on entry to CEEINT.
ETBL_A_IBMXXITA	Address of PL/I or C user exit.
ETBL_A_CEEBPUBT	Address of the product unique binding table, which contains the name for the first dynamically-loaded runtime library module.
ETBL_A_I EWBLIT	Address of the loader information table (IEWBLIT), which is created by the Binder for modules that were built without using the prelinker. For example, these modules would include reentrant C programs and all C++ programs. If the Binder does not need to create this table, this field will contain a zero (0).

Event handler routines

Each member in an enclave that provides a signature CSECT or appears in the dependent member list of a signature CSECT is also required to have an event handler routine. The load name of this routine must be CEEEV nnn , where nnn is the decimal member number. As an example, COBOL's event handler routine name is CEEEV005. This routine must be available to the Language Environment load service. All calls of the event handlers use an OS-style parameter list. R1 points to an address list which points to the specific parameters. The event handlers are always called in AMODE 31. The interface to the event handler routines for member-specific initialization and termination is described in "Member event codes for initialization and termination" on page 177. For each of the event handlers, see "Event handler calls" on page 481.

CEEELLST — language list

The member list is a vector of WXTRNs of the signature CSECTs and is generated by Language Environment. Language Environment checks for the presence of a member in the application in the language list. If the member represented by a specific offset in this list is not present or requires no special initialization, its WXTRN is unresolved. If the WXTRN is resolved or the member appears in the dependent member list of any signature CSECT in the language list, then Language Environment dynamically loads the event handler routine for that member, and stores the address in the member list. Language Environment then calls the event handler, passing an event code to the event handler routine.

The language list has zero through seventeen entries statically allocated in Language Environment. Language Environment uses the number of entries in the language list as a loop counter when it is necessary to loop through the language list entries. Refer to the LLISTENT as the number of valid entries within the language list. The format of the language list is shown in Figure 53 on page 150.

Language List

```
CEEELLST CSECT ,      LANGUAGE ENVIRONMENT LANGUAGE LIST HEADER
CEEELLST RMODE ANY
CEEELLST AMODE ANY
          DC   CL4'LLHD'
          DC   AL2(CEELLIST-CEEELLST)   Length of list header
          DC   AL2(1)                   Lang Env list version number
          DC   A((LLISTEND-CEELLIST)/4) Number of list entries
          DC   A(CEELLIST)              Pointer to the language list
CEEELLST DS   0D                      Lang Env language list
          WXTRN CEESG000
          DC   A(CEESG000)   00 RSVD
          WXTRN CEESG001
          DC   A(CEESG001)   01 Language Environment
          WXTRN CEESG002
          DC   A(CEESG002)   02 RSVD
          WXTRN CEESG003
          DC   A(CEESG003)   03 C/C++
          WXTRN CEESG004
          DC   A(CEESG004)   04 RSVD
          WXTRN CEESG005
          DC   A(CEESG005)   05 COBOL
          WXTRN CEESG006
          DC   A(CEESG006)   06 Debug Tool
          WXTRN CEESG007
          DC   A(CEESG007)   07 Fortran
          WXTRN CEESG008
          DC   A(CEESG008)   08 RSVD
          WXTRN CEESG009
          DC   A(CEESG009)   09 RSVD
          WXTRN CEESG010
          DC   A(CEESG010)   10 PL/I
          WXTRN CEESG011
          DC   A(CEESG011)   11 Enterprise PL/I for z/OS
          WXTRN CEESG012
          DC   A(CEESG012)   12 Berkeley Sockets
          WXTRN CEESG013
          DC   A(CEESG013)   13 RSVD
          WXTRN CEESG014
          DC   A(CEESG014)   14 RSVD
          WXTRN CEESG015
          DC   A(CEESG015)   15 assembler
          WXTRN CEESG016
          DC   A(CEESG016)   16 RSVD
          DC   A(0)          Dummy entry must contain X'00'
          DS   0D          This boundary requirement is mandatory.
*
*                          It is needed to save processing time when
*                          CEE is being initialized.
LLISTEND DC   A(0)          Mark the end of list
          END
```

Figure 53. CEEELLST format

Initialization parameter list

As Figure 54 on page 151 shows, the initialization parameter list is presented in two parts. The first part contains two items:

- The address of a fullword which contains the address of the entry point. For HLLs that do not have multiple entry points, the entry point is the address of the main routine.
- An offset from offset 0 of the first part of the initialization parameter list to the second part of the initialization parameter list. The offset is treated as a signed offset.

The second part of the initialization parameter list consists of the following information:

- Number of entries in this part, including this counter; this number is 6 or 7.
- Address of a fullword containing the address of the main entry point of the application. In COBOL, the fullword contains the primary entry point of the compile unit. This is provided in the user exit.
- Address of CEESTART, or zero.
- Address of the CEEBETBL CSECT.
- A fullword of the member identifier that created this instance of the initialization parameter list.
- A fullword that is used by the member identified by the above member ID.
- The main-opts word indicates attributes of the main program which is being initialized. The main-opts word is optional. If omitted, the number of entries is then 6. Also, if omitted, the information for the main-opts word is obtained by calling the event handler whose member identifier is in the INPL.

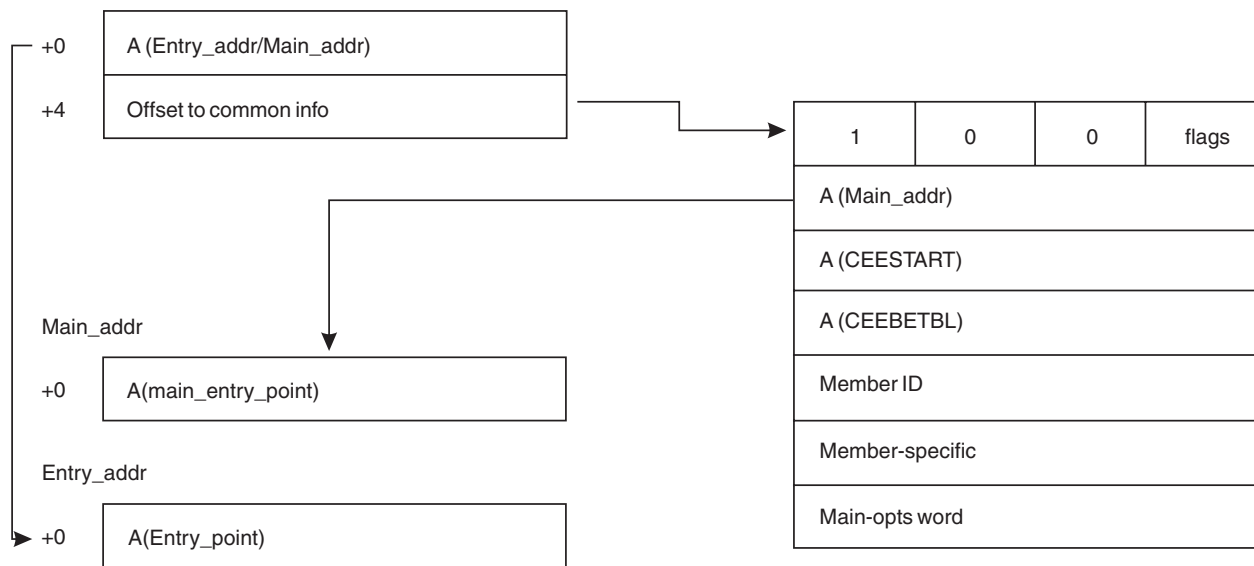


Figure 54. Format of the initialization parameter list

Updated INPL

Figure 55 on page 152 shows the updated format of the initialization parameter list (INPL).

Language List

Updated INPL				Old INPL					
1	1	0	0	flags	1	1	0	0	6 or 7
2	A (Main_addr)				2	A (Main_addr)			
3	A (CEESTART)				3	A (CEESTART)			
4	A (CEEBETBL)				4	A (CEEBETBL)			
5	Member ID				5	Member ID			
6	Member-specific				6	Member-specific			
7	Main-opts word				7	Main-opts word (optional)			

Figure 55. Updated format of the initialization parameter list

The Language Environment initialization parameter list is defined, as follows:

- The first word of the INPL is as follows:
 1. Byte 0 of the INPL contains a control level set to 0 or 1.
 2. Byte 3 of the INPL is dependent on the value of the control level.
 - For control level 0, byte 3 is the total number of words in the INPL; this value is either 6 or 7 (the seventh word of the INPL is optional for control level 0). An INPL that is marked control level 0 is identical in format to, and is compatible with, the Version 1 Release 1.1 level INPL.
 - For control level 1, byte 3 contains flags. The seventh word of the INPL is always present for control level 1. The flags are defined as follows:

reserved	B'xxxxxx..'	must be zero
mult_environ	B'.....x.'	0 - not enabled for multiple environments 1 - enabled for multiple environments
mainops_valid	B'.....x'	0 - not valid, call main-opts event 1 - valid, use word 7

The `mainops_valid` flag must contain a 0, which indicates that the main-opts event must be invoked, or a 1, which indicates the main-opts event must not be invoked. INPLs marked control level 1 are fixed length and contain the main-opts word field.

The `mult_environ` flag must contain a value of 0 if there must be only one Language Environment-enabled application in the Task Control Block (TCB). If it is set to 1, a complete new environment, including region, process, enclave, and thread, will be unconditionally created. The following applies:

- a. The TRAP and INTERRUPT runtime options are ignored. The application will always run as if TRAP(OFF) and INTERRUPT(OFF). The restriction is necessary because the ESTAE/ESTAX that gets control for errors is always the last one issued, but with multiple environments (which can be preempted and scheduled asynchronously) it is not possible to have the last error exit always match the currently executing environment.
- b. Anchor lookup must not be performed, either through explicit calls to CEEARLU or the documented assembler anchor lookup code sequence. For Language Environment that means it will not be able to create nested enclaves. These translate into the following application restrictions:

- 1) No COBOL programs can run in multiple environments.
 - 2) POSIX functions cannot be used in multiple environments.
 - 3) Debugger cannot be used in multiple environments.
 - 4) Nested enclaves cannot be created in multiple environments. The following are not legal:
 - SVC LINK from assembler
 - PL/I Fetchable main
 - C system() function
 - 5) Applications that rely on error handling semantics associated with TRAP(ON) will not be able to run in multiple environments.
 - 6) An application that is not enabled for multiple environments cannot be initialized while one or more applications enabled for multiple environments exist on the TCB.
- c. Library routine retention (LRR) cannot be used.
 - d. Preinitialization (PreInit) cannot be used.
 - e. The application cannot run under CICS or IMS.
- The second through sixth words of the INPL are unchanged from the INPL format in Language Environment Version 1 Release 1.1.
 - The seventh word of the INPL (the main-opts word) is always present in the INPL if the control level is 1 and is optionally present if the INPL control level is 0.
 - Byte 0 (Execops) of the seventh word is as follows:

prealloc	B'x.....'	0 Does not require preallocated storage event
		1 Requires preallocated storage event
thdappl	B'.x.....'	0 Does not require threading features to run
		1 Requires threading features to run
defoptreq	B'..x.....'	0 Does not require default options event
		1 Requires default options event
execops_off	B'...x....'	0 Execops
		1 Noexecops
reqcmdequ	B'....x...'	0 Does not require command line equivalent process
		1 Require command line equivalent process
invmaindir	B'.....x..'	0 Invoke main through event handler
		1 Invoke main directly
inheritop	B'.....x.'	0 Merge runtime options
		1 Inherit Run ops
propcond	B'.....x'	0 Ignore unhandled conditions
		1 Propagate conditions
 - Byte 1 of Word 7 is the PLIST (parameter list) style with the following values:

CEEINPL_PLIST_CMS	Fixed(8)	Constant(1)
CEEINPL_PLIST_HOST	Fixed(8)	Constant(2)
CEEINPL_PLIST_MVS	Fixed(8)	Constant(3)
CEEINPL_PLIST_TSO	Fixed(8)	Constant(4)
CEEINPL_PLIST_CICS	Fixed(8)	Constant(5)
CEEINPL_PLIST_IMS	Fixed(8)	Constant(6)
CEEINPL_PLIST_OS	Fixed(8)	Constant(7)
 - Byte 2 and 3 of Word 7 are reserved

CEEINT interface

CEEINT is link-edited with the user's load module. For fully Language Environment-enabled main programs without old object code no system service requests (such as GETMAINs and LOAD) can occur prior to calling the Language Environment enclave initialization routine.

CEEINT

Call this CWI interface as follows:

CEEINT Interface

L R15,=V(CEEVINT)
BALR R14,R15

Members should follow standard calling conventions by saving the registers in the application's caller's save area and then use the following register interface to call CEEINT:

- R0** Contents should be the same as when the application was called.
- R1** Contains the address of the application's parameter list. The parameter list can be a standard (for example, runtime options and user parameters) OS-style PLIST (on z/OS), a TSO CPPL, or a standard OS-style call interface.
- R2** Contains the address of the initialization parameter list. The initialization parameter list is designed so that it can be statically built by the compiler. (For example, COBOL could add the initialization parameter list in the constant area following the BRANCH at the start of the compilation unit.) Language Environment does not alter the contents of the initialization parameter list. If, for example, Language Environment needs to fold the runtime options to uppercase, this is performed in a Language Environment-obtained work area.

When the member event handler is called for enclave initialization, the initialization parameter list is passed to the event handler as an argument.

For compatibility support, the initialization parameter list can be dynamically constructed. To do so, storage must be obtained prior to Language Environment services being available. The initialization parameter list is shown in Figure 54 on page 151 and discussed in "Initialization parameter list" on page 150.

- R13** Contains the address of the main program's caller's save area, usually the operating system's save area. Note that during termination, this save area is used as the source of register contents and the return address when Language Environment has completed its termination processing.
- R14** Return address register.
- R15** Entry address register.

The registers upon return are:

- R0** Unknown.
- R1** Contains the address of the application's parameter list without the runtime options or the slash. This can contain the original R1 upon entry. In some cases, Language Environment constructs the application's parameter list.
- R2–R7** Language Environment work registers. These registers' contents are not preserved across the interface.
- R8–R11**
These registers' contents are preserved.
- R12** Contains the address of the CAA.
- R13** Contains the address of the **dummy** DSA for return codes of 0 or 8. The register remains unchanged for return code 4.
- R15** Contains the return code from CEEINT, which is:
 - 00** Successful initialization.

- 04 Environment already established, no implicit enclave created. R13 remains unchanged. All other registers are as shown above.
- 08 Environment already established, implicit enclave created. R13 points to the dummy DSA within the new enclave. All other registers are as shown above.

Initialization failures

If CEEINT cannot successfully initialize the environment, it abends with completion code 4093. The reason code associated with the abend 4093 indicates the cause of the failure. The reason codes are described in *z/OS Language Environment Runtime Messages*.

Usage Notes:

1. R13 points to the dummy DSA in the user stack for return codes 00 and 08. The Next Available Byte (NAB) of this DSA points to the beginning of the stack. For return codes 00 and 08 from CEEINT, a DSA can be allocated using the code sequence shown in Figure 33 on page 94. The application should not store register values into the dummy DSA. The user can store a forward chain into the dummy DSA, and can allocate another DSA using the stack allocation code found in Figure 33 on page 94.
2. The back chain of the dummy DSA points to the save area that was passed by the caller of initialization.

CEEBCRLM — cancel/release load module

This CWI is to be used by member languages before canceling or releasing a load module that had been previously added to an enclave.

Syntax

```
void CEEBCRLM (token, lang_list, [fc])
```

```
POINTER *token;
POINTER *lang_list;
FEED_BACK *fc;
```

CEEBCRLM

Call this CWI interface as follows:

```
L R15,CEECAACELV-CEECAA(,R12)
L R15,3984(,R15)
BALR R14,R15
```

token (input)

The token returned by the CEEPLD2 service when the module was loaded.

lang_list (input)

The pointer to the language list; it could be one of following:

1. The language list found in the load module and returned by the CEEBADDM CWI.
2. The language list found by a member and input to the CEEBMBR CWI.
3. Zero if the load module was not recognized by CEEBADDM or the language list was not saved from 1 or 2.

fc (output/optional)

The parameter into which the callable service feedback code is placed. The following conditions might result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE39K	Severity	1
	Msg_No	3380
	Message	The load module was not recognized Language Environment.
CEE38N	Severity	4
	Msg_No	3351
	Message	An event handler was unable to initialize properly.

Usage Notes:

- Language Environment recognizes the following entry point styles:
 - C/C++ for MVS/ESA-style PPA
 - C/370-style PPA
 - Language Environment routine entry layout (see “Routine layout” on page 6)
 - Language Environment-format CEESTART
 - Language Environment AWI stubs
- If *lang_list* is zero and the entry style is not recognized, all members that are currently active within the enclave will be called with the cancel/release load module event.
- If *lang_list* is zero and the entry style is recognized, or *lang_list* is provided, all members that are present in the load module will be called with the cancel/release load module event.
- CEEBCRLM should be called by the members before using the CEEPDEL2 service to delete the module. For more information about this service, see “CEEPDEL2 — enclave level delete service” on page 297.

CEEBSENM — set the enclave name

This CWI sets the name for the current enclave. The name is used in reports such as the options report and the dump output.

Syntax

```
void (*CEELIBVBSENM) (enclave_name, [fc])
```

```
VSTRING    *enclave_name;
FEED_BACK  *fc;
```

CEELIBVBSENM

A field in the Language Environment LIBVEC that points to the Set Enclave Name Routine (CEEBSENM). Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,3360(,R15)
BALR R14,R15
```

enclave_name (input)

The name by which the enclave is to be known.

fc (output/optional)

The parameter in to which the callable service feedback code is placed. The following conditions might result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE394	Severity	1
	Msg_No	3364
	Message	The enclave name was truncated by the enclave naming service during initialization.
	Explanation	The enclave naming service is used by the language in which the main program is written during enclave initialization. It was passed a name longer than 32 characters. This is an internal problem.
	Programmer Response	Contact your service representative.
	System Action	The truncated name is used as the enclave name.
CEE395	Severity	3
	Msg_No	3365
	Message	The enclave naming service was called, but not during enclave initialization, or not by a member corresponding to the main program.
	Explanation	The enclave naming service is used by the language in which the main program is written during enclave initialization. It was used in an illegal manner. This is an internal problem.
	Programmer Response	Contact your service representative.
	System Action	The requested service is not performed and the enclave name might not be correct.

Usage Notes:

1. This service can only be called during event handler processing of the Enclave Create event.
2. This service can only be called by the member corresponding to the language of the main routine, as specified in the initialization parameter list. A member can determine if it corresponds to the language of the main routine by checking the field in the initialization parameter list, which contains the member identifier of the creator of the initialization parameter list.
3. If the name is longer than 32 characters it is truncated to 32 characters and CEE394 is returned or signaled.
4. If this service is not used the name is taken from the main routine, if possible, or the members are polled for the name.

CEEBSRCM — set the enclave return code modifier

This routine is intended to provide a mechanism to update the return code modifier when errors are encountered during enclave termination. For example, in at least two cases, it is necessary to increase the enclave return code modifier to a specified value if the existing value is less than that value. One case, for example, is during termination when the Fortran library closes files that have not yet been closed. During this process, an error might occur but the remaining files still need

to be closed. On each occurrence of one of these errors, the return code modifier needs to be made at least as high as the severity of the condition.

This CWI will set the return code modifier for the enclave during enclave termination. The return code modifier will first be established by the enclave termination services (CEETREN or CEETREC) or set by condition handling when an unhandled condition causes termination of the enclave. Once established, the return code modifier can only be increased.

Syntax

void (*CEELIBVBSRCM) (rc_modifier, [fc])

```
INT4      *rc_modifier;
FEED_BACK *fc;
```

CEELIBVBSRCM

A field in the Language Environment LIBVEC that points to the Set Enclave Return Code Modifier Routine (CEEBSRCM). Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)
L      R15,3012(,R15)
BALR   R14,R15
```

rc_modifier (input)

The enclave return code modifier must in the range of 1 to 4 inclusive.

fc (output/optional)

Is an optional parameter in which the callable service feedback code will be placed. The following conditions may result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE38S	Severity	2
	Msg_No	3356
	Message	The rc_modifier must be in the range of 1 through 4. The return code modifier was not changed.
	Explanation	The rc_modifier was not in the range of 1 through 4. The return code modifier that was first established by the enclave termination services or by the condition handling was kept.
	Programmer Response	Provide a valid return code modifier.
	System Action	No system action is taken.

Condition		
CEE38T	Severity	2
	Msg_No	3357
	Message	The service was invoked outside of the member enclave termination; no action was taken.
	Explanation	CEEBSRCM is to be called during the member enclave termination; it was invoked outside of the member enclave termination.
	Programmer Response	Ensure that the routine is called during the enclave termination.
	System Action	No system action is taken.

CEEPGFD — get function pointer

The CEEPGFD CWI returns a function pointer to a function that resides in a separate load module. Functions that are called by function pointers that are created by CEEPGFD will have access to the writable static area, if it exists.

Syntax

```
void CEEPGFD (*load_addr, *func_pointer, [fc])
```

```
POINTER *load_addr;
POINTER *func_pointer;
FEED_BACK *fc;
```

CEEPGFD

Call this CWI interface as follows:

```
L R15,CEECAACELV-CEECAA(,R12)
L R15,3976(,R15)
BALR R14,R15
```

load_addr (**input/mandatory**)

is the address of an executable module.

func_pointer (**output**)

is a function pointer that can be used to call the function.

fc (**output/optional**)

specifies the optional feedback token where the CWI feedback code will be placed. If this argument is omitted and the CWI will return a feedback code other than **CEE000**, the CWI will “raise” this feedback code as an error condition. The following feedback tokens and associated severities may be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE39K	Severity	1
	Msg_No	3380
	Message	The load module was not recognized by Language Environment.

Condition		
CEE3NA	Severity	3
	Msg_No	3818
	Message	The event handler encountered an error.

Usage Notes:

1. After loading an executable module, CEE3ADDM (add members to the enclave) must be called prior to calling CEEPGFD to obtain a function pointer for the newly-loaded module. CEE3ADDM will augment the set of currently active members and notify members that a new load module has been introduced into the enclave.
2. When Language Environment returns the function pointer, the high order bit indicates the AMODE of the routine. You must provide the AMODE switching code when passing control to the function pointer.
3. Before deleting the load module containing the associated function, the CEEPRFD function must be called to release each function pointer obtained.
4. If the load module contains any ILC or the loading and loaded modules are written in different languages, the load module should not be deleted. The CEEFETCH and CEERELES assembler macros can be used to load and delete any ILC modules.
5. An AMODE 31 routine that is called using a pointer returned by CEEPGFD will have access to the writable static area, if it exists.
6. To use CEEPGFD to obtain a function pointer for a C function, the C function must either:
 - Be compiled with the pragma linkage(...,fetchable) directive, or
 - Have the function name specified as the entry point when the module is linked.
 In addition, a C++ routine must be compiled as extern "C".
7. CEEPGFD cannot be used to obtain a function pointer for a C main() routine.
8. If you use CEEPGFD to obtain a pointer for a C or C++ routine, calling the function pointer will give control to a glue routine. This routine will perform AMODE switching, if necessary, before calling the C/C++ routine.
9. If you use CEEPGFD to obtain a pointer for a C++ routine that is compiled as a DLL, the routine cannot export any variables or functions.

CEEPRFD — release function pointer

The CEEPRFD CWI will release a function pointer that was obtained by calling CEEPGFD. All function pointers must be released before deleting the load module which contains the associated function.

Syntax

```
void CEEPRFD (*load_addr, *func_pointer, [fc])
POINTER     *load_addr;
POINTER     *func_pointer;
FEED_BACK   *fc;
```

CEEPRFD

Call this CWI interface as follows:

```

L    R15,CEECAACELV-CEECAA(,R12)
L    R15,3980(,R15)
BALR R14,R15

```

load_addr (input)

is the load module address for which the function pointer was obtained.

func_pointer (input)

is a function pointer obtained by a call to CEEPGFD.

fc (output/optional)

specifies the optional feedback token where the CWI feedback code will be placed. If this argument is omitted and the CWI will return a feedback code other than **CEE000**, the CWI will “raise” this feedback code as an error condition. The following feedback tokens and associated severities may be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE39K	Severity	1
	Msg_No	3380
	Message	The load module was not recognized by Language Environment.
CEE39A	Severity	3
	Msg_No	3818
	Message	An event handler encountered an error.

CEE3ADDM — add new members to the enclave

This CWI interface dynamically augments the set of currently active members to an established environment. In addition, Language Environment notifies the members that a new load module was introduced into the enclave. This function is intended to be used when a new HLL is introduced into the currently executing mix of HLLs after a FETCH or dynamic call is performed.

Syntax

```
void CEE3ADDM (entry_point, lang_list, [fc])
```

```

POINTER    *entry_point;
POINTER    *lang_list;
FEED_BACK  *fc;

```

CEE3ADDM

Call this CWI interface as follows:

```

L    R15,CEECAACELV-CEECAA(,R12)
L    R15,2888(,R15)
BALR R14,R15

```

entry_point (input)

The entry point returned by a Language Environment load service.

lang_list (output)

The pointer to the language list found in the load module if the load module is recognized by Language Environment. If the load module is not recognized by Language Environment, a zero is returned.

fc (output/optional)

The parameter in to which the callable service feedback code is placed. The following conditions might result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE38M	Severity	4
	Msg_No	3350
	Message	A member event handler was not found.
CEE38N	Severity	4
	Msg_No	3351
	Message	An event handler was unable to initialize properly.
CEE38V	Severity	2
	Msg_No	3359
	Message	The module or language list is not supported in this environment.
CEE39K	Severity	1
	Msg_No	3380
	Message	The load module was not recognized Language Environment.

Usage Notes:

1. Language Environment recognizes the following *entry_point* styles; Language Environment does not recognize any other entry styles:
 - C/C++ for MVS/ESA-style PPA
 - C/370-style PPA
 - A Language Environment routine entry layout (see “Routine layout” on page 6)
 - Language Environment-format CEESTART
 - Language Environment callable service stubs
2. For Language Environment-recognized load modules, the following series of event handlers is called:
 - a. For those members that are contained within the newly-loaded load module and that have not yet been called for process initialization, Language Environment loads the member event handler and calls it for process initialization.
 - b. For those members that are contained within the newly-loaded load module and that have not yet been called for enclave initialization, Language Environment calls the member event handler for enclave initialization.
 - c. For those members that are contained within the newly-loaded load module, Language Environment calls the member event handler with the new load module event (see “Event code 8 — new load module event” on page 495).
 - d. For an application running with the POSIX(ON) runtime option or a PL/I tasking application, those members that are contained within the newly-loaded load module and that have not yet been called for POSIX

thread initialization, Language Environment calls the member event handler for POSIX thread initialization.

3. For load modules Language Environment does not recognize, the following member event handler is called:
 - For all members that are currently active within the enclave, Language Environment calls the member event handler with the new load module event passing a zero for the CEESTART parameter; see “Event code 8 — new load module event” on page 495.
4. When a member event handler is driven for the enclave initialization event due to the introduction of a new load module, Language Environment constructs the Initialization Parameter List (INPL) that is passed to the event handler. The INPL contains the following items:
 - a. The entry point and the main entry point contain the *entry_point* that was passed into CEE3ADDM.
 - b. The number of entries in the second half of the INPL is 7.
 - c. The address of CEESTART is the CEESTART found in the newly introduced load module.
 - d. The address of CEEBETBL points to a Language Environment constructed externals table.
 - e. The member identifier is that of Language Environment.
 - f. The main-opts word is zero.
5. The *lang_list* returned can be used to determine if the load module is a candidate for deletion using release() or a COBOL CANCEL statement. The *lang_list* is a read-only entity.
6. CEE3ADDM should be called by the member that issued the dynamic load.
7. Option processing does **not** occur.
8. No user exits are driven at this time.
9. The program mask is adjusted to accommodate the presence of new members within the environment. However, the program mask is not adjusted if the member appears only in the dependent member list of a signature CSECT in the language list.
10. No user code is called by Language Environment as a result of this call.
11. The dependency list is honored; see “Signature CSECT” on page 147 for details on the dependency list.

CEE3CRE — create enclave

The CEE3CRE CWI creates a new explicit enclave and initiates its execution. CEE3CRE can be called only from an already executing environment. The execution of the caller of CEE3CRE is suspended until the newly created enclave completes its execution and returns.

Syntax

```
void CEE3CRE (name, run_opts, inherit, user_arg, prop_cond, rtn_cd, rsn_cd, encl_fc, [fc]
)
VSTRING    *name;
VSTRING    *run_opts;
INT4       *inherit;
void       *user_arg;
INT4       *prop_cond;
```

```

INT4      *rtn_cd;
INT4      *rsn_cd;
FEED_BACK *encl_fc;
FEED_BACK *fc;

```

CEE3CRE

Call this CWI interface as follows:

```

L      R15,CEECAACELV-CEECAA(,R12)
L      R15,3356(,R15)
BALR   R14,R15

```

***name* (input)**

is a halfword prefixed character string containing the name of the Language Environment-enabled load module that is to start the enclave being created. The character string must be the platform-specific name identifying the load module. The name will be used as specified with no mapping by CEE3CRE.

***run_opt* (input)**

is a halfword prefixed character string containing the CEE runtime options and/or the user parm string applicable to the execution of the enclave. The format and interpretation of this string follows the same rules as the invocation parameter string. The CBLOPTS, EXECOPS, and PLIST options for the created enclave affect the interpretation of this parm string.

***inherit* (input)**

is a fullword integer which will determine if the explicitly created enclave inherits all the runtime options from its creating enclave.

0 the explicitly created enclave does not inherit its creating enclave's runtime options. Runtime options are established through the normal merge but with the *run_opts* argument taking the place of invocation options in the precedence order.

otherwise

the explicitly created enclave inherits its creating enclave's runtime options. When the value is specified, the input string in *run_opts* is ignored.

***user_arg* (input)**

the argument that will be passed to the first routine of the enclave.

- If this argument is non-zero it is the R1 value which is passed to the main routine. Any user parm string present in the *run_opts* argument will be ignored.
- If this argument is zero, the user argument will be taken from the user parm string, if present in the *run_opts* argument.

***prop_cond* (input)**

is a fullword integer which will determine if unhandled conditions or ABENDs that occur in the created enclave are propagated in or ignored by the creating enclave.

0 ignore the condition or ABEND in the creating enclave

otherwise

propagate conditions in the creating enclave

***rtn_cd* (output)**

is a full word integer with the return code from the created enclave. This is valid only when *fc* is CEE000.

***rsn_cd* (output)**

is a full word integer with the reason code from the created enclave. This is valid only when *fc* is CEE000.

encl_fc (output)

the feedback code produced by the execution of the enclave created by this call. This is valid only when *fc* is CEE000. When *encl_fc* is nonzero, it will be signaled in the CEE3CRE caller's enclave or be passed back based upon the value of *prop_cond*. The following feedback codes are possible:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE391	Severity	1
	Msg_No	3361
	Message	The created enclave, <i>name</i> , completed with an unhandled condition of severity two or greater.

fc (output/optional)

The feedback code from the service indicates how the service performed, and not the created enclave. The following feedback codes are possible:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3DC	Severity	3
	Msg_No	3500
	Message	Not enough storage available to load <i>name</i> .
CEE3DD	Severity	3
	Msg_No	3501
	Message	The <i>name</i> module not found.
CEE3DE	Severity	3
	Msg_No	3502
	Message	<i>name</i> module name too long.
CEE3EF	Severity	3
	Msg_No	3503
	Message	Load service request for module <i>name</i> was unsuccessful.

Table 43 lists the condition behavior for the different combinations of TRAP(ON/OFF) in the creating and created enclaves.

Table 43. Unhandled condition behavior summary

Behavior / Enclave Type	Creating TRAP(ON) Created TRAP(ON)	Creating TRAP(ON) Created TRAP(OFF)	Creating TRAP(OFF) Created TRAP(ON)	Creating TRAP(OFF) Created TRAP(OFF)
Propagate (<i>prop_cond</i> is not 0)				
Abend/Prog. check	Signal CEE391	Percolate original	Percolate U4094-40	Percolate original
Signal sev >= 2	Signal CEE391	Signal CEE391	Percolate U4094-40	Percolate U4094-40
Ignore (<i>prop_cond</i> is 0)				
Abend/Prog. check	Resume	Percolate original	Resume	Percolate original

Table 43. Unhandled condition behavior summary (continued)

Behavior / Enclave Type	Creating TRAP(ON) Created TRAP(ON)	Creating TRAP(ON) Created TRAP(OFF)	Creating TRAP(OFF) Created TRAP(ON)	Creating TRAP(OFF) Created TRAP(OFF)
Signal sev >= 2	Resume	Resume	Resume	Resume

Usage Notes:

1. The current thread waits for the created enclave to complete and for control to return to it.
2. There are two feedback codes in this service. One, *fc*, indicates how CEE3CRE behaved; the other, *encl_fc*, indicates how the created enclave executed. If the *fc* is zero, the result of the created enclave can be determined by the *encl_fc*.
3. The message file, specified by the runtime option MSGFILE, is shared across created enclaves if the MSGFILE name is the same.
4. When *inherit* parameter has value of zero, the CEEUOPT that is linked with the created enclave's load module is used during the option merge process.
5. The assembler user exit that is invoked for the created enclave is found in the created enclave load module (or the system default if not found in the created enclave load module).
6. If present, the HLL user exit that is invoked for initialization is found in the created enclave load module.
7. User parms will be available through the CEE3PRM, as follows:
 - If *user_arg* is zero, then the user parm string, if present in the *run_opts* argument, will be available.
 - If *user_arg* is non-zero, then no user parms will be available.
8. Debug Tool operation in a created enclave is documented by Debug Tool. Language Environment will not honor the TEST initial command string in a created enclave in which the debugger is already active.
9. This service is not supported in a CICS environment.
10. The PLIST option in the created enclave load module will be used to determine how the main routine anticipates the argument list. CEE3CRE normally makes register one contain either the address of *run_opts* or the value specified in *user_arg* and passes register one to the created enclave load module. After the explicit enclave is created and the main program gets control, register one may contain one of the following forms as the parameter list to the main program:
 - Value specified in *user_arg*.
 - Address of *run_opts* when NOEXECOPS is in effect.
 - Address of *run_opts* with runtime options removed when EXECOPS is in effect.

One exception is for the case of PLIST(TSO). One level of indirection to the parameter list is added. The rationale is that PLIST specified in the seventh word of the INPL does not effect the inbound character string for the created enclave in its process of runtime options and user arguments since the format has been defined in CEE3CRE. However, PLIST affects the format of the parameter list passed to the main program and one level of indirection should be added in for the case of PLIST(TSO).
11. The *name* field must contain a name that refers to a Language Environment-enabled target load module that starts with a main program. Target load modules that are not Language Environment-enabled are not supported. Unpredictable results will occur if a non-Language Environment-enabled module is the target of CEE3CRE.

CEE3CSYS — creating nested enclave

This CWI passes control to a target program. Control is passed in such a way that, if the target is a Language Environment-enabled application, the first call (if any) to CEEINT from the target program or its descendents results in the creation of a new nested enclave. CEE3CSYS can be called only from an already executing Language Environment environment. The execution of the caller of CEE3CSYS is suspended until the newly created enclave or non-Language Environment service, command, or EXEC completes its execution and returns.

Syntax

```
void (*CEECELVBCSYS) (name, user_arg, rsvd_word, rsvd_word, rtn_cd,[fc])
```

```
VSTRING *name;
void *user_arg;
void *rsvd_word;
void *rsvd_word;
INT4 *rtn_cd;
FEED_BACK *fc;
```

CEECELVBCSYS

Call this CWI interface as follows:

```
L R15,CEECAACELV-CEECAA(,R12)
L R15,2988(,R15)
BALR R14,R15
```

name (input)

a halfword-prefixed character string containing the name of the entry point in the target load module that is to receive control. The character string must be the platform-specific name identifying the entry point. The name is used as specified with no mapping by CEE3CSYS. The search order for the load module is consistent with that used for SVC LINK.

user_arg (input)

the equivalent of an R1 value. This can pass a single argument in the form of a halfword-prefixed character string that can contain user parameters and optionally runtime options.

rsvd_word (input)

A fullword reserved for future use.

rsvd_word (input)

A fullword reserved for future use.

rtn_cd (output)

A full word integer with the return code from the target enclave. This is valid only when *fc* is CEE000.

fc (output/optional)

The feedback code from the service indicates how the service performed, and not the target enclave. The following feedback codes are possible:

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE3DC	Severity	3
	Msg_No	3500
	Message	Not enough storage available to load <i>name</i> .

Condition		
CEE3DD	Severity	3
	Msg_No	3501
	Message	<i>name</i> module not found.
CEE3DE	Severity	3
	Msg_No	3502
	Message	<i>name</i> module name too long.
CEE3DF	Severity	3
	Msg_No	3503
	Message	Load service request for module <i>name</i> was unsuccessful.

Usage Notes:

1. The current thread waits for the new enclave or non-Language Environment service, command, or EXEC to complete and for control to return to it.
2. The unhandled condition behavior in the target enclave is always ignored.
3. The message file, specified by the runtime option MSGFILE, is shared across nested enclaves if the MSGFILE name is the same.
4. Runtime options are always obtained by normal merge in the target enclave. The *user_arg* string is used as command line equivalent. The availability of runtime options is subject to the EXECOPS setting of the target main program). The CEEUOPT that is linked with the nested enclave's load module is used during this option merge process.
5. If the assembler user exit that is invoked for the nested enclave creation is found in the target load module (user-supplied) it is used. Otherwise the system default user exit is used.
6. The HLL user exit that is invoked for nested enclave initialization is found in the target load module.

CEE3MBR — member bootstrap routine

This CWI interface dynamically augments the set of currently active members in an established environment. In addition, Language Environment notifies the members that a new load module was introduced into the enclave. This function is intended to be used on the callee side of a newly introduced load module. Specifically, it should be used when a HLL discovers the Language Environment environment established (using the anchor lookup) but the HLL-specific portion has not yet been initialized.

Syntax

void CEE3MBR (*lang_list*, *entry_point*, *inpl*, [*fc*])

```

POINTER    *lang_list;
POINTER    *entry_point;
POINTER    *inpl;
FEED_BACK  *fc;

```

CEE3MBR

Call this CWI interface as follows:

```

L    R15,CEECAACELV-CEECAA(,R12)
L    R15,2872(,R15)
BALR R14,R15

```

lang_list (input)

The language list found in the load module. If this needs to be manufactured by the caller, it should contain the entire header section. If the load module is already Language Environment-enabled, the language list can be obtained from the Language Environment externals table.

entry_point (input)

The entry point or entry point address for which CEE3MBR is called.

inpl or zero (input)

If the load module contains an INPL, it should be passed on this call. If the load module does not contain an INPL, pass a zero.

fc (output/optional)

The parameter in to which the callable service feedback code is placed. The following conditions might result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE38M	Severity	4
	Msg_No	3350
	Message	A member event handler was not found.
CEE38N	Severity	4
	Msg_No	3351
	Message	An event handler was unable to initialize properly.
CEE38V	Severity	2
	Msg_No	3359
	Message	The module or language list is not supported in this environment.
CEE39K	Severity	1
	Msg_No	3380
	Message	The load module was not recognized Language Environment.

Usage Notes:

- For the members that are contained within the member list that is passed to CEE3MBR the following events are called:
 - For those members that are contained within the newly-loaded load module and that have not yet been called for process initialization, Language Environment loads the member event handler and calls it for process initialization.
 - For those members that are contained within the newly-loaded load module and that have not yet been called for enclave initialization, Language Environment calls the member event handler for enclave initialization.
 - For an application running with the POSIX(ON) runtime option or a PL/I Tasking application, those members that are contained within the newly-loaded load module and that have not yet been called for POSIX thread initialization, Language Environment calls the member event handler for POSIX thread initialization.

- For those members that are contained within the newly-loaded load module, Language Environment calls the member event handler with the new load module event; see “Event code 8 — new load module event” on page 495.
2. Under CICS, a call to CEE3ADDM calls the new load module event and does not perform any ERTLI calls. The new load module event allows members to retain their current logic for both CICS and non-CICS paths. All other fields of PGMINFO1 and PGMINFO2 are zero.
 3. If nonzero, Language Environment uses *inpl* that is passed into CEE3MBR. If *inpl* is zero, Language Environment constructs an INPL for the event handlers using the *lang_list* passed. The INPL can be omitted only in the compatibility case.

When Language Environment constructs the INPL that is passed to the event handlers, the INPL contains the following items:

- The entry point and the main entry point contains the entry point that was passed into CEE3MBR.
 - The number of entries in the second half of the INPL is 7.
 - The address of CEESTART is the CEESTART found in the newly-introduced load module, if not found, zero.
 - The address of CEEBETBL points to Language Environment constructed externals table containing:
 - Fullword of 6
 - Zero (BAL user exit)
 - Zero (HLL user exit)
 - Address of the *lang_list* that was passed
 - Zero (CEEUOPT)
 - Zero (termination stub)
 - The member identifier is Language Environment's
 - The main-opts word is zero
4. If Language Environment recognizes the entry point as a program that object with deferred classes, the constructed externals table will have the following format:
 - Fullword of 9
 - Zero (BAL user exit)
 - Zero (HLL user exit)
 - Address of the *lang_list* that was passed.
 - Zero (CEEUOPT)
 - Zero (termination stub)
 - Zero (PL/I or C user exit)
 - Zero (unique binding table)
 - Address of loader information table from the executable program identified by the *entry_point* value.

CEE3SRSA — set return save area

This CWI interface dynamically sets the save area through which Language Environment returns on termination of the current enclave.

Syntax

```
void CEE3SRSA (rsa_address)
```

```
*void      *rsa_address;
```

CEE3SRSA

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,2904(,R15)
BALR R14,R15
```

rsa_address

The register save area address that is used for return after Language Environment termination. Registers are restored from *rsa_address* and control transferred using R14 which is contained within the *rsa_address*.

Usage Notes:

1. This service is provided explicitly for compatibility support and is not intended for general use.
2. This service allows members to identify the register save area that Language Environment uses as the target save area for its termination.
3. No verification is performed on the parameter.
4. This routine is has no effect when running under CICS and in a preinitialized environment.

CEE3DDBC — set dummy DSA back chain

This CWI interface dynamically sets the back chain of the dummy DSA.

Syntax

```
void CEE3DDBC (rsa_address)
*void      *rsa_address;
```

CEE3DDBC

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,2900(,R15)
BALR R14,R15
```

rsa_address

The register save area address that is stored in the back chain slot of the Dummy DSA.

Usage Notes:

1. This service is provided explicitly for compatibility support and is not intended for general use.
2. No verification is performed on the parameter.
3. This routine is has no effect when running under CICS and in a preinitialized environment.

CEE3PLST — PLIST manipulation

This CWI allows the member that requested initialization to specify the parameter list that is passed to the user application code.

Syntax

```
void CEE3PLST (R1_value, [fc])
POINTER  *R1_value;
FEED_BACK *fc;
```

CEE3PLST

Call this CWI interface as follows:

```

L    R15,CEECAACELV-CEECAA(,R12)
L    R15,2920(,R15)
BALR R14,R15

```

R1_value (input)

A fullword containing the value to be returned by CEEINT in R1 to the user's main routine.

fc (output/optional)

The feedback code passed by reference. The following conditions might result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE38P	Severity	3
	Msg_No	3353
	Message	The service was not called during event handler processing of the enclave create service or it was called by other than the member corresponding to the language of the main routine.
CEE38Q	Severity	3
	Msg_No	3354
	Message	The service was called in a CICS environment. The parameter list pointer is not modified.

Usage Notes:

1. This service can only be called during event handler processing of the enclave create event.
2. This service can only be called by the member corresponding to the language of the main routine, as specified in the initialization parameter list. A language can determine this by checking the field in the initialization parameter list containing the member identifier of the creator of initialization parameter list. This service does not modify the main parameter list pointer in a CICS environment.
3. This service does not modify the default parameter list pointer CEEEDBDEFPLPTR.
4. Use of this service allows members to repackage the parameter list during initialization.

CEEGIN — obtain the program's invocation name

This CWI returns the program name used to initiate this enclave.

Syntax

```
void CEEGIN (pname,[fc])
```

```
CHAR8    *pname;
FEED_BACK *fc;
```

CEEGIN

Call this CWI interface as follows:

```
L R15,CEECAACELV-CEECAA(,R12)
L R15,0120(,R15)
BALR R14,R15
```

pname (output)

An 8-character fixed length string, left-justified and right-padded, containing the name of the routine that called the enclave.

fc (output/optional)

The feedback code passed by reference. The following conditions might result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE39A	Severity	1
	Msg_No	3370
	Message	The program invocation name is unknown and the returned name is blank.

Usage Notes:

1. If the application is running under TSO and a CPPL is passed, the invocation name is obtained from the TSO command buffer, which is pointed to by the first word of the CPPL. (The CPPL is mapped by the IKJCPPL DSECT.)
2. If the application is running under CICS, the 4-character transaction name is obtained from the EIB.
3. If the application is running under TSO without a CPPL, or under z/OS, the invocation name is obtained from the Contents Directory Entry (CDE) that has the same address as the application in question.

CEERELU — RCB lookup

This routine sets R12 to the address of the RCB when the following conditions are met:

1. A Language Environment environment is not currently initialized.
2. Library routine retention has been successfully initialized and a Language Environment environment has been initialized and terminated at least one time with library routine retention in effect.

Syntax

Call CEERELU

Usage Notes:

1. Upon return, R12 contains the address of the RCB if the above conditions are met; otherwise, R12 contains a zero.
2. R14 and R15 are used as linkage registers. R0 is destroyed across the call. R13 is not used.
3. The routine must be entered in AMODE(31).
4. It must be link edited with its caller. It is not meant to be called from a HLL program.

Member interfaces for termination

The following section covers enclave termination and includes information on the CEETREC CWI, the CEETREN CWI, and the CEEATTRM CWI.

CEETREC — explicit termination through HLL constructs

CEETREC is a CWI that is intended for graceful enclave termination, supporting explicit termination through a HLL language construct such as a STOP statement or exit() function. The T_I_S condition notifies the stack frames on the current thread's stack of the intent to terminate the thread. The T_I_S condition is signaled using CEETREC.

Syntax

```
void CEETREC ([encl_modifier], [user_rtn_code])
```

```
INT4 *encl_modifier;
```

```
INT4 *user_rtn_code;
```

CEETREC

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)
```

```
L      R15,2880(,R15)
```

```
BALR  R14,R15
```

encl_modifier (input/optional)

The *encl_modifier* can be 0, 1, 2, or 3; if you do not specify one of these values or if you omit this parameter, a zero is assumed. Calculate the enclave reason code, sometimes called a return code modifier, by multiplying the *encl_modifier* by 1000. For more information, refer to *z/OS Language Environment Programming Guide*.

user_rtn_code (input/optional)

The user's specified return code for the enclave. If this is omitted, the return code is assumed to be zero.

Usage Notes:

1. Control does not return to the invoker of this service.
2. The actions taken by this service are:
 - a. Call CEESGL (T_I_S, <omitted>, frcode). Note that the CIB contains the enclave termination return code.
 - b. Quiesce all threads within the enclave.
 - c. Calculate the enclave composite return code from the parameters.
 - d. Set the termination condition token to zero.
 - e. All enclave level members and user exits are executed.
 - f. Return to the caller of the enclave with the appropriate return/reason code. Control is not returned to the caller of CEETREC.
3. If the *user_rtn_code* is omitted the enclave level user return code (CEEEDBURC) is used in the calculation of the enclave composite return code.
4. CEETREC cannot terminate the enclave if the resume cursor has been moved during the T_I_S processing.
5. Calculate the enclave return code by adding the *user_rtn_code* to the enclave reason code. For more information, see *z/OS Language Environment Programming Guide*.

6. The intended use of this service is to raise T_I_S and to provide graceful termination.
7. Control is given to the R14 value that is saved in the save area presented at enclave initialization.

CEETREN — terminate without raising T_I_S

CEETREN is a CWI that is intended for graceful enclave termination, supporting voluntary termination such as a return from main without raising T_I_S. The T_I_S condition notifies the stack frames on the current thread's stack of the intent to terminate the thread.

Syntax

```
void CEETREN ([encl_modifier], [user_rtn_code])
```

```
INT4 *encl_modifier;
INT4 *user_rtn_code;
```

CEETREN

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,2876(,R15)
BALR R14,R15
```

encl_modifier (input/optional)

The *encl_modifier* can be 0, 1, 2, or 3; if you do not specify one of these values or if you omit this parameter, a zero is assumed. Calculate the enclave reason code, sometimes called a return code modifier, by multiplying the *encl_modifier* by 1000. For more information, see *z/OS Language Environment Programming Guide*.

user_rtn_code (input/optional)

The user's specified return code for the enclave.

Usage Notes:

1. Control does not return to the caller of this service.
2. The actions taken by this service are:
 - a. Calculate the enclave composite return code from the parameters.
 - b. Set the termination condition token to zero.
 - c. All enclave level members and user exits are run.
 - d. Return to the caller of the enclave with the appropriate return/reason code. Control is not returned to the caller of CEETREN.
3. If the *user_rtn_code* is omitted, then the enclave-level user return code (CEEEDBURC) is used in the calculation of the enclave composite return code.
4. A GOBACK from a main COBOL program calls CEETREN.

Note the difference between the COBOL main program issuing a STOP RUN versus a GOBACK. The STOP RUN raises T_I, thus allowing the application to resume. A GOBACK from the main program terminates the enclave without the possibility of resumption.
5. The intended use of this service is to provide graceful termination without raising T_I_S.
6. Calculate the enclave return code by adding the *user_rtn_code* to the enclave reason code. For more information, see *z/OS Language Environment Programming Guide*.
7. Control is given to the R14 value that is saved in the save area presented at enclave initialization.

CEEATTRM — register event handler

CEEATTRM is a CWI that is used to register a member to gain control, through the member event handler, during termination. The member event handler gains control before any language termination activity is started.

Syntax

```
void CEEATTRM (member_id)
```

```
INT4 *member_id;
```

CEEATTRM

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L    R15,2912(,R15)
BALR R14,R15
```

member_id (input)

A fullword Language Environment member identifier number whose event handler is called with function code 15.

Termination sequence

For normal termination, the following steps occur:

1. CALL CEETREN or CEETREC to request termination.
2. Normal handling of the condition occurs without regard to the condition itself.
3. Those members that have been registered using the CEEATTRM service are called. The order of invocation of the members for the atterm event is unpredictable. When the atterm exit is driven, member termination has not yet started.
4. The debug tool is called for enclave termination, if the debug tool were already initialized.
5. Call the HLL specific enclave termination routines in the order defined using the signature CSECTs.
6. Call the assembler user exit.

Termination failures

If Language Environment cannot successfully terminate the enclave, it abends with completion code 4094. For example, this can occur when the program has overwritten Language Environment storage, causing Language Environment control blocks to become invalid. The reason code associated with the ABEND 4094 indicates the cause of the failure. The reason codes are described in *z/OS Language Environment Debugging Guide*.

Note: The reason and return codes are passed to the assembler user exit during termination processing. The exit can examine these values and change them. The return code and reason code returned by the user exit are used by Language Environment as the values returned in R15 and R0.

T_I_S condition

The T_I_S condition is defined as follows:

Condition		
CEE067	Severity	1
	Msg_No	0199

Signaling the T_I_S condition notifies the stack frames on the current thread's stack of the intent to terminate the thread. Notice that if the T_I_S condition is not handled, control returns to the next sequential instruction following the point where the Termination_Imminent condition was raised.

Member event codes for initialization and termination

Language Environment calls the event codes listed in Table 44 for preinitialization and for batch initialization. Language Environment calls member-specific initialization (MSI) routines for process initialization and again for enclave initialization. The resources and capabilities differ between the two events. For a description of the calling method, see “Language Environment member list and event handler” on page 83. (CICS initialization is discussed in Chapter 13, “Subsystem considerations,” on page 431.)

Table 44. Event codes called for initialization and termination

Event	Overview
Process Initialization Event	The process initialization event code is 17. This event is used to bring up HLL portions at the process level. The order in which the member event handlers are called is undefined. In particular, the dependency list is not honored. For a description of the parameters, see “Event code 17 — process initialization event” on page 502.
Process Termination Event	The process termination event code is 21. This event is used to terminate HLL portions at the process level. The order in which the member event handlers are called is undefined. In particular, the dependency list is not honored. For a description of the parameters, see “Event code 21 — process termination event” on page 508.
Enclave Initialization Event	The enclave initialization event code is 18. This event is used to initialize HLL portions at the enclave level. The order in which the member event handlers are driven is first based on the ascending order of the member identifier. However, if the member identifier is identified by a numerically lower ID in the dependencies part of the signature CSECT it could be called prior to a lower ID. For more information about the signature CSECTs, see “Signature CSECT” on page 147. For a description of the parameters, see “Event code 18 — enclave initialization event” on page 503.
Enclave Termination Event	The enclave termination event code is 19. This event is used to terminate HLL portions at the enclave level. The order in which the member event handlers are called is in the reverse order of initialization. The dependencies are determined from the signature CSECTs. For more information about the signature CSECTs, see “Signature CSECT” on page 147. For a description of the parameters, see “Event code 19 — enclave termination event” on page 506.
Runtime Options Event	The runtime options event code is number 4. This event has limited capabilities. There is no stack available, nor any Language Environment callable services. The purpose is to allow the members to handle runtime options in a compatible fashion. For a description of the parameters, see “Event code 4 — runtime options event” on page 486.

Member Event Codes

Table 44. Event codes called for initialization and termination (continued)

Event	Overview
Atterm Event	<p>The atterm event code is number 15. The atterm event is called during termination of an enclave. It is called after all user stack frames have been removed from the stack and prior to calling the members for the enclave termination event. Only the members that have been explicitly registered using the CWI CEEATTRM is called. For a description of the parameters, see “Event code 15 — atterm event” on page 501.</p> <p>Note: For information on Language Environment return codes, reason codes, existing language semantics, processing, and conventions, refer to <i>z/OS Language Environment Programming Guide</i>.</p>

Language Environment expects its registers to be restored to their original value upon return, conforming to normal calling conventions. The event handler must set the return code in R15 to one of the valid return codes (in decimal), as follows:

- 4 No action was taken for this event.
- 0 The termination event was successfully processed.
- 16 The event was not successfully processed and/or the program must be immediately terminated.

Language Environment abends the program if the event handler returns a value of 16 or a value not in the preceding list.

During initialization, Language Environment determines the members present in the application by interrogating the language list identifying those members present in the application and that require member-specific initialization. Each member found in the list has its event handler routine (CEEV mmm) loaded and called by Language Environment initialization in AMODE 31. The address of each event handler routine is stored into the Language Environment member list at the enclave level.

Language Environment expects a return in AMODE 31, and its registers to be restored to their original values using normal calling conventions. If an exception occurs during the execution of an MSI routine, the Language Environment exception manager issues an ABEND 4093 and the Language Environment environment terminates.

If there are multiple occurrences of a member within an enclave, its MSI routine is called only once per enclave. In addition, the order in which the MSI routines are called is determined from the list of member identifiers contained within the signature CSECTs.

If the MSIs need to be called in a specific order, it is indicated in the signature CSECT. For the format of the signature CSECT, see Figure 51 on page 148. Language Environment calls the MSI routines in the order dictated by the signature CSECTs. Termination is performed in the reverse order. If the signature CSECTs do not indicate any dependencies, the order of MSI invocation is undefined.

Language Environment abend summary

The normal paradigm for Language Environment (with TRAP ON) is to transform abends into signaled conditions, which if unhandled result in nonzero return and feedback codes.

In the rare case that Language Environment finds that its operation is severely compromised then it terminates the process with a 4xxx abend. The range of abends treated this way is 4000 to 4095. Termination is immediate (using SVC 13 or EXEC CICS ABEND, when in the CICS environment). Enclosing enclaves, if any, percolate the abend such that the whole process is taken down and no member or user termination exits at any level are driven.

When an implicit enclave created using system assisted linkage (such as LINK or CICS LINK) terminates with an unhandled condition, the system/subsystem abend function is called by Language Environment after normal cleanup.

The user (assembler) termination exit can transform the return, results and feedback codes in accordance with application needs. It can also request that an abend with a user code be issued by Language Environment on behalf of this enclave. This abend occurs as the last action in the enclave at the point where return would normally be passed back to the creator of the enclave. Because the current enclave has already canceled its STAE and SPIE exit, it does not get control, however the calling (if there is one) enclave's STAE does and thus its condition handler processes the exception in the normal way.

CEECOPP — runtime option compiler service

A callable service allows for compilers to convert runtime options strings specified in a source program to an options control block (OCB). This interface also supports the runtime options that are not part of the OCB, specifically ENV, PLIST, REDIR, EXECOPS, and ARGPARSE. These options are returned in the Supplementary Options Control Block (SOCB). The compiler would then create the OCB in the same format as the CEEUOPT CSECT file. This service is loadable and requires multiple calls, one to obtain the size of the working storage block (which includes the size of the OCB), and subsequent calls for the HLL to pass the runtime options string and the working storage and receive the parsed output.

CEECOPP is called by loading the executable named CEECOPP (using the LOAD SVC service) which resides in the SCEERUN data set. Then call the entry point returned from the load using:

Syntax

void CEECOPP (*function_code, storage_size, storage_addr, options, ocb_addr, socb_addr, roet_addr, ocb_status, socb_status, rc*)

```

INT4      *function_code;
INT4      *storage_size;
POINTER   *storage_addr;
PREFIXSTR *options;
POINTER   *ocb_addr;
POINTER   *socb_addr;
POINTER   *roet_addr;
POINTER   *ocb_status;
POINTER   *socb_status;
INT4      *rc;

```

***function_code* (input)**

Indicates the type of request. The valid function codes and meanings are:

- 1 Obtain the size of working storage. The first call is required to communicate to the caller how much storage is required by Language Environment to parse the options, the size of the resulting OCB, and the size of the error table. It is the caller's responsibility to acquire the storage and return the address to Language Environment in the second call.
- 2 Initialize OCB and parse the supplied options. The second call is used to initialize the OCB and to parse the options and save them in the OCB.
- 3 Parse the supplied options. Subsequent calls are used to parse the options save them in the OCB created by function code 2.

***storage_size* (output)**

The amount of storage required by Language Environment to do the parse. This size includes the amount of working storage needed to parse the string, the resulting OCB, and an error table. This is used in conjunction with *function_code* equal to 1.

***storage_addr* (input)**

The address of storage of the length returned by Language Environment in the first call. This is used with *function_code* 2 and 3.

***options* (input)**

A character string containing the runtime options. This is a halfword-prefixed length string. The string is not altered and can reside in read-only storage. This is used in conjunction with *function_code* 2 and 3.

***ocb_addr* (output)**

The address of the options control block that was created with the parsed options. The compiler should convert this block into a CEEUOPT CSECT. The storage used for the OCB is obtained from the storage provided by the caller. The length of the OCB is found directly within the OCB itself. The OCB is constructed so that there are no relocatable address constants and is essentially a stream of hex information. This is used with *function_code* 2 and 3. For an example of an options control block, see Appendix A, "Options control block and supplementary options control block," on page 811.

***socb_addr* (output)**

The address of a supplementary options control block (SOCB) that was created with the parsed options. The compiler should convert this block into a format that is suited to the caller. Language Environment does not retain this information. The storage used for the SOCB is obtained from the storage provided by the caller. The length of the SOCB is found directly within the SOCB itself. The SOCB is constructed so that there are no relocatable address constants and is essentially a stream of hex information. This is used in conjunction with *function_code* 2 and 3. For an example of a supplementary options control block, see Appendix A, "Options control block and supplementary options control block," on page 811.

***roet_addr* (output)**

The address of the runtime options error table created. The caller could convert this error table into error messages as part of the compiler output in its normal way of outputting errors. This is used in conjunction with *function_code* 2 and 3. The format of the runtime options error table is shown in Figure 56 on page 182.

ocb_status (output)

A fullword integer that contains the status of output OCB. If zero, no OCB entries were made. If nonzero, OCB entries have been made.

socb_status (output)

A fullword integer that contains the status of output SOCB. If zero, no SOCB entries were made. If nonzero, SOCB entries have been made.

rc (output)

A fullword integer that contains the return code. This is used in conjunction with both *function_code* 2s. The possible values are:

- 0 Options parsed with no errors, OCB entries made.
- 4 Invalid function code detected. No action performed.
- 8 Invalid function code sequence. Function code 3 (parse only) was received before function code 2 (initialize and parse).

Usage Notes:

1. In the OCB there are no address constants; therefore, no RLDs need to be created.
2. Options string length limitation is 64K bytes.
3. CEECOPP is reentrant and is marked AMODE(31)/RMODE(ANY). It is the caller's responsibility to insure the proper AMODE upon entry. CEECOPP does not switch AMODEs.
4. Invocation of CEECOPP is through BALR 14,15.
5. If the *OCB_status* parameter is zero, the compiler should not generate the CEEUOPT CSECT.
6. If the *roet_error_count* field in the ROET is not zero, errors occurred in the parse of the options string. The errors are contained in the table.
7. The *roet_error_code* field is in the format of a Language Environment condition token which is described in Figure 60 on page 227. The message numbers associated with the feedback codes that could be found in the runtime options error table are between CEE3601I and CEE3629I. For a description of these messages, see *z/OS Language Environment Debugging Guide*.
8. Figure 56 on page 182 shows the format of the runtime options error table.

Run-Time Options Error Table

0	Reserved
1	Error count
2	Error array entries · · ·
401	

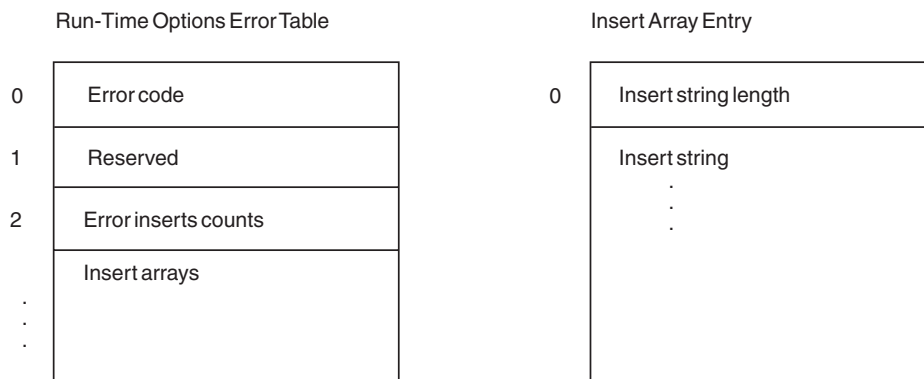


Figure 56. Runtime options error table

Options processing event

Event 4, which is used for compatibility options processing, has an update in its parameter list; an INPL is passed instead of the address of the main entry point. For a description of the parameters, see “Event code 4 — runtime options event” on page 486.

User exits

The assembler user exit for initialization is called when the enclave is initialized. This occurs during the CEEPIPI(init_sub) call and during the CEEPIPI(call_main) call.

The assembler user exit for termination is called when the enclave is terminated. This occurs at the end of a CEEPIPI(call_main) call, and at a CEEPIPI(term) for an environment for subroutines.

Similarly, the HLL user exit is called during the CEEPIPI(init_sub) invocation, and during the CEEPIPI(call_main) invocation.

CEEBSHL — exit from/re-entry to Language Environment shell

CEEBSHL is a CWI routine, which is called just before exit from, and just after re-entry to, Language Environment. This allows the Language Environment to be appropriately altered to accommodate the switch in states from a dormant environment to an active environment and vice versa.

Syntax

CEEBSHL (*function_code*)

CEEBSHL

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,3968(,R15)
BALR R14,R15
```

function_code (input)

Fullword integer set to one of the following values:

- 1 Application is exiting Language Environment and wants to put the Language Environment in a dormant state.
- 2 Application is re-entering a dormant Language Environment and wants to put the Language Environment in an active state.

Usage Notes:

1. The Language Environment ESTAE will percolate all abends when the environment is dormant.
2. The Language Environment ESTAE is enabled to handle abends and program interrupts when the environment is activated.
3. Calls to CEEINT while the environment is dormant will cause the environment to be activated.

Language Environment interface validation exit

The binder that is part of DFSMS/MVS Version 1 Release 3 allows a user-specified exit, called the *interface validation exit*, to examine and possibly modify names given as external references. This exit is given control immediately before the external references are bound to specific entry point names. It can invoke any of the language-specific interface validation exit routines it contains (see “Language-specific interface validation exit” on page 187). If the exit requests that an external reference be changed, the binder attempts to bind the reference to an entry point with the new name, using the binder's autocall facilities if necessary.

The language-specific interface validation exit routines (see “Language-specific interface validation exit” on page 187) that are invoked from the language-specific interface validation exit must conform to the conventions described in “Language-specific interface validation exit” on page 187 and must not use any additional features that the binder provides in general to an arbitrary interface validation exit. Here are two such features whose use is prohibited:

- The character string that the binder passes to any interface validation exit.
- The setting of the exit signature.

The name of an interface validation exit to be used is indicated to the binder by giving the name of the exit routine on an execution parameter to the binder.

Invocation of the exit is requested of the binder on a binder execution-time option as follows:

Invoking the Language Environment Interface Validation Exit

```
PARM='[... ] EXITS(INTFVAL(CEEPINTV))[... ]'
```

Interface validation exit

When an exit is specified in this manner, the named exit is invoked during binder execution. The exit is invoked once for each control section containing external references that have not been bound to entry names and validated. As the exit may be invoked quite often, it is important that the code in the exit be as efficient as possible so as not to degrade binder performance. Because of this performance consideration and the fact that contributions to the exit may be supplied by multiple languages, the exit is shipped as part of the Language Environment component of Language Environment. This exit is called the *Language Environment interface validation exit*.

Structure of the Language Environment interface validation exit

The Language Environment interface validation exit is a load module in SCEELKED with the name CEEPINTV. (This load module is shipped in SCEELKED rather than in SCEERUN because its actions apply to names that are in a specific level of SCEELKED rather than to anything in a specific level of SCEERUN. The renaming actions do not apply to execution of the application or to any changes that occur in SCEERUN from release to release of Language Environment.) It invokes one or more language-specific interface validation routines (see “Language-specific interface validation exit” on page 187).

The load module CEEPINTV contains these separately assembled routines:

CEEPINTV

This routine receives control from the binder, screens for possible renaming actions based on IDR information, and passes control to one of several language-specific interface validation exit routines for further analysis. In addition to some fixed code, CEEPINTV contains a series of CEEXVSEL macro instructions indicating that, when certain selection criteria are satisfied, a specific language-specific interface validation exit should be invoked to determine which external names, if any, should be renamed. The macro CEEXVSEL is described in “CEEXVSEL — high-level selection criteria.”

CEEfffX_n

These *language-specific interface validation exit* routines receive control from CEEPINTV based on the selection criteria indicated in a CEEXVSEL macro instruction. They determine whether an external reference should be renamed, and, if so, specify the new name. In the naming convention, *fff* is the component's three-character module prefix (for example, AFH for Fortran or IGZ for COBOL), and *n* is any digit. The conventions for coding a language-specific interface validation exit, along with the arguments passed to it, are described in “Language-specific interface validation exit” on page 187.

CEEfffXM

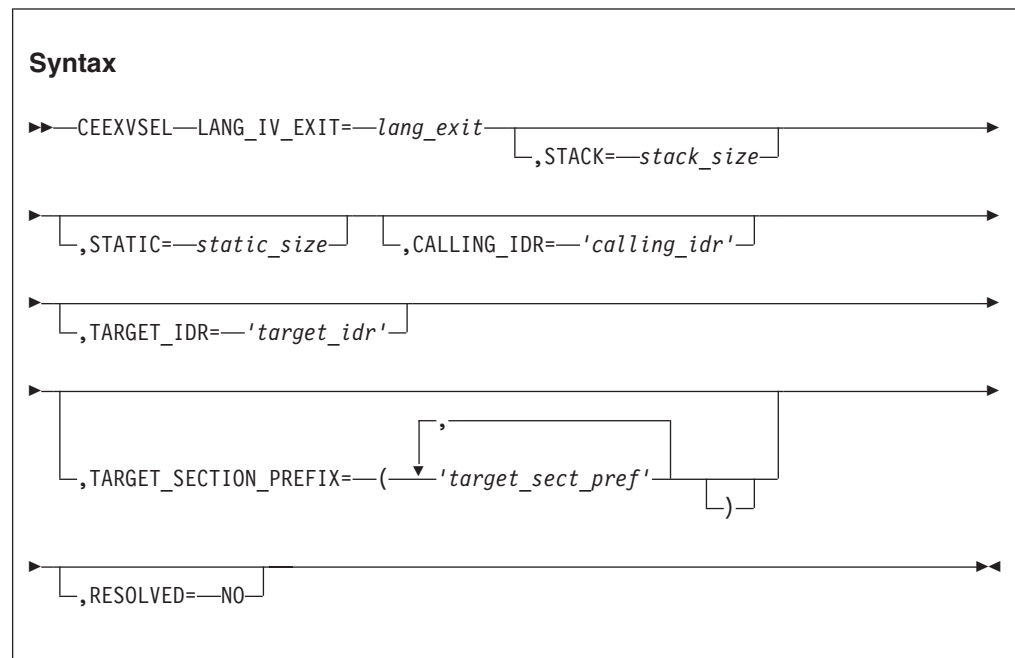
These language-specific interface validation message modules provide the text of messages, if any, that are to be printed by the binder in exceptional cases. For more information about message handling, see “Message handling” on page 191.

CEEXVSEL — high-level selection criteria

Each CEEXVSEL macro instruction in the routine CEEPINTV provides selection criteria that control whether a certain language-specific interface validation exit is to be invoked. The selection criteria include:

- IDR information of the control section whose external references are being examined,
- IDR information of the target control sections; that is, the control sections that contain the entry points that are about to be bound with the external references,
- Prefix of the names of the target control sections, and
- Any unresolved external references.

When the Language Environment interface validation exit finds that a set of selection criteria is satisfied, it passes control to a designated language-specific interface validation exit. This exit then determines if any of the external references must be renamed.



lang_exit

the name of the language-specific interface validation exit routine that is to be invoked if the criteria specified by the CALLING_IDR, TARGET_IDR, and TARGET_SECTION_PREFIX are satisfied. The conventions for coding this exit, along with the arguments passed to it, are described in “Language-specific interface validation exit” on page 187.

stack_size

the maximum total amount of stack storage needed for the DSA or DSAs of *lang_exit* and any routines that it calls. If the STACK parameter is omitted, the maximum length provided is 400 bytes. No stack extensions are possible.

static_size

the length of the storage area required for communication among successive invocations of *lang_exit*. A storage area of this length is provided on the first invocation and the same area passed on each successful invocation, as an argument on each call to *lang_exit*. On the first call, the first eight bytes of the area are cleared to 0 so that *lang_exit* can determine if it has previously stored any of its data in the area.

If the STATIC parameter is omitted or is coded with a value of 0, then no such storage area is made available to *lang_exit*.

Interface validation exit

calling_idr

Is a string of characters that is compared with the IDR information associated with the control section whose external references are to be examined by the interface validation exit. The selection criterion associated with the CALLING_IDR parameter is satisfied if *calling_idr* matches the control section's IDR information.

The value of *calling_idr* is compared only with the leading characters of the IDR information associated with the control section. This allows, for example, a selection based on the first few characters of the IDR information which contain the program number of a compiler without regard to any additional characters that indicate the version and release of the compiler.

If the *calling_idr* is omitted, the IDR information for the control section containing the external references is not used to eliminate the control section for further analysis of its external references. In this case, at least one of the other selection parameters is required.

target_idr

Is a string of characters that is compared with the IDR information associated with the control sections that contain the entry points in the reference list. (These are the control sections that contain the entry points that the binder is about to bind with the external references that are to be validated.) The selection criterion associated with the TARGET_IDR parameter is satisfied if there is at least one entry point whose control section has IDR information that matches *target_idr*.

The value of *target_idr* is compared only with the leading characters of the IDR information associated with the control section. This allows, for example, a selection based on the first few characters of the IDR information which contain the program number of a compiler without regard to any additional characters that indicate the version and release of the compiler. The *target_sect_pref* values are compared only with the leading characters of the section names in the reference list.

If the *target_idr* is omitted, the IDR information for control sections containing entry points in the reference list is not used to suppress further analysis of external references. In this case, at least one of the other selection parameters is required.

If the *target_idr* parameter is used, then the RESOLVED=NO parameter must not be used in the same macro invocation.

target_sect_pref

Is a string of characters that is compared with the names of the control sections that contain the entry points in the reference list. (These are the control sections that contain the entry points that the binder is about to bind with the external references that are to be validated.) The selection criterion associated with the TARGET_SECTION_PREFIX parameter is satisfied if the reference list has at least one entry point in a control section whose name begins with one of the *target_sect_pref* values.

If the TARGET_SECTION_PREFIX parameter is omitted, then the names of control sections containing the entry points in the reference list are not used to suppress further analysis of external references. In this case, at least one of the other selection parameters is required.

If the TARGET_SECTION_PREFIX parameter is used, then the RESOLVED=NO parameter must not be used in the same macro invocation.

RESOLVED=NO

This selection criterion is satisfied if there is at least one unresolved reference. If you use the RESOLVED=NO parameter, you cannot specify the TARGET_IDR or the TARGET_SECTION_PREFIX parameters in the same macro invocation.

Selection criterion parameters

The following parameters are called *selection criterion parameters*; at least one is required on each CEEXVSEL macro instruction to control if the language-specific interface validation exit, *lang_exit*, is to be entered.

- CALLING_IDR
- TARGET_IDR
- TARGET_SECTION_PREFIX
- RESOLVED=NO

If more than one of the selection criterion parameters is given on a single CEEXVSEL macro instruction, all of the specified criteria must be satisfied for the exit to be invoked.

The order of the CEEXVSEL macro instructions in the routine CEEPINTV controls the order in which the selection criteria are evaluated. Once the selection criteria on any CEEXVSEL macro instruction are satisfied, the corresponding language-specific interface validation exit is invoked, and the selection criteria on subsequent CEEXVSEL macro instructions, if any, are not evaluated.

Language-specific interface validation exit

A language-specific interface validation exit is a routine that is entered if the selection criteria specified by a CEEXVSEL macro invocation are satisfied. Its name should follow the naming conventions for exits described in 184. It should be written as a Language Environment-enabled non-XPLINK assembler language program, which conforms to the restrictions that follow.

Restrictions on the use of Language Environment services

Only a limited Language Environment-style environment is established for use by the language-specific interface validation exit(s). This means that there are no initialization services, no condition handling services, no message services, no program management services, and no heap services. The mini-environment that is made available is what is provided through the system programming C facility. (It is a “Persistent C Environment”, as described in *z/OS XL C/C++ Programming Guide*.) There is enough of an environment to provide a stack for use by a language-specific interface validation exit so long as no library services are used.

If the language-specific interface validation exit is written in assembler, it must be reentrant. Should any persistent data need to be kept between successive invocations of the same exit, the communication area must be used; the length of this area is specified in the STATIC parameter of the CEEXVSEL macro instruction, and its address is available through the anchor word argument that is passed to the exit (see “Arguments passed to language-specific interface validation exits”).

Arguments passed to language-specific interface validation exits

The following nine arguments are passed to each language-specific interface validation exit. These arguments are similar to, but not identical to, those passed by the binder to an interface validation exit (see “Language Environment interface validation exit” on page 183). Standard linkage conventions are followed, that is, register 1 contains the address of a list of addresses of these arguments:

Interface validation exit

1. Function Code. A 1-byte function code with value of 'V' indicating the Validate function. This differs from what the binder provides because the language-specific interface validation exit is entered only for the Validate function and not for the Start and End functions.
2. Anchor Word. A 4-byte pointer to a structure consisting of the following two fields:
 - a. A 4-byte pointer to a communication area of the length given by the STATIC parameter of the CEEXVSEL macro instruction. The same area is provided for each call to the same language-specific interface validation exit. On the first call, the first eight bytes of the area are cleared to 0 so that language-specific interface validation exit can determine whether it has previously stored any of its data in the area.
 - b. A 4-byte pointer to the first applicable reference list entry (see "Structure of the Language Environment interface validation exit" on page 184 for information about the reference list).

If there is neither a TARGET_IDR nor a TARGET_SECTION_PREFIX parameter on the CEEXVSEL macro instruction, this is a pointer to the first list entry for the section being validated and has the same value as the "Reference List" parameter.

If there is one or both of the TARGET_IDR and a TARGET_SECTION_PREFIX parameters, this is a pointer to the first list entry that satisfies these selection criteria. Note, however, that subsequent list entries may not satisfy these selection criteria.
3. This parameter has no meaning in this context.
4. Section name. A varying string containing the name of the section being validated.
5. Section vaddr. A 4-byte pointer to the beginning of the first text element in the section being validated. This may not be useful in a multi-class module, since there is no designated "primary" class. This field is reserved for future use.
6. Section IDRL. A 4-byte pointer to the IDR entry for the section in process. The IDR data entry consists of a halfword length field containing the length of the data followed by the data.
7. Reference List. A 4-byte pointer to a list of unchecked references. For a discussion of the reference list, see "Reference list" on page 189.
8. Return code. A fullword return code in which the overall status of the exit is to be returned. It will be initialized to zero on invocation of the exit. The following values may be set by the language-specific interface validation exit:

0	OK, no further processing required of this section. The action code for all references is zero.
4	Further processing required by the binder, as indicated in the returned action codes.
12	Severe error. Make no more calls to the exit and do not save the module (unless the binder option LET=12).
16	Terminate binder processing immediately.
9. Returned message. A 4-byte pointer in which may be returned the address of a halfword length field and a string allocated by the language-specific interface validation exit, and containing a message to be printed by the binder. The returned message must not be longer than 1000 bytes. The binder will prefix the returned message with its own message number. The returned message will be initialized to the null string so that the exit routine need not take any action

unless a message is to be issued. For more information about message handling, see “Message handling” on page 191.

Reference list

The seventh argument passed to the language-specific interface validation exit is the reference list. The reference list is a linked list containing one entry for each unchecked ER in the section. References marked NOCALL or NEVERCALL will not be included in the list. The last entry in the list contains zero in the link field. A reference entry is 64 bytes in length. The reference entry fields are shown in Table 45.

Table 45. Interface validation exit reference entry fields

Offset	Type	Len	Name	Description
(0)	Address	4	REFL_NEXT	Address of next list entry ³
(4)	Address	4	REFL_T_SYMBOL	Address of referenced symbol ²
(8)	Address	4	REFL_T_SECTION	Address of target section name ^{2, 8}
(C)	Address	4	REFL_T_ELEMENT	Address of target element ^{1, 8}
(10)	Address	4	REFL_T_DESCR	Address of target descriptors ^{1, 8}
(14)	Address	4	REFL_T_IDR	Address of target IDR ^{1, 5, 8}
(18)	Bit	4	REFL_T_ENVIR	Target environment ^{1, 6, 8}
(1C)	Character	8	REFL_T_SIGN	Target signature ⁸
(24)	Address	4	REFL_T_ADCONS	Adcon list anchor ¹
(28)	Address	4	REFL_C_DESCR	Address of caller descriptors ¹
(2C)	Bit	4	REFL_C_ENVIR	Caller's environment ^{1, 6}
(30)	Character	8	REFL_X_SIGN	Exit signature ^{4, 9}
(38)	Address	4	REFL_X_SYMBOL	New symbol (Char(*) varying) ^{4, 7, 9}
(3C)	Unsigned	2	REFL_X_ACTION	Action code ^{4, 9, 10}
(3E)	Unsigned	2		Reserved ¹

Notes for Interface Validation Exit Reference Entry Fields

- Field must be zero.
- Address points to varying character string, which must begin with a halfword length field containing current length, excluding length field.
- Last entry in list set to zero.
- Output field; set by exit routine.
- IDR data is returned in the following format; this 21-byte structure is preceded by a halfword length. The length may contain zero or any multiple of 21, allowing for multiple IDRs.

0	CHAR	10	Processor Identification
10	CHAR	2	Processor Version
12	CHAR	2	Processor Modification Level
14	CHAR	7	Date Compiled or Assembled (yyyyddd)
- Environmental bit settings are not yet defined.
- The exit routine must allocate and initialize a varying length character string, consisting of a halfword length field, containing the length of the symbol, immediately followed by the symbol itself. The address of this varying string must be stored in the REFL_X_SYMBOL field in the reference list.

Interface validation exit

8. Target fields will contain binary zeros for unresolved references.
9. Output fields will be initialized to binary zeros on invocation of the exit routine.
10. One of the following action codes should be returned for each reference entry:
 - 0 No special processing, such as changing the bind status flags, renaming the reference, or storing signatures required for this reference.
 - 1 Validation successful. Store the exit signature in both LD and ER records.
 - 2 Validation successful. Store glue code address in all referring adcons and store the exit signature in both LD and ER records.
 - 3 Accept unresolved reference. Do not store the exit signature in the ER record. Reference will be treated as a weak reference and will not affect the return code from the binder.
 - 4 Retry. New symbol has been provided for reference. Do not store signatures at this time. Reprocess autocall, if necessary, and re-validate.
 - 5 Validation failed. Mark reference unresolved and do not store signatures at this time. The return code from the binder will reflect that there was at least one unresolved reference.

Changing an external reference

The language-specific interface validation exit must decide whether a given external reference is to be changed, and if it is, to provide the new name. The routine should assume that the selection criterion specified by the `CALLING_IDR` parameter, if any, on the applicable `CEEXVSEL` macro instruction is already satisfied and does not need to be examined again. However, if one or both of the `TARGET_IDR` and `TARGET_SECTION_PREFIX` parameters are given, the applicable selection criteria are satisfied for the identified reference list entry but may not be for subsequent entries.

The language-specific interface validation exit can decide whether to rename an external reference based on one or more of the following pieces of information. Note that the first two are constant for each binder entry to the Language Environment interface validation exit and each of its calls to the language-specific interface validation exits. The others apply to the individual external references as reflected in the reference list entries. For more information about the reference list, see "Reference list" on page 189.

1. The name of the control section containing the external references.
2. The IDR information associated with the section that contains the external references. (This indicates the compiler, including its release, that produced the compiled code.)
3. The name of an external reference.
4. The name of a target control section, that is, the control section that contains the entry point with which the external reference is about to be bound. (In the event that the external reference is as yet unresolved, the address of the section name (`REFL_T_SECTION`) has a value of 0.)
5. The IDR information associated with a section that contains the entry point with which the external reference is about to be bound.

The language-specific interface validation exit can rename an external reference by:

1. Placing the address of the new name to be used as the external reference in the REFL_X_SYMBOL field in the reference list entry
2. Setting a value of 4 as the action code (REFL_X_ACTION field) in the reference list entry
3. Setting a value of 4 in the return code parameter that was provided to the exit.

When an external reference is renamed in this manner, the binder then uses autocall, if necessary, to locate the new name.

The language-specific interface validation exit can cause an external reference to become unresolved by:

1. Setting a value of 3 or 5 as the action code (REFL_X_ACTION field) in the reference list entry;
2. Setting a value of 4 in the return code parameter that was provided to the exit.

Message handling

If the language-specific interface validation exit detects an error, it can request that the binder print a message which is contained in a **CEEfffXM** module, which contains messages text (see page 184.) The exit requests that a message be printed by placing the address of the appropriate message text in the pointer which is the last argument to the language-specific interface validation exit.

Communication between the language-specific interface validation exit and the **CEEfffXM** module is a private interface between these two modules. A reasonable scheme would be for there to be a vector of address constants at the beginning of **CEEfffXM**. Each would point to a halfword-prefixed message text string so that in response to a request that a specific message be printed the language-specific interface validation exit would move one of those address constants into the pointer given as the argument for the returned message.

There is no provision for message inserts. Only one message can be requested per invocation of the Language Environment interface validation exit by the binder. No message should be requested except in the event of error.

Example of a language-specific interface validation exit

Figure 57 on page 192 shows an example of the interaction of the main Language Environment interface validation exit (module CEEPINTV) with a language-specific interface validation exit. First, assume that the module CEEPINTV contains the following two CEEXVSEL macro instructions. These two CEEXVSEL macro instructions will select sections meeting either of the two sets of selection criteria:

1. The section being validated has IDR information that begins with the seven characters '5655121', which represents the AD/Cycle C/370 compiler, and there is at least one external reference that is unresolved.
2. The section being validated has IDR information that begins with the eight characters 5668-806, which represents the VS FORTRAN Version 2 compiler, and there is at least one external reference that is resolved in to a section whose name begins with the three characters EDC.

Interface validation exit

CEEXVSEL	LANG_IV_EXIT=CEEEDCX0, CALLING_IDR='5655121', RESOLVED=NO	X X
*		
CEEXVSEL	LANG_IV_EXIT=CEEAFHX0, CALLING_IDR='5668-806', TARGET_SECTION_PREFIX='EDC'	X X

Figure 57. Language-specific interface validation exit

Remember that the Language Environment interface validation exit is entered with a Validate function code once for each section that contains external references.

The main Language Environment interface validation exit (CEEPINTV) examines the section being validated and its external references. If the first set of criteria is satisfied, control is passed to the C-specific interface validation exit whose name is CEEEDCX0. Upon return from CEEEDCX0, control returns to the binder without examination of the second set of selection criteria. However, if the first set of criteria is not satisfied but the second set is, control is passed to the Fortran-specific interface validation exit whose name is CEEAFHX0.

For this example, assume that only the second set of selection criteria is satisfied so that the Fortran-specific language validation exit named CEEAFHX0 would be entered. In this case, this means that the section being validated was produced by the VS FORTRAN Version 2 compiler and that at least one of its external references is being resolved into a section whose name begins with the characters EDC, which likely indicates a C library routine. Also, assume that the purpose of CEEAFHX0 is to change the external reference SQRT to AFHFSSQS if the binder was about to bind SQRT to an entry point in the C library routine EDC1@0C4. The code example (below) shows how CEEAFHX0 could do this.

```

CEEAFHX0 CEEENTRY PPA=PPAX0,MAIN=NO,BASE=11
*
* INITIALIZE POINTERS
*
      L   2,4(,1)           PTR TO PTR TO ANCHOR BLOCK
      L   2,0(,2)           PTR TO ANCHOR BLOCK USING ANCHOR_BLOCK,2
      L   2,AB_REFL         FIRST REFERENCE LIST ENTRY
      USING REFL,2
*
* CHECK FOR SQRT AS EXTERNAL REFERENCE IN REFERENCE LIST ENTRY*
*
LOOP_NXT L   3,REFL_T_SYMBOL   EXTERNAL REFERENCE
      LA  0,L'SQRT_SY         SQRT NAME LENGTH
      CH  0,0(,3)             IS REF LIST ER NAME SAME LENGTH?
      BNE LOOP_CTL            NO, CAN'T BE THE ER WE'RE SEEKING
      CLC SQRT_SY,2(3)        IS REF LIST ER NAME WHAT WE WANT?
      BE  GOT_SQRT            YES, GO ANALYZE IT FURTHER
*
* NOT SQRT SO GO CHECK NEXT REFERENCE LIST ENTRY IF ANY
*
LOOP_CTL L   2,REFL_NEXT      NEXT REFERENCE LIST ENTRY
      LTR 2,2                 IS THERE ANOTHER ENTRY?
      BNE LOOP_NXT            YES, GO CHECK IT
*
* RETURN *
*
DONE     CEETERM ,           RETURN TO CALLER
*
* EXTERNAL REFERENCE IS SQRT; CHECK IF IN C LIBRARY TARGET SECTION
*
GOT_SQRT L   3,REFL_T_SECTION  TARGET SECTION NAME
      LA  0,L'CSQRT_SY        C SQRT TARGET SECTION NAME LENGTH

```

```

          CH  0,0(,3)          IS REF LIST TARGET SECTION NAME
*                               THIS SAME LENGTH?
          BNE  DONE           NO, CAN'T BE SECT WE'RE SEEKING
          CLC  CSQRT_SY,2(3)  IS TARGET SECTION THE NAME OF
*                               C LIBRARY ROUTINE WE WANT?
          BNE  DONE           NO, EXIT (CAN'T BE ANOTHER SQRT)
*
* CHANGE EXTERNAL REFERENCE FROM SQRT TO AFHFSSQS*
*
          LA  0,FSSQS_LN      NEW SYMBOL TO USE AS ER
          ST  0,REFL_X_SYMBOL SAVE IN REFERENCE LIST ENTRY
          LA  1,4             ACTION CODE FOR NEW ER NAME
          STH 1,REFL_X_ACTION SAVE IN REFERENCE LIST ENTRY
          B   DONE           DONE. (CAN'T BE ANOTHER SQRT)
*
* EXTERNAL REFERENCE BEING VALIDATED AND REPLACEMENT SYMBOL
*
SQRT_SY DC  C'SQRT'
*
FSSQS_LN DC  Y(L'FSSQS_SY)
FSSQS_SY DC  C'AFHFSSQS'
*
* SECTION NAME OF C LIBRARY ROUTINE THAT CONTAINS THE C SQRT LD
*
CSQRT_SY DC  C'EDC1@0C4'
*
* PPA1 AND PPA2
*
PPAX0  CEEPPA
*
* ANCHOR BLOCK (POINTED TO THROUGH SECOND ARGUMENT)
*
ANCHOR_BLOCK DSECTAB_STATIC DS  A          ADDRESS OF STATIC AREA
AB_REFL      DS  A          ADDRESS OF FIRST APPLICABLE
*                               REFERENCE LIST ENTRY
*
* REFERENCE LIST ENTRY*
REFL        DSECT
*
REFL_NEXT   DS  A          ADDRESS OF NEXT LIST ENTRY
REFL_T_SYMBOL DS  A          ADDRESS OF REFERENCED SYMBOL
REFL_T_SECTION DS  A          ADDRESS OF TARGET SECTION NAME
REFL_T_ELEMENT DS  A          ADDRESS OF TARGET ELEMENT
REFL_T_DESCR DS  A          ADDRESS OF TARGET DESCRIPTORS
REFL_T_IDR   DS  A          ADDRESS OF TARGET IDR
REFL_T_ENVIR DS  XL4        TARGET ENVIRONMENT
REFL_T_SIGN  DS  CL8        TARGET SIGNATURE
REFL_T_ADCONS DS  A          ADCON LIST ANCHOR
REFL_C_DESCR DS  A          ADDRESS OF CALLER DESCRIPTORS
REFL_C_ENVIR DS  XL4        CALLER'S ENVIRONMENT
REFL_X_SIGN  DS  CL8        EXIT SIGNATURE
REFL_X_SYMBOL DS  A          NEW SYMBOL (CHAR(*) VARYING)
REFL_X_ACTION DS  H          ACTION CODE
*                               RESERVED
*
          CEEDSA ,
          CEECAA ,
          END

```

Interface for preinitialization

This section describes the preinitialization functions that are intended for use as CWIs. For information about other valid invocations, see *z/OS Language Environment Programming Guide*.

CEEPIPI — invocation for subroutine by address

Each invocation of CEEPIPI (call_sub_addr_nochk) or CEEPIPI (call_sub_addr_nochk2) invokes a specified routine by address, which is similar to

Preinitialization Interface

CEEPIPI(*call_sub_addr*), but does not perform Language Environment anchor look-up, set, or reset. Both of these CWI interfaces to CEEPIPI are intended to be used when the Language Environment environment is initialized and terminated in one task control block (TCB) or address space but is used from a different TCB or from an SRB, in the same or a different address space.

Both CWIs are supported only in the CEEPIPI(*init_sub_dp*) environment, which must be initialized with TRAP(ON,NOSPIE), INTERRUPT(OFF), and NOTEST. If the CEEPIPI(*init_sub_dp*) interface is used to establish multiple Language Environment environments under the same address space, the routine must not use z/OS UNIX functions. For additional information, see *z/OS XL C/C++ Runtime Library Reference*.

The Language Environment environment identified by the *token* is activated before the called routine is invoked. After the called routine returns, the environment is dormant.

Syntax

call CEEPIPI (*call_sub_addr_nochk*, *routine_addr*, *token*, *parm_ptr*, *sub_ret_code*, *sub_reason_code*, *sub_feedback_code*)

```
INT4      *call_sub_addr_nochk;
POINTER   *routine_addr;
INT4      *token;
POINTER   *parm_ptr;
INT4      *sub_ret_code;
INT4      *sub_reason_code;
INT4      *sub_feedback_code;
```

Syntax

call CEEPIPI (*call_sub_addr_nochk2*, *routine_addr*, *token*, *parm_ptr*, *sub_ret_code*, *sub_reason_code*, *sub_feedback_code*)

```
INT4      *call_sub_addr_nochk2;
POINTER   *routine_addr;
INT4      *token;
POINTER   *parm_ptr;
INT4      *sub_ret_code;
INT4      *sub_reason_code;
INT4      *sub_feedback_code;
```

***call_sub_addr_nochk* (input)**

a fullword function code (integer value of 12) that specifies the CEEPIPI request for calling a C main routine and obtaining writable static. The *routine_addr* specified must be CEESTART. The entry point called will then be the main entry point specified in the CEEMAIN referenced by that CEESTART. For more information on CEESTART and CEEMAIN, see “CEESTART” on page 140 and “CEEMAIN” on page 145 and *z/OS XL C/C++ User’s Guide*.

***call_sub_addr_nochk2* (input)**

a fullword function code (integer value of 14) that specifies the CEEPIPI request for calling a C, C++, PL/I, or Language Environment-conforming assembler subroutine.

***routine_addr* (input)**

a doubleword containing the address of the routine that should be invoked. The first fullword contains the entry point address; the second fullword must be zero.

***token* (input)**

a fullword with the value of the token returned by CEEPIPI(init_sub_dp) when the Language Environment environment is initialized. The *token* must identify a previously pre-initialized environment that is not active at the time of 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. Runtime options are not obtained from this parameter.

***sub_ret_code* (output)**

the subroutine return code.

***sub_reason_code* (output)**

the subroutine reason code; this is 0 for normal subroutine returns.

***sub_feedback_code* (output)**

the feedback code for enclave termination; this is the CEE000 feedback code for normal subroutine returns. 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 <i>function_code</i> is not valid. |
| 8 | CEEPIPI was called from a Language Environment-conforming HLL. |
| 12 | The indicated environment was not initialized to allow multiple Language Environment environments for subroutines. |
| 16 | The token is invalid. |
| 28 | A PL/I STOP, C exit(), or unhandled condition with severity 2 or greater occurred. |
| 36 | The language of the subroutine is not present in the environment identified by token. |

Usage Notes:

1. This CWI is supported in the init_sub_dp environment only.
2. The init_sub_dp environment must be initialized with TRAP(NOSPIE), INTERRUPT(OFF), and NOTEST.
3. The routine must be written in PL/I, C, C++, and must be reentrant or written in Language Environment-conforming assembler.
4. The routine must not contain PL/I STOP or C exit() calls. PL/I STOP and C exit() will cause a Language Environment enclave termination. Such termination will cause an unpredictable result because the TCB for the CALL time is different from the TCB for the INIT/TERM time.
5. If the PL/I or C routine calls an Assembler routine, the Assembler routine must not contain an SVC LINK; LINK will cause a Language Environment enclave initialization. Such initialization will cause unpredictable results because the TCB for the CALL time is different from the TCB for the INIT/TERM time.
6. The caller of this CWI is responsible to establish its own error recovery for hardware- and software-detected errors; otherwise, the Language Environment condition manager will be in control. The Language Environment condition manager will terminate the current Language Environment enclave and/or process for any unhandled condition with severity 2 or greater. Such termination will cause unpredictable results because the TCB for the CALL time is different from the TCB for the INIT/TERM time.

Preinitialization Interface

7. If the CEEPIPI(*init_sub_dp*,...) interface is used to establish multiple Language Environment environments under the same address space, the routine must not use z/OS UNIX functions.
8. The Language Environment Math services can be called when using the *call_sub_addr_nochk* function.
9. Nested enclaves are not supported when *call_sub_addr_nochk* is used while in System Request Block (SRB) mode.
10. The language of the routine must already be present in Language Environment, identified by *token*. This is done by including a routine coded in the same language in the PreInit table used during initialization of the environment.

Preinitialization environment and system request block mode

The following topics describe the preinitialization environment and system request block (SRB) mode.

Initializing the preinitialization environment

Language Environment requires that a preinitialization environment be initialized while running in task mode. For a preinitialization environment to be able to run in SRB mode, initialize the preinitialization environment by using the CEEPIPI *init_sub_dp* function (function code 9).

Calling the preinitialization environment in SRB mode

To call the preinitialization environment while running in SRB mode, the call must be made using the CEEPIPI *call_sub_addr_nochk* function (function code 12 if calling a C main routine and obtaining writable static, function code 14 if calling a C subroutine). "nochk" indicates that Language Environment will not perform any processing that depends on a TCB address, such as anchor look-up, set, or reset.

Preinitialization service routines

Restrictions exist when running a routine in SRB mode. For instance, an SRB routine cannot issue any SVCs (except for ABEND). This restriction causes difficulties when attempting to use Language Environment in SRB mode; since the default operating system services that Language Environment uses make calls to SVCs.

The preinitialization services offer a solution. By specifying a Service Routine Vector while initializing the preinitialization environment, an application can replace the basic operating system service routines that Language Environment provides, supplying alternative services or mechanisms to accomplish the same function. The following service routines can be replaced through the use of the Service Routine Vector:

- Load Module
- Delete Module
- Get Storage
- Free Storage
- Handle Exception
- Process Message

To run in SRB mode, each of the listed service routines must be replaced. The following sections explain ways to perform the function for each service routine while in SRB mode. For details on the interfaces to these services, see *z/OS Language Environment Programming Guide*.

Module Load/Delete Routines: One way to provide module loads while in SRB mode requires:

1. Forcing all modules to be loaded during initialization of the preinitialization environment. During CEEPIPI `init_sub_dp` processing, Language Environment does not necessarily load all of the modules required by the application. One way to force Language Environment to load these modules is to provide a dummy C function in the PreInit table passed to Language Environment during initialization of the preinitialization environment. Language Environment will detect that the C language is present, load the C event handler, and call C for initialization, which will cause C to load additional modules.

Language Environment's math functions reside in a separate load module that is not normally loaded. To get the math module loaded while still in task mode, include a call to a math function in the dummy C function. Once Language Environment has completed initialization of the preinitialization environment, the application's initialization routine can then call the dummy function by using the CEEPIPI `call_sub` function (function code 4). The dummy function's call to the math function forces Language Environment to load the math module.

2. Keeping track of the name and entry point of each module that is loaded during initialization. One way is for the application to provide its own load and delete service routines. During initialization of the preinitialization environment, when Language Environment calls the application's load service routine for each module, it performs a normal load, saving the module name and entry point address in a table. When the load service routine is called in SRB mode, it simply looks up the module name in the table and returns its entry point address to the caller. Unless cleanup of the modules is required at some point, the delete service can simply return to its caller, leaving the module in storage for the next caller.

Another method for providing module loads while in SRB mode is to have the application set up "worker" tasks in its address space during initialization. The application's load service routine can create a work request, queue it to a work queue, and then SUSPEND the SRB. Once the worker task has processed the load, it can return the module entry point to the SRB, and RESUME it. Once loaded, the module can be left in storage, and the load service routine can track its location.

Storage Get/Free Routines: The Storage Get and Free routines can be replaced with routines that use the z/OS STORAGE OBTAIN and STORAGE RELEASE services, respectively. These macros do not issue SVCs; instead, they issue a Program Call (PC) instruction, which is allowable in SRB mode.

Exception Routine: To use a preinitialization environment in SRB mode, Language Environment requires that the environment be initialized with the TRAP(OFF), INTERRUPT(OFF), and NOTEST runtime options. These options prevent Language Environment from establishing exception handlers under the current task, which does no good when the preinitialization environment is called from an SRB routine.

If the application requires exception handling, it can establish its own by providing an exception routine in the service vector routine. Language Environment calls the exception routine during `init_sub_dp` processing to inform the application of the address of the Language Environment condition handling routine to call when an exception occurs, as well as providing a list of exceptions in which the condition handler is interested. Figure 58 on page 198 shows when the exception routine is

Preinitialization Interface

called during initialization.

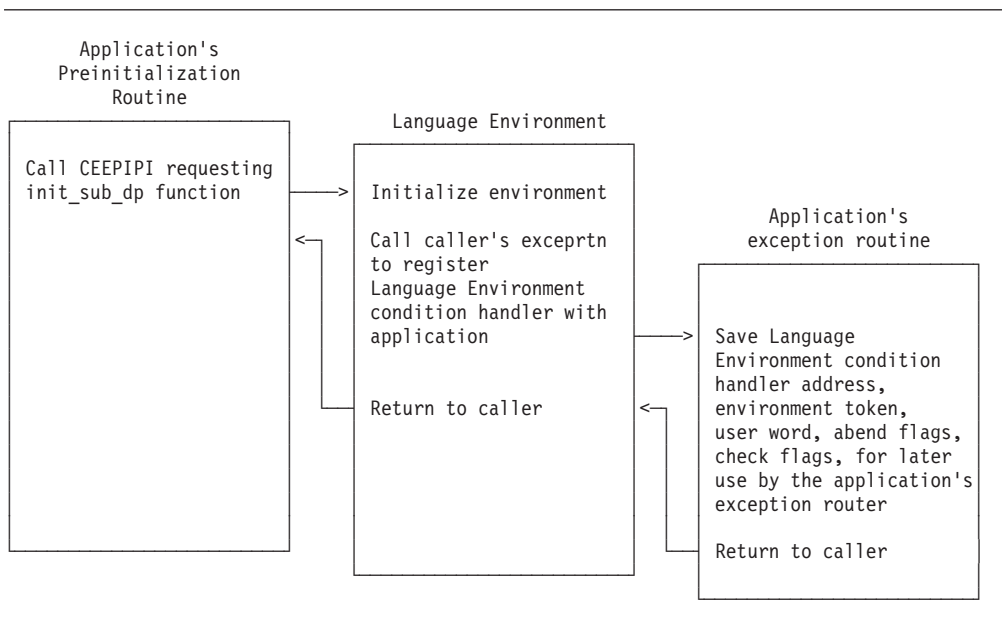


Figure 58. Preinitialization environment initialization exceptn flow (task mode)

When the application's SRB routine gets control, it must establish its own exception handler before calling the preinitialization environment. It can do so by invoking the SETFRR service to establish an FRR.

If an exception occurs during a call to the preinitialization environment, the application's exception handler receives control. By examining the SDWA and the information provided by Language Environment during the initial call to the exception routine, the exception handler can determine whether Language Environment is interested in the exception. If so, then the exception handler calls the Language Environment condition handler. Language Environment then drives whatever HLL exception handling routines the application has established.

Under certain conditions, Language Environment calls the application's exception routine to register another level of exception handling. This call will not occur while the application's exception handler is in control. Figure 59 on page 199 shows the control flow during exception handling for an SRB mode application.

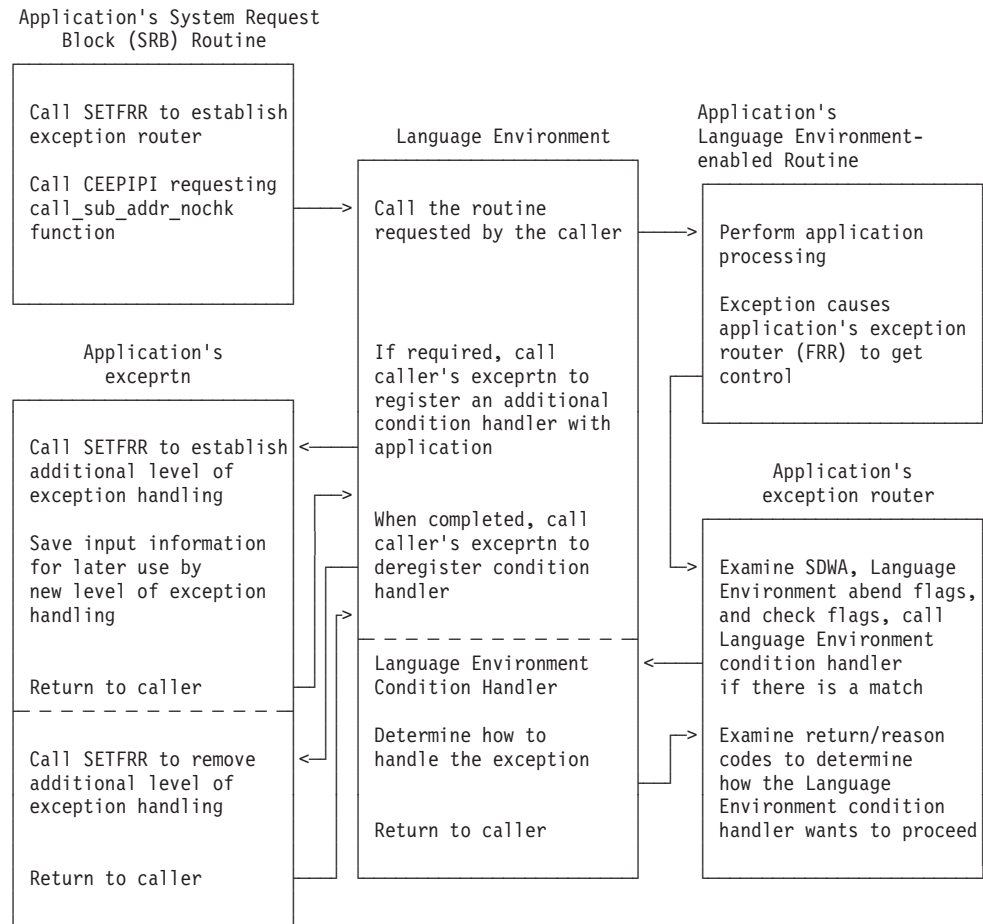


Figure 59. Preinitialization environment exception flow (SRB mode)

Message Routine: One method for providing message handling while in SRB mode is to have the application set up "worker" tasks in its address space during initialization. The application's message service can create a work request from the message, queue it to a work queue, and post the worker task to process it asynchronously. In cases where the application is not interested in the messages, the message routine can ignore the message by simply returning to its caller.

CEEXPIT macro keyword

Language Environment supports the following keyword for the CEEXPIT macro:

CICS=YES

indicates that the preinitialization environment being created allows EXEC CICS commands to be executed.

CICS=NO

indicates that the preinitialization environment being created will not be used to execute EXEC CICS commands; this is the default.

Preinitialization Interface

Chapter 4. Storage management

The Language Environment storage manager provides services that control the stack and heap storage used at run time. These services extend to Language Environment member languages and assembler language routines which adopt this storage protocol. Storage Management consists of:

- Storage allocation
- Storage clean-up
- Service call routines

The initial allocation of STACK storage is done by Language Environment initialization routines. The initial allocation of HEAP storage is done upon the first request for HEAP storage. The storage manager:

- Manages STACK and HEAP storage above and below the 16 MB line
- Manages any subsequent allocations of STACK or HEAP
- Interfaces with host operating system to allocate/free storage
- Detects the short-on-storage condition and signal the exception handler
- Releases (or keeps track) of free storage segments
- Cleans-up resources at termination

Stack storage is managed in two stack queues. The primary stack queue is the user stack. The user stack is directed above or below the 16 MB line by the STACK runtime option. The secondary stack queue is the library stack which is always directed below 16 MB for use by routines needing a DSA (dynamic storage area) below the 16 MB line. Library DSAs can be obtained from the user stack, but user DSAs can not be obtained from the library stack. All stack storage is managed at each individual thread level; stacks are not shared across threads.

Heap storage can either be directed anywhere or below the 16 MB line.

In addition to storage manager, Language Environment provides an interface to a vendor heap manager for use with C/C++ applications.

Dynamic storage (heap) services

This section describes the various types of heap storage and covers the services provided to acquire, release, and manage the heap storage.

Storage model

The Language Environment storage model is based on a model of multiple heaps that can be dynamically created and discarded. Each heap has a unique *heap-ID*. The Language Environment storage model includes a single heap sub-model for languages whose intrinsic storage model does not comprehend multiple heaps. A single heap model does not have all the function of the multiple heap model. Missing from the model are group de-allocation capabilities. The initial heap cannot be discarded. Table 46 on page 202 lists all Language Environment heaps and their purpose.

Storage Management

Table 46. Heap IDs recognized by Language Environment heap manager

Heap Name	Heap-ID	Intended Purpose	Created by	Disposed of by
Initial Heap	0	Application program data.	Language Environment initialization; size and location are determined from HEAP runtime option.	Language Environment termination
Extended Initial Heap	(returned by CEECRHP)	Application program data.	Call CEECRHP. Size and location determined from HEAP runtime option.	Call CEEDSHP, or Language Environment termination
Language Environment/ language anywhere heap	CEEEDBANHP	Runtime library data that can reside above 16 MB (Language Environment and member language control blocks).	Language Environment initialization. Storage can reside either above or below 16 MB; size determined from ANYHEAP runtime option.	Language Environment termination
Language Environment/ language below heap	CEEEDBBEHP	Runtime library data required to reside below 16 MB (Language Environment and member language DCBs).	Language Environment initialization. Storage always resides below 16 MB; size determined from the BELOWHEAP runtime option.	Language Environment termination
Additional Heaps	(returned by CEECRHP)	Collections of application program data that can be quickly disposed with a single CEEDSHP call.	Call CEECRHP. Arguments define heap size and location.	Call CEEDSHP, or Language Environment termination

CWI to the heap services

Language Environment provides the following sets of CWIs for heap services:

- Routines for process-level heap storage (acquisition and release)
- Routines for region-level heap storage interface.
- Routines using a parameter list (PLIST) interface. Call these CWI interfaces as follows (xxxx has the appropriate offset value listed in Table 47):

```
L    R12,A(CAA)           Get the address of CAA in R12
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,xxxx(,R15)
BALR R14,R15
```

Table 47 lists the interfaces and their corresponding callable services, which are described in more detail in the *z/OS Language Environment Programming Guide*. In each case, the parameter list for the callable service also applies to the CWI.

Table 47. Routines using a parameter list interface

CWI Name	Description	Callable Service	Decimal Offset
CEEVGTST	Allocate storage	CEEGTST	144
CEEVFRST	Free storage	CEEFRST	132
CEEVCRHP	Create a new heap	CEECRHP	164
CEEVDSHP	Discard heap	CEEDSHP	168
CEEVCZST	Reallocate storage	CEECZST	2820
CEEVGTSB	Allocate storage unconditionally below 16 MB	None	2936

Member-language intrinsic functions such as *malloc* must generate a call to a member-language stub routine. The stub routine, in turn, must call the corresponding Language Environment service (for example, CEEVGTST with heap-ID 0) to allocate the heap storage.

Member-language control blocks should be allocated in the private Language Environment/language below heap only if they must reside below the 16 MB line. Most other internal control blocks should be allocated in the private Language Environment/language anywhere heap. The heap-IDs of both the Language Environment/language below heap and Language Environment/language anywhere heap are stored in the enclave data block (EDB) for easy access, but these heap-IDs are not exposed to application code.

Process-level heap storage management

Language Environment provides the following process-level storage management services. The addresses of the process-level storage routines are found in the Process Control Block (PCB) at labels CEEPCB_ZGETST and CEEPCB_ZFREEST. AMODE switching is not performed for the process-level GETMAIN and FREEMAIN.

CEEPCB_ZGETST

This routine allocates storage on behalf of the storage manager. This routine can rely upon the caller to provide a save area, which can be the **@Workarea**.

Syntax

void CEEPCB_ZGETST (*amount, subpool_no, user word, flags, stg_address, obtained, return code, reason code*)

```
INT4    *amount;
INT4    *subpool_no;
POINTER *user word;
INT4    *flags;
POINTER *stg_address;
INT4    *obtained;
INT4    *return code;
INT4    *reason code;
```

CEEPCB_ZGETST

Call this CWI interface as follows:

```
L      R15,CEECAAPCB-CEECAA(,R12)
L      R15,36(,R15)
BALR  R14,R15
```

amount (input)

Fixed-binary(31) amount of storage requested.

subpool_no (input)

Fixed-binary(31) subpool number 0-127.

user word (input)

Pointer to a fullword user field.

flags (input)

Fullword flag area. Bit zero in the flags is ON if the storage is required below the 16 MB 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 (output)

Fullword address of the storage obtained or zero.

Storage Management

obtained (output)

Fixed-binary(31) number of bytes obtained.

return code (output)

Return code from CEEPCB_ZGETST service.

reason code (output)

Reason code from the CEEPCB_ZGETST service.

Return Code	Reason Code	Description
0	0	Successful
16	0	Unsuccessful — uncorrectable error occurred

CEEPCB_ZFREEST

The CEEPCB_ZFREEST routine frees storage on behalf of the storage manager.

Syntax

void CEEPCB_ZFREEST (*amount, subpool_no, user word, stg_address, return code, reason code*)

```
INT4    *amount;
INT4    *subpool_no;
POINTER *user word;
POINTER *stg_address;
INT4    *return code;
INT4    *reason code;
```

CEEPCB_ZFREEST

Call this CWI interface as follows:

```
L      R15,CEECAAPCB-CEECAA(,R12)
L      R15,40(,R15)
BALR  R14,R15
```

amount (input)

Fixed-binary(31) amount of storage to free.

subpool_no (input)

Fixed-binary(31) subpool number 0-127.

user word (input)

Pointer to a fullword user field.

stg_address (output)

Fullword address of the storage to free.

return code (output)

Return code from the CEEPCB_ZFREEST service.

reason code (output)

Reason code from the CEEPCB_ZFREEST service.

Return Code	Reason Code	Description
0	0	Successful
16	0	Unsuccessful — uncorrectable error occurred

Region-level heap storage management

This section describes the region-level storage management services that are provided by Language Environment. The addresses of the process-level storage

routines are found in the Region Control Block (RCB) at labels CEERCB_ZGETST and CEERCB_ZFREEST. AMODE switching is not performed for the region-level GETMAIN and FREEMAIN.

CEERCB_ZGETST

This routine allocates storage on behalf of the storage manager. This routine can rely upon the caller to provide a save area, which can be the @Workarea. The parameter list that is passed contains the following:

Syntax

void CEERCB_ZGETST (*amount, subpool_no, user word, flags, stg_address, obtained, return code, reason code*)

```
INT4    *amount;
INT4    *subpool_no;
POINTER *user word;
INT4    *flags;
POINTER *stg_address;
INT4    *obtained;
INT4    *return code;
INT4    *reason code;
```

CEERCB_ZGETST

Call this CWI interface as follows:

```
L      R15,CEECAARCB-CEECAA(,R12)
L      R15,32(,R15)
BALR  R14,R15
```

amount (input)

Fixed-binary(31) amount of storage requested.

subpool_no (input)

Fixed-binary(31) subpool number 0-127.

user word (input)

Pointer to a fullword user field.

flags (input)

Fullword flag area. Bit zero in the flags is ON if the storage is required below the 16 MB 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 (output)

Fullword address of the storage obtained or zero.

obtained (output)

Fixed-binary(31) number of bytes obtained.

return code (output)

Return code from CEERCB_ZGETSTR service.

reason code (output)

Reason code from the CEERCB_ZGETST service.

Return Code	Reason Code	Description
0	0	Successful
16	0	Unsuccessful — uncorrectable error occurred

CEERCB_ZFREEST

This routine frees storage on behalf of the storage manager. The parameter list passed contains the following:

Syntax

void CEERCB_ZFREEST (*amount, subpool_no, user word, stg_address, return code, reason code*)

```
INT4      *amount;
INT4      *subpool_no;
POINTER   *user word;
POINTER   *stg_address;
INT4      *return code;
INT4      *reason code;
```

CEERCB_ZFREEST

Call this CWI interface as follows:

```
L      R15,CEECAARCB-CEECAA(,R12)
L      R15,36(,R15)
BALR   R14,R15
```

amount (input)

Fixed-binary(31) amount of storage to free.

subpool_no (input)

Fixed-binary(31) subpool number 0-127.

user word (input)

Pointer to a fullword user field.

stg_address (output)

Fullword address of the storage to free.

return code (output)

Return code from the CEERCB_ZFREEST service.

reason code (output)

Reason code from the CEERCB_ZFREEST service.

Return Code	Reason Code	Description
0	0	Successful
16	0	Unsuccessful — uncorrectable error occurred

CEEVGTSB — unconditional get heap below

The CEEVGTSB CWI service obtains enclave heap storage below the 16 MB line and, if unsuccessful, CEEVGTSB signals a condition when the feedback code is omitted.

Syntax

void CEEVGTSB (*heap_id, size, address, [fc]*)

```
INT      *heap_id;
INT      *size;
POINTER  *address;
FEEDBACK *fc;
```

CEEVGTSB

Call this CWI interface as follows:


```

L    R12,A(CAA)           Get the address of CAA in R12
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,2936(,R15)
BALR R14,R15

```

heap_id (input)

A token denoting the heap in which the storage is allocated. If *heap_id* is not valid, the *address* is undefined and CEEVGTSB signals a condition.

size (input)

A number representing the amount of storage to be allocated. The amount of storage obtained is rounded to the next higher multiple of 8 bytes. Storage is always allocated below the line on a doubleword boundary. If the specified amount cannot be obtained, a condition is signaled.

address (output)

The machine address of the first byte of allocated storage.

fc (output)

The resulting feedback code. The following conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE0P2	Severity	4
	Msg_No	0802
	Message	Storage headers are damaged.
CEE0P3	Severity	3
	Msg_No	0803
	Message	The heap identifier <i>heap_id</i> did not match any existing heap.
CEE0P8	Severity	3
	Msg_No	0808
	Message	The <i>size</i> was not a positive number.
CEE0PD	Severity	3
	Msg_No	0813
	Message	The request was larger than the storage available.
CEE3JN	Severity	0
	Msg_No	3704
	Message	Expected data at address <i>address: data</i>
CEE3JO	Severity	0
	Msg_No	3705
	Message	Pointer at <i>address</i> should point to a valid <i>controlblock</i>

CEEV#GTS — get heap storage

The CEEV#GTS CWI allocates heap storage from a user-specified heap.

Syntax

void (ceev#gts)

CEEV#GTS

Call this CWI interface as follows:

```
L   R12,A(CAA)
L   R15,CEECAACELV-CEECAA(,R12)
L   R15,124(,R15)
BALR R14,R15
```

Parameters are passed to CEEV#GTS in registers:

R1 (Input) Heapid (0 for user heap)
R1 (Output) Address of storage obtained
R2 (Input) Number of bytes of storage to obtain

Usage Notes:

1. Storage below the 16 MB line is always returned under the following conditions:

- The caller is in AMODE 24
- HEAP(,BELOW) is in effect
- The ensm_below16m_flag is set

Storage above the 16 MB line will only be returned if the caller is in AMODE 31 and HEAP(,ANY) is in effect.

The caller's AMODE is determined by the high order bit of R14.

```
0   AMODE 24
1   AMODE 31
```

2. The caller must test for errors. When an error occurs, R15 will be nonzero. The caller must either handle the error or build a 96-bit feedback token and signal it.
3. The conditions that can result from this service are the same as the conditions from the CEEGTST AWI.
4. The heapid (R1 on input) must be 0 (for the user heap) or a value returned from the CEECRHP AWI callable service.

CEEV#FRS — free heap storage

The CEEV#FRS CWI frees heap storage from a user-specified heap.

Syntax

void (ceev#frs)

CEEV#FRS

Call this CWI interface as follows:

```
L   R12,A(CAA)
L   R15,CEECAACELV-CEECAA(,R12)
L   R15,3452(,R15)
BALR R14,R15
```

Parameter is passed to CEEV#FRS in register 1:

R1 (Input) Address of storage to free

Usage Notes:

1. The caller must test for errors. When an error occurs, R15 will be nonzero. The caller must either handle the error or build a 96-bit feedback token and signal it.
2. The conditions that can result from this service are the same as the conditions from the CEEFRST AWI.

CEEVHRPT — obtain dynamic heap storage report

CEEVHRPT returns information about an application's user heap storage (specifically, enclave-level heap ID 0). CEEVHRPT returns information that is similar to the information in the user heap storage section of the report that is generated when you specify the RPTSTG(ON) runtime option.

Using CEEVHRPT, an application that is running can obtain information about heap storage. However, CEEVHRPT will not report any information that relates to the heap pool manager; rather, storage information about heap pools will be included in the number of allocated user heap bytes that is returned by the service.

Syntax

CEEVHRPT (*uheap_size*, *uheap_bytes_alloc*, *uheap_bytes_free*, [*fc*])

CEEVHRPT

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,4032(,R15)
BALR R14,R15
```

uheap_size (input)

The total amount of user heap storage that is currently allocated by the application.

uheap_bytes_alloc (input)

The amount of user heap storage that is currently in use by the application.

uheap_bytes_free (input)

The amount of user heap storage that is currently available to the application. Note that the available storage may not be contiguous.

fc (output/optional)

A 12-byte feedback code that indicates the results of this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE0P2	Severity	4
	Msg_No	802
	Message	Heap storage control information was damaged.
CEE3JN	Severity	0
	Msg_No	3704
	Message	Expected data at address <i>address: data</i>
CEE3JO	Severity	0
	Msg_No	3705
	Message	Pointer at <i>address</i> should point to a valid <i>controlblock</i>

Condition		
CEE4VG	Severity	3
	Msg_No	5104
	Message	The z/OS UNIX System Services callable service BPX1PTQ failed.

User-created heap services

This section describes the various types of services provided to acquire, release, and manage heap storage resulting from user-provided storage.

CEEVUHCR — create a heap using user-provided storage

The CEEVUHCR CWI creates a heap out of storage that is provided by the caller. The heap is divided into cell pools based on the information provided in the `cellpool_attrib_table`. Up to 6 cell pools can be created within the heap. Note that this is a fixed-size heap; when storage within a given cell pool is exhausted, no additional storage will be allocated. CEEVUHCR returns a heap token that is used to identify the heap on subsequent user-created heap CWI calls, such as CEEVUHGT, CEEVUHFR, and CEEVUHRP.

Syntax

void CEEVUHCR (*block, size, cellpool_attrib_table, heap_token, rsvd1, rsvd2, rsvd3, rsvd4, [fc]*)

```

POINTER    *block;
INT4       *size;
POINTER    *cellpool_attrib_table;
POINTER    *heap_token;
POINTER    *rsvd1;
POINTER    *rsvd2;
POINTER    *rsvd3;
POINTER    *rsvd4;
FEED_BACK  *fc;

```

CEEVUHCR

Call this CWI interface as follows:

```

L      R15,CEECAACELV-CEECAA(,R12)
L      R15,4060(,R15)
BALR  R14,R15

```

block (input)

A pointer to the storage which is to be used for the heap.

size (input)

The size of the block of storage. Note that Language Environment reserves approximately 328 bytes of this storage for use in allocating heap management control blocks. Additional storage is reserved if storage report usage statistics are being collected for the heap. The amount of this storage is related to the largest cell size and the granularity of the statistics, and is calculated as:
 $\text{storage amount} = ((\text{largestcellsize} + \text{granularity} - 1) / \text{granularity}) * 4.$

cellpool_attrib_table (input)

A pointer to a structure describing the attributes of the cell pools to be created by CEEVUHCR.

The first field of the structure, `number_of_pools`, indicates the number of cell pools to be created. Up to 6 cell pools can be created in the heap.

The second field of the structure, *granularity*, indicates the granularity to which storage usage statistics are to be collected. This value must be zero, or a power of 2 greater than or equal to 8. If the value is zero, then statistics are not collected.

Following these words are pairs of words describing the attributes of each cell pool in the heap. The first field in the pair, *size*, is the size of the cell in the cell pool. The cell size must be a multiple of 8 and greater than or equal to 8. Note that Language Environment adds an additional 8 bytes to the size of the cell for use in managing the cells. The second field in the pair, *percentage*, is the percentage of the total block size to be allocated for the cell pool.

heap_token (output)

A token representing the heap that was created.

rsvd1-rsvd4

Reserved for future use.

fc (output/optional)

The parameter into which the callable service feedback code is placed. The following conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE0P7	Severity	3
	Msg_No	0807
	Message	An input parameter to the CEEVUHCR CWI was not valid.

CEEVUHGT — allocate storage from a user-created heap

The CEEVUHGT CWI allocates storage from the heap identified by the *heapid*. CEEVUHGT will search for an available cell within the cell pool that contains cells at least as large and closest in size to the requested size. CEEVUHGT uses the C-style parameter interface. If successful, CEEVUHGT returns the address of the reserved cell in register 15. The returned value is NULL if a cell of the required size is not available, if size was larger than the largest available cell size, or if size was specified as 0. If CEEVUHGT returns a NULL because there is not enough storage or if the requested size was too large, it will also return an error value in *errno*. The following are the possible values of *errno*:

ENOMEM

Insufficient memory is available

E2BIG Requested amount of storage is larger than the largest available cell size

Syntax

void CEEVUHGT (*heap_token*, *size*)

POINTER *heap_token*;
INT4 *size*

CEEVUHGT

Call this CWI interface as follows:

CEEVUHGT

```
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,4064(,R15)
BALR R14,R15
```

heap_token (input)

The identifier of the user-created heap from which the storage is to be allocated.

size (input)

The amount of storage to be allocated.

CEEVUHFR — return storage to a user-created heap

The CEEVUHFR CWI returns storage to the heap identified by the *heapid*. If the returned storage does not belong to the given heap, the result is unpredictable. CEEVUHFR uses the C-style parameter interface.

Syntax

```
void CEEVUHFR (heap_token, ptr)
```

```
POINTER  heap_token;
POINTER  ptr
```

CEEVUHFR

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,4068(,R15)
BALR R14,R15
```

heap_token (input)

The identifier of the user-created heap to which the storage is to be returned.

ptr (input)

A pointer to the storage to be returned to the heap.

CEEVUHRP — produce a storage report for a user-created heap

The CEEVUHRP CWI generates a report of the storage used within the user-created heap identified by *heapid*. The report is directed to the *ddname* specified in the MSGFILE runtime option. The report format is similar to the heap pools portion of the storage report generated for the RPTSTG runtime option.

Statistics for the user-created heap will only be collected if the *granularity* field of the *cellpool_attrib_table* passed to CEEVUHCR is non-zero and a valid value.

Syntax

```
void CEEVUHRP (heap_token)
```

```
POINTER  *heap_token
```

CEEVUHRP

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,4072(,R15)
BALR R14,R15
```

heap_token (input)

The identifier of the user-created heap for which a report is to be produced.

Vendor heap manager interface

The vendor heap manager interface allows an external heap manager product to support C/C++ applications by an event driven interface. The following routines are supported for non-XPLINK and XPLINK:

- malloc() (C++ default operator new and default operator new[] are included)
- calloc()
- realloc()
- free() (C++ default operator delete and default operator delete[] are included)

Note: The vendor heap manager does not manage the following.

- ANYHEAP
- BELOWHEAP
- CEECZST
- CEEFRST
- CEEGTST
- CEEVCZST
- CEEVFRST
- CEEVGTSB
- CEEVGTST
- additional heaps (CEECRHP)
- user created heaps (__ucreate, __umalloc, __ufree)

Requirements from the vendor

A vendor, wishing to provide a replacement for functions that obtain or release storage from the user heap, needs to provide a DLL that:

- Resides in either the z/OS UNIX file system or a PDSE
- Must be a program object so the writable static area acquired for each load of the vendor heap manager does not come from the user heap storage
- Must not be XPLINK, since it must work for both XPLINK and non-XPLINK applications
- Must contain the following exported function:

```
void __cee_heap_manager(int, void *);
```

The purpose of this routine is to be the communication vehicle between Language Environment and the vendor heap manager (VHM). The communication will be in the form of event codes and data areas. The prototype for the function is in the header file,

```
<edcwccwi.h>.
```

The replacement should provide a "memory manager" that is:

- fast, when not running in debug mode and thread-safe
- storage efficient

What the vendor should know

The communication between Language Environment and the vendor heap manager (VHM) is through events and data structures. The C header, <edcwccwi.h>, contains the interfaces required to create a vendor heap manager. This includes the C structures required as input to the VHM event calls.

Vendor heap manager interface

Support provided for the vendor heap manager interface

The following events, which are described below, are supported and defined in the Vendor Interfaces header file <edcwccwi.h>. This file is located in member EDCWCCWI of the SCEESAMP data set. To include <edcwccwi.h> in an application, the header file must be copied into a PDS or into a directory in the z/OS UNIX file system where the z/OS XL C/C++ compiler will find it.

- `_VHM_INIT` - Initialization event
- `_VHM_TERM` - Termination event

Each of these events is described below:

- Initialization event (`_VHM_INIT`)

This event is driven during initialization of the Language Environment enclave before any user code is given control. The purpose of this event is for the VHM to give Language Environment the addresses of the replacement services. Language Environment will use these routines, instead of its own, to manage the user heap. The VHM can, at this time, use `getenv()` to query any environment variables it has defined that will customize its operation. The VHM should initialize its environment at this time, possibly allocating its own control blocks and the initial user heap segment. The data area passed is defined as follows:

```
struct __event1_s {
    void * __ev1_free;
    void * __ev1_malloc;
    void * __ev1_realloc;
    void * __ev1_calloc;
    void * __ev1_xp_free;
    void * __ev1_xp_malloc;
    void * __ev1_xp_realloc;
    void * __ev1_xp_calloc;
    unsigned int __ev1_le_xplink : 1,
                __ev1_le_reserved : 31;
    unsigned int __ev1_vhm_xplink : 1,
                __ev1_vhm_reserved : 31;
};
```

The elements are as follows:

`__ev1_free` (output)

This field is set by the VHM to be the address of the `free()` replacement routine.

`__ev1_malloc` (output)

This field is set by the VHM to be the address of the `malloc()` replacement routine.

`__ev1_realloc` (output)

This field is set by the VHM to be the address of the `realloc()` replacement routine.

`__ev1_calloc` (output)

This field is set by the VHM to be the address of the `calloc()` replacement routine.

`__ev1_xp_free` (output)

This field is set by the VHM to be the address of the `free()` replacement routine for XPLINK support.

`__ev1_xp_malloc` (output)

This field is set by the VHM to be the address of the `malloc()` replacement routine for XPLINK support.

`__ev1_xp_realloc` (output)

This field is set by the VHM to be the address of the `realloc()` replacement routine for XPLINK support.

`__ev1_xp_calloc` (output)

This field is set by the VHM to be the address of the `calloc()` replacement routine for XPLINK support.

`__ev1_le_xplink` (input)

This bit is set when the application is running under XPLINK. It is expected that the VHM load the XPLINK version of its replacement routines, set their addresses in the above fields, and turn on the `__ev1_vhm_xplink` bit indicating support. If the `__ev1_vhm_xplink` bit is not turned on, then Language Environment will not use the VHM.

`__ev1_vhm_xplink` (output)

This bit is set by the VHM when it sees that the `__ev1_le_xplink` bit is set and it successfully loads the XPLINK versions of the replacement routines and sets their addresses into the above fields. If the `__ev1_le_xplink` bit is not set, then the VHM does not need to consider the XPLINK replacement routines.

- Termination event (`_VHM_TERM`)

This event is driven during termination of the Language Environment enclave, after all application code has completed, but before the C library resources are terminated. There is no data area passed with this event. The purpose of this event is for the VHM to write, to `stderr`, any reports, as necessary, and then cleanup the user heap storage its has managed for the enclave.

XPLINK considerations

If the VHM intends to support XPLINK applications, then it must provide a second DLL containing the XPLINK versions of the replacement routines. During the initialization event, the VHM must load the XPLINK DLL when the `__ev1_le_xplink` bit is set. The addresses of the XPLINK replacement routines must be obtained from the XPLINK DLL and placed into the `__event1_s` structure.

Serialization

The VHM must be thread-safe. One way to detect a multi-threaded environment is to test the `ceedbmultithread` bit; see page Table 15 on page 66.

Nested enclaves

The VHM must be aware that it can be driven for initialization while already being active. This is possible in a nested enclave environment where the parent and child enclaves both specify that the VHM is to be used. Language Environment will drive the DLL load for each enclave, producing a unique writable static area.

Usage notes

The VHM should not use `malloc()`, `free()`, `calloc()`, or `realloc()` from within the replacement services, to avoid potential recursive calls.

Activating the vendor heap manager

Users choose the option to use the vendor heap manager at run time by setting the `_CEE_HEAP_MANAGER` environment variable. This environment variable is set by the end-user or the application to indicate that the vendor heap manager (VHM) DLL will be used to manage the user heap. This environment variable must be set using one of the following mechanisms:

- ENVAR runtime option

Vendor heap manager interface

- inside the file specified by the `_CEE_ENVFILE` environment variable
- export `_CEE_HEAP_MANAGER`

Each of these locations is before any user code gets control, meaning prior to the HLL user exit, static constructors, and/or `main()` getting control. Setting of this environment variable, once the user code has begun execution, will not activate the VHM, but the value of the environment variable will be updated.

`__vhm_event()` API

This API drives an event into any vendor heap manager. It drives the `_VHM_REPORT` event argument with `_VHM_REPORT_CLEAR` as the optional argument in MEMCHECK VHM.

Restrictions:

1. The API supports C/C++ and Enterprise PL/I applications. COBOL and FORTRAN are not supported
2. The vendor heap manager `_CEE_HEAP_MANAGER` environment variable must be active.

Syntax

```
#include <edcwccwi.h>
```

```
int __vhm_event (int event,...)
```

event

The VHM event to execute. The API calls the `__cee_heap_manager()` function with the event as argument. The `_VHM_REPORT` event generates the 'Heap Leak Report' and writes it in the Language Environment *OutputFileName* . The `edcwccwi` header contains the prototype of `__vhm_event()` API: `Int __vhm_event(Int, char *)`.

...

An optional argument that can be used to set special options in the event to be driven.

For more information on the Heap Leak Report and the heap manager, see *z/OS Language Environment Debugging Guide*.

XPLINK DSA extension services

This section describes the services provided to extend an XPLINK downward-growing stack frame.

CEEVXPAL — XPLINK DSA extension

This CWI is invoked to extend an XPLINK downward-growing stack frame.

Input/Output	Register	Used for
Input Registers	R0	Not used
	R1	Storage size
	R2 - R3	Not used
	R4	Stack pointer
	R5	Not used
	R6	Entry point
	R7	Return address
	R8 - R11	Not used
	R12	CAA address
	R13- R15	Not used
Output Registers	R0	Not preserved
	R1 - R2	Not preserved
	R3	Address of allocated storage
	R4	New stack pointer (saved R4, R6, R7 modified)
	R5 - R6	Not preserved
	R7 - R15	Unchanged

CEEVXPAL

Call this CWI interface as follows:

```
L      6,CEECAALEOV-CEECAA(,12)
L      6,260(,6)
BASR   7,6
DC     X'4700'
DC     Y(signed offset/8 to entry marker)
```

This CWI will return control to its invoker at the return address:

```
BR     7
```

This CWI will always;

- update R4 to point to new beginning of stack frame (maintaining quadword alignment),
- copy the register save area,
- adjust the backchain.

If allocating this storage causes a stack expansion, this CWI will also modify the saved R7 value (return address) in the stack frame so that when the routine that did the DSA extension returns it will give control to a glue routine which will fix the upward-growing stack fields in the CAA and SMCB.

Note: The argument area is never copied. The caller must never assume that something placed in the argument area is still there across a call to this CWI.

__alcaxp() — XPLINK DSA extension (alloca)

This CWI is invoked by z/OS XL C/C++ compiler generated code to extend an XPLINK downward-growing stack frame. The linkage will be normal XPLINK

`__alcxpc()`

conventions for call-by-name. It will appear like a function that takes an integer for input and returns void. It is used by the compiler to implement the compiler built-in function `alloca()`.

Syntax

```
#include <edcwcwi.h>
```

```
void __alcxpc (int storage_size)
```

storage_size

the amount of additional stack storage being requested in bytes. This value will be rounded up to a multiple of 16 to ensure that the stack frame remains on a quadword boundary.

Usage Notes:

1. This CWI changes the value of the stack pointer (R4) and moves the register save area.
2. The argument area is never copied. The compiler must never assume that something placed in the argument area is still there across a call to this CWI.
3. The address of this CWI will be resolved like other C-RTL functions for XPLINK (using a side deck). There will not be a stub for non-XPLINK.
4. If there is not sufficient room in the current stack segment, this routine will pass control to the CEL CWI CEEVXPAL which will handle stack expansion.
5. It is the responsibility of the caller to calculate the address of the allocated storage. The allocated storage is located immediately following the argument area. The reason for this is that the compiler, which will know the size of the argument area, can generate more efficient code to perform the calculation.
6. The Vendor Interfaces header file, `<edcwcwi.h>`, is located in member EDCWCCWI of the SCEESAMP data set. To include `<edcwcwi.h>` in an application, the header file must be copied into a PDS or into a directory in the z/OS UNIX file system where the z/OS XL C/C++ compiler will find it.

XPLINK compatibility stack swapping services

This section describes the services provided to allow non-XPLINK and XPLINK routines to run on the correct upward- or downward-growing stack.

CEEVROND — run on downward-growing stack

This CWI is invoked from a non-XPLINK routine running on the upward-growing stack. It is used to invoke an XPLINK routine that runs on the downward-growing stack. It performs stack swapping, moves the parameter list, and loads appropriate parameters into registers before invoking the routine. After the routine returns, it will adjust the return value and swap the stacks back.

This CWI has two entry points: +0 for standard linkage and +16 for FASTLINK linkage.

Linkage	Input/Output	Register	Used for
Standard	Input Registers	R0	Function descriptor
		R1	Parameter List
		R2 - R11	Not used
		R12	CAA address
		R13	Caller's DSA
		R14	Return address
		R15	Entry point (CEEVROND)
	Output Registers	R0	Extended return value
		R1 - R14	Unchanged
		R15	Return value
	FASTLINK	Input Registers	R0
R1 - R3			Parameters
R4 - R11			Not used
R12			CAA address
R13			Caller's DSA
R14			Return address
R15			Entry point (CEEVROND)
NAB + BC			Parameter list
Output Registers		R0	Not preserved
		R1 - R3	Return value
		R4 - R14	Unchanged
		R15	Not preserved

CEEVROND

Call this CWI interface as follows:

```
L    15,CEECAACELV-CEECAA(,12)  Address of CAA in R12
L    15,3408(,15)
BALR 14,15
```

This CWI will return control to its invoker at the return address:

```
BR   14
```

Function descriptor

A 24 byte function descriptor that contains the environment and entry point for the XPLINK routine to be invoked at offset +16 ('10'x).

Parameter List

The parameter list for the routine to be invoked.

Note: CEEVROND has a stub called @@ROND.

CEEVRONU — run on upward-growing stack

This CWI is invoked from an XPLINK routine running on the downward-growing stack. It is used to invoke a non-XPLINK routine that runs on the upward-growing stack. It performs stack swapping, moves the parameter list, and stores appropriate parameters from registers before invoking the routine. After the routine returns, it will adjust the return value and swap the stacks back.

Input/Output	Register	Used for
Input Registers	R0	Not used
	R1 - R3	Parameters
	R4	Caller's stack pointer
	R4 + 2112	Parameter list
	R5	Function descriptor
	R6	Entry point (CEEVRONU)
	R7	Return address
	R8 - R11	Not used
	R12	CAA address
	R13- R15	Not used
Output Registers	R0	Not preserved
	R1 - R3	Return value
	R4	Unchanged
	R5 - R6	Not preserved
	R7 - R15	Unchanged

CEEVRONU

Call this CWI interface as follows:

```

|      L      6,CEECAALEOV-CEECAA(,12)
|      L      6,272(,6)
|      BASR   7,6
|      DC     X'4700'
|      DC     Y(signed offset/8 to entry marker or call descriptor)

```

This CWI will return control to its invoker at the return address:

```
BR      7
```

Function descriptor

A function descriptor that contains the entry point and writable static area address for the non-XPLINK routine to be invoked, or the actual function entry point.

Parameter List

The parameter list for the routine to be invoked.

Note: CEEVRONU has a stub called @@RONU.

CEEVH2OS — XPLINK to OS linkage on upward-growing stack

This CWI is invoked from an XPLINK routine running on the downward-growing stack. It is used to invoke an OS linkage routine that runs on the upward-growing stack. It performs stack swapping, moves the parameter list, and stores appropriate parameters from registers before invoking the routine. After the routine returns, it will adjust the return value and swap the stacks back.

```

|
| This CWI can not be used to invoke a routine that requires a call descriptor with
| non-zero return adjust field or parameter descriptor fields. Use the CWI
| CEEVRONU instead, see "CEEVRONU — run on upward-growing stack" on page
| 219.

```

Input/Output	Register	Used for
Input Registers	R0	Not used
	R1 - R2	Must be zero
	R3	Entry point of OS linkage routine
	R4	Caller's stack pointer
	R4 + 2124	OS style parameter list
	R5	Not used
	R6	Entry point (CEEVH2OS)
	R7	Return address
	R8 - R11	Not used
	R12	CAA address
	R13- R15	Not used
Output Registers	R0	Not preserved
	R1 - R2	Not preserved
	R3	Return value
	R4	Unchanged
	R5 - R6	Not preserved
	R7 - R15	Unchanged

CEEVH2OS

Call this CWI interface as follows:

```
L      6,CEECAACELV-CEECAA(,12)
L      6,3444(,6)
BASR   7,6
DC     X'4700'
DC     Y(signed offset/8 to entry marker)
```

This CWI will return control to its invoker at the return address:

```
BR     7
```

OS Style Parameter List

The parameter list for the routine to be invoked.

Note: CEEVH2OS has two stubs -- @D2U@OS and @D2U@C

__stack_info() - stack segment ranges

The `__stack_info()` CWI returns the stack segment information for a specific thread owned by the caller. The stack information returned is the beginning and ending address of each stack segment. The beginning and ending address of each stack segment will be adjusted to include only the stack frames on the active stack. If `__stacktop`, which is the address of the top of the stack, is not null, the last stack segment returned will be the one containing the stack frame pointed to by the `__stacktop`. Only information about the user stack is returned.

The caller must provide the storage that Language Environment will use to return a structure that contains the information about the stack segments that comprise the user stack. If the storage provided is insufficient to contain all of the stack segment addresses, the CWI will fail and return information about the minimum

`__stack_info()`

number of bytes required to store the segment information. Also, the caller must also supply a null pointer as the second parameter to the CWI.

Syntax

```
#define_OPEN_THREADS
#include <edcwccwi.h>
```

```
int __stack_info (struct StackInfo *StackSegmentInfo, struct __thdq *thdq)
```

```
struct StackInfo *StackSegmentInfo
```

The storage for this StackInfo structure is provided by the caller of the CWI.

The caller must supply the values of `__structsize` and `__stacktop` in the StackInfo structure. The StackInfo structure parameters are defined as follows:

`__structsize`

The total number of bytes of storage provided by the user for the StackInfo structure.

`__numsegs`

The total number of stack segments belonging to this thread that have been returned in this invocation of this CWI.

`__stacktop`

Zero or the address of the stack frame at which to end the search. If the `__stacktop` is zero, the stack is scanned from the top to the bottom of the stack. If it is non-zero, the stack is scanned from the specified stack frame until the bottom is reached.

`__startaddr`

This address is the beginning, the numerically-lowest bound address, of the stack segment. For an upward-growing stack, this is the address of the beginning of the segment. For a downward-growing stack, this will be the last byte used within the segment.

`__endaddr`

This address is the end, the numerically-highest bound address, of the stack segment. For an upward-growing stack, this is the last byte used within the segment. For a downward-growing stack, this will be the address of the end of the stack segment.

`__segtype`

This indicates if the stack is upward-growing or downward-growing. The allowed values are:

- `__EDCWCCWI_UP` for a upward-growing stack
- `__EDCWCCWI_DOWN` for a downward-growing stack

If Language Environment cannot determine the top of the stack, the `__endaddr` field will contain the end address of the last segment. When a thread has more than one stack, the stack segment information will be returned for both the downward-growing stack and the upward-growing stack. It will begin with the initial stack segment, which contains the first, that is, the oldest, stack frame allocated, and end with the stack segment containing the most recent stack frame (or the segment containing the stack frame pointed to by `__stacktop`).

```
struct __thdq *thdq
```

A null pointer, which indicates that the caller is requesting information about its own thread.

Returned Values:

- If successful, __stack_info() returns zero.
- If unsuccessful, __stack_info() returns:
 - - 1, when errno is set to EINVAL or EMVSERR
 - a number greater than zero, when errno is set to ENOMEM

EINVAL

This error indicates that an invalid thread ID or *__stacktop* has been supplied by the user.

EMVSERR

This error indicates that an MVS internal error has occurred.

ENOMEM

This error indicates that the storage provided by the user to store the stack segment information is not large enough to hold the information. In this case, __stack_info() returns the minimum number of bytes required to hold all the information.

Usage Note: The Vendor Interfaces header file, <edcwccwi.h>, is located in member EDCWCCWI of the SCEESAMP data set. To include <edcwccwi.h> in an application, the header file must be copied into a PDS or into a directory in the z/OS UNIX file system where the z/OS XL C/C++ compiler will find it.

Saving the stack pointer

Language Environment provides two fields, CEECAA_SAVSTACK and CEECAA_SAVSTACK_ASYNC, where the stack pointer can be saved.

For either field, when the stack pointer does not point to the stack, the user code must not use Language Environment interfaces, nor invoke a routine that uses Language Environment interfaces. This includes implicitly referencing a DLL.

CEECAA_SAVSTACK

This field can be used by an application or a compiler to save the stack pointer before calling a routine by using OS_NOSTACK linkage. After the call returns, the CEECAA_SAVSTACK field must be set back to zero.

The value in CEECAA_SAVSTACK is used as the current stack frame in the following conditions:

1. The Language Environment ESPIE exit routine, ESTAE exit routine, or signal interface routine (SIR) gets control.
2. The value in CEECAA_SAVSTACK is not zero.

For asynchronous signal processing, typically the interrupt PSW is outside the routine that owns the stack frame and the signal is put back.

The c macro `__LE_SAVSTACK_ADDR`, which is defined in the sample header file, <edcwccwi.h>, is the address of the CEECAA_SAVSTACK field.

CEECAA_SAVSTACK_ASYNC

This field can be used by applications that have large sections of code that does not require access to the Language Environment stack but can benefit from having an additional register available. The CEECAA_SAVSTACK_ASYNC field is a pointer to the field where the stack pointer will be saved. Language Environment initializes CEECAA_SAVSTACK_ASYNC to zero. The application needs to set up the field where the stack pointer will be saved and store the address of that field in CEECAA_SAVSTACK_ASYNC. The storage for the field must be in the application key and persist for the life of the thread.

`__stack_info()`

When the application sets `CEECAA_SAVSTACK_ASYNC`, appropriate action needs to be taken if `CEECAA_SAVSTACK_ASYNC` is not zero. Because it is possible to directly access the field where the stack pointer will be stored, consider the consequences if some part of the application is doing so.

Whenever the Language Environment stack is being used, either `CEECAA_SAVSTACK_ASYNC` must be zero or the field pointed to by `CEECAA_SAVSTACK_ASYNC` must be zero.

The value in the field pointed to by `CEECAA_SAVSTACK_ASYNC` is used as the current stack frame in the following conditions:

1. The Language Environment ESPIE exit routine, ESTAE exit routine, or signal interface routine (SIR) gets control.
2. `CEECAA_SAVSTACK_ASYNC` is not zero.
3. The value in the field pointed to by `CEECAA_SAVSTACK_ASYNC` is not zero.

For asynchronous signal processing, the signal is always handled as if the interrupt PSW was inside the routine that owns the stack frame.

The c macro `__LE_SAVSTACK_ASYNC_ADDR` defined in sample header file `<edcwcwi.h>` is the address of the `CEECAA_SAVSTACK_ASYNC` field.

Chapter 5. Condition representation

This chapter describes the format and use of condition representation within Language Environment. Conditions can be defined in many ways. Some examples are hardware- or software-detected events (which might or might not be critical for the application to run properly), asynchronous events, or the completion of a unit of work (successfully or unsuccessfully).

Systems communicate information about conditions in a variety of ways. Return and condition codes are examples of condition information. Also, common usage is almost nonexistent in representing or communicating these conditions across IBM products or platforms. Therefore, Language Environment is required to define a consistent data type to represent conditions and communicate information about them to enable ILC and cross-system source code portability of applications. The methodology presented here is required for the representation and communication of condition-related information:

- As a feedback code (return information) from all Language Environment callable services
- As input to the Language Environment condition manager
- As input to the Language Environment message services

Condition representation model

A condition in Language Environment is communicated with a 12-byte (96-bit) condition token data type. The return information (feedback code) from a Language Environment callable service is an instance of this data type. The advantages of the condition token data type include:

- A condition handler can be established to process return information from called services, thus freeing the programmer from coding **invoke then check** calls. Instead, a centralized location handles return information.
- The shared data type ties together the Language Environment callable services, condition management, and message services components of Language Environment.
- A message that can be displayed or logged in a file is associated with each instance of a condition.
- As a feedback code, the data type can be stored or logged for later processing (if the message associated with the feedback code has inserts, the message must be obtained before it is saved).
- Symbolic names can be equated to defined feedback codes and hardware conditions for those languages which support symbolic names.

The format of the condition token data type allows four different cases, or types, of conditions to be represented. Two of the four types are **cross-system consistent**. The other two are reserved for future expansion or describe platform-specific conditions.

All Language Environment callable services use this condition token data type to return information as a feedback code.

Condition Representation

The condition token data is input to Language Environment condition management to reduce the amount of overhead and the lack of completeness associated with the traditional call method of **invoke then check**. The input method has the following characteristics:

- The caller has the option of passing an address parameter for a feedback code on the call statement to the service.
- A feedback code is returned to the caller if the address parameter is supplied and the result of the service is not critical.
- Critical conditions (severity = 4) are always signaled to the Language Environment condition manager.
- The called service signals the condition manager passing the condition token if the parameter is not supplied and the result of the called service is not totally successful.
- The service returns if the parameter is not supplied and the result of the called service is completely successful.

Note: Language Environment-enabled languages must allow optional parameters on their call statements to use the optional parameter method. In the case where a language does not allow optional parameters, the feedback code parameter is always coded by the caller. Optional parameters are supported by passing a zero by value in the parameter address list. When the optional parameter is the last parameter in the parameter address list, Language Environment tolerates the high order bit being on.

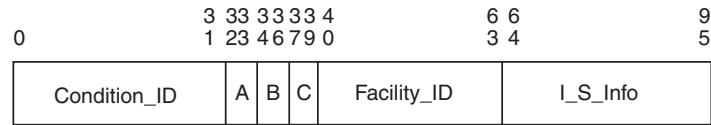
Data objects

Language Environment condition representation data objects are defined in this section.

Condition token data type (CEECTOK)

The CEECTOK communicates with message services, condition management, Language Environment callable services, and user applications; Figure 60 on page 227 shows the layout.

Condition Representation



A = Case
 B = Severity
 C = Control

Cases of Condition_ID are:

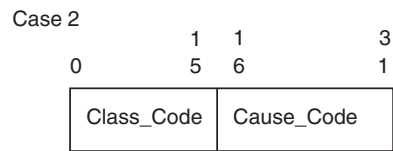
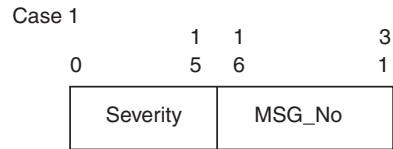


Figure 60. Language Environment Condition token (CEECTOK)

An instance of a CEECTOK can be built dynamically by the callable service CEENCOD or, more typically, constructed statically. An instance of a condition token is 12 bytes (96 bits) long, as shown in Figure 61 on page 228.

Condition Representation

```
CEECTOK      DSECT
CONDITION_ID DS  0F
*
* Case 1 definitions for CONDITION_ID
*
SEVERITY      DS  H      Condition severity (0-4)
MSG_NUMBER    DS  H      Related message number
*
* Case 2 definitions for CONDITION_ID
*
          ORG  CONDITION_ID
CLASS_CODE    DS  H      Message associated with the class
CAUSE_CODE    DS  H      Message associated with the cause
*
* Common part of the feedback code
*
FLAGS         DS  X      Bits for Case/Severity/Control
*
* Case definitions
*
          B'xx.....'
CASE1         EQU  B'01000000'
CASE2         EQU  B'10000000'
*
* Severity definitions
*
          B'...xxx...'
SEV0          EQU  B'00000000'  Severity 0 condition
SEV1          EQU  B'00001000'  Severity 1 condition
SEV2          EQU  B'00010000'  Severity 2 condition
SEV3          EQU  B'00011000'  Severity 3 condition
SEV4          EQU  B'00100000'  Severity 4 condition
*
* Control definitions
*
          B'.....xxx'
IBM_ASSIGN    EQU  B'00000001'  IBM assigned the facility id
CTL_RSVD1     EQU  B'00000010'  Reserved - must be 0
CTL_RSVD2     EQU  B'00000100'  Reserved - must be 0
*
* Facility ID
*
FACILITY_ID   DS  CL3      3 char string that ids the product
*
* Instance Specific Information Token
*
I_S_Info      DS  F      Token to the ISI
```

Figure 61. Condition token

CONDITION_ID

A 4-byte identifier that describes the condition with the FACILITY_ID. The case field determines the type of identifier. Two identifiers are defined to be CSC:

1. **Case 1 - Service Condition**, which is used by all Language Environment callable services and most application programs.

```
struct Condition_ID {
    INT2 Severity;
    INT2 Msg_No;
};
```

Severity

A 2-byte binary integer with the following possible values:

- | | |
|---|--|
| 0 | Information only (or, if the entire token is zero, no information). |
| 1 | Warning — service completed, probably correctly. |
| 2 | Error detected — correction attempted; service completed, perhaps incorrectly. |
| 3 | Severe error — service not completed. |

- 4 Critical error — service not completed; condition signaled.

Although the field is capable of containing other values, these are not architected. If a critical error (severity = 4) occurs during a Language Environment callable service, it is always signaled to the condition manager, rather than returned synchronously to the caller.

Msg_No

A 2-byte binary number that identifies the message associated with the condition. The combination of Facility_ID and Msg_No uniquely identifies a condition.

2. **Case 2 - Class/Cause Code Condition**, which is used by some operating systems and compiler runtime libraries.

```
struct Condition_ID {
    INT2 Class_Code;
    INT2 Cause_Code;
};
```

Class_Code

A 2-byte, binary number that identifies the message subid associated with the **class** of the condition.

Cause_Code

A 2-byte, binary number that identifies the message ID associated with the **cause** of the condition.

Note: The message subid and the message identifier are tags found in the message source file.

Facility_ID

A 3-character, alphanumeric string that identifies a product or component within a product. Note that special characters, including space, cannot be used. The Facility_ID is associated with the repository (for example, a file) of the runtime messages. The conventions for naming the message repository, however, are platform-specific. The Facility_ID need not be unique within the system and can be determined by the application writer. If a unique ID is required (for IBM and non-IBM products), an ID can be obtained by contacting an IBM project office.

A Facility_ID assigned by IBM to an IBM product must begin with one of the letters A through I, inclusive. A Facility_ID assigned by IBM to a product other than an IBM's must not begin with a letter A through I. For information on how to indicate if the Facility_ID has been assigned by IBM, see Control below. There are no constraints (other than the alphanumeric requirement) on a Facility_ID not assigned by IBM.

Language Environment constructs a load name consisting of the form **T | | Facility_ID | | MSGT**:

- T** The character 'T' if the Facility_ID was assigned by IBM, or the character 'U' if the Facility_ID was **not** assigned by IBM.

Facility_ID

The three character facility ID as described above.

MSGT

The four characters MSGT.

For example, given an IBM assigned facility ID of CEE, the constructed load name would be ICEEMSGT.

Condition Representation

Note: The `Msg_No/Facility_ID` identifies a condition for a Language Environment-enabled product. This identification is required to be persistent beyond the scope of a single session. This allows the meaning of the condition and its associated message to be determined after the session that produced the condition has ended. The message inserts and the `I_S_Info` need to be explicitly saved to allow persistence after the session has concluded.

Case A 2-bit field that defines the format of the `Condition_ID` portion of the token. The value 1 identifies a case 1 condition, the value 2 identifies a case 2 condition. The values 0 and 3 are reserved.

Severity

A 3-bit field indicating a condition's severity. Severity values are the same as defined under a case 1 `Condition_ID`. When evaluating the severity, the same rules apply for signaling case 2 conditions as for case 1 conditions. For a case 1 condition, this field contains the same value as the `Severity` field in the `Condition_ID`.

Note: This field is valid for both case 1 and 2 conditions. It can be used with either condition token to evaluate the condition's severity.

Control

A 3-bit field containing flags describing or controlling various aspects of condition handling, as follows:

- ..1 Indicates `Facility_ID` has been assigned by IBM.
- .1 Reserved.
- 1.. Reserved.

ISI A fullword field containing a token that identifies the Instance Specific Information (ISI) associated with the given condition. If an ISI is not associated with a given condition token, the ISI field contains binary zero. The ISI token provides access to various instance specific information such as message inserts and qualifying data.

Feedback code

A feedback code is an instance of a condition token (CEECTOK). A feedback code is returned from a Language Environment service call if the caller has passed a reference to an area to hold it. To test a feedback code for equivalence, the first 8 bytes should be compared because they are static. The last four bytes can change from instance to instance.

CEEGETFB — Construct a condition token given a facility ID and a message number

Purpose

CEEGETFB is an S/370-specific CWI that constructs a case 1 condition token given a facility identifier and a message number. The severity is retrieved from the appropriate message file containing the message number.

Syntax

```
void CEEGETFB (facility_id, message_no, cond_token, [fc])
CHAR3      *facility_id;
INT4       *message_no;
CEECTOK    *cond_token;
FEED_BACK  *fc;
```

CEEGETFB

Call this CWI interface as follows:

```
L   R12,A(CAA)           Get the address of CAA in R12
L   R15,CEECAACELV-CEECAA(,R12)
L   R15,2816(,R15)
BALR R14,R15
```

facility_id (input)

The 3-character facility identifier that is placed into the resulting condition token. It is used to determine the file containing the message definition and message text.

message_no (input)

A 4-byte binary integer representing the message number for the resulting condition token.

cond_token (output)

A case 1 style 12-byte condition token (CEECTOK) that is constructed from *facility_id*, *message_no*, and the severity, which is obtained from the appropriate file containing the message definition. The I_S_Info field is set to binary zero.

fc (output/optional)

A 12-byte feedback code passed by reference. If specified as an argument, feedback information (condition token) is returned to the calling routine. If not specified as an argument and the requested operation was not successfully completed, the condition is signaled to the condition manager. The following symbolic conditions can result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE0CJ	Severity	3
	Msg_No	0403
	Message	Invalid severity code found.
CEE0EA	Severity	3
	Msg_No	0458
	Message	The message repository cannot be located.
CEE3CT	Severity	3
	Msg_No	3485
	Message	<i>message_no</i> was not found within the library specified.

CEEGETFB

Chapter 6. National language support and message handler

This chapter describes Language Environment National Language Support (NLS) and message handling services.

National language support

The Language Environment message handler provides services to support many NLS machine readable information (MRI) requirements, such as: message formatting, message delivery, casing, folding, and normalization. The message facility formats messages for any national language known to Language Environment. Language Environment provides runtime messages for the following national languages:

- **ENU** (Mixed-case English USA)
 - Message text is made up of SBCS characters and consists of both uppercase and lowercase letters.
 - Message inserts can contain DBCS characters.
 - Long messages are split at an SBCS blank if possible or split by the output line length if a blank separator does not exist.
- **UEN** (Uppercase American English)

This is identical to the mixed-case American English language except the message text consists of uppercase letters. Message inserts can be in lowercase or might use lowercase codepoints to make use of SBCS Katakana capabilities.
- **JPN** (Japanese)

This language supports devices that have both DBCS and SBCS capabilities; its characteristics are:

 - Message text can be made interchangeably of SBCS and DBCS characters.
 - If a long message extends beyond the print line and the text is SBCS, it is split at a blank when possible. If a blank separator does not exist, text is split by the output line length. If the text is DBCS, the message is split at a DBCS blank if possible. If a blank separator does not exist, it is split at the last DBCS character that allows a shift-in to be inserted. The next line begins with a shift-out character.

The national language can be set using the NATLANG runtime option or the CEE3LNG callable service. One current language is maintained at the enclave level and remains in effect until it is changed. For example, if JPN is specified in the NATLANG runtime option but ENU is later specified by the CEE3LNG callable service, ENU is considered the current national language. If the message text is not available for the current national language setting, the system-level or region-level default is used instead.

The current value of the COUNTRY runtime option controls the following values:

- Date format
- Time format
- Currency symbol
- Decimal separator character
- Thousands separator

The value can be set by the COUNTRY runtime option or by the CEE3CTY callable service. The IBM-supplied default COUNTRY(US) indicates the default country is USA.

Introduction to Language Environment message services

Language Environment provides message handling services to format and deliver runtime messages. The following items are described in this section:

- The format of the message source files
- How to create a loadable message library
- How to establish inserts for messages
- How to format a message
- How to deliver the message to a given destination

The Language Environment message services can be divided into two categories:

- Cross System Consistent (CSC) interfaces
- Compatibility interfaces

The CSC interfaces are callable services. The CSC-callable services supported in Language Environment are:

CEEMOUT

Dispatches a message string to the platform's defined output device.

CEEMSG

Given a condition token, this service gets, formats, and dispatches a message string to the defined output device.

CEEMGET

Gets, formats, and stores a formatted message in a buffer.

The CSC-CWI interface supported in Language Environment is:

CEECMIB

Populates a feedback token with an ISI.

The compatibility interfaces provided are listed below. The services are provided to manipulate the insert area and to dispatch a message.

CEEXMGET

Obtain an insert block.

CEEXMDFL

Populate all inserts with a default.

CEEXMFRE

Release an insert area.

CEEXMINS

Place an insert into the insert area.

CEEXMFMT

Format a message into a user specified buffer.

CEEXMOUT

Dispatch a message to a specified destination.

CEEMFNDM

Given a feedback token, return the pointer to the ISI.

MSGFILE — related CWIs

Language Environment provides some message services that aid the HLLs in mapping their message files to Language Environment's MSGFILE.

CEECL0S — close ddname

Purpose

The CEECL0S CWI closes the specified ddname.

Syntax

```
void CEECL0S (ddname, [fc])
```

```
CHAR8      *ddname;
FEED_BACK *fc;
```

CEECL0S

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L      R15,2924(,R15)
BALR   R14,R15
```

ddname (input)

An 8-character fixed-length string, left-justified and right-padded, containing the ddname that should be closed. If Language Environment owns the related DCB, Language Environment closes the file. If the ddname is blank, then the current MSGFILE ddname is used.

fc (output/optional)

The feedback code passed by reference. The following conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3C5	Severity	1
	Msg_No	3484
	Message	The file was not currently open.
CEE3D5	Severity	3
	Msg_No	3493
	Message	Language Environment did not own the specified ddname's DCB.
CEE3D6	Severity	3
	Msg_No	3494
	Message	Uncorrectable I/O error encountered while closing the file.

CEEODMF — open an input ddname

Purpose

The CEEODMF CWI opens an input ddname.

Syntax

```
void CEEODMF (ddname,[fc])
```

```
CHAR8      *ddname;
FEED_BACK *fc;
```

CEEODMF

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L      R15,3988(,R15)
BALR   R14,R15
```

ddname (input)

An 8-character fixed-length string, left-justified and right-padded, containing the ddname to be opened.

fc (output/optional)

The feedback code passed by reference. The following conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3DA	Severity	1
	Msg_No	3498
	Message	The MSGFILE was already open.
CEE3DB	Severity	3
	Msg_No	3499
	Message	The MSGFILE could not be opened.

CEEOPMF — open the MSGFILE ddname

Purpose

The CEEOPMF CWI opens the current MSGFILE ddname.

Syntax

```
void CEEOPMF ([fc])
```

```
FEED_BACK *fc;
```

CEEOPMF

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L      R15,2984(,R15)
BALR   R14,R15
```

fc (output/optional)

The feedback code passed by reference. The following conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3DA	Severity	1
	Msg_No	3498
	Message	The MSGFILE was already open.
CEE3DB	Severity	3
	Msg_No	3499
	Message	The MSGFILE was unable to be opened.

CEEQDMF — query an input ddname

Purpose

CEEQDMF returns the status of the file, the effective LRECL if the file is open, and the file descriptor, if the file is in the POSIX file system, for an input ddname.

Syntax

```
void CEEQDMF (ddname, status, elrecl, fdesc, [fc])
```

```
CHAR8      *ddname;
INT4       *status;
INT4       *elrecl;
INT4       *fdesc;
FEED_BACK *fc;
```

CEEQDMF

Call this CWI interface as follows:

```
L   R15,CEECAACELV-CEECAA(,R12)   Address of CAA in R12
L   R15,3984(,R15)
BALR R14,R15
```

ddname (input)

An 8-character fixed-length string, left-justified and right-padded, containing the ddname to be queried.

status (output)

A fixed-binary(31) integer that contains one of the following values:

- 1** The message file was already open.
- 0** The message file was not open.

elrecl (output)

A fixed-binary(31) integer that contains the effective length of the record, thus providing the number of bytes available for character data. If the file is not open, the *elrecl* is set to zero.

fdesc (output)

A fixed-binary(31) integer that contains the file descriptor of the Language Environment message file if it is in the POSIX file system; otherwise this field contains a value of -1.

fc (output/optional)

The feedback code passed by reference. The following condition can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.

CEEQUMF — query the MSGFILE ddname

Purpose

This CWI returns the current MSGFILE ddname, status of the file, the effective LRECL if the file is open, and the file descriptor if the file is in the POSIX file system.

Syntax

void CEEQUMF (*ddname, status, elrecl, fdesc, [fc]*)

```
CHAR8    *ddname;
INT4     *status;
INT4     *elrecl;
INT4     *fdesc;
FEED_BACK *fc;
```

CEEQUMF

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L    R15,2928(,R15)
BALR R14,R15
```

ddname (output)

An 8-character fixed-length string, left-justified and right-padded, containing the current MSGFILE ddname.

status (output)

A fixed-binary(31) integer that contains one of the following values:

- 1** The message file was already open.
- 0** The message file was not open.

elrecl (output)

A fixed-binary(31) integer that contains the effective length of the record. Thus providing the number of bytes available for character data. If the file is not open, the *elrecl* is set to zero.

fdesc (output)

A fixed-binary(31) integer that contains the file descriptor of the Language Environment message file if it is in the POSIX file system, otherwise this field contains a value of -1.

fc (output/optional)

The feedback code passed by reference. The following condition can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.

CEECHMF — change the MSGFILE ddname

Purpose

The CEECHMF CWI allows the specified ddname to become the new MSGFILE ddname.

Syntax

```
void CEECHMF (ddname, [fc])
```

```
CHAR8      *ddname;
FEED_BACK  *fc;
```

CEECHMF

Call this CWI interface as follows:

```
L   R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L   R15,2932(,R15)
BALR R14,R15
```

ddname (input)

An 8-character fixed-length string, left-justified and right-padded, containing the ddname that becomes the new MSGFILE ddname.

fc (output/optional)

The feedback code passed by reference. The following conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.

Usage notes

Note:

1. The OCB is not updated by this service.
2. The ddname is not validated by this service.
3. The ddname is not opened at this time. It is opened at the first request to write to the ddname.
4. When the Message File ddname is changed using this service, it does not inherit the ENQ/NOENQ characteristic of the ddname specified on the MSGFILE runtime option.

Relationship between date/time and COUNTRY settings

Some date/time callable services allow the specification of a blank or null picture string. This directs Language Environment to use the current country value to obtain the default picture string for the date or time. The names of the months and days of the week are obtained based upon the current national language value. It is obtained from the national language message's file, as selected by the NATLANG runtime option. The message numbers assigned to the days of the week and the months are in Figure 62 on page 240.

CEE0001 - JANUARY	CEE0021 - january
CEE0002 - FEBRUARY	CEE0022 - february
CEE0003 - MARCH	CEE0023 - march
CEE0004 - APRIL	CEE0024 - april
CEE0005 - MAY	CEE0025 - may
CEE0006 - JUNE	CEE0026 - june
CEE0007 - JULY	CEE0027 - july
CEE0008 - AUGUST	CEE0028 - august
CEE0009 - SEPTEMBER	CEE0029 - september
CEE0010 - OCTOBER	CEE0030 - october
CEE0011 - NOVEMBER	CEE0031 - november
CEE0012 - DECEMBER	CEE0032 - december
CEE0013 - SUNDAY	CEE0033 - sunday
CEE0014 - MONDAY	CEE0034 - monday
CEE0015 - TUESDAY	CEE0035 - tuesday
CEE0016 - WEDNESDAY	CEE0036 - wednesday
CEE0017 - THURSDAY	CEE0037 - thursday
CEE0018 - FRIDAY	CEE0038 - friday
CEE0019 - SATURDAY	CEE0039 - saturday

Figure 62. Message numbers assigned to the days of the week and months

Message handling services

This section describes the message handling CWIs CEECMIB and CEEMFNDM.

CEECMIB — create a message insert area entry

Purpose

The CEECMIB CWI provides a mechanism by which an MIB can be populated; an MIB is managed by Language Environment. The number of ISIs per thread is determined by the MSGQ(x) runtime option. MIBs are released when CEEMSG issues the message, or when the MSGQ(n) runtime option is exceeded. The least recently used MIB is overwritten.

Syntax

void CEECMIB (*cond_rep*, *Insert_Seq_Num*, *Insert_Data*, [*fc*])

```
FEED_BACK *cond_rep;  
INT4      *Insert_Seq_Num;  
VSTRING   *Insert_Data;  
FEED_BACK *fc;
```

CEECMIB

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECOA(,R12)  Address of CAA in R12  
L    R15,2748(,R15)  
BALR R14,R15
```

cond_rep (input)

A condition token defining the condition for which the Q_Data-Token is to be retrieved.

Insert_Seq_Num (input)

A 4-byte integer containing the insert sequence number (for example, insert 1, insert 2). It corresponds to that specified with the **ins** tag in the message source file.

Insert_Data (input)

The insert data. The data type is a halfword-prefixed fixed-length string. The entire length that is described in the halfword prefix is used without truncation. DBCS needs to be enclosed within SO/SI.

fc (output/optional)

A condition token which can return the following conditions:

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE0EB	Severity	3
	Msg_No	0459
	Message	An invalid MIB Sequence number was found.
CEE0H9	Severity	3
	Msg_No	0553
	Message	An invalid <i>insert_seq_num</i> was found.

CEEMFNDM — return the MIB address

Purpose

The CEEMFNDM CWI returns the MIB address given a feedback token.

Syntax

```
void CEEMFNDM (FB_token, MIB_Addr, [fc])
```

```
FEED_BACK *FB_token;
POINTER *MIB_Addr;
FEED_BACK *fc;
```

CEEMFNDM

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L    R15,2868(,R15)
BALR R14,R15
```

FB_token (input)

The 12-byte feedback token returned from a callable service.

MIB_Addr (output)

The address of the MIB for this condition.

fc (output/optional)

A 12-byte feedback code passed by reference. Feedback information (condition token) is returned to the calling routine. The following conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.

Condition		
CEE3D8	Severity	1
	Msg_No	3496
	Message	The MIB was not found.

Usage notes

Once the MIB is obtained, message inserts can also be located. The procedure for finding the message insert information is described below. Figure 63 on page 243 represents the access to message insert information.

1. Offset X'0' into the MIB is an EBCDIC eyecatcher "CMIB".
2. Offset X'24' into the MIB points to an array of 9 pointers of message insert data. If the pointer is 0, this insert is not used. If a pointer is non-zero, this points to the message insert data in EBCDIC.
3. Offset X'20' into the MIB points to an array of 9 quadwords of message insert information. The fourth word (the last word) contains the length of the message insert data.

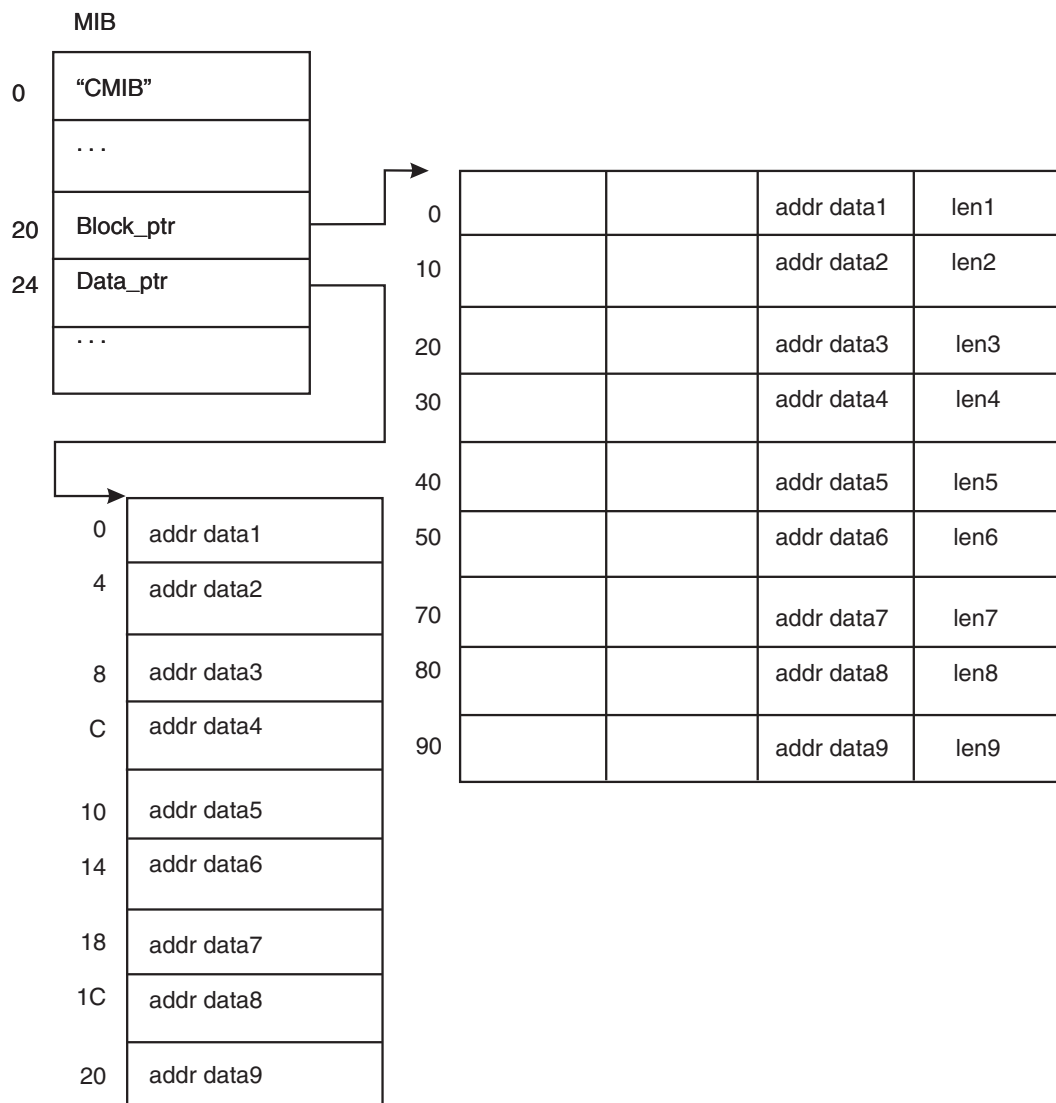


Figure 63. Access to message insert information

CEE3SMO — suppress printing of messages

Purpose

CEE3SMO is a callable service that suppresses the printing of any message, traceback, and dump (as indicated by the `TERMTHDACT` option) for any condition that has been signaled and allowed to percolate. This service must be called by a user condition handler.

Syntax

```
void CEE3SMO ([fc]);
```

```
FEED_BACK *fc;
```

fc (output/optional)

A 12-byte feedback code passed by reference. Feedback information (condition token) is returned to the calling routine. The following conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3B0	Severity	3
	Msg_No	3424
	Message	CEE3SMO called from outside a condition handler. This condition is signaled when there is only one CIBH in the CIBH chain and it is not in use.

C/C++-specific vendor interfaces

This section describes the C/C++-specific vendor interfaces.

__cttbl() — returns address of **_LC_ctype_t** structure Header

The `_LC_info.h` header file contains definitions for the `__cttbl()` function.

Standards

Standards/Extensions	C or C++	Dependencies
Language Environment	both	None

Syntax

```
#include <localdef.h>
#include <_LC_info.h>

_LC_ctype_t * __cttbl(void);
```

General description

This function provides the location of the `CTYPE` class, which defines character membership in a character class.

Return values

`__cttbl()` returns a pointer of type `_LC_ctype_t` which is defined in `<localdef.h>`.

ASCII/EBCDIC mixed mode support for enhanced ASCII C-RTL

ASCII/EBCDIC bimodal support for enhanced ASCII facilitates the development of bimodal C++ libraries and DLLs. Bimodal class libraries and DLLs eliminate the need for the development, maintenance and distribution of separate ASCII and EBCDIC class libraries.

The end user application is not bimodal. Users of C++ class libraries or DLLs need to use either the ASCII/EBCDIC bimodal version of the library or the ASCII or EBCDIC version of the library which matches the mode of their application.

It is the responsibility of the user to ensure that the character mode of data passed as arguments to, or values returned from, function calls are in the correct character mode for the functions being used. The C-RTL does not convert arguments or return values.

Header information

Enhanced ASCII support requires that all headers required by all Enhanced ASCII functions used in an application be included. Enhanced ASCII support uses headers to dynamically map generic function calls such as `printf()` to either an ASCII version of `printf()` or an EBCDIC version of `printf()` based on how the application was compiled. Additionally, the headers dynamically map explicit ASCII or EBCDIC function calls such as `__printf_a()` or `__printf_e()` to ASCII or EBCDIC versions of `printf()` respectively. For example, the snippet of code in `stdio.h` regarding the `printf()` function is as follows:

```
#ifndef __AE_BIMODAL_F
    #pragma map (__printf_a,  "\174\174A00118")
    #pragma map (__printf_e,  "PRINTF")
    __new4102(int, __printf_a, (const char *, ...));
    __new4102(int, __printf_e, (const char *, ...));
#endif /* __AE_BIMODAL_F */

#ifndef __NATIVE_ASCII_F
    #pragma map (printf,  "\174\174A00118")
#endif /* __NATIVE_ASCII_F */

#ifndef _NO_PROTO
    int    printf ();
#else
    int    printf (const char *, ...);
#endif /* _NO_PROTO */
```

The `__AE_BIMODAL_F` feature test is for ASCII/EBCDIC Bimodal support. The `__AE_BIMODAL_F` feature is defined in `features.h` if the application was compiled using the z/OS V1R2 C/C++ Compiler, the user compiled their code using the XPLINK compile option and `_AE_BIMODAL` was defined. If the `__AE_BIMODAL_F` feature test is satisfied, the explicit `printf()` function calls, `__printf_a()` and `__printf_e()` get pragma mapped to the ASCII and EBCDIC versions of `printf()` respectively. In addition, the prototypes for `__printf_a()` and `__printf_e()` are exposed. Similar header logic is also used for ASCII/EBCDIC Mixed Mode versions of macros and structures.

Usage example

The design point for ASCII/EBCDIC Mixed Mode support for Enhanced ASCII was to make it possible for C++ class library and DLL developers to develop a common mixed mode version of their library instead of producing separate ASCII and EBCDIC versions. The class library or DLL developer can accomplish this by calling explicit ASCII and EBCDIC versions of C-RTL functions based on the results of a call to `__isASCII()`. `__isASCII()` is used to determine the character mode of the user application. It is assumed that the end user application is not bimodal. A simple ASCII/EBCDIC bimodal "Hello World!" program shows how a bimodal class library or DLL can be produced. For this example, it is assumed that the user application (the code containing `main()`) is compiled ASCII while the bimodal code, which is contained in a separate compile unit, is compiled EBCDIC.

```
#include <stdio.h>
void printItOut(const char *, const char *);

void main(void) {
    printItOut("%s\n", "Hello World!");
}
```

Assuming the preceding code was compiled using the ASCII compile option, the C/C++ Compiler will generate values for the characters in the format string and the "Hello World!\n" string in the ISO8859-1 code page. A separate compile unit contains the bimodal printItOut() function, as shown below:

```
#define _AE_BIMODAL 1
#include <stdio.h>
#include <_Nascii.h>

void printItOut(const char *format, const char *string {
    if (__isASCII())
        __printf_a(format, string);
    else
        __printf_e(format, string);
}
```

In the example, the format and string arguments passed on the call to printItOut will be in the ISO8859-1 code page. __isASCII() returns the character mode of the current thread. In this example, the character mode of the initial processing thread is ASCII. This was set during C-RTL initialization since the compile unit containing main() was compiled ASCII.

Since __isASCII() returns the value one, __printf_a() is called, passing along the format and string arguments. The format and string arguments are encoded using ISO8859-1. Since the code in our ASCII/EBCDIC Bimodal part was compiled XPLINK and _AE_BIMODAL is defined, the __printf_a() function call is pragma mapped by stdio.h to be \174\174A00118, which is the Enhanced ASCII version of the printf() function. Hello World! in ISO8859-1 will be sent to stdout. By default, stdout is assumed EBCDIC and the Hello World! string will show up on stdout as unreadable characters. The Hello World! string will show up legibly on stdout if the application is being run with auto conversion on or the output of the "Hello World!" program is piped into iconv as follows:

```
hellow 2 >&1 | iconv -f ISO8859-1 -t IBM-1047
```

__ae_thread_setmode() — set character mode: ASCII or EBCDIC

Standards

Standards/Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1R2 or later

Syntax

```
#include <ctype.h>

void __ae_thread_setmode(int aemode);
```


General description

The __ae_thread_setmode() function sets the current thread's character mode to ASCII or EBCDIC based on the value of the argument *aemode*:

- __AE_ASCII_MODE - set thread character mode to ASCII
- __AE_EBCDIC_MODE - set thread character mode to EBCDIC

If the value for *aemode* is other than the values shown above, the thread's ASCII/EBCDIC mode will remain unchanged.

The TCP/IP resolver is reinitialized, if already initialized, in the new character mode. This function or __ae_thread_swapmode() must be used before and after calls between EBCDIC and ASCII portions of an application.

Return values

If successful, __ae_thread_setmode() changes the character mode.

If unsuccessful, __ae_thread_setmode() will terminate with either message EDC6254 or EDC6255.

There are no documented errnos for this function.

Related information

- “__ae_autoconvert_state() — returns automatic conversion state of thread” on page 249
- “__ae_thread_swapmode() — swap character mode to ASCII or EBCDIC”
- “__isASCII() — determine character mode: ASCII or EBCDIC” on page 248

__ae_thread_swapmode() — swap character mode to ASCII or EBCDIC

Standards

Standards/Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1R2 or later

Syntax

```
#include <ctype.h>

int __ae_thread_swapmode(int aemode);
```

General description

The __ae_thread_swapmode() function sets the current thread's character mode to ASCII or EBCDIC, based on the value of the argument *aemode*. If any other value is specified for *aemode*, the thread's ASCII/EBCDIC mode will remain unchanged.

- __AE_ASCII_MODE - set thread character mode to ASCII
- __AE_EBCDIC_MODE - set thread character mode to EBCDIC

The TCP/IP resolver is reinitialized, if already initialized, in the new character mode. This function or __ae_thread_setmode() must be used before and after calls

__ae_thread_swapmode()

between EBCDIC and ASCII portions of an application.

Return values

If successful, __ae_thread_swapmode() changes the character mode and returns the mode value corresponding to the thread's previous mode.

If unsuccessful, __ae_thread_setmode() will terminate with either message EDC6254 or EDC6255.

There are no documented errors for this function.

Related information

“__ae_autoconvert_state() — returns automatic conversion state of thread” on page 249

“__ae_thread_setmode() — set character mode: ASCII or EBCDIC” on page 246

“__isASCII() — determine character mode: ASCII or EBCDIC”

__isASCII() — determine character mode: ASCII or EBCDIC Standards

Standards/Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1R2

Syntax

```
#include <ctype.h>
int __isASCII(void);
```

General description

The __isASCII() function determines the current thread's character mode of ASCII or EBCDIC. If the character mode is ASCII, it returns 1. If the character mode is EBCDIC, it returns 0.

Return values

For ASCII character mode, __isASCII() returns 1.

For EBCDIC character mode, __isASCII() returns 0.

There are no documented errors for this function.

Related information

“__ae_autoconvert_state() — returns automatic conversion state of thread” on page 249

“__ae_thread_setmode() — set character mode: ASCII or EBCDIC” on page 246

“__ae_thread_swapmode() — swap character mode to ASCII or EBCDIC” on page 247

__ae_autoconvert_state() — returns automatic conversion state of thread

Standards

Standards/Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1R2

Syntax

```
#define _CVTSTATE_OFF    0
#define _CVTSTATE_ON    1

#define _CVTSTATE_SWAP  2
#define _CVTSTATE_QUERY 3

int __ae_autoconvert_state(int action);
```

_CVTSTATE_OFF

Automatic conversion for the current thread is set to OFF.

_CVTSTATE_ON

Automatic conversion for the current thread is set to ON.

_CVTSTATE_SWAP

Automatic conversion is swapped to the state opposite that of the current thread state.

_CVTSTATE_QUERY

Current thread's automatic conversion state remains unchanged (only return value is significant).

Return values

Regardless of the action argument, the returned integer value for __ae_autoconvert_state is the current thread's automatic conversion state before any changes were made based upon the action requested. This returned value will be _CVTSTATE_OFF, CVTSTATE_ON.

If the C runtime library is unable to access or set the automatic conversion state, or an invalid action argument is supplied, __ae_autoconvert_state will fail by returning -1.

Related information

- “__ae_thread_setmode() — set character mode: ASCII or EBCDIC” on page 246
- “__ae_thread_swapmode() — swap character mode to ASCII or EBCDIC” on page 247
- “__isASCII() — determine character mode: ASCII or EBCDIC” on page 248

`__ae_autoconvert_state`

Chapter 7. Condition management

This section describes what constitutes a condition in Language Environment, how Language Environment supplements existing HLL condition handling methods, and how the Language Environment condition handling model works. It describes in detail the steps involved in condition handling under Language Environment, HLL-specific condition handling considerations, Language Environment — POSIX signal handling interactions, and how you can communicate events that happen in a routine to another routine.

For a discussion of Language Environment condition handling models in the POSIX(ON) and POSIX(OFF) environments, see *z/OS Language Environment Programming Guide*.

Compiler-writer interfaces (CWIs)

Language Environment provides the following CWIs for condition management. The CWIs beginning with the “CEE” prefix are available for a non-64-bit environment only. The others are for use in both non-64-bit and 64-bit environments.

- CEE3ERP
- CEE3RSUM
- CEE3SGLN
- CEE3SGLT
- CEE3SMS
- CEE3SMS2
- CEEGOTO
- CEEHDHDL
- CEEMRCM
- CEEYDSAF
- __dsa_prev()
- __far_jump()
- __set_stack_softlimit()

CEE3ERP — support for user-provided error recovery

The CEE3ERP callable service enables user-written applications that have established their own ESTAE/ESPIE exit routines to notify Language Environment when an abend or program check occurs. With this support, Language Environment can analyze and process an error that was captured by the application's ESPIE or ESTAE exit before the error is passed to the user application.

Syntax

```
void CEE3ERP;
```

CEE3ERP

Call this CWI interface as follows:

```
L    R15,CEECAAHERP-CEECAA(,R12)
BALR R14,R15
```

R0 (output)

If ESTAE processing is in effect and register 15 contains 4, register 0 contains the retry address that the user's ESTAE exit must use for resumption; otherwise, this register can be ignored.

R1 (input)

Contains the address of the EPIE, which was passed to the ESPIE exit, or the SDWA, which is passed to the ESTAE exit.

R15 (output)

Register 15 contains a value that indicates the actions that Language Environment wants the user application's ESPIE/ESTAE exit routine to take as a result of Language Environment processing the error condition. The following values are returned in register 15:

- 4 Language Environment is not active in this environment; the application continues with its own error recovery processing.
- 0 Language Environment is not interested in the error; the application continues with its own error recovery processing.
- 4 Language Environment can handle the error. If SPIE processing is in effect, Language Environment sets up the EPIE; the user application's EPIE exit must return to the system to resume processing. If STAE processing is in effect, Language Environment sets up the SDWA for retry; the user application's ESTAE exit must retry at the address specified in register zero.
- 16 Language Environment CAA has been overlaid
- 20 Language Environment condition manager is disabled; retry the operation.

Usage Notes:

1. This service should always be used with a user's ESPIE or ESTAE exit routine, regardless of the setting of the TRAP runtime option. It must also be invoked immediately by the user's ESPIE or ESTAE exit routine, before any of its own error recovery processing.
2. This service supports AMODE 31 only.
3. This service is primarily looking for a "shunt routine". When the CEECAADMC field contains a non-zero value, a "shunt routine" is active. Language Environment will set up the EPIE or SDWA to resume or retry at the "shunt routine" address that was in the CEECAADMC field.

When Language Environment indicates for STAE processing that it is interested in the error, Language Environment would have already issue the SETRP macro to set up the SDWA for the retry. Language Environment also returns the retry address in register 0.

4. Program checks can also occur when the Language Environment XPLINK stack needs to be expanded. In this case, Language Environment sets up the EPIE or SDWA to resume or retry at an appropriate point, and sets the return code to 4.
5. This service can be used in the z/OS and pre-initialization environments. The CICS and POSIX environments are not supported.

CEE3RSUM — resume an interrupted program

CEE3RSUM is used to resume execution of an interrupted Language Environment program with a specified PSW, registers, and access registers. The resume point would normally be the point of interruption. This service does not return, so there

is no feedback token. If the program cannot be resumed with the requested PSW and registers, CEE3RSUM will cause an ABEND.

Syntax

void CEE3RSUM (*CSRL16J_parms*, *flags*, *resume_info*)

```
void *CSRL16J_parms
INT4  *flags
void *resume_info
```

CEE3RSUM

From a non-XPLINK routine, call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)  Address of CAA in R12
L   R15,120(,R15)
BALR R14,R15
```

CSRL16J_parms (input)

This is a pointer to a data area that can be passed to the CSRL16J callable service.

flags (input)

This parameter is not used, and should point to a fullword of zero bits.

resume_info (input)

This parameter points to an optional area that contains resume information; the code sample below shows the layout. If *resume_info* pointer is NULL, the result is the same as pointing to a resume information area with all validity flags off.

```
+-----+
+00 | Version = 1 |
+-----+
+04 | Validity flags: |
    | X'80000000' - valid_int_DSA field is present |
    |   On  : Indicates that the DSA to be |
    |         resumed is in the Valid interrupt |
    |         DSA field (below). |
    |   Off : Get resume DSA address from |
    |         reg 4 or reg 13 in the CSRL16J |
    |         registers (1st parm to CEE3RSUM) |
    | X'40000000' - sigset field is present |
    |   On  : Indicates that the signal mask |
    |         is to be restored to the value in |
    |         the sigset field (below) before |
    |         the resume is done. |
    |   Off : Indicates that the signal mask |
    |         is not to be restored before the |
    |         resume is done. |
    | X'20000000' - oldest_up_DSA field is present |
    |   On  : Indicates that the oldest up DSA |
    |         field is present (see below). |
    |   Off : Indicates that the active DSA |
    |         chain needs to be scanned to find |
    |         the oldest Upstack DSA. |
    | X'10000000' - hr field is present |
    |   On  : Indicates that the hr field |
    |         contains the high registers to |
    |         be restored before resuming. |
    |   Off : Indicates that the high registers |
    |         are not available in the resume |
    |         information. |
    | X'08000000' - CEECAA_SAVSTACK needs to be restored |
    |   On  : Indicates that the CEECAA_SAVSTACK |
    |         field needs to be restored from |
    |         CEERSMI_VALID_INT_DSA field |
    |         before resuming. |
    |   Off : Indicates that no restore of |
    |         CEECAA_SAVSTACK is needed. |
    | X'04000000' - CEECAA_SAVSTACK_ASYNC needs to be |
    |         restored |
+-----+
```

CEE3RSUM

	On : Indicates that the CEECAA_SAVSTACK_ASYNC field needs to be restored from CEERSMI_VALID_INT_DSA field before resuming.
	Off : Indicates that no restore of CEECAA_SAVSTACK_ASYNC is needed.
	X'03FFFFFF' (reserved -- should be 0)
+08	Valid interrupt DSA (filled in if the X'80000000' flag is set on)
+0C	(reserved - should be 0)
+08	Valid interrupt DSA (filled in if the X'80000000' flag is set on)
+0C	(reserved - should be 0)
+10 +14	Sigset -- signal mask to be restored before the resume is to be done. (filled in if the X'40000000' flag is set on)
+18	Resume DSA (reserved - should be 0) format(up=0, down=1)
+1C	(reserved -- should be 0)
+20	Oldest up DSA -- this is the oldest Upstack DSA that is not older than the DSA to be resumed. (If the DSA to be resumed is Upstack, this must be the same as the DSA being resumed.)
+24	(reserved - should be 0)
+28	(reserved - should be 0)
.	.
+3F	(reserved - should be 0)
+40	(reserved - should be 0)
+7F	(reserved - should be 0)
+80	(reserved - should be 0)
.	.
+FC	(reserved - should be 0)
+100	(reserved - should be 0)

Usage Notes:

1. When CEE3RSUM is called, the CAA stack direction must be valid and the caller must have a proper DSA chained into the Language Environment stack.
2. All fields in the CSRL16J parm area must be filled in properly. If the L16JSUBPOOL, L16JLENGHTHOFREE, and L16JAREATOFREE fields are set up, the CSRL16J area will be freed before the program is resumed (this area may not be freed up if an ABEND is declared). If the CSRL16J area is to be freed, the CEE3RSUM service uses CSRL16J rather than the RP instruction (so that the free can be done by directly by z/OS).
3. If the area to free includes any part of the CSRL16J parameter area, this parm area must not lie in any DSA on the Language Environment stack. (It must be in GETMAINed storage.) If there is no area to free, or the area to free does not include any part of the CSRL16J parm area, this area may lie in a DSA on the Language Environment stack (including a DSA that will be freed up when the resume occurs).
4. If the *valid_interrupt_dsa* field in the resume information area is not filled in, register 4 or 13 in the CSRL16J parms must point to a valid XPLINK or non-XPLINK Language Environment DSA on the stack. Register 4 or 13 may point to a transitional or overflow stack frame, but the PSW and registers

must not point back to a place where the stack direction in the CAA is not valid. If these registers are not valid, ABEND 4091-42 may occur.

If the *valid_interrupt_dsa* field in the resume information area is filled in, reg 4 or 13 in the resume registers does not need to point to a valid DSA. However, if the *valid_interrupt_dsa* is not correct, the same ABEND 4091-42 may occur.

5. The Resume PSW in the CSRL16J area must be complete, with all AMODE, ilc, cc etc. fields properly set. If the PSW is incorrect, ABEND 4091-43 or 4091-45 can occur.
6. The caller must have restored the floating point registers (and control registers, if required) before calling CEE3RSUM. CEE3RSUM does not alter any floating point or control registers before resuming the program.
7. CEE3RSUM does not notify the debugger or member languages of the resume. If required, the caller must do this before calling CEE3RSUM.
8. CEE3RSUM cannot be used to jump over:
 - user-code stack frames
 - any stack frames that require PL/I exit GOTO processing.
 - any stack frames that require event 11 calls (PL/I DSA exit event stack frames)
 - any stack frames that have run CEE3SMS or CEE3SRT
9. CEE3RSUM can be used to jump over XPLINK transitional routines that are invoked as Exit DSAs.
10. CEE3RSUM must not be used to resume back into a function that has done any *alloca()* requests since the time of interruption. If this restriction is violated, ABEND 4091-42 or other problems may occur.
11. CEE3RSUM must run in 31-bit addressing mode.
12. The CEE3RSUM service uses either the RP instruction, CSRL16J, or some other method to resume. In all cases, the main input to CEE3RSUM is a CSRL16J parm area.

CEE3SGLN — signal invalid resume request

The CEE3SGLN callable service signals a condition to the Language Environment condition manager; optionally, this service can also provide qualifying data and create an ISI for a condition for which resumption is not supported.

Syntax

void (*CEELIBVxSGLN) (*cond_rep*, [*q_data_token*])

```
FEEDBACK *cond_rep;
INT4     *q_data_token;
```

CEELIBVxSGLN

A field in the Language Environment LIBVEC that points to the signal invalid resume request routine. Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L    R15,3008(,R15)
BALR R14,R15
```

cond_rep (input)

A condition token that defines the condition to be raised; it is passed by reference.

q_data_token (input/optional)

A 32-bit data token that is passed to the condition manager when a condition is signal; this value may be a pointer or any other information that may be

required. This information is placed in the ISI for use in accessing the qualifying data associated with the given instance of the condition.

Usage Note: CEESLGN cannot signal a severity 0 or 1 condition. If this is attempted, the following condition is passed to CEEHDSP.

Condition		
CEE3B1	Severity	3
	Msg_No	3425
	Message	Severity 0 or 1 condition signaled with CEESGLN.

CEE3SGLT — signal a condition and terminate

The CEE3SGLT callable service signals a condition for which resumption, without moving the resume cursor, is not supported.

Syntax

void (*CEELIBVxSGLT) (cond_rep, [q_data_token], [fc])

```
FEED_BACK *cond_rep;
INT4      *q_data_token;
FEED_BACK *fc;
```

CEELIBVxSGLT

A field in the Language Environment LIBVEC that points to the signal and terminate routine (CEESERC). Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L    R15,2764(,R15)
BALR R14,R15
```

cond_rep (input)

A condition representation that is passed by reference.

q_data_token (input/optional)

A 32-bit data object to be placed in the ISI for use in accessing the qualifying data associated with the given instance of the condition.

fc (output/optional)

The parameter in which the callable service feedback code is placed. The following conditions can result from this service.

Condition		
CEE000	Severity	0
	Msg_No	None
	Message Text	The service completed successfully.
CEE069	Severity	0
	Msg_No	0201
	Message Text	An unhandled condition was returned in a feedback code.
CEE0CE	Severity	1
	Msg_No	0398
	Message Text	Resume with new input.

Condition		
CEE0CF	Severity	1
	Msg_No	0399
	Message Text	Resume with new output.
CEE0EB	Severity	3
	Msg_No	0459
	Message Text	Not enough storage was available to create a new instance-specific information block
CEE0EE	Severity	3
	Msg_No	0462
	Message Text	Instance-specific information for the condition token with message number <i>message number</i> and facility ID <i>facility ID</i> could not be found.

Usage Notes:

1. Control is never returned to the next sequential instruction following the call to this routine.
2. The intent of CEESGLT is to provide a way for members to raise a condition and not allow resumption unless the resume cursor has been moved explicitly to a new position.
3. Requesting resumption when the resume cursor has not been moved causes CEE088 to be signaled. If resumption is once again requested without moving the resume cursor, the environment is terminated with abend 4091-12. CEE088 is defined as follows:

Condition		
CEE088	Severity	3
	Msg_No	0264
	Message	An invalid request to resume from a condition was detected.
	Explanation	A condition handler attempted to resume for a condition for which resumption is not allowed without moving the resume cursor. This condition can not be handled and resumed without moving the resume cursor. If resumption is requested without moving the resume cursor, the environment is terminated with abend 4091-12.
	Programmer Response	Move the resume cursor as part of handling the condition.
	System Action	The resume request that triggered this condition is ignored.

CEE3SMS — set machine state

This CWI interface dynamically builds a machine state block that contains the necessary machine state information for use with CEEMRCM.

Syntax

void (*CEECELVBSMS) (*gprs, float0, float2, float4, float6, stackframe, psw, [ars], machine_state, [fc]*)

```

INT      *gprs[16];
FLOAT    *float0;
FLOAT    *float2;
FLOAT    *float4;
FLOAT    *float6;
POINTER  *stackframe;
CHAR     *psw[8];
INT      *ars[16];
TOKEN    *machine_state;
FEED_BACK *fc;

```

CEECELVBSMS

Call this CWI interface as follows:

```

L      R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L      R15,3464(,R15)
BALR   R14,R15

```

CEECELVBSMS

A field in the Language Environment LIBVEC that points to the set machine state routine (CEE3SMS).

gprs (input)

An array of the 16 general purpose registers arranged in the order of gpr 0 through gpr 15.

float0 (input)

The value of the floating-point register 0 associated with the machine state.

float2 (input)

The value of the floating-point register 2 associated with the machine state.

float4 (input)

The value of the floating-point register 4 associated with the machine state.

float6 (input)

The value of the floating-point register 6 associated with the machine state.

stackframe (input)

The stack frame for this *label_var*. It must be a stack frame that is active on the call chain.

psw (input)

The program status word that contains information for the code point that gains control. In particular, it contains the code address that is to gain control and the program mask that is to be restored. The PSW must be complete and correct for execution at the indicated address. The instruction address must contain the correct high-order bit indicating the addressing mode.

ars (input/optional)

An array of the 16 access registers arranged in the order of AR 0 through AR 15. When omitted, the access registers are assumed inconsequential for this state block.

machine_state (output)

A token that represents the machine state block. The machine state block is allocated by Language Environment from heap storage. The machine state block is automatically freed by Language Environment when the code associated with the *stackframe* returns to its caller.

***fc* (output/optional)**

The parameter in which the callable service feedback code is placed. The following conditions can result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE390	Severity	3
	Msg_No	3360
	Message	The stack frame was not found on the call chain.

Usage Notes:

1. This is intended for building a machine state for use by the CEEMRCM routine. The token returned by this routine can be used as input to CEEMRCM.
2. Language Environment automatically frees the heap storage for the machine state block when the routine that is associated with the *stackframe* returns to its caller. Attempts to use the machine state block after it is freed result in unpredictable behavior.
3. If the saved machine state points into an XPLINK routine that does *alloca()*, the value of register 4 in the *gprs* parameter must point to the DSA currently on the Language Environment stack for that routine. In other words, the routine owning the DSA cannot have done any *alloca()* requests since the value of register 4 was captured.

CEE3SMS2 — set machine state 2

This CWI interface dynamically builds a machine state block that contains the necessary machine state information for use with CEEMRCM. It builds the machine state block at a storage location provided by the caller.

Syntax

void (*CEECELVBSMS2) (*gprs, float0, float2, float4, float6, stackframe, psw, [ars], machine_state, [fc]*)

```

INT      *gprs[16];
FLOAT    *float0;
FLOAT    *float2;
FLOAT    *float4;
FLOAT    *float6;
POINTER  *stackframe;
CHAR     *psw[8];
INT      *ars[16];
POINTER  *machine_state;
FEED_BACK *fc;

```

CEECELVBSMS2

Call this CWI interface as follows:

```

L      R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L      R15,4076(,R15)
BALR   R14,R15

```

CEECELVBSMS2

A field in the Language Environment LIBVEC that points to the set machine state routine 2 (CEE3SMS2).

***gprs* (input)**

An array of the 16 general purpose registers arranged in the order of gpr 0 through gpr 15.

***float0* (input)**

The value of the floating-point register 0 associated with the machine state.

***float2* (input)**

The value of the floating-point register 2 associated with the machine state.

***float4* (input)**

The value of the floating-point register 4 associated with the machine state.

***float6* (input)**

The value of the floating-point register 6 associated with the machine state.

***stackframe* (input)**

The stack frame for which this machine state block is built. It must be a stack frame that is active on the call chain.

***psw* (input)**

The program status word that contains information for the code point that gains control. In particular, it contains the code address that is to gain control and the program mask that is to be restored. The PSW must be complete and correct for execution at the indicated address. The instruction address must contain the correct high-order bit indicating the addressing mode.

***ars* (input/optional)**

An array of the 16 access registers arranged in the order of AR 0 through AR 15. This parameter is optional and is ignored by this service. The access registers are not affected..

***machine_state* (output)**

A pointer containing the address of storage into which the machine state block is built. Storage for the machine state block is allocated by the caller of CEE3SMS2. It must be large enough to contain a machine state block and mapped by CEEMCH.

***fc* (output/optional)**

The parameter in which the callable service feedback code is placed. The following conditions can result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service was successful.
CEE390	Severity	3
	Msg_No	3360
	Message	The stack frame was not found on the call chain.

Usage Notes:

1. This is intended for building a machine state for use by the CEEMRCM routine. The machine state block returned by this routine can be used as input to CEEMRCM.
2. It is the responsibility of the calling application to ensure that the storage for the machine state block is not freed prematurely but it is freed when it is no longer required. This helps to prevent memory leaks.

3. If the saved machine state points into an XPLINK routine that does `alloca()`, the value of register 4 in the `gprs` parameter must point to the DSA currently on the Language Environment stack for that routine. The routine owning the DSA cannot have done any `alloca()` requests since the value of register 4 was captured.

CEEGOTO — restart execution at specified label

CEEGOTO is used to restart execution at a specified label within a stack frame. It is supported to work only from one language to that same language.

CEEGOTO operates within a single thread (and thus, on one stack) and can only target earlier stack frames on that stack. If the Language Environment condition manager is on the stack and the range of CEEGOTO is from a stack frame more recent than the Language Environment condition manager to a stack frame less recent than the Language Environment condition manager, the Language Environment condition manager pops off the stack and the corresponding condition handler is terminated at that point. For more deeply nested conditions, several can be canceled at once.

A return to the caller occurs only when the feedback token is provided and a condition is detected.

Syntax

void CEEGOTO (*target_id*, [*fc*])

LABEL *target_id;
FEED_BACK *fc;

CEEGOTO

Call this CWI interface as follows:

```
L     R15,CEECAACELV-CEECAA(,R12)
L     R15,20(,R15)
BALR  R14,R15
```

target_id (input)

A label variable passed by reference. The first word of the label points to an active DSA. The DSA does not necessarily need to exist within the Language Environment-managed stack. However, it does need to be on the back chain of save areas. The second word of the label variable points to the instruction that receives control when CEEGOTO is run. If this address is zero, then the address is obtained from the saved R14 in the DSA specified in the first word. AMODE information is obtained from the high-order bit of the address.

The `target_DSA_address` field in the first 4 bytes of a non-XPLINK label variable is always non-zero. The `base_register_instruction` contains an instruction that, when run using an assembler EX instruction, restores the base register(s) needed by the `target_instruction`. A non-XPLINK label variable cannot be used to GOTO or resume an XPLINK routine. Figure 64 on page 262 shows the format of the non-XPLINK label variable.

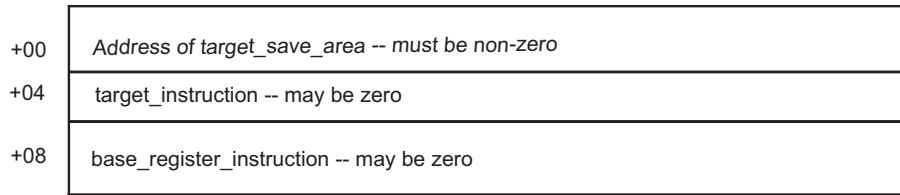


Figure 64. Format of a non-XPLINK label variable

The first 4 bytes of an extended label variable are zero. An extended label variable can be used to GOTO or resume either an XPLINK or non-XPLINK routine. When resuming an XPLINK routine, the resume registers are contained in the extended label variable itself, not the DSA to be resumed. Figure 65 shows the XPLINK extended format label variable.

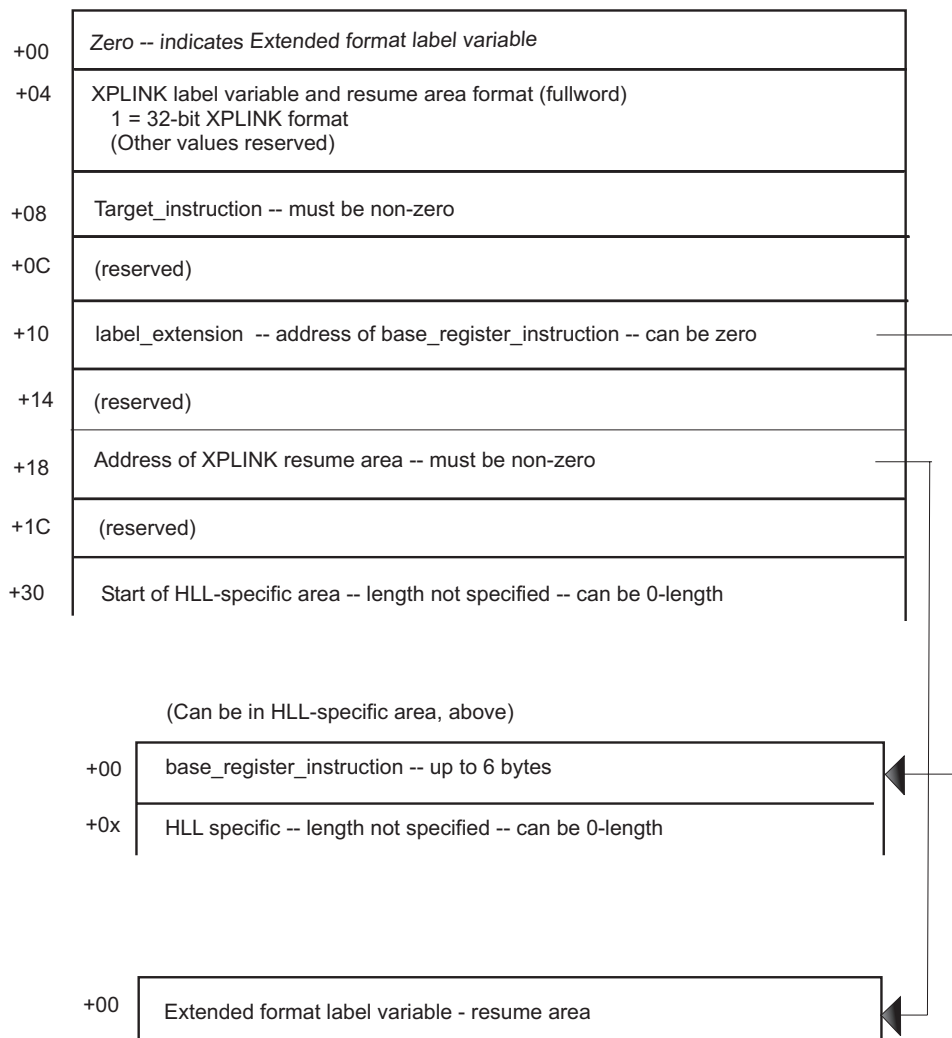


Figure 65. XPLINK extended format label variable

Note: The storage for the label variable is expected to be allocated within the storage of the lexical scope of the label variable so that the storage is released when the lexical scope is collapsed.

Figure 66 shows the XPLINK extended format label variable – resume area.

+00	Flags - x'80000000' -- FP regs 0,2,4,6 valid x'40000000' -- FP regs 0-15 valid (overrides x'80000000' bit) x'20000000' -- FP control Register valid x'10000000' -- Call chain DSA fixup complete x'08000000' -- High Registers Valid x'01000000' -- Restarting XPLINK alloca() routine
+04	(reserved)
+08	(reserved)
+0C	(reserved)
+10	target_dsa_address -- address of DSA to be resumed. This address may be stale (point to the place where the DSA used to start before alloca() was called. alloca() changes the starting address of the DSA.)
+14	(reserved)
+18	
+1C	
+20	
+24	
+28	reference_dsa -- If the target routine does not issue alloca(), this field must be zero. The target_dsa_address field is still valid. If the target routine is an XPLINK routine that issues alloca(), this field must point to the address of the DSA of the logical caller of the routine being branched to. The target_dsa_address field may be stale in this case (see above).
+30	
+34	(reserved)
+38	DSA reg 7 for alloca() backout -- needed only if reference_dsa is non-zero. This saved Register 7 value is stored back into the moved alloca() DSA, before CEEGOTO branches back to target_instruction
+3C	(reserved)
+40	Registers 0-15, 4-bytes each. Selected values are loaded into regs before CEEGOTO branches to target_instruction. When going back to a non-XPLINK routine, these values may also be copied into the target_DSA.
+44	
+78	
+7C	
+80	High Registers 0-15, 4-bytes each. These are valid only if the x'08000000' bit in the Flags is set. These values are loaded into the high registers before CEEGOTO branches to the target, if the X'08000000' bit is set in the Flags field.
+BC	(reserved)
+C0	
+FC	Floating-point registers 0-15 (8 bytes each)
+100	Note: Slots 0,2,4,6 or slots 0,1,2,..15 are valid, depending on setting of X'80000000' and X'40000000' bits in flags
+17C	Floating-point Control Register (Valid, if X'20000000' flag bit is on)
+180	(reserved)
+184	(reserved)
+188	
+1FC	Start of HLL-specific area -- length not specified -- can be 0-length
+200	

Figure 66. XPLINK extended format label variable – resume area

fc (output/optional)

One of the following condition tokens, which are passed by reference. A return to the caller occurs only when a condition is detected.

Condition		
CEE07Q	Severity	2
	Msg_No	0250
	Message	The <i>target_id</i> was not found on the stack.
CEE07R	Severity	2
	Msg_No	0251
	Message	An invalid <i>target_id</i> was provided.

Usage Notes:

1. As routines are popped off the stack, the DSA exit routines are invoked for those DSAs marked as an exit DSA.
2. The *base_register_instruction* in the label variable is run using the assembler instruction EX and is intended to restore the base register needed by the target instruction. This instruction is executed immediately prior to the branch to the *target_instruction*. The values of the registers are as follows:

R0-R12

Restored from the *target_dsa*

R13 The *target_dsa*

R14 The address of the *target_instruction*

R15 The address of the *base_register_instruction*

When resuming with an extended-format LABEL variable that resides in an XPLINK DSA or *alloca()* area, that area may have been freed before the *base_register_instructon* is executed.

When resuming into an XPLINK function, the register values when the EX instruction is executed are:

R0-R3

Restored from extended format resume area in the LABEL variable

R4 Address of the *target_dsa*

R5 Restored from Extended format resume area in the LABEL variable

R6 The address of the *base_register_instruction*. This instruction is not copied to a safe place before it is executed, so it must not reside in an XPLINK DSA or XPLINK *alloca()* area that will get freed when CEEGOTO runs.

R7 The address of the *target_instruction*

R8-R15

Restored from extended format resume area in the LABEL variable

3. If the *base_register_instruction* is zero, the EXecute is not performed.
4. CEEGOTO requests from a POSIX application (using *longjmp()* or *siglongjmp()*) are intercepted so that proper cleanup routine and destructor function invocation can take place. While executing a cleanup routine (for example, a routine established using CEECP SH), a CEEGOTO results in execution of all of the *pushed, but not popped* cleanup routines for more recent stack frame's than the target stack frame. This rule applies for both normal

processing and for the execution of cleanup routines during thread termination. If the jump buffer was established in the cleanup routine (for example, the target of the CEEGOTO is in the same cleanup routine) control continues at that point.

CEEGOTO invocation while processing a destructor function (during thread termination) is allowed, but the target of the jump must be established by the destructor function. (This is required since all of the user code has been removed from the stack.)

5. When resuming an XPLINK routine that issues an `alloca()`, any `alloca()` requests that were done after the LABEL variable was set up will be undone. When resuming into a non-XPLINK routine, `alloca()` requests already made by that routine are not undone.
6. A non-extended format LABEL variable passed to CEEGOTO must not reside in an XPLINK DSA or `alloca()` area that will get freed when the program is restarted.
7. An extended format LABEL variable passed to CEEGOTO can reside in an XPLINK or non-XPLINK DSA that will be freed when execution is restarted. It can also reside in storage obtained using XPLINK `alloca()` that will be freed when execution is restarted. The LABEL variable can also reside elsewhere, in which case CEEGOTO will not free it.
8. When an XPLINK program is restarted, register 7 always points to the *target_instruction*. When a non-XPLINK program is restarted, register 14 always points to the *target_instruction*. This may limit the use of CEEGOTO to restarting programs at a return point after a call.
9. The *base_register_instruction* will not be copied into a safe place before CEEGOTO is executed with the EX instruction. The *base_register_instruction* and any operands it uses must not reside in an XPLINK DSA or XPLINK `alloca()` area that is freed during the processing in CEEGOTO.

CEEHDHDL — register an event handler for stack frame zero processing

CEEHDHDL is used to register a member event handler for stack frame zero. This register condition handler is called following the normal stack frame zero condition handler.

Syntax

```
void CEEHDHDL (memberid, [fc])
```

```
LABEL      *memberid;
FEED_BACK *fc;
```

CEEHDHDL

Call this CWI interface as follows:

```
L      CEECAACELV-CEECAA(,R12)    CAA address is in R12
L      R15,3376(,R15)
BALR   R14,R15
```

memberid (input)

The member ID of the member event handler that is to gain control at stack frame zero conditions that remain unhandled. This condition handler is to gain control at the termination of the condition manager as opposed to gaining control at the termination of stack frame zero.

***fc* (output/optional)**

A condition token passed by reference. The following conditions are returned in *fc*:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE36S	Severity	2
	Msg_No	3292
	Message	Member already registered.

CEEMRCM — move the resume cursor

The callable service CEEMRCM allows the resume cursor to be moved to a specific predefined location within the active call chain. A recommended approach for using this service is to start with the current resume cursor machine state. This can be obtained from the CIB's resume cursor. Changes then can be made to the registers, PSW, or other components in your local copy of the machine state. Later, if a resume function code is returned to condition management, then the information from the updated machine state is used to resume the application program.

Initially, the resume cursor is placed after the machine instruction that caused the condition. Whenever the resume cursor is moved, as each stack frame is passed, any associated exit is invoked. This moving also cancels any associated user handlers. The direction of movement is always toward older stack frames and never toward newer stack frames. The action occurs only after the condition handler has returned to the condition manager. Multiple calls to CEEMRCM yield the NET results of the calls; that is, if two calls move the resume cursor to different places for the same stack frame, the most recent call is used for that stack frame.

Syntax

void CEEMRCM (*position*, [*fc*])

```
POINTER *position;
FEED_BACK *fc;
```

CEEMRCM

Call this CWI interface as follows:

```
L      CEECAACELV-CEECAA(,R12)   CAA address is in R12
L      R15,2856(,R15)
BALR   R14,R15
```

position (input)

A pointer to a valid machine state block to which the resume cursor is be moved.

fc (output/optional)

A condition token passed by reference; conditions returned in *fc* include:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE07V	Severity	2
	Msg_No	0255
	Message	<i>position</i> parameter is not a machine state block.

Usage Notes:

1. No stub is shipped for the CWI CEEMRCM.
2. Exit DSA routines are invoked as the resume cursor is moved across stack frames.
3. The machine state provided to the CWI CEEMRCM has the format as found in the resume cursor.
4. When a resume is requested, the state of the machine indicated in the machine state block is established prior to the resume point being entered.
5. If the resume point is in an XPLINK routine, all storage obtained by any `alloca()` requests done after the machine state was saved (perhaps by an earlier call to `CEE3SMS`) will be freed up before the XPLINK routine is resumed. In other words any `alloca()` requests issued after the machine state was saved and before this CEEMRCM call will be undone. If the resume point is in a non-XPLINK routine that issues `alloca()`, `alloca()` requests issued after the machine state was saved and before the CEEMRCM call are not undone. When resuming either an XPLINK or non-XPLINK routine, any `alloca()` requests issued before the machine state was saved are not undone.
6. When an interrupt has occurred in a routine that has saved the stack pointer in the `CEECAA_SAVSTACK` field or in the field pointed to by the `CEECAA_SAVSTACK_ASYNC` field, the resume cursor is initially set up so that the stack pointer is restored to that field if the application is resumed. However, if the resume cursor is moved, the stack pointer is not restored to that field unless certain fields in the machine state are set. To restore the stack pointer to the `CEECAA_SAVSTACK_ASYNC` field, the flags `INT_SF_VALID` and `SAVSTACK` must be set to 1 and the field `INT_SF` must contain the stack pointer. To restore the stack pointer to the field pointed to by the `CEECAA_SAVSTACK` field, the flags `INT_SF_VALID` and `SAVSTACK_ASYNC` must be set to 1 and the field `INT_SF` must contain the stack pointer.

Note: Only the stack pointer that was saved at the time of the interrupt can be restored and only be restored to the field where it was saved.

CEEYDSAF — find the previous DSA

CEEYDSAF is used to identify the DSA prior to the passed DSA. It requires that the DSA used as input be a valid OS stackframe or an XPLINK stack frame. It also requires that the stack format be passed so it uses the proper unwind technique.

Recommendation: For performance reasons, whenever possible, the DSA format should be passed to this service instead of determining it dynamically.

Syntax

void CEEYDSAF (*dsa_in*, *dsa_prev*, *dsa_format*, (*physical*), (*ph_callee*), (*ph_callee_dsa_format*), (*fc*))

```

POINTER    *dsa_in;
POINTER    *dsa_prev;
INT4       *dsa_format;
INT4       *physical;
POINTER    *ph_callee;
INT4       *ph_callee_dsa_format;
FEED_BACK  *fc;

```

CEEYDSAF

From a non-XPLINK routine, call this CWI interface as follows:

```

L    R15,CEECAALEOV-CEECAA(,R12)   Address of CAA in R12
L    R15,0(,R15)
BALR R14,R15

```

dsa_in (input)

Address of an OS or XPLINK format DSA.

dsa_prev (output)

Address of the OS or XPLINK format DSA behind *dsa_in*.

dsa_format (input/output)

Format of DSA:

0 OS

1 XPLINK

-1 The CWI determines the *dsa_format*. On input, it pertains to the format of the DSA of the *dsa_in* parameter. On output, it pertains to the format of the returned DSA in the *dsa_prev* parameter. The -1 indicates the CWI will attempt to determine the format of the passed DSA first. In all cases, the DSA format returned will be for the DSA returned by the service in *dsa_prev*.

physical (input/optional)

When *physical* = 1, physical unwinding requested. This means library-injected and XPLINK transitional stack frames are to be skipped over.

ph_callee (output/optional)

This parameter is designed to be used with logical unwinding. It provides a pointer to the stack frame physically located "in front" of the DSA returned as the previous logical. If no transitionals or library-injected DSAs are present, this is simply the DSA passed as input. If a transitional or an injected DSA is present, this is a pointer to it.

ph_callee_dsa_format (output)

Format of DSA of Physical Callee : 0 = OS 1 = XPLINK Used with the *ph_callee* parameter, this is the DSA format of the returned callee.

fc (output/optional)

The parameter in which the callable service feedback code is placed. The following conditions can result from this service.

Condition		
CEE000	Severity	0
	Msg_No	n/a
	Message	The service completed successfully

Condition		
CEE3EQ	Severity	2
	Msg_No	3546
	Message	An error occurred while attempting to find the previous DSA.
CEE3ER	Severity	2
	Msg_No	3547
	Message	The physical callee DSA was requested and the physical callee format was not.
CEE3ES	Severity	2
	Msg_No	3548
	Message	The callable service was passed a DSA format of -1 and was unable to determine the format of the passed DSA.

__dsa_prev() — chain back to previous DSA

The `__dsa_prev()` function returns the address of the DSA prior to `dsa_p` on the Language Environment stack. Two types of backchaining request are supported: logical and physical. The `req_type` parameter is used to select either logical or physical backchaining. For physical backchaining, the address of the DSA immediately prior to `dsa_p` is always returned. That DSA can be a transition or overflow DSA, or the DSA of a normal routine. For logical backchaining, `__dsa_prev()` keeps looking backward on the Language Environment stack until a normal DSA is found, skipping over any transition or overflow DSAs.

If the dummy Language Environment DSA is reached while backchaining, a NULL pointer is returned, and `errno` is set to **ESRCH**.

`__dsa_prev()` can be used when the Language Environment stack of interest is not in the current address space. To access storage outside the current address space, the user must provide the `callback_p` parameter. `callback_p` is a pointer to a user-written function that fetches all required data for `__dsa_prev()`. Generally, the `(*callback_p)()` function would obtain the data using some application-dependent method (like `BPX1PTR`) and move it into the current address space, where `__dsa_prev()` can access it directly. If the Language Environment stack of interest is in the same address space and is directly accessible to `__dsa_prev()`, `callback_p` can be NULL.

Syntax

```
#include <edcwccwi.h>
```

```
void __dsa_prev (const void * (dsa_p, int req_type, int dsa_fmt, void *
(*callback_p)(void *data_p, size_t data_l), const void *caa_p, int *prev_fmt, void
**ph_callee_dsa_p, int *ph_callee_dsa_fmt; fc));
```

```
const void *dsa_p
```

Pointer to the current DSA. `__dsa_prev()` returns a pointer to the DSA logically or physically previous to `dsa_p`, depending on the value of the `req_type` parameter. `dsa_p` may point to a DSA in another address space or in some other place not directly accessible by `__dsa_prev()`. If this address is not

`__dsa_prev()`

directly accessible, the *callback_p* parameter must be non-NULL. The callback function will be used to access *dsa_p* indirectly.

int req_type

Controls if transition and overflow DSAs are returned. The allowed values for *req_type* are:

`__EDCWCCWI_PHYSICAL`

Physical backchaining causes `__dsa_prev()` to return the address of the DSA immediately prior to *dsa_p*. The returned DSA may be either a transition, overflow, or normal DSA.

`__EDCWCCWI_LOGICAL`

Logical backchaining causes `__dsa_prev()` to skip over any transition or overflow DSAs that it finds while backchaining, and not pass them back. The address of the most recent normal DSA previous to *dsa_p* is returned. Doing logical backchaining is the same as doing physical backchaining one or more times, stopping when a normal DSA is found.

int dsa_fmt

The format of the DSA pointed to by *dsa_p*. The allowed values for *dsa_fmt* are:

`__EDCWCCWI_UP`

Indicates that *dsa_p* points to a non-XPLINK DSA.

`__EDCWCCWI_DOWN`

Indicates that *dsa_p* points to an XPLINK DSA.

*void * (*callback_p)()*

Pointer to a user-provided function that fetches data not normally accessible by `__dsa_prev()`. If *callback_p* is NULL, `__dsa_prev()` accesses *dsa_p* and any other required Language Environment data areas directly in the current address space. The Language Environment stack and all other data needed for backchaining must be directly accessible to `__dsa_prev()` in this case.

The user-provided *(*callback_p)()* function is passed the address and length of data to access. It must fetch the data in some application-dependent manner, and make the data available in the current address space in a place accessible to `__dsa_prev()`. *(*callback_p)()* must return a pointer to the copied data. This data must remain available to `__dsa_prev()` until the next call to *(*callback_p)()*, or until `__dsa_prev()` returns to its caller, whichever happens first. On subsequent calls, *(*callback_p)()* is allowed to reuse the same data passback area.

There is no provision for *(*callback_p)()* to pass back an error return code, indicating that the requested data could not be obtained. If *(*callback_p)()* cannot return the requested data, it must not return to `__dsa_prev()`. When an error occurs, *(*callback_p)()* may:

- `longjmp()` back to some error return point in the user code that called `__dsa_prev()`
- ABEND or otherwise terminate abnormally
- `exit()`, `pthread_exit()`, etc.
- Raise a caught signal where the catcher does `longjmp()` so as not to return to `__dsa_prev()`
- Use Language Environment condition management to bypass `__dsa_prev()` after the error and resume in user code.
- Recover in some other way that does not involve returning to `__dsa_prev()`.

`__dsa_prev()` calls *(*callback_p)()* with two parameters:

*void *data_p*

Pointer to the start of the required data. This address might not be in the current address space.

size_t data_l

The number of bytes of data required. *data_l* will never exceed 16 bytes. If *(*callback_p)()* cannot pass back the complete data requested, it must not return to `__dsa_prev()`.

*const void *caa_p*

Pointer to the Language Environment CAA for the thread owning the *dsa_p* DSA. This parameter must be non-NULL whenever *callback_p* is non-NULL, and it may point to a CAA in some other address space. If *callback_p* is NULL, *caa_p* may also be NULL. If *caa_p* is NULL, the current CAA (of the thread where `__dsa_prev()` is running) is used. In this case, it is assumed that *dsa_p* points to a DSA on the Language Environment stack for the caller's thread.

*int *prev_fmt*

Pointer to an optional passback area, where `__dsa_prev()` will return the DSA format of the prior DSA. The possible values passed back in this field are the same as the values for *dsa_fmt*. If *prev_fmt* is NULL, the DSA format for the previous DSA is not passed back. If `__dsa_prev()` cannot find the previous DSA and returns a NULL value, the field pointed to by *prev_fmt* is not altered.

*void **ph_callee_dsa_p*

Pointer to an optional passback area where `__dsa_prev()` will return the address of the DSA of the physical callee. The physical callee is the function called by the function owning the returned DSA. The physical callee can be a Language Environment overflow or stack expansion routine, or it can be a normal user or Language Environment function. If physical backchaining is requested, *ph_callee_dsa_p* will be the same as *dsa_p* after `__dsa_fmt()` returns.

If *ph_callee_dsa_p* is NULL, the address of the physical callee DSA is not passed back. If `__dsa_prev()` cannot find the previous DSA and returns a NULL value, the field pointed to by *ph_callee_dsa_p* is not altered.

*int *ph_callee_dsa_fmt*

ph_callee_dsa_fmt is a pointer to an optional passback area where `__dsa_prev()` will return the DSA format of the physical callee's DSA. The possible values passed back in this field are the same as the values for *dsa_fmt*. If *ph_callee_dsa_fmt* is NULL, the format of the physical callee DSA is not passed back. If `__dsa_prev()` cannot find the previous DSA and returns a NULL value, the field pointed to by *ph_callee_dsa_fmt* is not altered.

If successful, `__dsa_prev()` returns the address of the previous DSA. In addition, if `errno` is zero when `__dsa_prev()` is called, one of the following `errno` values may be set to pass back additional information:

EACCES

Indicates that the returned DSA pointer is for the Language Environment dummy DSA (pointed to by the CAA `ceecaaddsa` field). This is not an error, and all returned or passed-back information is valid.

EALREADY

Indicates that the input DSA pointer (*dsa_p*) is for the Language Environment dummy DSA (pointed to by the CAA `ceecaaddsa` field). This is not an error, and all returned or passed-back information is valid.

If unsuccessful, `__dsa_prev()` returns a NULL pointer, and sets `errno` to one of the following values:

__dsa_prev()

ESRCH

This error indicates that there was no DSA previous to *dsa_p* that could satisfy the physical or logical backchaining request. This error also occurs if *dsa_p* is NULL when `__dsa_prev()` is called.

EINVAL

This error can occur if:

- *caa_p* was NULL and *callback_p* was not NULL.
- *req_type* was not `__EDCWCCWI_PHYSICAL` or `__EDCWCCWI_LOGICAL`.
- *dsa_fmt* was not `__EDCWCCWI_UP` or `__EDCWCCWI_DOWN`.

Usage Notes:

1. If the return code from `__dsa_prev()` is NULL, the listed errno values are set even if errno was non-zero when `__dsa_pr()` was called. When the return code from `__dsa_pr()` is not NULL, errno is not changed if it was not zero when `__dsa_prev()` was called.
2. `__dsa_prev()` may cause program checks if it accesses invalid addresses. This is especially likely to happen if *callback_p* is NULL and the Language Environment stack being looked at is corrupted. For this reason, the caller should consider having a signal catcher set up to handle SIGSEGV with appropriate error recovery.
3. The Vendor Interfaces header file, `<edcwccwi.h>`, is located in member EDCWCCWI of the SCEESAMP data set. In order to include `<edcwccwi.h>` in an application, the header file must be copied into a PDS or into a directory in the z/OS UNIX file system where the C/C++ compiler will find it.

__far_jump() — perform far jump (C/C++ and XPLINK only)

The `__far_jump()` interface performs a function similar to `longjmp()`. However, it does not require a `setjmp()` to be performed previously. The information required to perform this "nonlocal goto" is provided by the user in the `__jumpinfo` structure. This information is normally provided in a `jmp_buf` and saved by the library when a `setjmp()` is invoked. When used in conjunction with `__set_stack_softlimit()`, described in "`__set_stack_softlimit() — set stack soft limit (C/C++ and XPLINK only)`" on page 274, the information needed to jump to the point of retry can be obtained from data passed to a signal handler set up to field a softlimit stack overflow signal. This information includes registers, psw, and signal mask.

Syntax

```
#include <edcwccwi.h>
```

```
void __far_jump (struct __jumpinfo * JumpInfo);
```

```
struct __jumpinfo * JumpInfo
```

The `__jumpinfo` structure appears in the following format:

```
struct __jumpinfo
{
    char          __ji_u1[68];
    char          __ji_mask_saved;
    char          __ji_u2[3];
    sigset_t      __ji_sigmask;
    char          __ji_u3[11];
    unsigned      __ji_fl_fp4      :1;
    unsigned      __ji_fl_fp16     :1;
    unsigned      __ji_fl_fpc      :1;
```

```

unsigned    __ji_fl_res1a    :1;
unsigned    __ji_fl_hr      :1;
unsigned    __ji_fl_res2    :1;
unsigned    __ji_fl_exp     :1;
unsigned    __ji_fl_res2a   :1;
char        __ji_u4[36];
long        __ji_gr[16];
long        __ji_hr[16];
int         __ji_u5[16];
double      __ji_fpr[16];
int         __ji_fpc;

```

`__ji_gr`

Contains the following:

- The values of the 16 general purpose registers that are restored
- The value in Register 7 will be used as the target address of the jump
- The value of Register 4 will be used as the target DSA address

`__ji_hr`

Contains the values of the high halves of the 16 general purpose registers (0-15) that are restored. This is valid only on 64 bit hardware when running in 31 bit mode.

`__ji_fpr`

Contains the values of either 4 or 16 floating-point registers. If all 16 floating-point registers are present, registers 0-15 are saved in `__ji_fpr[0]` through `__ji_fpr[15]`. If only 4 floating-point registers are present, they are registers 0, 2, 4, and 6; these are saved in `__ji_fpr[0]`, `__ji_fpr[2]`, `__ji_fpr[4]`, and `__ji_fpr[6]`.

`__ji_mask_saved`

Indicator field that is set to non-zero value when the signal mask field (`__ji_sigmask`) is valid.

`__ji_sigmask`

Contains the signal mask value.

`__ji_fpc`

Contains the floating point control register value.

`__ji_fl_fp4`

Set to one when values for the 4 floating-point registers 0, 2, 4, and 6 are provided in `__ji_fpr`. This bit should also be set to one whenever all floating-point registers 0-15 are present (when `__ji_fl_fp16` is also set to one.)

`__ji_fl_fp16`

Set to one when values for all 16 floating-point registers 0-15 are provided in `__ji_fpr`. In this case, `__ji_fl_fp4` should also be set to one.

`__ji_fl_fpc`

Set to one when the value of the floating-point control register is provided in `__ji_fpc`.

`__ji_fl_exp`

Set to one when explicit backchaining is complete to the target stack.

`__ji_fl_hr`

Set to one when values for the high halves of general registers 0-15 are provided in `__ji_hr`. This flag is set only in 31-bit addressing mode. In 64-bit addressing mode, `__ji_gr` contains 64-bit values for the general registers, and `__ji_fl_hr` is not set.

`__far_jump()`

The `__far_jump()` function has no returned value. When `__far_jump()` completes, program execution continues at the target address.

Usage Notes:

1. The library does not attempt to verify the contents of the `__jumpinfo` structure. Incorrect data can lead to unpredictable results.
2. The caller of `__far_jump()` can optionally supply a signal mask suitable to the target of the jump. It is usually required in the soft overflow scenario because the signal handler, which is the `__far_jump` invoker, is driven with SIGSEGV disabled. However, SIGSEGV must be enabled at resumption in the target.
3. The caller of `__far_jump()` provides the GPR and FPR sets needed for the target of the `__far_jump()`. The GPR set is always complete. For example, it has all 16 registers, including the target DSA address in R4 and target code address in R7. The FPR set is 4 or 16 registers long, indicated by the accompanying switches.
4. The contents of all registers at the point of resumption after a `__far_jump()` are the values specified in the `__jumpinfo` buffer. The target address of the jump is not supplied separately. It is supplied as two of the register values in the GPR set in the `__jumpinfo` buffer, R4 for the target DSA address and R7 for the target code address.
5. The Vendor Interfaces header file, `<edcwccwi.h>`, is located in member EDCWCCWI of the SCEESAMP data set. In order to include `<edcwccwi.h>` in an application, the header file must be copied into a PDS or into a directory in the z/OS UNIX file system where the C/C++ compiler will find it.

`__set_stack_softlimit()` — set stack soft limit (C/C++ and XPLINK only)

When Language Environment attempts to expand the stack and the additional stack segment could cause the total stack size to exceed the *MaximumStackSize*, a SIGSEGV with an `si_code` of `_SEGV_SOFTLIMIT` is generated. As a result of the SIGSEGV, a signal handler is driven and is passed information that represents the environment at the point of stack overflow. This includes register contents, psw contents, and signal mask contents. The signal handler has the option of releasing stack storage and using the passed data to perform a `__far_jump()` to the original point of overflow in Language Environment, trying the stack segment request again. If a signal handler was registered but the `SA_INFO` flag was not set, the SIGSEGV signal is delivered but no extra information is passed to the signal handler.

The initial stack softlimit value that existed before issuing any `__set_stack_softlimit()` requests is the `ULONG_MAX` value. This disables the softlimit from being reached. Because this function returns the current softlimit value, the first time it is invoked, it returns the `ULONG_MAX` value. The function always sets the soft limit to the passed *MaximumStackSize* value and returns the previous soft limit value.

Syntax

```
#include <edcwccwi.h>
```

```
unsigned long __set_stack_softlimit (unsigned long MaximumStackSize);
```

```
unsigned long MaximumStackSize
```

MaximumStackSize is the stack size, in bytes. This is a thread-specific value. It is

also a soft limit, which means that the actual stack size can grow beyond this limit. You can specify *MaximumStackSize* back to the `ULONG_MAX` value, which disables the softlimit.

The `__set_stack_softlimit()` returns the previous value of the soft limit. This function does not fail and no errors are defined.

Usage Notes:

1. The SIGSEGV is generated for the thread whose stack has grown beyond the maximum size.
2. The SIGSEGV is generated regardless of whether a signal handler function for SIGSEGVs has been registered.
3. No attempt is made to guarantee that there is sufficient available stack space to deliver the signal, or that there is a minimum amount of available stack space.
4. If a signal handler function for SIGSEGV was registered with the SA_SIGINFO flag and using the `sa_sigaction` field to identify the handler function, an `si_code` of, `_SEGV_SOFTLIMIT`(defined in `signal.h`), will be reported to the signal handler.
5. The soft limit overflow is not detected until a stack extension is requested. Therefore if a stack initial size has been selected that is greater than the soft limit the stack size will grow past the soft limit, and will not be detected until the initial stack size is exceeded.
6. The Vendor Interfaces header file, `<edcwcwi.h>`, is located in member `EDCWCCI` of the `SCEESAMP` data set. In order to include `<edcwcwi.h>` in an application, the header file must be copied into a PDS or into a directory in the z/OS UNIX file system where the C/C++ compiler will find it.

Other Language Environment routines and handlers

Along with compiler-writer interfaces, Language Environment provides the following routines and handlers for condition management:

- Language-specific handler interface
- DSA exit routines
- Shunt routines
- Attention handling
- Error processing

Interface to the language-specific handlers

For information on the condition handlers, see the following sections:

- For handling conditions represented by the CEECIB (not for stack frame zero), see “Event code 1 — handle condition represented by the CIB event” on page 482.
- For performing enablement for this stack frame, see “Event code 2 — perform enablement for this stack frame event” on page 483.
- For handling conditions in accordance with the language defaults (stack frame zero), see “Event code 3 — handle condition according to language defaults event” on page 485.
- For information about a resumption from a condition handler within a *target_dsa*, see “Event code 10 — resume from a condition handler event” on page 497.

DSA exit routines

A DSA exit routine is used to perform activities on behalf of a stack frame when the stack is being collapsed as the result of a return from a main, an immediate STOP request, a GOTO out of block, or a move resume cursor request.

Exit routines allow for activities such as the closing of files and releasing of system resources that are held.

Members not requiring exit DSAs may, for performance reasons, request that this processing be disabled. This applies to normal, or non-abend, enclave terminations initiated by a call to the CEETREN or CEETREC services. This is implemented with a parameter used on the Enclave Initialization Event, Event Code 18. Refer to this event for more information on enabling this feature. When this feature is on, the traverse of the stack for exit DSA routines is not executed and the DSA exit event call is skipped. If multiple language members are present in an enclave, all must indicate that the DSA exit scan may be skipped. Stack traverse and DSA Exit processing continues to occur for terminations with an abend pending or a GOTO out of block or move resume cursor request whether the feature is enabled or not. If the exit DSA scan is to be skipped, a flag in the EDB, 'CeeEdb_Term_Noedsa' is activated.

An exit routine is established by one of two mechanisms, as described below.

1. The PPA1 has the exit DSA flag on.
2. The stack frame (DSA) is marked as requiring DSA exit processing by flags set within the DSA.

The exit routine has two different interfaces, depending upon the mechanism used to establish the exit.

PPA-marked exit routines

For the event handler when a stack frame is abnormally collapsed, see “Event code 11 — DSA exit routines event” on page 498. You can use this for both non-64-bit and 64-bit environments.

DSA-marked exit routines

An exit can be marked in the first word of the DSA. The first byte of the DSA must be marked with bit 4 on. The second byte of the DSA must have the X'08' flag on indicating this DSA is an exit DSA.

When the exit routine is to be driven, the X'08' in the second byte is turned off and a return using R14 is made to the routine. In addition, the Language Environment condition manager takes the return address of one level back. Turning the flag off allows the routine to interrogate whether the return was due to a normal return or as an exit routine. When the return is due to an abnormal collapse of the stack frame, there are no parameters passed back to the routine.

To establish the DSA exit, the FORTRAN I/O library routines must place the following value into the first word of the calling application's DSA (in binary) (x means any setting of the bit is valid in bytes 3 and 4.) The FORTRAN event handler will be driven for the exit DSA event (Event Code 11)

```
00000000 01000000 xxxxxxxx xxxxxxx1
```


To remove the exit, the FORTRAN library can place any other pattern into the first word of the DSA that will not match the pattern above and will not conflict with other conventions of word zero established in previous releases of Language Environment.

An exit can also be marked in two words of the DSA. Byte 0 of the DSA is nonzero. In byte 1 of the DSA, either bit 6 is non-zero, or bit 0 is nonzero and in byte 77 (hex), bit 0 is nonzero.

Upon completion of the exit routine, the exit routine returns to its caller, which is the Language Environment condition manager.

Shunt routine

A shunt is a low-level error handling routine intended for use by language library routines and debug tools. A shunt is typically used when a segment of code needs to protect itself from a likely error. An incorrect address while following a control block chain is an example of an error that activates a shunt routine.

A shunt is usually established for short periods of time while the library routines or debug tools are providing services to the application. Language Environment establishes an ESPIE error recovery routine for program interrupts and an ESTAE recovery routine for abends. These recovery routines check for and setup for retry to a shunt, as appropriate. Shunt routines do not return to the Language Environment condition manager. There is no return code from the shunt routine.

Establishing a program interrupt shunt service

A program interrupt shunt routine is established by setting its address in the CAA (CEECAADMC). When the shunt address gains control, the AMODE is the AMODE at the time of the program interrupt. Setting an address in the CEECAADMC effectively cancels the previously established shunt routine, if any. Only one shunt routine can be in effect at a time. Language Environment does not provide any facility for stacking the shunt addresses. A save is not needed prior to establishing your own shunt routine.

The shunt routine is removed by removing its address from the CEECAADMC. A value of zero should be assigned to CEECAADMC as soon as possible. A shunt routine should be removed as soon as it is not needed. Information about the error is provided to the shunt routine through the CEECAAPRGCK field in the CAA, which is set to the value of the program interrupt code.

Usage Notes:

1. R15 is set to the address of the shunt routine upon entry to the shunt routine for a resume into non-XPLINK code. For shunts activated in XPLINK routines, there is no specific register set to the shunt address when the shunt routine receives control.
2. R0 through R14 have the same value when the shunt routine gains control as they did when the program check occurred. For shunts active in XPLINK routines, R15 is also set to its contents at the time of the interrupt for the resume.
3. The shunt routine cannot assume that the range of the base registers used at the time that the program check occurred extends to the shunt routine. The shunt routine might need to re-establish addressability upon entry.
4. The CEECAADMC field should be cleared as soon as it is no longer needed.

Condition Management

5. A shunt routine should never span a call statement. A shunt routine that gains control with another program's registers will usually fail on the first branch attempt. The routine that is called does not have to save the address of your shunt routine.
6. The Language Environment condition manager clears the CEECAADMC field when the program interrupt shunt routine is called.

Abend shunt routine

An abend shunt routine is established by setting its address in the CAA (CEECAASHAB). Setting an address in the CEECAASHAB effectively cancels the previously established abend shunt routine, if any. Only one abend shunt routine can be in effect at a time. Language Environment does not provide any facility for stacking the abend shunt addresses. A save is not needed prior to establishing your own abend shunt routine.

The abend shunt routine is removed by removing its address from the CEECAASHAB. A value of zero should be assigned to CEECAASHAB as soon as possible. An abend shunt routine should be removed as soon as it is not needed.

After an abend occurs that is shunted, the abend shunt routine gains control in the addressing mode in effect when the error recovery routine was established. Table 48 lists the external fields of the CAA that will contain information about the abend. This information is taken from fields of the SDWA associated with the shunted abend. The SDWA does not exist after the abend shunt routine is given control.

Table 48. CAA fields that contain information about abends

Field	Description
CEECAAAB_GR0_VALID	A bit indicating, if on, that the CEECAAAB_GR0 field contains valid data about the last abend.
CEECAAAB_GR0	Register 0 contents at the time of the abend. This is only valid if the CEECAAAB_GR0_VALID bit is on.
CEECAAAB_ICD1_VALID	A bit indicating, if on, that the CEECAAAB_ICD1 field contains valid data about the last abend.
CEECAAAB_ICD1	The eight bit interrupt code from SDWAICD1 field of the SDWA for the abend. This is only valid if the CEECAAAB_ICD1_VALID bit is on.
CEECAAAB_ABCC_VALID	A bit indicating, if on that the CEECAAAB_ABCC field contains valid data about the last abend.
CEECAAAB_ABCC	The abend completion code, taken from SDWAABCC field of the SDWA for the shunted abend. This is only valid if the CEECAAAB_ABCC_VALID bit is on.
CEECAAAB_CRC_VALID	If on, this bit indicates that the CEECAAAB_CRC field contains valid data about the last abend.
CEECAAAB_CRC	Component reason code, or return code associated with the abend, taken from the SDWACRC field of the SDWA for the shunted abend. This is only valid if the CEECAAAB_CRC_VALID bit is on.
CEECAAAB_GR15_VALID	A bit indicating, if on, that the CEECAAAB_GR15 field contains valid data about the last abend.
CEECAAAB_GR15	Register 15 contents at the time of the abend. This field is only valid if the CEECAAAB_GR15_VALID bit is on.

Usage Notes:

1. The abend shunt routine is intended to be used when the PSW key that is in effect at the time the shunt is established matches the PSW key in effect at the time the Language Environment ESTAE or user-provided error recovery routine was established. Since Language Environment does not support retry in a specified key, two fields in the CAA are provided to help effect the behavior of the retry being done in the correct key. The CEECAASHAB_KEY field is to be set to the IPK result just before setting CEECAASHAB to the address of the abend shunt routine. This establishes the PSW key in effect at the time the shunt is established. The CEECAASHAB_RECOVER_IN_ESTAE_MODE field (a flag bit) is to be set on. This flag, when on, instructs Language Environment to set up for retry to the abend shunt routine in the PSW key that was in effect when the recovery routine was established. Since recovery routines are given control by the system in the same PSW key as when they were established, the flag simply tells the recovery routine to honor the abend shunt only when the current IPK result matches the CEECAASHAB_KEY field. When the flag is off, the recovery routine does not compare the IPK result and will setup for retry using old methodology, which might include a retry in the wrong key. When the flag is on, but the CEECAASHAB_KEY field is not set properly, the recovery routine might ignore the abend shunt.
2. The abend shunt routine will receive control in the addressing mode that was in effect when the recovery routine was established. This could be different than the addressing mode in effect when the abend shunt routine was established. Therefore, the abend shunt routine might need to change addressing mode to execute properly.
3. The abend shunt routine cannot assume the contents of any of the general purpose registers when it receives control. Generally, the registers will contain the values at the time of the abend. The abend shunt routine might need to re-load the general purpose registers that were saved prior to setting the abend shunt routine address.
4. The CEECAASHAB field should be set to zero, the CEECAASHAB_KEY field set to X'8F', and the CEECAASHAB_RECOVER_IN_ESTAE_MODE flag set to off, as soon as the abend shunt routine is no longer needed.
5. The Language Environment ESTAE, CEE3ERP, and the exception handler routine used with Language Environment preinitialization, will reset these fields when setting up for retry to the abend shunt routine.

Attention handling

When the runtime option INTERRUPT(ON) is specified, the Language Environment condition manager issues a STAX macro, which requests attention interrupts to be directed to a STAX exit.

In the CAA at offset X'120', label CEECAAATTN, is initially set to the address of a routine that runs a BR 14.

If an attention interrupt occurs and the STAX exit is entered, the STAX exit changes the address of the routine at CEECAAATTN to a routine that issues a CALL CEESGL raising the ATTENTION condition.

Polling code is contained in both library- and compiler-generated code and is the following code sequence; there is no parameter for this routine and R1 is not used:

```
L    15,CEECAAATTN
BALR 14,15
```

Condition Management

When polling code calls the routine that calls CEESGL, the attention condition is raised. Condition handling proceeds with the defined sequence of condition handling events, as if a synchronous condition were raised.

Error processing

Language Environment allows you to write exit routines that can be added to CEE_ABEND_EXIT. All exit routines that were added to CEE_ABEND_EXIT are invoked during condition management error processing using the CSVDYNEX macro. Note the following restrictions:

- The TRAP runtime option must specify TRAP(ON,NOSPIE).
- The address space must be APF-authorized.

Return codes

The return codes (in decimal) for CEE_ABEND_EXIT are:

- 0 The exit routine did not take any action. Continue with Language Environment default dump processing.
- 4 The exit routine took a dump. Language Environment does not take a SYSDUMP but might produce a CEEDUMP based on the TERMTHDACT option.
- 8 The exit routine took a dump and gathered all appropriate diagnostic information. Language Environment does not take a SYSDUMP or a CEEDUMP.

Usage notes

1. Upon entry, GPR 1 contains the address of the SDWA.
2. Upon entry, GPR 13 contains the address a 336-byte work area.
3. For more information about CSVDYNEX macro, see *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.
4. For more information about the TRAP runtime option, see *z/OS Language Environment Programming Reference*.

Examples of condition management routines

This section contains code examples that demonstrate condition management routines. The following code (sample) shows an example that adds two exit routines called MYEXIT and MYEXIT2.

```
TITLE 'ADD EXIT ROUTINE'
PRINT GEN
DYNEXADD CEEENTRY PPA=MAINPPA,MAIN=YES,BASE=11,AUTO=WORKSIZE
* =====
        USING WORKAREA,13
        CSVDYNEX REQUEST=ADD,EXITNAME=LEEXIT,MODNAME=MYEXIT,      X
                DSNAM=MYPDS,RETCODE=LRETCODE,RSNCODE=LRSNCODE,    X
                MF=(E,DYNEXL)
        L      15,LRETCODE
        LTR   15,15          TEST RETURN CODE
        BZ    DYNGOOD
DYNFAIL  NOPR  0
        WTO   'CSVDYNEXIT FAILED',ROUTCDE=12
        B     DONE
DYNGOOD  NOPR  0
        WTO   'CSVDYNEXIT WAS SUCCESSFUL'
DONE     CEETERM RC=0,MODIFIER=0
* =====
*  CONSTANTS
* =====
LEEXIT   DC    CL16'CEE_ABEND_EXIT'
MYEXIT   DC    CL8'MYEXIT'
```

```

MYPDS   DC   CL44'POSIX.MYEXIT.LOADLIB'
* =====
MAINPPA CEEPPA ,           CONSTANTS DESCRIBING THE CODE BLOCK
* =====
*       THE WORKAREA AND DSA
* =====
WORKAREA DSECT
        ORG   **CEEDSASZ           LEAVE SPACE FOR THE DSA FIXED PART
LRETCODE DS   F
LRSNCODE DS   F
        CSVDYNEX MF=(L,DYNEXL)
*
        DS   0D
WORKSIZE EQU   *-WORKAREA
        CEEDSA ,           MAPPING OF THE DYNAMIC SAVE AREA
        CEECAA ,           MAPPING OF THE COMMON ANCHOR AREA
*
*
        END   DYNEXADD
        TITLE 'ADD EXIT ROUTINE'
        PRINT GEN
DYNEXAD2 CEEENTRY PPA=MAINPPA,MAIN=YES,BASE=11,AUTO=WORKSIZE
* =====
        USING WORKAREA,13
        CSVDYNEX REQUEST=ADD,EXITNAME=LEEXIT,MODNAME=MYEXIT2,      X
                DSNAME=MYPDS,RETCODE=LRETCODE,RSNCODE=LRSNCODE,    X
                MF=(E,DYNEXL)
        L     15,LRETCODE
        LTR  15,15           TEST RETURN CODE
        BZ   DYNGOOD
DYNFAIL  NOPR  0
        WTO  'CSVDYNEXIT FAILED',ROUTCDE=12
        B    DONE
DYNGOOD  NOPR  0
        WTO  'CSVDYNEXIT WAS SUCCESSFUL'
DONE     CEETERM RC=0,MODIFIER=0
* =====
*   CONSTANTS
* =====
LEEXIT  DC   CL16'CEE_ABEND_EXIT'
MYEXIT2 DC   CL8'MYEXIT2'
MYPDS   DC   CL44'POSIX.MYEXIT.LOADLIB'
* =====
MAINPPA CEEPPA ,           CONSTANTS DESCRIBING THE CODE BLOCK
* =====
*       THE WORKAREA AND DSA
* =====
WORKAREA DSECT
        ORG   **CEEDSASZ           LEAVE SPACE FOR THE DSA FIXED PART
LRETCODE DS   F
LRSNCODE DS   F
        CSVDYNEX MF=(L,DYNEXL)
*
        DS   0D
WORKSIZE EQU   *-WORKAREA
        CEEDSA ,           MAPPING OF THE DYNAMIC SAVE AREA
        CEECAA ,           MAPPING OF THE COMMON ANCHOR AREA
*
*
        END   DYNEXAD2

```

The following code sample shows a code example that describes the exit routines MYEXIT and MYEXIT2.

```

* =====
* Standard entry code.
* =====
MYEXIT  CSECT
MYEXIT  AMODE 31
MYEXIT  RMODE ANY

```

Condition Management

```

        STM R14,R12,12(R13)   Save caller's registers
        LR  R11,R15           Establish base address
        USING MYEXIT,R11     Identify base register
        STORAGE OBTAIN,LENGTH=WORKALEN,LOC=ANY
        LTR R15,R15          Test return code
        BNZ STOFAIL          Storage not available
        ST  R13,8(R1)        Back-chain the save area
        LR  R13,R1
        WTO 'GET STORAGE SUCCESSFUL'
        WTO 'INSIDE MYEXIT'
* =====
* Process condition.
* =====
        LA  R15,4
        B   DONE
STOFAIL NOPR R0
        WTO 'GET STORAGE FAILED',ROUTCDE=11
* =====
* Standard exit code.
* =====
DONE    NOPR R0
        L   R13,8(,R13)
        L   R14,12(R13)      Reload caller's register 14
        LM  R0,R12,20(R13)  Reload caller's registers 0-12
        BR  R14
*
* =====
* CONSTANTS and SAVE AREA.
* =====
WORKAREA DSECT
SAVE     DC 18F'0'
WORKALEN EQU *-WORKAREA
*
        LTORG
R0       EQU 0
R1       EQU 1           entry: points to parameter list
R2       EQU 2           work register
R3       EQU 3           copy of R1 at entry (preserves value)
R4       EQU 4           A(amount of storage to free)
R5       EQU 5           A(A(storage to be freed)
R6       EQU 6           A(return code)
R7       EQU 7           A(reason code)
R8       EQU 8           Amount of storage to free
R9       EQU 9
R10      EQU 10
R11      EQU 11
R12      EQU 12          code base address
R13      EQU 13          savearea address
R14      EQU 14          entry: return point address
R15      EQU 15          entry: entry point address
*
        END
* =====
* Standard entry code.
* =====
MYEXIT2 CSECT
MYEXIT2 AMODE 31
MYEXIT2 RMODE ANY
        STM R14,R12,12(R13)   Save caller's registers
        LR  R11,R15           Establish base address
        USING MYEXIT2,R11     Identify base register
        STORAGE OBTAIN,LENGTH=WORKALEN,LOC=ANY
        LTR R15,R15          Test return code
        BNZ STOFAIL          Storage not available
        ST  R13,8(R1)        Back-chain the save area
        LR  R13,R1
        WTO 'GET STORAGE SUCCESSFUL'
        WTO 'INSIDE MYEXIT2'
* =====
* Process condition.
* =====

```

```

        LA    R15,0
        B    DONE
STOFAIL NOPR R0
        WTO  'GET STORAGE FAILED',ROUTCDE=11
* =====
* Standard exit code.
* =====
DONE    NOPR R0
        L    R13,8(R13)
        L    R14,12(R13)      Reload caller's register 14
        LM   R0,R12,20(R13)   Reload caller's registers 0-12
        BR   R14
*
* =====
* CONSTANTS and SAVE AREA.
* =====
WORKAREA DSECT
SAVE     DC    18F'0'
WORKALEN EQU  *-WORKAREA
*
        LTORG
R0       EQU  0
R1       EQU  1      entry: points to parameter list
R2       EQU  2      work register
R3       EQU  3      copy of R1 at entry (preserves value)
R4       EQU  4      A(amount of storage to free)
R5       EQU  5      A(A(storage to be freed)
R6       EQU  6      A(return code)
R7       EQU  7      A(reason code)
R8       EQU  8      Amount of storage to free
R9       EQU  9
R10      EQU  10
R11      EQU  11
R12      EQU  12      code base address
R13      EQU  13      savearea address
R14      EQU  14      entry: return point address
R15      EQU  15      entry: entry point address
*
        exit : return code
END

```

Other Language Environment condition manager topics

For information about Language Environment default condition handling, see *z/OS Language Environment Programming Guide*. For information about Language Environment runtime options, see *z/OS Language Environment Programming Reference*.

Language Environment condition information block

Each condition is represented by a condition information block (CIB). The CIB is built by the condition manager and is used as an information repository for data required by the condition handling facilities. The CIB is not presented to user condition handlers and is available only to member condition handlers. The CIB is not intended to be viewed or altered by the user. The complete CIB is listed in *z/OS Language Environment Debugging Guide*.

Errors during condition handling

Every effort should be made to ensure that further exceptions do not occur during the condition handler process. However, errors may still occur. To identify the state (or point in time) of the Language Environment condition manager, a state setting is contained in the CIB. The valid states, constant values, and actions taken by the Language Environment condition manager are listed in Table 49 on page 284.

Condition Management

When a language-specific exception handler determines that it is safe to incur a nested condition, it should alter the CEECIB state variable to indicate nested conditions are tolerated (cib_state_recursion).

Table 49. CEECIB State Variable, Constant values, and associated actions

State Value	Value	Variable Meaning	Condition Manager Actions with Nested Condition
cib_state_enable	1	The language-specific enablement handler is in control. This is set by the Language Environment condition manager.	Terminate the enclave with abend 4087-1.
cib_state_hdl	2	A user condition handler, registered from CEEHDLR, is in control. This is set by the Language Environment condition manager.	Terminate the enclave with abend 4087-2.
cib_state_memb	3	A language-specific exception handler is in control. This is set by the Language Environment condition manager.	Terminate the enclave with abend 4087-3.
cib_state_SF0	4	A language-specific exception handler is in control for stack frame zero. This is set by the Language Environment condition manager.	Terminate the enclave with abend 4087-4.
cib_state_evnt	5	A language-specific exception handler is in control for incidental service. This is set by the Language Environment condition manager.	Terminate the enclave with abend 4087-5.
cib_state_ipat	6	The debug tool is in control. This is set by the Language Environment condition manager.	Call the debug tool event handler indicating this event, then terminate the enclave with abend 4087-6.
cib_state_msg	7	Language Environment message services are being called by the Language Environment condition manager; this is set by the Language Environment condition manager.	Terminate the enclave with abend 4087-7.
cib_state_dump	8	Used when traceback or dump services are being called.	Terminate the enclave with abend 4087-8.
cib_state_Memb_AR_MODE	9	Used for member processing when recursion is allowed.	While in this state, the Language Environment condition manager tolerates the occurrence of a nested condition.
cib_state_ab_term_exit	10	Used when an abnormal termination exit is called; the cib_state_ab_term_exit variable contains the name of the exit.	End the enclave with abend 4087-A.
cib_state_recursion	100	A language-specific user handler is in control, such as a PL/I On Unit. This value is set by the language-specific exception handler. While in this state, the Language Environment condition manager tolerates the occurrence of a nested condition. This is set by subordinate condition handlers and debug tools when calling user code.	Tolerate nested conditions.

HLL conventions and information

To work together with each other, the HLL condition handlers must adhere to a set of conventions. Also, some extensions to the current HLL error handling schemes are required so that the condition handling model is complete. By doing so, a consistent, cooperative view of the condition handling model is produced. This section lists these conventions and requirements on the HLLs, as well as some additional information provided to aid the HLL writer.

HLL condition handling conventions

The HLL condition handling models basically fall into two categories: the stack frame-based model (PL/I), and the Global Error Table (GET) model (FORTRAN). To ensure a consistent view of condition handling in a multi-language thread that can involve both models, cooperation is required among the HLL condition handlers. The conventions for the HLL condition handling are:

- A stack frame is defined as a register save area that is back chained in the logical invocation stack.
- All HLL condition handlers must percolate all unknown conditions. An unknown condition is one for which the HLL has no defined action.
- When an HLL condition handler is called for enablement, unknown conditions must be enabled.
- All (enabled) conditions of severity 0 or 1 must permit a resume at the next sequential instruction without requiring any fix-up. If needed, any critical fix-up can be performed at enablement by the member deciding that the condition was enabled.
- When the action for a given condition is “ignore”, the condition is considered to be **disabled** and the HLL condition handler must return **not enabled** when it is called for enablement.
- Some hardware conditions can be detected through software prior to raising the hardware condition. For example, the software can check for the **ZeroDivide** condition by checking for a zero divisor. If a condition is defined to be enabled by the HLL, and software detects a potential hardware condition, then the equivalent Language Environment condition must be signaled using CEESGL.
- Despite the above, statement-oriented language constructs are most appropriately handled directly in the HLL. Any corresponding condition is defined to be disabled. For example, the ON SIZE clause of a COBOL DIVIDE verb (which includes the logical equivalent of the **ZeroDivide** condition) can be handled by COBOL without ever raising a condition.
- For HLLs that employ the GET model, some conventions must be followed so that in a multi-language application that contains both the GET model HLL and the stack frame based model HLL the two models can work in concert. These conventions are:
 - For enabled conditions, the actions defined in the GET model are divided into two groups:
 1. Fix-up and resume
 2. Other than fix-up and resume
 If the default action is fix-up and resume, then that action must be taken at the owning stack frame.
 - If a user explicitly alters the action that needs to be taken for a particular condition within the GET (for example, registering his own handler to field the condition instead of taking the default action), the user-specified action must be honored.

Condition Management

If the user changes the number of messages that are displayed for a particular condition, then the system action is still enforced at the zeroth stack frame.

- If the condition is presented to a stack frame other than the owning stack frame (implying that the condition occurred in a non-GET language) or if the default action is something other than fix-up and resume, then the HLL condition handler must percolate the condition. Specifically, any existing GET actions that specify termination **must** be changed to percolation. These rules allow current semantics to be followed, but also permit inter-language cooperation.
- If the HLL condition handler for the GET model is called for the zeroth stack frame, the “true” system action must be enforced at this time.
- If a HLL condition handling routine needs to permit nested conditions, and thus would be recursively entered, they must set the state variable in the CIB to *cib_state_recursion*.
- Those HLLs that employ a GET must provide a mechanism to allow a condition to be percolated. Users must be able to specify the percolation action.

HLL condition handling information

The following list provides some information and suggestions intended to be helpful to implementers of HLLs:

- Language Environment math library routines that are called as an intrinsic in a given HLL must behave as if the logic within the math service had been generated in-line by the compiler. That is, the characteristics of the HLL are inherited by the Language Environment math service.
- Some conditions are considered to be HLL-specific. For example, I/O related conditions are currently HLL-specific. Other HLL-specific conditions include the AREA, CONDITION, and SIZE conditions in PL/I (PL/I examples are included in this section for illustrative purposes).
- Enablement is performed for **all** conditions regardless of the origin, hardware or software (CEESGL, for instance).
- Enablement allows the HLL to enforce constructs such as PL/I's prefix conditions and COBOL's ON SIZE clause. (COBOL can generate in-line code to check for this and honor the ON SIZE without ever signaling a condition.)
- The HLLs should use the severity that is contained within the condition representation to advantage. For example, PL/I could signal the PL/I ENDPAGE condition at severity 1. If no handler acted on the condition, Language Environment would take the default action for unhandled severity 1 conditions (in the absence of a feedback token), which is to resume at the next sequential instruction following the signal.

Language Environment-issued abends

Language Environment issues abends for some fatal errors. For these errors, the Language Environment exception manager terminates the process without the subordinate exception handlers being called. Note that all enclaves within the process are terminated.

While executing under CICS, the abend code is the 4-character EBCDIC representation of the abend codes; the reason code is not provided. Reason codes are only included in the CEE100 messages that are issued to the console. In this case, the abend code and the reason code are provided in hexadecimal notation. Reason codes are zero unless stated otherwise.

Language Environment issues user abends with codes of 4000 and above. When Language Environment issues an abend, the normal condition processing does not occur. Language Environment percolates the abend if the abend drives Language Environment's ESTAE exit. User abends of 4000 and above that are not issued by Language Environment are **not** percolated.

The products running under Language Environment should be aware that abend codes that are 4000 through 4095 are reserved for Language Environment use. These abend codes are used by Language Environment and possibly the members to signify that the environment is no longer usable.

In general, other abend codes are intercepted by the Language Environment exception manager. These produce messages and possibly dumps. The philosophy of the Language Environment exception manager is to provide diagnostic messages and not abend.

Chapter 8. Program management

This section describes the functions supported by the Language Environment program manager and how to invoke them. The Language Environment program manager utilizes current underlying system support for load and delete services. The Language Environment program manager is responsible for:

- Loading and deleting of routines
- Managing quick access to library subroutines through library vector tables (LIBVECs)

It should be noted that fetches and dynamic calls remain the responsibility of individual high level languages (HLLs). The HLLs can use the load and delete services of Language Environment to physically load and delete subroutines, but the actual management of routines is up to the HLLs.

Loading and deleting programs in different environments

This section describes load module name support, the search order for the loading and deleting of library subroutines, and the interface to load and delete services. This process can vary, depending on the environment (MVS, CICS or z/OS UNIX System Services).

Under MVS, the search order can be specified on the interface to the CEEPLD2 service. This service allows the CWI writer to specify the search order in the following:

- Search data sets only
- Search the UNIX file system only
- First search data sets, if not found then search the UNIX file system
- First search the UNIX file system, if not found then search data sets

The name specified on a load request can affect the search order. The following rules apply to the name and search order:

- If the single slash ('/') character is anywhere in the name and is not covered by the rules below, then search the UNIX file system only.
- If the characters ('./') are the first 2 characters in the name, then search the UNIX file system only.
- If the characters ('//') are the first 2 characters, then search data sets only. The first 2 characters are then deleted from the name that is passed to the operating system.

When searching data sets, the name is always folded to uppercase. When going to the UNIX file system, the name is passed "as is".

When an MVS module search is performed, the load macro is issued and the usual program search order prevails. When searching for the executable module in the UNIX file system, then the BPX1LOD service is used.

In a CICS environment, Language Environment uses the EXEC CICS load services.

Finally, in a z/OS UNIX System Services environment, under CICS, loading from the hierarchical file system is not supported.

CWI to program management process services

This section describes the interface to the process level load and delete services provided by program management for other Language Environment library routines and the HLLs library routines. AMODE switching is not performed for the process level load and delete services.

CEEZLOD — process load service

This service is used to load routines that are maintained in storage across enclaves. The address of CEEZLOD is held in CEEPCB_ZLOD. It is the user's responsibility to delete routines loaded by this service.

Syntax

void CEEZLOD (*name, name_len, rsvd, epoint, rc*)

```
POINTER *name;  
INT4     *name_len;  
INT4     *rsvd;  
POINTER *epoint;  
INT4     *rc;
```

CEEZLOD

Call the process load service CWI interface as follows:

```
L   R15,CEECAAPCB-CEECAA(,R12)   Get address of PCB  
L   R15,CEEPCB_ZLOD-CEEPCB(,R15) Get address of CEL subroutine  
BALR R14,R15                     Invoke process load
```

name (input)

The address of the name to load.

name_len (input)

The length of the name, in bytes, to load.

rsvd (input)

A reserved field that must be zero.

epoint (output)

The address of the entry point returned as a result of the load.

rc (output)

A return code indicating the success of the service. This was chosen over feedback codes because message services are not yet available during process level initialization. The return codes (in decimal) are defined as follows:

```
00    Successful load  
08    Module not found  
12    Not enough storage to load  
16    Unsuccessful load
```

CEEZDEL — process delete service

This service is used to delete routines that were loaded by CEEZLOD. It is the user's responsibility to delete routines loaded by CEEZLOD. The address of CEEZDEL is held in CEEPCB_ZDEL.

Syntax

void CEEZDEL (*name, name_len, rsvd, rc*)

```

POINTER *name;
INT4    *name_len;
INT4    *rsvd;
INT4    *rc;

```

CEEZDEL

Call the process delete service CWI interface as follows:

```

L   R15,CEECAAPCB-CEECAA(,R12)    Get address of PCB
L   R15,CEEPZDEL-CEEPCB(,R15)    Get address of CEL subroutine
BALR R14,R15                      Invoke process delete

```

name (input)

The address of the name to delete.

name_len (input)

The length of the name, in bytes, to delete.

rsvd (input)

A reserved field that must be zero.

rc (output)

A return code indicating the success of the service. This was chosen over feedback codes because message services are not yet available during process level initialization. The return codes (in decimal) are defined as follows:

```

00    Successful delete
04    Unsuccessful delete

```

CWI to program management region services

This section describes the interface to the region level load and delete services provided by program management for other Language Environment library routines and the HLLs library routines. AMODE switching is not performed for the region level load and delete services.

CEEZLODR — region load service

This service is used to load routines that are maintained in storage across processes. The address of CEEZLODR is held in CEERCB_ZLOD. It is the user's responsibility to delete routines loaded by this service.

Syntax

void CEEZLODR (*name, name_len, rsvd, epoint, rc*)

```

POINTER *name;
INT4    *name_len;
INT4    *rsvd;
POINTER *epoint;
INT4    *rc;

```

CEEZLODR

Call the region load service CWI interface as follows:

```

L   R15,CEECAARCB-CEECAA(,R12)    Get address of RCB
L   R15,CEERCB_ZLOD-CEEPCB(,R15)  Get address of CEL subroutine
BALR R14,R15                      Invoke region load

```

name (input)

The address of the name to load.

name_len (input)

The length of the name, in bytes, to load.

CEEZLODR

rsvd (input)

A reserved field that must be zero.

epoint (output)

The address of the entry point returned as a result of the load.

rc (output)

A return code indicating the success of the service. This was chosen over feedback codes because message services are not yet available during process level initialization. The return codes (in decimal) are defined as follows:

00	Successful load
08	Module not found
12	Not enough storage to load
16	Unsuccessful load

Usage Note:

1. On CICS, when a region level load is requested, the HOLD option is specified when the CICS LOAD command is performed.

CEEZDEL — region delete service

This service is used to delete routines that were loaded by CEEZLODR. It is the user's responsibility to delete routines loaded by CEEZLODR. The address of CEEZDEL is held in CEERCB_ZDEL.

Syntax

void CEEZDEL (*name, name_len, rsvd, rc*)

```
POINTER *name;  
INT4 *name_len;  
INT4 *rsvd;  
INT4 *rc;
```

CEEZDEL

Call the region delete service CWI interface as follows:

```
L R15,CEECAARCB-CEECAA(,R12)    Get address of RCB  
L R15,CEERCB_ZDEL-CEECRCB(,R15)  Get address of CEL subroutine  
BALR R14,R15                    Invoke region delete
```

name (input)

The address of the name to delete.

name_len (input)

The length of the name, in bytes, to delete.

rsvd (input)

A reserved field that must be zero.

rc (output)

A return code indicating the success of the service; message services are not yet available during process level initialization. The return codes (in decimal) are defined as follows:

00	Successful delete
08	Unsuccessful delete

CWI to program management enclave services

This section describes the interface to the enclave load and delete services provided by program management for other Language Environment library routines and the HLLs library routines. The load/delete services are accessed through LIBVEC.

CEEPLD — enclave level load service

The CEEPLD CWI callable service loads the named routine into storage. It uses system services depending on the environment; MVS, z/OS UNIX, or CICS. For a discussion of the search orders for the various host systems, see “Loading and deleting programs in different environments” on page 289.

Syntax

```
void (*CEECELVPLD) (name_len, name, address, mod_size, [fc])
```

```
INT4    *name_len;
CHAR8   *name;
POINTER *address;
INT4    *mod_size;
FEEDBACK *fc;
```

CEECELVPLD

A field in the Language Environment LIBVEC that points to the Program Load CWI. Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L    R15,0096(,R15)
BALR R14,R15
```

name_len (input)

The number of bytes of the routine name to be loaded.

name (input)

The name of the routine to load into storage.

address (output)

The address of the entry point returned as a result of the load.

mod_size (output)

The number of bytes occupied by the newly-loaded load module. If the size cannot be determined, a zero is returned.

fc (output/optional)

A parameter which contains the condition token. The possible conditions are:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3DC	Severity	3
	Msg_No	3500
	Message	Not enough storage available to load [module_name]
CEE3DD	Severity	3
	Msg_No	3501
	Message	[module_name] module not found.
CEE3DE	Severity	3
	Msg_No	3502
	Message	[module_name] module name too long.

Condition		
CEE3DF	Severity	3
	Msg_No	3503
	Message	Load service request for module <i>[module_name]</i> was unsuccessful.
CEE3EJ	Severity	3
	Msg_No	3539
	Message	The load request for program object <i>[module_name]</i> was unsuccessful for the current level of CICS.
CEE3EK	Severity	3
	Msg_No	3540
	Message	The load request for program object <i>[module_name]</i> was unsuccessful.

Usage Notes:

1. Language Environment maintains a list of all modules loaded by the low-level load service. This list is maintained so that Language Environment can delete all the modules it loaded with the low-level load service during the life of the enclave.
2. The user must issue a corresponding low-level delete service for each load service request.
3. The search order for the module is system dependent. For details, see “Loading and deleting programs in different environments” on page 289.
4. The CEEPL0D service does not support loading program objects with deferred load classes; for example, CEEPL0D does not support a reentrant C program that was built with using the Pre-linker utility. The CEEPL0D2 CWI should be used instead.

CEEPL0D2 — enclave/thread level load service

The CEEPL0D2 CWI callable service loads the named routine into storage. The underlying environment (MVS, z/OS UNIX, CICS) system services are used. For a discussion of the search orders for the various host systems, see “Loading and deleting programs in different environments” on page 289. This service can be used to request loads at the thread or enclave level.

Syntax

void CEEPL0D2 (*name_length, name, flag, token, entry_point_address, [fc]*)

```

INT4      *name_length;
CHAR      *name;
INT4      *flag;
INT4      *token;
POINTER   *entry_point_address;
FEED_BACK *fc;

```

CEEPL0D2

Call this CWI interface as follows:

```

L      R15,CEECAACELV-CEECAA(,R12)      CAA address is in R12
L      R15,3948(,R15)
BALR   R14,R15

```


name_length (input)

Specifies the name of a fullword containing the length of the name of file (program) to be loaded. The length can be up to 1023 bytes.

name (input)

Specifies the name of a field of length *name_length* containing the name of the file (program) to be loaded. The file name can be up to 1023 characters long, and does not require a terminating null character.

flag (input)

A fullword binary value indicating the load search order and the service level (enclave/thread) request. The following bits are defined:

0-17 reserved

18-23 flag_search. The value indicates the search order for the load request. The values are defined as follows:

- 0** Data sets only
- 1** UNIX file system only
- 2** Data sets then UNIX file system
- 3** UNIX file system then data sets

24-31 flag_level. The value indicates if this load request is to be done at the enclave or thread level. The values are defined as follows:

- 0** thread
- 1** enclave

token (output)

A 32-bit field of information that is returned. This token must be passed to the query/delete service.

entry_point_address (output)

The address of the entry point returned as a result of the load.

fc (output/optional)

Specifies the optional feedback token where the CWI feedback code will be placed. If this argument is omitted and the CWI will return a feedback code other than CEE000, the CWI will 'raise' this feedback code as an error condition. The following conditions can result from this CWI service.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE3D0	Severity	3
	Msg_No	3512
	Message	A UNIX file system load of module <i>module name</i> failed. The system return code was [<i>return_code</i>]; the reason code was [<i>reason_code</i>].
CEE3DC	Severity	3
	Msg_No	3500
	Message	Not enough storage available to load [<i>name</i>].
CEE3DD	Severity	3
	Msg_No	3501
	Message	[<i>name</i>] module not found.

Condition		
CEE3DE	Severity	3
	Msg_No	3502
	Message	[<i>name</i>] module name to long.
CEE3DF	Severity	3
	Msg_No	3503
	Message	Load service request for module [<i>name</i>] was unsuccessful.
CEE3EJ	Severity	3
	Msg_No	3539
	Message	The load request for program object [<i>module_name</i>] was unsuccessful for the current level of CICS.

Usage Notes:

1. Language Environment maintains a list of all modules loaded by this load service. This list is maintained so that Language Environment can delete all the modules it loaded using this load service during the life of the enclave or thread.
2. If more than one load is issued for a reentrant module, multiple loads are not performed. For the first load request, the module is brought into storage. If any subsequent load requests are made for that module, its address is returned and a use count is maintained for it.
3. The user must issue a corresponding delete service for each load service request.
4. The search order for the module is system dependent. For details, see "Loading and deleting programs in different environments" on page 289.
5. If the file name does not follow the name rules, see "Loading and deleting programs in different environments" on page 289, and the search order requests load from the UNIX file system, then a *getenv()* is done for the 'LIBPATH' environment variable. If the variable exists, it is passed to the BPXILOD service as the path name and BPXILOD would proceed to search for the requested file name in each of the directories specified in the LIBPATH. If LIBPATH does not exist, then it is assumed the path is the current working directory, unless the path name to load already contains a slash, then the LIBPATH is ignored.

CEEPDEL — enclave level delete service

The CEEPDEL CWI callable service deletes the specified routine. The underlying host services are used to delete the routine.

Syntax

```
void (*CEECELVDEL) (name_len, name, [fc])
```

```
INT4   *name_len;
CHAR8  *name;
FEEDBACK *fc;
```

CEECELVDEL

A field in the Language Environment LIBVEC that points to the program delete CWI. Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L      R15,0084(,R15)
BALR   R14,R15
```

name_len (input)

The number of bytes of the following load module name.

name (input)

The name of the routine to be deleted.

fc (output)

A parameter which contains the condition token. The possible conditions are:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3DG	Severity	3
	Msg_No	3504
	Message	Delete service request for module [<i>module_name</i>] was unsuccessful.

Usage Notes:

1. Language Environment maintains a list of all modules loaded by the low-level load service. This list is maintained so that Language Environment can delete all modules it loaded with the low-level load service during the life of the environment.
2. If more than one load is issued for a module, multiple loads are not performed. For the first load request, the module is brought into storage. If any subsequent load requests are made for that module, its address is returned and a use count is maintained for it.
3. If a delete request is made, the use count decrements and, when it reaches zero, the module is deleted from virtual storage.
4. It should be noted that calling a deleted entry point is an error and causes unpredictable results.

CEEPDEL2 — enclave level delete service

The CEEPDEL2 CWI will delete a module that was requested by the CEEPLD2 load service. The use count is decremented and, when it reaches zero, the module is deleted from virtual storage. It should be noted that calling a deleted entry point is an error and causes unpredictable results.

Syntax

```
void CEEPDEL2 (token, [fc])
```

```
INT4      *token;
FEED_BACK *fc;
```

CEEPDEL2

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)      CAA address is in R12
L      R15,3952(,R15)
BALR   R14,R15
```

token (input)

A 32-bit field of information that is returned from the load request.

fc (output/optional)

Specifies the optional feedback token where the CWI feedback code will be

placed. If this argument is omitted and the CWI will return a feedback code other than CEE000, the CWI will “raise” this feedback code as an error condition. The following message identifiers and associated severities may be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE516	Severity	3
	Msg_No	5158
	Message	The z/OS UNIX callable service, BPX1DEL was unsuccessful. The system return code was [return_code], the reason code was [reason_code].

When the message identifier is CEE516, the following qualifier data is also displayed

No.	Name	Input/Output	Type	Value
1	<i>parm_count</i>	Input	INT4	3
2	<i>return_code</i>	Input	INT4	return code from kernel, BPX1DEL function nn codes defined by z/OS UNIX
3	<i>reason_code</i>	Input	INT4	reason code from kernel, BPX1LOD function nn codes defined by z/OS UNIX

CEEPQLD — return information about loaded module

The CEEPQLD CWI returns information about the executable module that was loaded by the CEEPLD2 load service. The following information is returned:

- name length
- name
- load point address
- entry point address
- executable module size in bytes

Syntax

void CEEPQLD (*token, name_len, name, load_point_address, entry_point_address, module_size, [fc]*)

```

INT4      *token;
INT4      *name_length;
CHAR      *name;
POINTER   *load_point_address;
POINTER   *entry_point_address;
INT4      *module_size;
FEED_BACK *fc;
    
```

CEEPQLD

Call this CWI interface as follows:

```

L      R15,CEECAACELV-CEECAA(,R12)      CAA address is in R12
L      R15,3596(,R15)
BALR   R14,R15
    
```

token (input)

A 32-bit field of information that was returned on the load request.

name_length (output)

Specifies the name of a fullword containing the length of the name of file (program) to be queried. The length can be up to 1023 bytes long.

name (output)

Specifies the name of a field of length *name_length* containing the name of the file (program) to be queried. The file name can be up to 1023 characters long, and does not require a terminating null character.

load_point_address (output)

The address of the load point returned as a result of the load.

entry_point_address (output)

The address of the entry point returned as a result of the load.

module_size (output)

The module size in bytes of the executable module.

fc (output/optional)

Specifies the optional feedback token where the CWI feedback code will be placed. If this argument is omitted and the CWI will return a feedback code other than CEE000, the CWI will “raise” this feedback code as an error condition. The following message identifiers and associated severities may be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE3DR	Severity	0
	Msg_No	3515
	Message	No modules were loaded.

CEEPCB_DELETE — system dependent delete service

The system dependent delete service is either a delete service for MVS, CICS or a delete service provided by the user service routines in a pre-initialized environment.

Syntax

void CEEPCB_DELETE (*name_addr, name_length, user_word, load_point, return_code, reason_code*)

```
POINTER *name_addr;
INT4 *name_length;
INT4 *user_word;
INT4 *rsvd_word;
INT4 *return_code;
INT4 *reason_code;
```

CEEPCB_DELETE

Call this CWI interface as follows:

```
L R15,CEECAAPCB-CEECAA(,R12)
L R15,CEEPCB_DELETE-CEEPCB(,R15) System dependent delete
BALR R14,R15
```

CEEPCB_DELETE

name_addr (input)

Fullword Address of the name of module to delete.

name_length (input)

Fixed Binary(31) length of module name.

user_word (input)

A fullword user field.

rsvd_word (input)

A fullword address reserved for future use (input parameter); must be zero.

return_code (output)

Fullword return code from load.

reason_code (output)

Fullword reason code from load. The return and reason codes have the following values:

Return Code	Reason Code	Meaning
0	0	Successful
8	4	Unsuccessful; delete failed
16	4	Unsuccessful — uncorrectable error occurred

CEEPCB_LOAD — system dependent load service

The system dependent load service is either a load service for MVS, CICS or a load service provided by the user service routines in a pre-initialized environment.

Syntax

void CEEPCB_LOAD (*name_addr, name_length, user_word, load_point, entry_point, module_size, return_code, reason_code*)

```
POINTER *name_addr;  
INT4 *name_length;  
INT4 *user_word;  
POINTER *load_point;  
POINTER *entry_point;  
INT4 *module_size;  
INT4 *return_code;  
INT4 *reason_code;
```

CEEPCB_LOAD

Call this CWI interface as follows:

```
L R15,CEECAAPCB-CEECAA(,R12)  
L R15,296(,R15) System dependent load  
BALR R14,R15
```

name_addr (input)

Fullword Address of the name of module to load.

name_length (input)

Fixed Binary(31) length of module name.

user_word (input)

A fullword user field.

load_point (input/output)

Fullword address to load point address of the loaded routine. If zero on output, then load point address was not available.

entry_point (output)

Fullword entry point address of the loaded routine.

module_size (output)

Fixed Binary(31) size of module that was loaded.

return_code (output)

Fullword return code from load.

reason_code (output)

Fullword reason code from load. The return and reason codes have the following values:

Return Code	Reason Code	Meaning
0	0	Successful
0	12	Successful; loaded via SVC 8
4	4	Unsuccessful; module loaded above the 16 megabyte line when in AMODE(24)
8	4	Unsuccessful; load failed
16	4	Unsuccessful; uncorrectable error occurred

CEEPLD — thread level load service

The CEEPLD CWI callable service loads the named routine into storage. Underlying host system services are used. For a discussion of the search orders for the various host systems, see “Loading and deleting programs in different environments” on page 289.

Syntax

```
void (*CEECELVPLD) (name_len, name, address, mod_size, [fc])
```

```
INT4   *name_len;
CHAR8  *name;
POINTER *address;
INT4   *mod_size;
FEEDBACK *fc;
```

CEECELVPLD

A field in the Language Environment LIBVEC that points to the thread-level Program Load CWI. Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L      R15,3492(,R15)
BALR   R14,R15
```

name_len (input)

The number of bytes of the routine name to be loaded.

name (input)

The name of the routine to load into storage.

address (output)

The address of the entry point to the loaded module.

mod_size (output)

The number of bytes occupied by the newly-loaded load module. If the size cannot be determined, a zero is returned.

fc (output/optional)

A parameter which contains the condition token; possible conditions are:

CEEPLODT

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3DC	Severity	3
	Msg_No	3500
	Message	Not enough storage available to load <i>[module_name]</i> .
CEE3DD	Severity	3
	Msg_No	3501
	Message	<i>[module_name]</i> module not found.
CEE3DE	Severity	3
	Msg_No	3502
	Message	<i>[module_name]</i> module name too long.
CEE3DF	Severity	3
	Msg_No	3503
	Message	Load service request for module <i>[module_name]</i> was unsuccessful.
CEE3EJ	Severity	3
	Msg_No	3539
	Message	The load request for program object <i>[module_name]</i> was unsuccessful for the current level of CICS.
CEE3EK	Severity	3
	Msg_No	3540
	Message	The load request for program object <i>[module_name]</i> was unsuccessful.

Usage Notes:

1. Language Environment maintains a list of all modules loaded by the thread-level load service but as yet not deleted. This list is maintained so that Language Environment can delete all as not yet deleted modules it loaded with the thread-level load service during the life of the thread when the thread terminates.
2. The user can issue a corresponding thread-level delete service request for each thread load service request.
3. This service can be used in all environments, MVS, CICS, pre-initialization, multi-threaded and non-multi-threaded.
4. Each call to the service will result in a load system request. The operating system determines whether to load another copy or just return a pointer to another copy already in storage.
5. The search order for the module is system dependent; see "Loading and deleting programs in different environments" on page 289 for details.
6. The CEEPLODT service does not support the loading of program objects with deferred load classes. For example, CEEPLODT will not support a reentrant C program that was built without using the Pre-linker utility. The CEEPLOD2 CWI should be used instead.

CEEPDELT – thread level delete service

The CEEPDELT CWI callable service deletes the specified routine. Underlying host services are used to delete the routine.

Syntax

```
void (*CEECELVPDELT) (name_len, name, [fc])
```

```
INT4   *name_len;
CHAR8  *name;
FEEDBACK *fc;
```

CEECELVPDELT

A field in the Language Environment LIBVEC that points to the thread-level program delete CWI. Call this CWI interface as follows:

```
L   R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L   R15,3496(,R15)
BALR R14,R15
```

name_len (input)

The number of bytes of the following load module name.

name (input)

The name of the routine to be deleted.

fc (output/optional)

A parameter which contains the condition token. The possible conditions are:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3DG	Severity	3
	Msg_No	3504
	Message	Delete service request for module [<i>module_name</i>] was unsuccessful.

Usage Notes:

1. Language Environment maintains a list of all modules loaded by the thread-level load service but as yet not deleted. This list is maintained so that Language Environment can delete all as not yet deleted modules it loaded with the thread-level load service during the life of the thread when the thread terminates.
2. If the load module name is not in the list of modules, the request completes with a feedback code of CEE000 and no delete is done.
3. A thread may issue a thread-level delete only for modules for which it issued the thread-level load.

Library subroutine access

Library subroutines can be accessed from compiler-generated code, user-written assembly language code, and other subroutines. The following items support these methods of access:

- A specially designed vector table called a LIBVEC
- A LIBVEC descriptor (a CSECT stored within the owner's library)

LIBVECS

- Routines to build a LIBVEC and to load and delete library routines

These methods of access include: dynamic load, LIBPACKS, and AMODE switching that is transparent to both the caller and called subroutine.

The sections that follow describe the following items:

- LIBVECS
- LIBPACKS
- LIBVEC descriptors
- LIBVEC initialization
- CWI to LIBVEC low-level services
- Other LIBVEC functions

LIBVECS

A LIBVEC is provided for Language Environment's library routines. A pointer to it is kept in the CAA and in the EDB. Additional LIBVECS are provided for members to use for their own libraries. Members define their own LIBVEC pointer fields.

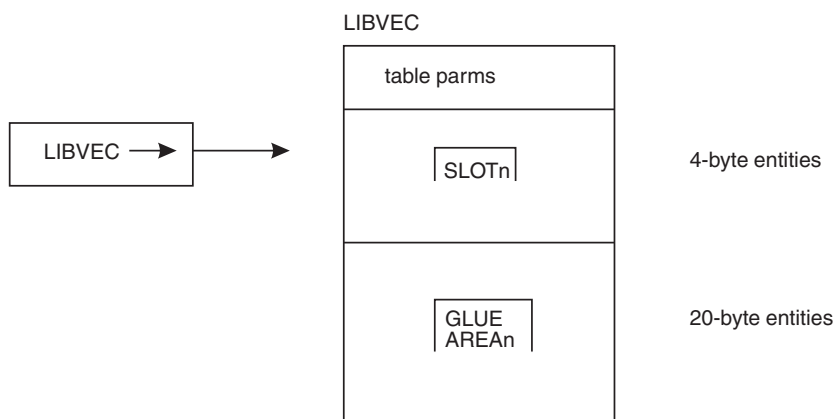


Figure 67. Library subroutine access table (LIBVEC)

As Figure 67 shows, a LIBVEC has three contiguous parts. The first part is made up of fields that contain parameters relating to the table itself. The subroutine loader uses this information. The second part is pointed to by the LIBVEC pointer and contains a 4 byte entry (SLOT n) at a fixed offset for each subroutine. If the subroutine has been loaded and AMODE switching is not being performed, each SLOT n contains the entry point address of its subroutine. Otherwise SLOT n points to its corresponding GLUE AREAn within the third part of the table. The third part is the GLUEAREAn. The size of each GLUE AREAn is 20 bytes. If the subroutine has been loaded and AMODE switching is being performed, the corresponding GLUE AREAn contains AMODE switching code.

Access to library routines through LIBVEC can be *direct* (though a known fixed LIBVEC offset) or *indirectly* (through an externally-defined address constant and a library owner supplied stub). Library owners can choose to provide a macro for direct access to their library routines and a stub for indirect access.

Note: Language Environment library routines can be accessed only through a stub in compiled code; Language Environment provides stubs for this purpose. Library to library calls can access the Language Environment LIBVEC vector table directly.

The following instructions are used to access library routines through LIBVEC directly using fixed offset:

L	R15,libvec_pointer	Get address of LIBVEC
L	R15,xxx(,15)	Get address of Subroutine or GLUE AREAn from LIBVEC slot
BALR	R14,R15	

When the access is indirect, through a stub,

1. The calling program performs the following instruction sequence:

AL	15,=V(LIBSUBx)
BALR	14,15

2. A stub routine is provided for each library routine entry point. It is link-edited with the user program.

There is one type of stub routine that generates the LIBVEC calling sequence. When the stub is entered from the above code, it performs the following calling sequence:

L	15,libvec_pointer	Get address of LIBVEC
L	15,xxx(,15)	Get address of Subroutine or GLUE AREAn from LIBVEC slot
BR	15	

3. If the subroutine has not yet been loaded, then the direct or stub branch goes to GLUE AREAn, which contains code to invoke the subroutine loader.
4. If the subroutine has been loaded and AMODE switching is being done, the direct or stub branch goes to GLUE AREAn, which contains switching code. AMODE switching is done whenever the ALL31(OFF) runtime option is in effect.
5. If the subroutine has been loaded and AMODE switching is not being done, the direct or stub branch goes directly to the subroutine's entry address.

LIBPACKs

A LIBPACK is a packaging mechanism for library packaged subroutines. Language Environment supports two types of LIBPACKs:

- An **INIT LIBPACK** is a LIBPACK loaded during LIBVEC initialization, provided it is not already link-edited with the LIBVEC descriptor (LVD) in the load module. For more information on LVDs, see "LIBPACK relationship to the LVD" on page 306. INIT LIBPACKs not link-edited with LVDs are explicitly loaded during LIBVEC initialization, as a part of the LIBVEC build process.
- A **dynamic LIBPACK** is a LIBPACK loaded upon first reference of a containing entry point.

The use of LIBPACKs improves performance by loading many library routines in one load. It eliminates directory searches for individual library routines.

HLLs using LIBVECS for their own libraries can organize LIBPACKs by RMODE, components, function, or frequency of library subroutine access.

LIBPACK relationship to the LVD

The LVD contains a LIBPACK information section that describes the name of the LIBPACK and if it was link-edited with the LVD. It also contains LIBPACK attributes such as the type of LIBPACK (INIT or dynamic), whether to invoke an address resolver routine, and the first related entry number on a forward chain of entries contained in that particular LIBPACK.

The entries contained in a LIBPACK are forward chained in the LIBPACK and LIBVEC information section of the LVD. For a detailed description of LVDs, see “LIBVEC descriptor (LVD).”

LIBPACK structure

Language Environment LIBPACKs are structured as a CSECT containing a table of WXTRNs and address constants. The dimension of the table must equal the dimension of its related LIBVEC (an entry for a particular routine must be in the same position within the table as its SLOtN is within LIBVEC). For an example LIBPACK CSECT definition, see Figure 68.

```

CEEPLPKA CSECT
CEEPLPKA AMODE ANY
CEEPLPKA RMODE ANY
      WXTRN CEEPxxx
      DC    A(CEEPxxx+X'80000000') Routine xxxx
      WXTRN CEEPyyy
      DC    A(CEEPyyy+X'80000000') Routine yyyy
      DC    A(0) This slot unused in this LIBPACK
      WXTRN CEEPzzz
      DC    A(CEEPzzz+X'80000000') Routine zzzz
      .
      .
      END CEEPLPKA

```

Figure 68. Partial LIBPACK CSECT definition

The X'80000000' value is added to the address constants in the preceding figure to provide an indication to the AMODE switching code that these are AMODE(31) routines. The linkage editor does not provide this for A-type address constants.

LIBPACK creation

A link-edit operation is required to create the Language Environment-supported LIBPACK. An INCLUDE combines the LIBPACK CSECT with library routine CSECTs to produce a load module. HLL library routines that must exist and be preloaded during member initialization, should be packaged an INIT LIBPACK. HLLs should not allow users to tailor this LIBPACK.

Note: Language Environment does not allow tailoring of any of its LIBPACKs.

If LIBVEC owners structure the LIBPACK differently from the Language Environment-supported LIBPACK structure, they should flag the LIBPACK attribute resolve and provide an address resolver routine. For additional information, see note 5 on page 308.

LIBVEC descriptor (LVD)

The LVD provides the entry point names and attributes of routines accessed through the LIBVEC. It also provides information about LIBPACKs associated with the LIBVEC. The LVD is provided by the LIBVEC owner. An LVD consists of 3 sections; its format Figure 69 on page 307 shows the format:

- Header information
- LIBPACK information
- LIBVEC information

```

CEEPLVD  CSECT                LIBVEC descriptor
CEEPLVD  AMODE ANY
CEEPLVD  RMODE ANY           Can reside anywhere in storage
*****
* Header Information Section *
*****
          DC  CL4'LVD '       Eyecatcher
          DC  HL2'n'          Number of LIBPACKs
          DC  HL2'm'          Number of LIBVEC slots
          DC  CL3'ppp'        Name prefix
          DC  XL1'nn'         Version Number
          DC  AL4(aaaaaaa)    Addr of Address Resolver or 0
*****
* LIBPACK Information Section - Contains an entry for each LIBPACK*
*****
          DC  CL8'ssssssss'   LIBPACK name
          WXTRN kkkkkkkk      Address of LIBPACK. See notes
          DC  AL4(kkkkkkkk)   following figure.
*                               LIBPACK attributes
          DC  BL8'flags'      "Dynamic" LIBPACK
BIT0     EQU  X'80'          "Resolve" invoke the Address
BIT1     EQU  X'40'          Resolver Routine
*                               **** Reserved ****
BIT27    EQU  X'37'          **** Reserved ****
          DC  AL1(0)          Entry number of first related
          DC  AL2(k)          entry on chain.
*                               .
*                               .
* (Additional LIBPACK entries are repeat of above 16 bytes)
*                               .
*                               .
*****
* LIBVEC Information Section - Contains an entry for each LIBVEC *
*                               slot in LIBVEC slot order. *
*****
          DC  CL5'eeeeee'     Library routine name suffix
          DC  BL8'flags'      Library routine attributes
BIT0     EQU  X'80'          This routine is part of a LIBPACK
BIT1     EQU  X'40'          Invoke address resolver
BIT2     EQU  X'20'          Do Not Perform AMODE switching
*                               code for this module.
BITS37   EQU  X'3F'          *** Reserved ***
          DC  AL1(j)          LIBPACK number; index into LIBPACK
*                               Information Section)
          DC  AL2(k)          Entry number of next related
*                               entry on chain or 0 to indicate
*                               end of chain (if this routine is
*                               part of a LIBPACK).
*                               .
*                               .
* (Additional LIBVEC entries are repeat of above 9 bytes)
*                               .
*                               .
          END  CEEPLVD

```

Figure 69. LIBVEC descriptor

Note:

1. One or more LIBPACKs (and their library routines) can be directly link-edited with the LVD. In the LIBPACK information section, the address constant of a LIBPACK can also be defined as an A-type address constant of zero or a V-type address constant. A V-type indicates the address of the LIBPACK is always link-edited with the LVD. An A-type address constant of zero indicates the LIBPACK should always be loaded.

2. The LVD must remain in memory for the life of the LIBVEC it represents. It is used by the subroutine loader as the source of the names of the modules to be dynamically loaded. However, it is designed to reside above the 16 megabyte line.
3. The LVD is reentrant and can exist in the (E)LPA.
4. Within the LIBVEC information section, the entries for the routines of a LIBPACK are forward chained together. When the subroutine loader determines that the routine to be loaded is part of a dynamic LIBPACK, it loads that LIBPACK. The chain is then followed to insure that the LIBVEC slots for all routines within the LIBPACK are updated.
5. The LIBVEC owner can provide an address resolver routine (as part of the LVD contained load module). If present and the LIBPACK attribute resolve is flagged, this routine is invoked by LIBVEC initialization or the subroutine loader. It is passed the address of the LVD LIBVEC information section, the entry number of the first related entry on the chain, the address of the load module's external entry point. It returns a temporary LIBVEC table with addresses resolved to their respective LIBVEC slots.

If the LIBPACK attribute resolve is flagged and no address resolver routine is present then a default address resolver internal to Language Environment program management is used. It assumes a table of address constants equal to the dimension of its related LIBVEC are located at the load module's external entry point. An entry for a particular routine must be in the same position within the table as its SLOTh is within LIBVEC. When the address resolver routine is invoked, R1 it points at the following parameter list.

ARXPARMS	DS	0F		
	DS	F		*** Reserved ***
ARXLVD1	DC	A(first-LVD-entry)	IN,	Addr of first LVD info entry.
ARXLVDM	DC	AL4(first-entry-number)	IN,	entry number of first related entry on the chain
ARXMODAD	DC	A(Module-Entry)	IN,	Addr of LIBPACK load module
ARXEPNAD	DC	A(vector-table)	OUT,	Addr of a temporary LIBVEC table with routine addresses in their respective slots.

The address resolver routine must be written as REENTRANT AMODE 31 RMODE ANY. It is entered using BALR 14,15 in AMODE 31 and, therefore, can return using BR 14.

LIBVEC initialization

The Language Environment LIBVEC is initialized as part of Language Environment initialization. Prior to first use, the LIBVECs of other library owners should be initialized as part of the owning member's initialization. A member's LIBVEC is initialized by calling the LIBVEC initialization routine CEEPLVI. It is passed the name or address of an LVD. If the name of the LVD is passed, it must exist as a load module in the LIBVEC owner's library.

During LIBVEC initialization, each INIT LIBPACK structured with the address constant table equal to the LIBVEC entries is ORed directly into the LIBVEC's SLOT section. This eliminates the individual handling of each LIBVEC SLOT and its associated overhead. The LIBVEC initialization routine (CEEPLVI) performs the following functions.

- Load the LVD if the address of the LVD module name was passed.
- Load any INIT LIBPACKs that are not a part of the LVD contained load module.
- Get heap storage for the LIBVEC.

- Insure that all LIBVEC entries are initialized properly.
- Return the address of the LIBVEC.

CWI to LIBVEC low-level services

This section describes the interfaces provided by the library subroutine access sub-component for use by other Language Environment library routines and high-level language library routines. These functions can be accessed as follows.

L	R1,parmptr	R1 points to a parameter list
L	R15,CEECAACELV-CEECAA(,R12)	Get address of CEL LIBVEC
L	R15,CEECELVPxxx-CEECAACELV(,R15)	Get address of CEL library routine GLUE AREAn from CEL LIBVEC slot
BALR	R14,R15	Invoke the library routine

The sections below define the following items for each function:

- The LIBVEC slot offset
- The parameter list format
- The feedback codes returned in GPRs 15 and 0

CEEPLVI — LIBVEC initialization

The CEEPLVI callable service performs the following functions:

- Load any INIT LIBPACKs that are not a part of the LVD contained load module.
- Get heap storage for the LIBVEC.
- Insure that all LIBVEC slots are initialized properly.
- Return the address of the LIBVEC.

During LIBVEC initialization, each INIT LIBPACK structured with the address constant table equal to the LIBVEC entries is ORed directly into the LIBVEC's slot section. This eliminates the individual handling of each LIBVEC slot and its associated overhead.

Syntax

void CEEPLVI (*libvec_descriptor*, *libvec_table*, [*fc*])

```
POINTER *libvec_descriptor;
POINTER *libvec_table;
FEED_BACK *fc;
```

CEEPLVI

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L    R15,0112(,R15)
BALR R14,R15
```

libvec_descriptor (input)

The address of the LIBVEC descriptor module that describes how to build the LIBVEC.

libvec_table (output)

The address of the LIBVEC table.

fc (output/optional)

The parameter in which the callable service feedback code is placed. The following conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3E1	Severity	3
	Msg_No	3505
	Message	Load of [module_name] LIBVEC descriptor module was unsuccessful.
CEE3E2	Severity	3
	Msg_No	3506
	Message	Load of [libpack_name] LIBPACK was unsuccessful.
CEE3E3	Severity	3
	Msg_No	3507
	Message	Not enough storage available for LIBVEC table.
CEE3E4	Severity	3
	Msg_No	3508
	Message	Number of LIBPACKs specified in the LIBVEC descriptor module exceeded maximum of 256 supported. No LIBPACKs were loaded.
CEE3E5	Severity	3
	Msg_No	3509
	Message	Number of LIBVEC slots specified in the LIBVEC descriptor module either exceeds maximum (1024) or less than minimum (1) allowed for a LIBVEC table.

CEEPLVE — verify load/delete

The subroutine verify load/delete CWI is provided for use by functions that require the subroutine to be preloaded. CEEPLVE is explicitly called when needed. When the function is load, it loads the indicated subroutine and updates its LIBVEC fields (unless this has already been done). It always returns the subroutine's address as an output parameter. When the function is delete, it deletes the indicated subroutine and updates its LIBVEC fields as if the subroutine was never loaded (unless this has already been done).

Syntax

void CEEPLVE (*function_code*, *libvec_slot*, *libvec_table*, *libvec_entry*, [*fc*])

```
INT      *function_code;
INT      *libvec_slot;
POINTER  *libvec_table;
POINTER  *libvec_entry;
FEED_BACK *fc;
```

CEEPLVE

Call this CWI interface as follows:

```
L   R15,CEECAACELV-CEECAA(,R12)   Address of CAA in R12
L   R15,0108(,R15)
BALR R14,R15
```


function_code (input)

One of the following values:

- 1 LOAD LIBVEC module
- 2 DELETE LIBVEC module

libvec_slot (input)

The LIBVEC slot offset into the LIBVEC table of which module verify load/delete.

libvec_table (input)

The address of the LIBVEC table built by the LIBVEC initialization routine(CEEPLVI).

libvec_entry (output)

The entry address of the LIBVEC module. This parameter is undefined for function code DELETE.

fc (output/optional)

The parameter in which the callable service feedback code is placed. The following conditions can result from this service. Feedback codes are:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3E6	Severity	3
	Msg_No	3510
	Message	[<i>module_name</i>] module is a member of the "init" LIBPACK [<i>libpack_name</i>] and was not deleted.
CEE3E7	Severity	3
	Msg_No	3511
	Message	Invalid function code.
CEE3E8	Severity	3
	Msg_No	3512
	Message	Verify Load/Delete service request for module [<i>module_name</i>] was unsuccessful.

Note: If the function is DELETE LIBVEC and its entry point is part of an INIT LIBPACK, the LIBPACK is not deleted.

CEEPLVT — LIBVEC termination

The CEEPLVT callable service deletes LIBVEC subroutines and frees up storage obtained for the LIBVEC table during termination of the last enclave of a process. Each LIBVEC owner should invoke the LIBVEC termination CWI for deletion of its LIBVEC subroutines and to free-up the storage obtained for its LIBVEC table.

Syntax

```
void CEEPLVT (libvec_table, [fc])
```

```
POINTER *libvec_table;
FEED_BACK *fc;
```

CEEPLVT

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L    R15,0116(,R15)
BALR R14,R15
```

***libvec_table* (input)**

The address of the LIBVEC table built by the LIBVEC initialization routine (CEEPLVI).

***fc* (output/optional)**

The parameter in which the callable service feedback code is placed. The following conditions can result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3E9	Severity	3
	Msg_No	3513
	Message	LIBVEC termination was unsuccessful.

CEEPPPOS — program object services

The CEEPPPOS CWI provides several functions for requesting information and data for programs that are Language Environment-conforming and contain classes with names prefixed by C_ (particularly C_WSA[64]), and limited support for all other programs.

Syntax

void CEEPPPOS (*function, class_name, entry_point, class_address, class_size, [fc]*)

```
INT4      *function;
CHAR16    *class_name;
POINTER   *entry_point;
POINTER   *class_address;
INT4      *class_size;
FEED_BACK *fc;
```

CEEPPPOS

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,4036(,R15)
BALR R14,R15
```

***function* (input)**

one of the following values, which indicates the requested function:

<i>function Value</i>	Function Name	Meaning
1	OBTAIN	<p>Request Language Environment to obtain and initialize storage for the writable static area of a program object. This function can only be specified with a <i>class_name</i> of C_WSA or CEE_ALL.</p> <p>The C_WSA storage will be obtained by the Loader, except when Language Environment obtains the storage from user HEAP for either CICS or preinitialization environment with user storage routines. Language Environment determines the RMODE of the storage from the HEAP runtime option. Storage is initialized to zeros unless it is explicitly set to a value due to compiler-generated "recipe cards".</p> <p>When the CEE_ALL class is specified, the address and size of C_WSA, C_@@DLLI, and C_@@STINIT are returned in a caller-provided 6 word area. The address for the caller-provided area is input in <i>class_address</i> and the size of the area (24 bytes) is input in <i>class_size</i>.</p>
2	RELEASE	Request Language Environment to release writable static area of a program object. This function can only be specified with the <i>class_name</i> C_WSA.
3	REFRESH	Request Language Environment to refresh writable static area of a program object. This function can only be specified with the <i>class_name</i> C_WSA. This function re-initializes the current WSA to its initial value, which was established by the OBTAIN function.
4	LOCATE	Request Language Environment to locate a specific class within the program object. This function can not be specified with the <i>class_name</i> C_WSA.
5	QUERY	<p>Request Language Environment to return information and characteristics of a program object or load module. When this function is used with the CEE_ALL class, information about the program object or load module specified by <i>entry_point</i> is returned in a caller-provided one word area. The address for the caller-provided area is input in <i>class_address</i> and the size of the area (four bytes) is input in <i>class_size</i>. The following bits are defined in the fullword returned to the caller.</p> <p>0 If this bit is on, then the program object or load module is a DLL (it exports at least one variable and/or function). If this bit is off, then it is not a DLL (but may import variables or functions from a DLL).</p> <p>1 If Query bit (1) is on, then the program object consists of XPLINK compiled functions. If this bit is off, then the program object is entirely non-XPLINK, or it is a load module.</p> <p>2 If this bit is on, then the program object or load module is reentrant and has an associated writable static area (WSA). If this bit is off, then it is not reentrant or does not have a WSA.</p> <p>3–31 Reserved.</p>

***class_name* (input)**

specifies the name of a 16-byte field, which is padded on the right with blanks, that contains one of the following class names:

C_WSA

Writable static area; the OBTAIN, RELEASE, and REFRESH functions are valid for this class.

C_@@DLLI

DLL static initialization routines; the LOCATE function is valid for this class.

C_@@STINIT

C++ Constructor and Destructor routines; the LOCATE function is valid for this class.

C_@@PPA2

Address of PPA2s; the LOCATE function is valid for this class.

CEE_ALL

Provides the address and size of the C_WSA, C_@@DLLI, and C_@@STINIT classes, if the OBTAIN function has been specified. Provides information about the program object or load module, if the QUERY function has been specified.

***entry_point* (input)**

The entry point returned by a Language Environment load service. It is used to identify this program object (or load module).

***class_address* (input/output)**

The address of the class (input or output) or address of a caller-provided area (input).

For the OBTAIN function with the *class_name* specified as C_WSA, the field is an output value and will contain the address of the obtained and initialized writable static area (WSA). For the OBTAIN function with *class_name* CEE_ALL, the field is a required input value and must contain the address of a caller-provided 6-word area.

For the RELEASE and REFRESH functions, this field is a required input value. It should contain the address returned on a previous OBTAIN request.

For the LOCATE function, the field is an output value; it will contain the address of the class requested by *class_name*.

For the QUERY function with *class_name* CEE_ALL, the field is a required input value and must contain the address of a caller-provided one word area.

***class_size* (input/output)**

The size of the class (output) or the size of the caller-provided area (input).

For the OBTAIN function and *class_name* specified as C_WSA, the field is an output value, which will contain the size of the obtained and initialized WSA. For the OBTAIN function with the *class_name* specified as CEE_ALL, this field is an input value and you must specify a size of 24 bytes.

For the LOCATE function, the field is an output value and will contain the size of the class requested by *class_name*.

For the QUERY function with *class_name* CEE_ALL, the field is an input value and must specify a size of four bytes.

For the RELEASE and REFRESH functions, this value is ignored.

***fc* (output/optional)**

A 12-byte feedback code that indicates the results of this service. The following symbolic conditions may result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3EA	Severity	3
	Msg_No	3530
	Message	The service was invoked for a load module.
CEE3EB	Severity	3
	Msg_No	3531
	Message	The entry point was not recognized by Language Environment.
CEE3EC	Severity	3
	Msg_No	3532
	Message	The requested class does not exist in the program object.
CEE3ED	Severity	3
	Msg_No	3533
	Message	The service invoked a system function that was unsuccessful. The system return code was <i>return_code</i> and the system reason code was <i>reason_code</i> .
CEE3EE	Severity	3
	Msg_No	3534
	Message	The requested <i>function</i> is not supported.
CEE3EF	Severity	3
	Msg_No	3535
	Message	The requested <i>class_name</i> is not supported.
CEE3EG	Severity	3
	Msg_No	3536
	Message	Not enough storage was available for the WSA.
CEE3EH	Severity	3
	Msg_No	3537
	Message	The request to release the WSA was unsuccessful.
CEE3EI	Severity	3
	Msg_No	3538
	Message	The request to refresh the WSA was unsuccessful.
CEE3ET	Severity	3
	Msg_No	3549
	Message	The service was invoked for a program object that contains both XPLINK and NOXPLINK-compiled parts.

Usage Notes:

1. Members should invoke this service to obtain the WSA when they are invoked for event code 8 or event code 18 . Members should also invoke this service to release the WSA when they are invoked for event code 41.

2. The CEE_ALL class is provided as a performance improvement; it enables members to obtain the WSA and the address and size of three of the classes (C_WSA, C_@@DLLI, and C_@@STINIT) in one call to CEEPPPOS.
3. When the WSA is obtained for a DLL, Language Environment saves the address and provides the necessary cleanup of the WSA when the implicitly-loaded DLLs are freed at enclave termination. When an explicitly-loaded DLL is explicitly freed, Language Environment will issue the Delete Module Event to allow the member to release the WSA. Otherwise, the member should release the WSA before deleting the program object. An example of a non-DLL program object is a re-entrant C program.
4. The RELEASE and REFRESH functions are not supported for DLL program objects. The member itself must use the RELEASE and REFRESH services with the appropriate serialization.
5. The QUERY function is the only CEEPPPOS function that is valid for program objects or load modules; otherwise a feedback code of CEE3EA is returned.

CWIs for explicit DLL reference

Language Environment provides the following CWI services to support the explicit reference of dynamic load libraries (DLLs).

CEEPLDE — load DLL

The CEEPLDE routine invokes the Language Environment multi-level load routine CEEPLD2, which supports loading a routine from a data set or the UNIX file system. This support causes the writable static area (WSA) for the DLL to be obtained and initialized; it also returns a *dll_token* to represent the DLL on future requests. Note that the user of the *dll_token* should not make any assumptions about the content of the token.

Note:

1. Usage by System Programmer C (SPC) is not supported.
2. Error diagnostics are available through the Language Environment DLL Failure (CEEDLLF) control block chain.

Syntax

```
void CEEPLDE (dll_name, dll_name_length, dll_token, [fc])
```

```
CHARn      *dll_name;
INIT4      *dll_name_length;
TOKEN      *dll_token;
FEED_BACK  *fc;
```

CEEPLDE

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)
L      R15,4016(,R15)
BALR   R14,R15
```

dll_name (input)

name of the DLL to be loaded.

dll_name_length (input)

length of the name of the DLL to be loaded; this length excludes any null terminator at the end of the name.

dll_token (output)

a 32-bit field that represents the DLL that was loaded. The *dll_token* must be

passed to other explicit requests for this DLL, such as: query function (CEEPQDF), query variable (CEEPQDV), and free (CEEPFDE).

fc (output/optional)

An optional 12-byte feedback code that indicates the results of this service. The following symbolic conditions may result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE38V	Severity	2
	Msg_No	3359
	Message	The module or language list is not supported in this environment.
CEE3EU	Severity	3
	Msg_No	3550
	Message	DLL <i>dll_name</i> does not contain a CEESTART CSECT.
CEE3EV	Severity	3
	Msg_No	3551
	Message	DLL <i>dll_name</i> does not contain any C functions.
CEE3F0	Severity	3
	Msg_No	3552
	Message	DLL <i>dll_name</i> does not export any variables or functions.
CEE3F1	Severity	3
	Msg_No	3553
	Message	DLL <i>dll_name</i> is part of a circular list.
CEE3F2	Severity	3
	Msg_No	3554
	Message	There is not enough storage to load the DLL.
CEE3FB	Severity	3
	Msg_No	3563
	Message	Attempted to load DLL <i>dll_name</i> while running C++ destructors.
CEE3FC	Severity	3
	Msg_No	3564
	Message	DLL constructors or destructors did not complete, so DLL <i>dll_name</i> cannot be used.
CEE3FI	Severity	3
	Msg_No	3570
	Message	The DLL name was not valid.
CEE3FJ	Severity	3
	Msg_No	3571
	Message	Storage for writable static was not available for DLL <i>dll_name</i> .

CEEPFDE — DLL free

The CEEPFDE service uses the *dll_token* to identify the DLL to be freed and to invoke the Language Environment multi-level delete routine CEEPDEL2, which supports deleting DLLs from a data set or from the UNIX file system, and to release the WSA of the DLL. After the DLL is deleted, any attempts to use the specified *dll_token* will produce the “invalid *dll_token*” condition.

Note:

1. Usage by System Programmer C (SPC) is not supported.
2. Error diagnostics are available through the Language Environment DLL Failure (CEEDLLF) control block chain.

Syntax

```
void CEEPFDE (dll_token, [fc])
```

```
TOKEN      *dll_token;
FEED_BACK  *fc;
```

CEEPFDE

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)
L      R15,4020(,R15)
BALR   R14,R15
```

***dll_token* (input)**

a 32-bit field that represents the DLL that is to be freed. This is the *dll_token* that is returned by the CEEPLDE service when the DLL was loaded.

***fc* (output/optional)**

an optional 12-byte feedback code that indicates the results of this CWI. The following symbolic conditions may result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3FC	Severity	3
	Msg_No	3564
	Message	DLL constructors or destructors did not complete, so DLL <i>dll_name</i> cannot be used.
CEE3FD	Severity	0
	Msg_No	3565
	Message	The input <i>dll_token</i> was NULL.
CEE3FE	Severity	0
	Msg_No	3566
	Message	There are no DLLs to be freed.
CEE3FF	Severity	0
	Msg_No	3567
	Message	A logical delete was performed for DLL <i>dll_name</i> , but the DLL was not physically deleted.

Condition		
CEE3FG	Severity	0
	Msg_No	3568
	Message	No DLL could be found that matched the input <i>dll_token</i> .
CEE3FH	Severity	2
	Msg_No	3569
	Message	The DLL function was not allowed because destructors are running for the DLL.
CEE3FR	Severity	3
	Msg_No	3579
	Message	Attempted to free DLL <i>dll_name</i> while running C++ destructors.

CEEPQDF — query DLL function

The CEEPQDF routine provides a pointer to an exported function in a specified DLL. Because the value returned is a pointer to a function descriptor, the address of the function's C_WSA is not returned; this information can be found within the descriptor itself.

Note:

1. Usage by System Programmer C (SPC) is not supported.
2. Error diagnostics are available through the Language Environment DLL Failure (CEEDLLF) control block chain.

Syntax

void CEEPQDF (*dll_token*, *func_name*, *func_name_length*, *func_pointer*, [*fc*])

TOKEN *dll_token;
 CHARn *func_name;
 INT4 *func_name_length;
 POINTER *func_pointer;
 FEED_BACK *fc;

CEEPQDF

Call this CWI interface as follows:

```
L     R15,CEECAACELV-CEECAA(,R12)
L     R15,4024(,R15)
BALR  R14,R15
```

dll_token (input)

a 32-bit field that represents the DLL that is to be queried for the named function (*func_name*). This is the *dll_token* returned by CEEPLDE when the DLL was loaded.

func_name (input)

name of the requested function exported from the DLL represented by the *dll_token*.

func_name_length (input)

length of the name of the requested function.

func_pointer (output)

pointer to the requested function, or 0.

fc (output/optional)

an optional 12-byte feedback code that indicates the results of this CWI. The following symbolic conditions may result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3FA	Severity	3
	Msg_No	3562
	Message	There is not enough storage to obtain a function pointer for external function <i>func_name</i> in DLL <i>dll_name</i> .
CEE3FC	Severity	3
	Msg_No	3564
	Message	DLL constructors or destructors did not complete, so DLL <i>dll_name</i> cannot be used.
CEE3FH	Severity	2
	Msg_No	3569
	Message	The DLL function was not allowed because destructors are running for the DLL.
CEE3FK	Severity	0
	Msg_No	3572
	Message	The input <i>dll_token</i> was not available for use.
CEE3FL	Severity	0
	Msg_No	3573
	Message	DLL <i>dll_name</i> does not export any functions.
CEE3FM	Severity	0
	Msg_No	3574
	Message	External function <i>func_name</i> was not found in DLL <i>dll_name</i> .
CEE3FP	Severity	0
	Msg_No	3577
	Message	The external function was not found in DLL <i>dll_name</i> .

CEEPQDV — query DLL variable

The CEEPQDV routine provides the virtual address of a particular exported variable of a specified DLL, which may then be used to reference the DLL's variable.

Note:

1. Usage by System Programmer C (SPC) is not supported.
2. Error diagnostics are available through the Language Environment DLL Failure (CEEDLLF) control block chain.

Syntax

void CEEPQDV (*dll_token*, *var_name*, *var_name_length*, *var_pointer*, [*fc*])

```
TOKEN      *dll_token;
CHARn     *var_name;
INT4      *var_name_length;
POINTER   *var_pointer;
FEED_BACK *fc;
```

CEEPQDV

Call this CWI interface as follows:

```
L   R15,CEECAACELV-CEECAA(,R12)
L   R15,4028(,R15)
BALR R14,R15
```

***dll_token* (input)**

a 32-bit field that represents the DLL that is being queried for the named variable (*var_name*). This is the *dll_token* returned by CEEPLDE when the DLL was loaded.

***var_name* (input)**

name of the requested variable exported from the DLL represented by the *dll_token*.

***var_name_length* (input)**

length of the name of the requested variable.

***var_pointer* (output)**

pointer to the requested variable, or 0.

***fc* (output/optional)**

an optional 12-byte feedback code that indicates the results of the CWI. The following symbolic conditions may result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3FC	Severity	3
	Msg_No	3564
	Message	DLL constructors or destructors did not complete, so DLL <i>dll_name</i> cannot be used.
CEE3FH	Severity	2
	Msg_No	3569
	Message	The DLL function was not allowed because destructors are running for the DLL.
CEE3FK	Severity	0
	Msg_No	3572
	Message	The input <i>dll_token</i> was not available for use.
CEE3FN	Severity	0
	Msg_No	3575
	Message	DLL <i>dll_name</i> does not export any variables.
CEE3FO	Severity	0
	Msg_No	3576
	Message	External variable <i>var_name</i> was not found in DLL <i>dll_name</i> .

Condition		
CEE3FQ	Severity	0
	Msg_No	3578
	Message	The external variable was not found in DLL <i>dll_name</i> .

CWIs for implicit DLL reference

Language Environment provides CWI support for the implicit loading of a DLL, which resolves the DLL's exported symbols with the referencing DLL application's matching imported symbols. CWIs are provided to perform the following functions:

- Trigger Load on Reference for imported variables
- Trigger Load on Call for imported functions
- Get Function support to provide linkage to a function that has been invoked through direct branching to a function pointer (as if it pointed to a function instead of a function descriptor).

In addition, when the enclave terminates, Language Environment provides the support to delete all implicitly-loaded DLLs and any explicitly-loaded DLLs that have not yet been deleted.

Note: The calling routine must ensure that AMODE switching is not necessary between the application and the DLL. The AMODE of the DLL entry is reflected in the high-order bit of the DLL's entry point address. Language Environment does not attempt to detect a mismatch in the AMODE of the DLL and DLL application and does not enforce this restriction.

The following messages are unique to implicit DLL reference. The CWIs will signal these messages as error conditions.

Condition		
CEE3F6	Severity	3
	Msg_No	3558
	Message	DLL <i>dll_name</i> does not export any variables.
CEE3F7	Severity	3
	Msg_No	3559
	Message	External variable <i>var_name</i> was not found in DLL <i>dll_name</i> .
CEE3F8	Severity	0
	Msg_No	3560
	Message	DLL <i>dll_name</i> does not export any functions.
CEE3F9	Severity	0
	Msg_No	3561
	Message	External variable <i>var_name</i> was not found in DLL <i>dll_name</i> .
CEE3FO	Severity	0
	Msg_No	3576
	Message	External function <i>func_name</i> was not found in DLL <i>dll_name</i> .

CEETLOC — stub for trigger load on call

The CEETLOC stub (Figure 70) provides an interface to the CEEPLTC routine on behalf of an unresolved function reference; it then invokes the routine. The function descriptor block (FDCB) (see “FDCB — function descriptor control block” on page 325) contains the address of the CEETLOC before the first reference to the function. This stub is directly referenced from the compilation unit's generated code; it is also link-edited in the same program object as the compilation unit. The code that is generated from the compiler will locate the function descriptor, load registers 15 and 0 from 8 bytes past the start of the function descriptor, and branch to the address in register 15. Register 15 will then contain the address of CEETLOC and register 0 will contain the address of the function descriptor.

```

/*****
*/ CEETLOC - Language Environment "trigger load on call" stub */
*/
*/ This stub contains dual entry points */
*/ + 0 for standard linkage */
*/ +16 for FASTLINK */
/*****
CEETLOC CSECT
CEETLOC RMODE ANY
CEETLOC AMODE ANY
        USING CEETLOC,15
        L    15,CEECAACELV-CEECAA(,12)
        L    15,CEECELVPTLC-CEECELV(,15)
        BR   15
        CNOP 0,8                force alignment on +16
*                                FASTLINK entry point
        L    15,CEECAACELV-CEECAA(,12)
        L    15,CEECELVPTLC-CEECELV(,15)
        B    16(15)
        CEEXCELV
        CEEXCAA
        END  CEETLOC

```

Figure 70. CEETLOC stub for trigger load on call

CEETLOC

Call this CWI interface as follows:

```

L    R15,CEECAACELV-CEECAA(,R12)
L    R15,4004(,R15)
BALR R14,R15

```

- Register Usage:

R15 Entry point address of CEETLOC; for the FASTLINK format, this is CEETLOC + X'10'.

R14 Return address; for FASTLINK, R14 points to the parm list size and the return address is 4 bytes past R14.

R13 DSA address

R12 CAA address

R0 Function descriptor that represents the function that is in the target DLL.

- Sample of compiler-generated code:

```

L    R15,=Q(<d11 func>)
AR   R15,R2
LM   R15,R0,8(R15)
ST   R0,500(,R12)
BALR R14,R15

```

Implicit DLL reference

Note: The compiler-generated code must save the current value of the CEECAACRENT field, set CEECAACRENT from the function descriptor field (CEEFDCEB_DLL_CWSA) at offset X'0C', and restore CEECAACRENT from the saved value. When Language Environment is called from CEETHLOC to invoke the function, it will set CEECAACRENT with the WSA address for the newly-loaded DLL from the value saved in CEEFDCEB_DLL_CWSA. On entry to the DLL function, register 0 will contain the WSA address from CEEFDCEB_DLL_CWSA.

CEETHLOC — stub for trigger load on XPLINK call by name

The function of the CEETHLOC stub (Figure 71) is to interface to the CEEPHTLC (XPLINK Trigger Load on Call) routine on behalf of an unresolved “by-name” function call, which will load the DLL and then invoke the function. This stub is directly referenced from the compilation unit's generated code, and is link-edited in the same program object as the compilation unit. The compiler-generated code will locate the function descriptor, load registers 5 and 6 from the start of the function descriptor, and branch to the stub address in register 6. Once inside the stub, register 7 contains the return address and register 5 contains the 'function_id' identifying this imported function (from the first word of the descriptor). Register 6 is used as a work register to load and branch to the entry point of the CEEPHTLC function.

```
*/*****  
*/  
*/ CEETHLOC - Language Environment */  
*/ "trigger load on XPLINK; call by name" stub */  
*/  
*/ This stub contains one entry point */  
*/ + 0 for XPLINK linkage */  
*/  
*/*****  
CEETHLOC CSECT  
CEETHLOC RMODE ANY  
CEETHLOC AMODE ANY  
*  
DS 0D Doubleword aligned...  
DC XL7'00C300C500C500' eyecatcher text ".C.E.E."  
DC XL1'F4' eyecatcher marker "4"  
*  
ENTRY CEETHLOC  
CEETHLOC XATTR LINKAGE(XPLINK)  
CEETHLOC DS 0D Real entry pt of stub  
L 6,CEECAACELV-CEECA(,12)  
L 6,CEECELVPHTLC-CEECELV(,6)  
BR 6  
*  
CEEXCELV  
CEEXCAA  
END CEETHLOC
```

Figure 71. CEETHLOC stub for trigger load on XPLINK call by name

- Register Usage:
 - R4** DSA address
 - R5** Identifier that represents the XPLINK function that is in the target DLL
 - R6** Entry point address of CEETHLOC
 - R7** Return address to caller
 - R12** CAA address
- Compiler-generated code sample:

Function Descriptor Control Block (FDCB)

LM	r5,r6,dd+0(envreg)	load environment & function addresses
*		... from function descriptor
BASR	r7,r6	call the function
DC	X'4700'	
DC	Y(signed offset/8 to entry point marker)	

FDCB — function descriptor control block

The function descriptor control block (FDCB) contains the information that is needed to call a function from an application. For example, a DLL application “implicitly” references imported functions using compiler-generated code that picks up the 3rd and 4th word of the FDCB. The code then branches to the address in the 3rd word and passes the contents of the 4th word in register 0. An FDCB will be created for every imported function known to the program object. Those functions that are referenced “implicitly” are built initially by the Binder in the WSA of the importing program object; they are then modified dynamically at run time by Language Environment. They are accessible through offsets that are carried in the importing program object's import/export table (IET). An IET is an internal structure that identifies the functions and variables that are imported or exported by an application.

FDCBs will usually reside in the C_WSA because they are placed there by the Binder. However, Language Environment will dynamically construct an FDCB at execution time for any function that is explicitly referenced by a Query DLL function request.

Figure 72 shows the format of the FDCB. It is identical to the old C/C++ function descriptors, with respect to the fields that can be referenced by the compiler-generated code.

Function Descriptor Control Block (FDCB)	
000000	ceeFDCB_Glue - Glue code for direct branches fo function pointer
000008	ceeFDCB_FuncAddr - Pointer to function (initial value = CEETLOC address)
00000C	ceeFDCB_DLL_CWSA - Pointer to exporting DLL's C_WSA (initial value = address of this function descriptor)
000010	ceeFDCB_MoreGlue - Address used by a glue code (above)
000014	ceeFDCB_DLLE - Address of DLL entry in import/export table
000018	ceeFDCB_CEESTARTPtr - Pointer to CEESTART
00001C	ceeFDCB_CWSA - Pointer to this program object's C_WSA

Figure 72. Function descriptor control block (FDCB) format

The fields in the FDCB are defined as follows:

CEEFDCB_GLUE

Code (8 bytes) for “old” modules that branch directly to a function pointer. This field is set by the Binder to a constant that represents a series of

Function Descriptor Control Block (FDCB)

executable instructions that are identical to the glue code in the C/C++ Pre-linker based support. The glue code has the following structure:

HEX Value	Assembler Instruction	Comment
180F	LR R0,R15	Save address of descriptor
58FF0010	L R15,16(R15)	Get address of CEETGFTN
07FF	BR R15	Branch to routine

CEEFD_CB_FUNCADDR

Address of the function. This is part of the descriptor that is picked up and branched to by the compiler-generated code. Initially, this field is set by the Binder to the address of CEETLOC, which is the Language Environment "load on call" stub routine that causes the address of the actual function to be resolved and placed here before invoking the function itself.

CEEFD_CB_DLL_CSWA

Pointer to the C_WSA of the exporting DLL. This is required to establish correct addressability to the DLL's C_WSA before invoking the DLL's function. Initially, this field is set by the Binder to the address of the function descriptor containing it. This enables the CEETLOC code to find the descriptor and thus find the name of the exporting DLL in the program object's Import/Export table. During execution, Language Environment then updates this field to point to the WSA of the exporting DLL.

CEEFD_CB_MOREGLUE

Address of the routine that fixes calls from old modules. This field is set by the Binder to the address of CEETGTFN, a CSECT in the application module, which is the Language Environment stub routine for fixing calls from old modules. The glue code at displacement +0 (mentioned above) picks up this address and branches to it.

CEEFD_CB_DLLE

Address in the referencing program object's Import/Export table entry, from which the name of the exporting DLL can be found. The Binder sets this field.

CEEFD_CB_CEESTARTPTR

Address of referencing program objects CEESTART CSECT. The Binder sets this field to the address of CEESTART, which is a CSECT in every Language Environment-enabled application module that provides a path to other information in the executable program.

CEEFD_CB_CWSA

Address of the program object's C_WSA, which is the base address of the area in which its descriptors for imported symbols are defined. It **must** be restored as the **current** C_WSA upon return from the function. The Loader sets this field to the address of the WSA in which this FDCB is defined.

__bldxfd() — build an XPLINK compatibility descriptor

The __bldxfd() function is passed a function pointer of unknown linkage as input and returns the address of an XPLINK compatibility descriptor that can be used in all situations in an XPLINK-compiled program. It can also be passed to a NOXPLINK-compiled program.

An XPLINK compatibility descriptor can be used like a C function pointer. It is built in such a way that it can be passed among functions compiled using different linkage conventions. For example, NODLL, DLL, XPLINK can use the descriptor. It

can be called to pass control to the function that it represents. The linkage of the caller does not matter. The format of the compatibility descriptor is defined based on the linkage of the function that it represents.

- Calls from a non-XPLINK function through a compatibility descriptor to a non-XPLINK function will first result in the RunOnDownStack CWI getting control to swap to the downward-growing stack.
- Calls from an XPLINK function through a compatibility descriptor to a non-XPLINK function will first result in the RunOnUpStack CWI getting control to swap to the upward growing stack.

Syntax

```
#include <edcwccwi.h>
```

```
char * __bldxfd (void *entry_point);
```

*void *entry_point*

A pointer to the entry point of the function for which an XPLINK compatibility descriptor is to be built. Depending on the linkage of the function that originally took the address of the function, the input could be either:

- a fullword function pointer containing the address of the function (NODLL).
- a 32 byte function descriptor (DLL).
- a 24 byte compatibility descriptor (XPLINK).

`__bldxfd()` returns the following values:

- If successful, `__bldxfd()` returns a pointer to storage containing an XPLINK compatibility descriptor.
- If unsuccessful, `__bldxfd()` does not return. It terminates with a message indicating the cause of failure. It is possible that an invalid *entry_point* was passed as input, or possibly that storage could not be obtained.

Usage Notes:

1. `__bldxfd()` returns one of the following storage addresses :
 - The address passed in *entry_point* if it already represents an XPLINK compatibility descriptor.
 - The address passed in *entry_point* if it represented a DLL-compiled function descriptor. In this case, the descriptor was rewritten in storage as an XPLINK compatibility descriptor.
 - The address of a newly obtained piece of storage if *entry_point* represented a NODLL-compiled function pointer (for example, the real address of a function). An XPLINK compatibility descriptor will be created in the obtain storage.
 - The address passed in *entry_point* if the entry point is not valid.
2. If `__bldxfd()` is passed a DLL function descriptor representing a function in a DLL that has not yet been loaded, that DLL will be loaded by `__bldxfd()` so that the DLL function address and WSA can be inserted into the constructed XPLINK compatibility descriptor. Note that the XPLINK compatibility descriptors always represent functions in loaded DLLs. Taking the address of a function in an XPLINK DLL forces that DLL to be preloaded during the initialization of the application.
3. `__bldxfd()` may cause program checks if it attempts to access an invalid address. The caller might wish to consider having a signal catcher set up to handle SIGSEGV with an appropriate error recovery routine.

4. Mixing XPLINK and non-XPLINK in the same program object is not supported. A NODLL function pointer passed as input must contain the address of a non-XPLINK function. NODLL-compiled functions cannot take the address of imported functions. This is the only way an XPLINK function can currently be directly accessed from non-XPLINK. In this case, `__bldxfd()` will construct an XPLINK compatibility descriptor representing a non-XPLINK function, so no XPLINK environment is necessary.
5. The Vendor Interfaces header file, `<edcwccwi.h>`, is located in member EDCWCCWI of the SCEESAMP data set. In order to include `<edcwccwi.h>` in an application, the header file must be copied into a partitioned data set or a UNIX file system directory in which the z/OS XL C/C++ compiler will find it.

CEETLOR — stub for trigger load on reference

The CEETLOR stub (Figure 73) provides an interface to the CEEPTLR routine on behalf of an unresolved variable reference and then returns to the calling routine. The variable descriptor (see “VDCB — variable descriptor control block” on page 329) contains a zero for the address of the variable before the first reference of the variable by the compiled code. This stub is directly referenced from the compilation unit's generated code; it is also link-edited in the same program object as the compilation unit. The compiler will generate code to locate the variable descriptor, examine the first word of the variable descriptor and, if it is zero, invoke CEETLOR. Register 15 will contain the address of CEETLOR and register 0 will contain the address of the variable descriptor.

```
*/*****  
*/  
*/ CEETLOR - Language Environment "trigger load on reference" stub */  
*/  
*/*****  
CEETLOR CSECT  
CEETLOR RMODE ANY  
CEETLOR AMODE ANY  
        USING CEETLOR,15  
        L    15,CEECAACELV-CEECAA(,12)  
        L    15,CEECELVPTLR-CEECELV(,15)  
        BR   15  
        NOPR 0  
        NOP  0  
        B    1(15)  
        CEEXCELV  
        CEEXCAA  
        END  CEETLOR
```

Figure 73. CEETLOR stub

- Register usage:
 - R15** Entry point address
 - R14** Return address
 - R13** DSA address
 - R12** CAA address
 - R0** Variable descriptor that represents the variable that is in the target DLL.
- Sample of compiler-generated code:

```

      ICM R15,B'1111',=Q(<d11var>)
      BM @4L5
      L R7,0(,R7)
      : OTHER CODE
      :
@4L5 DS 0F
      MVC 132(4,R13),296(R3)
      B 300(,R3)
      =F'1198534796'
      AL R15,128(,R13)
      ST R15,140(,R13)
      ICM R15,B'1111',0(R15)
      EX R0,132(,R13)
      L R15,=V(CEETLOR)
      ST R14,136(,R13)
      ST R0,144(,R13)
      L R0,140(,R13)
      BALR R14,R15
      =F'0'
      LM R14,R0,136(R13)
      B 308(,R3)

```

After the variable descriptor has been updated, the compiler-generated code can obtain the address of the variable within the DLL. To do so, add the value of the CEEVDCB_VARPQCON to the address of the WSA of the compiler-generated code.

VDCB — variable descriptor control block

The variable descriptor control block (VDCB) defines a structure that provides information that is need to reference a variable from an application. For example, a DLL application can use compiler-generated code to “implicitly” refer to imported variables. The compiler-generated code tests the first word for zero and, if it is zero, calls the CEETLOR stub routine to resolve the address of the variable. All VDCBs reside in the C_WSA because they are placed there by the Binder. Figure 74 shows the format of the VDCB.

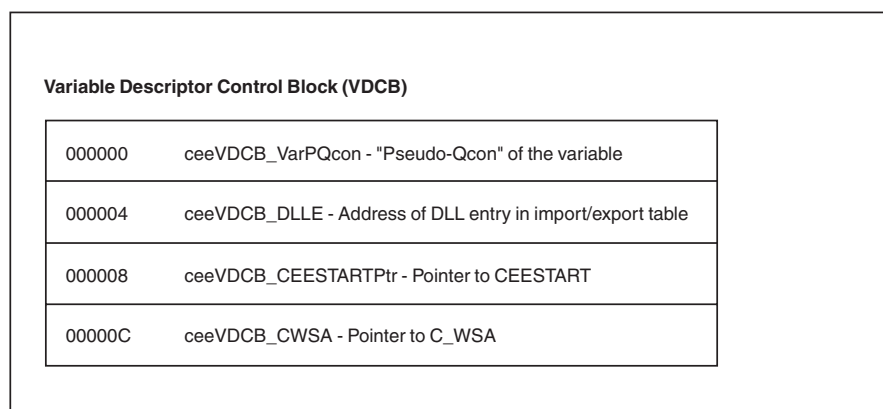


Figure 74. Variable descriptor control block (VDCB) format

The fields in the VDCB are defined as follows:

CEEVDCB_VARPQCON

“pseudo-Qcon” for the variable. A “pseudo-Qcon”, which is a concept introduced in C/C++, is a displacement that gives the address of the referenced DLL's variables when it is added to the base address of the area

Variable Descriptor Control Block (VDCB)

(C_WSA) in which the referencing program object's descriptors are defined. That is, this displacement is an offset from the base address of one area to a storage location in a different area.

CEEVDCB_DLLE

Address in the referencing program object's Import/Export Table (IET), from which the name of the exporting DLL can be found. The Binder sets this field.

CEEVDCB_CEESTARTPTR

The Binder sets this field to the address of CEESTART, which is a CSECT in every Language Environment-enabled application module that provides a path to other information in the executable program.

CEEVDCB_CWSA

Address of this program object's C_WSA, which is the base address of the area in which its descriptors for imported symbols are defined. The Loader sets this field to the address of the WSA in which the VDCB is define. This is used to decode each "pseudo-Qcon" of a DLL's variable, which provides the location of the imported variable on terms of the start of the referencing program object's C_WSA. It is also used, when working with the referencing program object's Import/Export table, to decode its Qcons for the descriptors of its imports.

CEETGTFN — stub for function invocation of old code

The CEETGTFN stub (Figure 75) supports levels of C code that branch directly to function pointers. With this support, all function descriptors are headed by "glue code" that consists of a constant. This constant is a series of instructions to save the contents of register 15 into register 0, load register 15 with a later slot in the same descriptor that contains the address of CEETGTFN, and branch to the CEETGTFN routine.

```
*/*****  
*/  
*/ CEETGTFN - Language Environment "get function" stub */  
*/  
*/*****  
CEETGTFN CSECT  
CEETGTFN RMODE ANY  
CEETGTFN AMODE ANY  
    USING CEETGTFN,15  
    L    15,CEECAACELV-CEECAA(,12)  
    L    15,CEECELVPGTFN-CEECELV(,15)  
    BR   15  
    NOPR 0  
    NOP  0  
    B    1(15)  
    CEEXCELV  
    CEEXCAA  
    END  CEETGTFN
```

Figure 75. CEETGTFN stub

CEETGTFN

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)  
L    R15,4012(,R15)  
BALR R14,R15
```

CWIs to find the writable static area (WSA)

CEEPFWSA — find writable static area (WSA)

The CEEPFWSA provides the ability to locate the writable static area (WSA) associated with a load module or a program object containing a specified entry point within the current enclave.

Syntax

void CEEPFWSA (*entry_point*, *wsa_address*, [*fc*])

```
POINTER *entry_point;
POINTER *wsa_address;
FEED_BACK *fc;
```

CEEPFWSA

Call this CWI interface as follows:

```
L R15,CEECAACELV-CEECAA(,R12)
L R15,2832(,R15)
BALR R14,R15
```

entry_point (input)

The entry point of a function whose WSA address is to be located. The entry point can be the address of the function or the CEESTART of the load module if the load module contains a main or a fetchable subroutine.

wsa_address (output)

The address of the caller provided area in which the WSA address will be returned if the call is successful.

fc (output/optional)

An optional 12-byte feedback code that indicates the results of this service. The following symbolic conditions may result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3EL	Severity	3
	Msg_No	3541
	Message	A writable static area (WSA) associated with the entry point was not found.

Usage Notes:

1. The CEEPFWSA service will verify that the entry point is a valid C/370 or Language Environment style entry point. It will then examine all loaded modules to find one that contains that entry point. Modules can be the main load module or any load modules loaded by fetch(), COBOL dynamic call, PIPI, CEEFETCH, or DLL load. When the load module is found and the load module has a WSA, the *wsa_address* associated with the module will be returned.
2. If the load module is not recognized as a Language Environment-conforming load module, then the feedback code will be CEE3EL and the *wsa_address* is undefined.

3. If the load module is recognized as a Language Environment-conforming load module, and the load module does not have a WSA, then the feedback code will be CEE000 and the `wsa_address` will be zero.
4. If the load module containing the entry point has been fetched more than once, the service will return the WSA of the last fetch.
5. If the load module containing the entry point has been fetched at least once and has been loaded as a DLL, this service will return the WSA associated with the DLL invocation.

__fnwsa() — CWI to find a writable static area

The `__fnwsa()` function returns the address of the writable static area (WSA) associated with the function represented by `entry_point`. `__fnwsa()` can be used when the passed-in `entry_point` is not in the current address space. To access storage outside the current address space, the user must provide the `callback_p` parameter. `callback_p` is a pointer to a user-written function that fetches all data required by `__fnwsa()`. Generally, the `(*callback_p)()` function would obtain the data using some application-dependent method (like BPX1PTR) and move it into the current address space, where `__fnwsa()` can access it directly. If the passed-in `entry_point` is in the same address space and is directly accessible to `__fnwsa()`, `callback_p` can be NULL.

Syntax

```
#include <edcwccwi.h>
```

```
void * __fnwsa (const void * entry_point, void * (*callback_p)(void *data_p, size_t data_l), const void * caa_p);
```

const void * entry_point

a pointer to the entry point of the function or CEESTART of a main or CEESTART of a fetchable subroutine whose WSA address is to be located. `entry_point` can point to a function or CEESTART in another address space or in a place not directly accessible by `__fnwsa()`. If this address is not directly accessible, both the `callback_p` and `caa_p` parameters must not be NULL. The callback function is used to access `entry_point` indirectly.

void * (*callback_p)()

a pointer to a user-provided function that fetches data not normally accessible by `__fnwsa()`. If `callback_p` is NULL, `__fnwsa()` accesses `entry_point` and any other required Language Environment data areas directly in the current address space. All required data must be directly accessible to `__fnwsa()` in this case.

The user-provided `(*callback_p)()` function is passed the address and length of data to access. It must fetch the data in some application-dependent manner, and make the data available in the current address space in a place accessible to `__fnwsa()`. `(*callback_p)()` must return a pointer to the copied data. This data must remain available to `__fnwsa()` until the next call to `(*callback_p)()`, or until `__fnwsa()` returns to its caller, whichever happens first. On subsequent calls, `callback_p()` is allowed to reuse the same data passback area.

There is no provision for `(*callback_p)()` to pass back an error return code, indicating that the requested data could not be obtained. If `(*callback_p)()` cannot return the requested data, it must not return to `__fnwsa()`. When an error occurs, `(*callback_p)()` may:

- `longjmp()` back to some error return point in the user code that called `__fnwsa()`

- ABEND or otherwise terminate abnormally
- exit(), pthread_exit()
- raise a caught signal where the catcher does longjmp() so as not to return to __fnwsa()
- use Language Environment condition management to bypass __fnwsa() after the error and resume in user code
- recover in some other way that does not involve returning to __fnwsa()

__fnwsa() calls (*callback_p)() with two parameters:

void *data_p

data_p is a pointer to the start of the required data. This address might not be in the current address space.

size_t data_l

data_l is the number of bytes of data required. *data_l* will never exceed 16 bytes. If (*callback_p)() cannot pass back the complete data requested, it must not return to __fnwsa().

const void * caa_p

Address of the Language Environment CAA control block, required only if the second parameter of __fnwsa() (that is, *callback_p*) is non-NULL. This is the address of the CAA in the address space containing *entry_point*.

__fnwsa() returns the following values:

- If successful, the WSA address of the function specified by *entry_point* is returned. If the function does not have a WSA, then __fnwsa() returns NULL.
- If unsuccessful, __fnwsa() returns -1 and sets errno to one of the following values:

ESRCH

Indicates that a matching load module could not be found that contains the passed-in *entry_point*.

EINVAL

Occurs if *entry_point* is not a valid C/370 or Language Environment style entry point. This error also occurs if *entry_point* is NULL when __fnwsa() is called.

EMVSPARM

A callback function was supplied as the second parameter, but the CAA address supplied as the third parameter is NULL.

Usage Notes:

1. __fnwsa() may cause program checks if it accesses invalid addresses. This is especially likely to happen if *callback_p* is NULL and the *entry_point* being looked at is not valid. For this reason, the caller should consider having a signal catcher set up to handle SIGSEGV with appropriate error recovery.
2. The __fnwsa() service will verify that the entry point is a valid C/370 or Language Environment style entry point. It will then examine all loaded modules to find one that contains the specified *entry_point*. Modules can be the main load module or any load modules loaded by fetch(), COBOL dynamic call, PIPI, CEEFETCH, or DLL load. When the load module is found and the load module has a WSA, the WSA address associated with the module will be returned.
3. If the load module containing the entry point does not have a WSA, then __fnwsa() will return NULL.

4. If the load module containing the entry point has been fetched more than once, the service will return the WSA of the last fetch().
5. The Vendor Interfaces header file, `<edcwccwi.h>`, is located in member EDCWCCWI of the SCEESAMP data set. In order to include `<edcwccwi.h>` in an application, the header file must be copied into a partitioned data set or a UNIX file system directory in which the z/OS XL C/C++ compiler will find it.

`__static_reinit()` — CWI to reinitialize writable static area

The `__static_reinit()` function reinitializes the writable static area (WSA) of a dynamic link library (DLL). When a DLL is loaded, Language Environment performs static initialization of the WSA. Additionally, C++ static constructors are run during initialization. When a DLL is deleted, C++ static destructors are run and `atexit` routines are unregistered from the `atexit` list during termination.

Syntax

```
#include <edcwccwi.h>
```

```
int __static_reinit (int func_code, void *fcn);
```

int *func_code*

func_code performs termination/initialization and should be `__STATIC_REINIT_FULL`.

void **fcn*

fcn is a DLL handle pointer returned from a previous successful call to the `dlopen()` or `dlopen()` function..

`__static_reinit()` returns the following values:

- If successful, returns 0.
- If unsuccessful, `__static_reinit()` returns -1 and sets *errno* to one of the following values:

EFAULT

Occurs if the *fcn* address is not valid.

EINVAL

Occurs if *fcn* is not a valid DLL handle pointer or if *func_code* is not valid.

Usage Notes:

1. `__static_reinit()` cannot be used with a DLL that is already in use.
2. `__static_reinit()` can only be used with a DLL that has been explicitly loaded once.
3. If a DLL (A) is loaded and Language Environment loads another DLL (B), B still exists if A is reinitialized.
4. The `__static_reinit()` service should not be used while any other DLL is being initialized.
5. The Vendor Interfaces header file, `<edcwccwi.h>`, is located in member EDCWCCWI of the SCEESAMP data set. To include `<edcwccwi.h>` in an application, the header file must be copied into a partitioned data set or a UNIX file system directory in which the z/OS XL C/C++ compiler will find it.
6. Figure 76 on page 335 shows an example of how to use this CWI.


```

...
/* Open a dynamic library and then reinitilizes its WSA*/

#include <edcwccwi.h>
#include <d1fcn.h>

void *handle;
int eret;

handle = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
....
eret = __static_reinit(__STATIC_REINIT_FULL, handle);

```

Figure 76. Example of using `__static_reinit`

CEEDLLF — DLL failure control block

The CEEDLLF control block contains error diagnostics corresponding to an implicit or explicit DLL failure. Diagnostics describing up to 10 of the most recent DLL failures are available in a circular list of CEEDLLF control blocks. When viewing a dump, the in-use CEEDLLF control blocks are displayed from newest to oldest. Table 50 shows the format of the 31-Bit Language Environment CEEDLLF.

Table 50. Format of the 31-Bit Language Environment CEEDLLF

Location	Content
000000	CEEDLLF Eye Catcher
000008	CEEDLLF version number
000009	CEEDLLF flags
00000A	CEEDLLF size
00000C	DLL service requested
00000D	DLL reference type
00000E	DLL explicit load type
00000F	Reserved
000010	Padding
000014	Pointer to previous CEEDLLF control block
000018	Padding
00001C	Pointer to next CEEDLLF control block
000020	Message feedback token
00002C	Padding
000030	Padding
000034	Pointer to DLL name
000038	Padding
00003C	Pointer to symbol name
000040	Length of DLL name
000044	Length of symbol name
000048	DLL service return code or UNIX file system explicit load return code
00004C	DLL service reason code or UNIX file system explicit load reason code
000050	MVS explicit load return code
000054	MVS explicit load reason code
000058	Reserved

DLL Failure Control Block

Table 50. Format of the 31-Bit Language Environment CEEDLLF (continued)

Location	Content
00005C	Reserved

Table 51 shows the format of the 64-Bit Language Environment CEEDLLF.

Table 51. Format of the 64-Bit Language Environment CEEDLLF

Location	Content
000000	CEEDLLF Eye Catcher
000008	CEEDLLF version number
000009	CEEDLLF flags
00000A	CEEDLLF size
00000C	DLL service requested
00000D	DLL reference type
00000E	DLL explicit load type
00000F	Reserved
000010	Pointer to previous CEEDLLF control block
000018	Pointer to next CEEDLLF control block
000020	Message feedback token
000030	Pointer to DLL name
000038	Pointer to symbol name
000040	Length of DLL name
000044	Length of symbol name
000048	DLL service return code or UNIX file system explicit load return code
00004C	DLL service reason code or UNIX file system explicit load reason code
000050	MVS explicit load return code
000054	MVS explicit load reason code
000058	Reserved
00005C	Reserved

Table 52 describes the fields in CEEDLLF.

Table 52. List of CEEDLLF fields

Field	Explanation
CEEDLLF_EYE	The CEEDLLF eye catcher. If eye catcher is in lower case, the CEEDLLF is currently unused. If eye catcher is in upper case, the CEEDLLF has been populated with DLL diagnostics.
CEEDLLF_VERSION	The CEEDLLF version number. 1 This is the first version of the CEEDLLF.

Table 52. List of CEEDLLF fields (continued)

Field	Explanation
CEEDLLF_FLAGS	CEEDLLF flag bits, defined as follows: 0 CEEDLLF_FIRST. Set to 1 if this is the first CEEDLLF control block in the contiguous CEEDLLF chain storage. 1 CEEDLLF_FRST_FAILED. Set to 1 if there was an error when attempting to free the storage allocated to CEEDLLF_DLL_NAME or CEEDLLF_SYMBOL_NAME. 2 CEEDLLF_GTST_FAILED. Set to 1 if there was an error when attempting to allocate storage for CEEDLLF_DLL_NAME or CEEDLLF_SYMBOL_NAME. 3 CEEDLLF_DLLNAME_FAILED. Set to 1 if there was an error when attempting to copy into CEEDLLF_DLL_NAME. 4 CEEDLLF_SYMNAME_FAILED. Set to 1 if there was an error when attempting to copy into CEEDLLF_SYMBOL_NAME. 5-7 Reserved
CEEDLLF_SIZE	Size of the CEEDLLF control block.
CEEDLLF_SERVICE	The DLL service that failed. 0 The DLL service was unknown. 1 The failure occurred during an implicit DLL Load. 2 Failing DLL service was a DLL Load. 3 Failing DLL service was a DLL Open. 4 Failing DLL service was a DLL Query Function. 5 Failing DLL service was a DLL Query Variable. 6 Failing DLL service was a DLL Explicit Symbol Lookup. 7 Failing DLL service was a DLL Close. 8 Failing DLL service was a DLL Free.
CEEDLLF_REFERENCE_TYPE	The DLL reference type. 0 The DLL reference type was unknown. 1 The DLL reference was implicit. 2 The DLL reference was explicit.
CEEDLLF_LOAD_TYPE	The type of load that was attempted by the failing DLL service. 0 A load was not attempted. 1 MVS load was attempted. 2 UNIX file system load was attempted. 3 MVS and UNIX file system loads were attempted.
CEEDLLF_PREV	Pointer to the previous CEEDLLF in the circular chain.
CEEDLLF_NEXT	Pointer to the next CEEDLLF in the circular chain.
CEEDLLF_FBTOK	Message feedback token associated with this failure.
CEEDLLF_DLL_NAME	Pointer to the DLL name. This value is null if there is no DLL name available at the time of failure.
CEEDLLF_SYMBOL_NAME	Pointer to the function or variable name. This value is null if there is no function or variable name available at the time of failure.
CEEDLLF_DLL_NAME_LEN	Length of CEEDLLF_DLL_NAME (the maximum length for a DLL name is 1024 bytes).
CEEDLLF_SYMBOL_NAME_LEN	Length of CEEDLLF_SYMBOL_NAME (the maximum length for a DLL function or variable name is 1024 bytes).
CEEDLLF_RETCODE1	Return code from the DLL service requested or the return code from an explicit UNIX file system load.

DLL Failure Control Block

Table 52. List of CEEDLLF fields (continued)

Field	Explanation
CEEDLLF_RSNCODE1	Reason code from a DLL service requested or the reason code from an explicit UNIX file system load.
CEEDLLF_RETCODE2	Return code from an explicit MVS load.
CEEDLLF_RSNCODE2	Reason code from an explicit MVS load.

Chapter 9. Debugging and performance analysis

Language Environment provides interfaces upon which a debug tool, such as Debug Tool, can be built. The interfaces defined by Language Environment to a debug tool fall into the following classes: callable service, event handlers, and data areas. These interfaces, and the actions Language Environment takes on the behalf of a debug tool, are described in the following sections .

Language Environment also provides interfaces upon which a performance analysis tool, which is often called a profiler, can be built. This support is described in “Performance analysis support” on page 361. Much of this support is similar to the support Language Environment provides for debugging tools. Therefore, a debugging tool and a profiler cannot be used at the same time.

Language Environment-provided CWIs for the debug tool

The following sections describe the CWIs that Language Environment provides for use with the debug tool.

__setHookEvents() — specify execute hook events for target process

The `__setHookEvents()` CWI sets the execute hook events state for all threads owned by the target enclave and referenced using `asfTargetThreadRef` as specified by the `eventsMask` parameter. Callback functions let you provide address space free access to storage in the target process.

Restriction: Because C and C++ linkage conventions are incompatible, `__setHookEvents()` cannot receive a C++ function pointer as one of the callback routine function pointers. If you attempt to pass a C++ function pointer to `__SetHookEvents()`, the compiler will flag it as an error. You can pass a C or C++ function to `__SetHookEvents()` by declaring it as `extern "C"`.

Syntax

```
int __setHookEvents (int eventsMask,  
*asfCallbacks,  
                    const asfTargetRef *asfTargetThreadRef,  
                    const threadSpec  
*reservedForFutureUse);
```

eventsMask

Used as a bit mask to specify which types of instruction hook events to enable and which events to disable. For each bit in `eventsMask` that is set to 1, the corresponding instruction hook event is enabled. For each bit that is set to 0, the corresponding instruction hook event is disabled. Bits that do not correspond to instruction hook events are reserved and must be set to 0. The following macros define the bit values corresponding to the instruction events:xm

```
THOOK_LABEL  
THOOK_STATEMENT  
THOOK_ACALL  
THOOK_DO  
THOOK_IFTRUE
```

__setHookEvents()

```
THOOK_IFFALSE
THOOK_WHEN
THOOK_OTHER
THOOK_POST
THOOK_BCALL
THOOK_GOTO
THOOK_EXIT
THOOK_MEXIT
THOOK_MULTIEVT
THOOK_ALLOC
THOOK_ENTRY
```

const asfCallbackFunctions *asfCallbacks

Specifies the callback functions for copying data between the controlling process and the target process. If the controlling and target processes are the same or if they are running in the same address space, asfCallbacks can be a null pointer. The addresses of the callback functions are specified by the following structure type:

```
typedef struct {
    /******
    /* callback function copies data to controlling */
    /* process buffer from target process memory */
    /******
    asfCallbackResult (*asfGetStoreCallback)(
        void *localDest,
        const asfTargetRef *targetSrce,
        size_t *dataLength);

    /******
    /* callback function copies data to target process */
    /* memory from controlling process buffer */
    /******
    asfCallbackResult (*asfSetStoreCallback)(
        const asfTargetRef *targetDest,
        const void *localSrce,
        size_t *dataLength);
} asfCallbackFunctions;
```

- *asfGetStoreCallback* is a pointer to a function that copies the amount of data specified by **dataLength* bytes from the target process memory specified by *targetSrce* to *localDest*. *localDest* must point to a buffer with a capacity of at least **dataLength* bytes. On return, **dataLength* is set to the number of bytes actually copied into *localDest*. If any of the requested target process data cannot be copied, all bytes starting from the target process address specified by *targetSrce* up to the first non-copyable byte are copied to *localDest*. **dataLength* is set to the number of bytes copied, and (**asfGetStoreCallback*()) returns the appropriate error value. If all the requests are copied successfully, **dataLength* is unchanged and (**asfGetStoreCallback*()) returns *asfResultOK*.
- *asfSetStoreCallback* is a pointer to a function that copies **dataLength* bytes of data from *localSrce* to the target process memory specified by *targetDest*. On return, **dataLength* is set to the number of bytes that could have been copied into *targetDest*. If any of the requested target process data cannot be updated, none of the target process' memory is changed, **dataLength* is set to the difference between the target process address specified by *targetDest* and the next lowest non-updatable target process address, and (**asfSetStoreCallback*()) returns the appropriate error value. If all of the target

process memory was updated successfully, **dataLength* is unchanged and (**asfSetStoreCallback*()) returns *asfResultOK*.

The two callback functions must return an appropriate value to the caller. They must not *exit()*, *longjmp()*, execute a PL/I ON clause or C++ throw statement, or transfer control to any routine that bypasses returning to the caller. The type of a target process memory reference is defined as follows:

```
typedef struct {
    int asid;          /* target address space identifier */
    void *addr;       /* memory address within target address
                       * space */
} asfTargetRef;
```

- *asid* contains the identifier of the address space that contains the referenced target process memory.
- *addr* is the virtual address of the target process memory within the specified address space.

The return type of the address space free callback functions is defined as follows:

```
typedef enum {
    asfResultOK,
    asfResultAddressSpaceNotAvailable,
    asfResultPageNotMapped,
    asfResultPageNotAvailable,
    asfResultPageNotAccessable
} asfCallbackResult;
```

- *asfResultOK* specifies that the callback function returned successfully. Memory in the controlling process or target process is updated as requested.

The remaining values indicate an error in locating or accessing the target process memory. If one of the following values is returned, no memory in the target process is updated. If data is being copied from the target process to the controlling process, the largest contiguous length of memory is copied, starting from the specified target process address:

- *asfResultAddressSpaceNotAvailable*: the *asid* member of the target process memory reference is not valid, or the address space to which it refers is not available to the controlling process.
- *asfResultPageNotMapped*: the target process address space is available to the controlling process, but the specified virtual address is not mapped within that address space.
- *asfResultPageNotAvailable*: the target process address space is available and the virtual address is mapped, but the data contained in that page is not available to the controlling process. For example, the target process memory is paged out and the target process is suspended, or the target process memory is contained in a dump that does not include the requested memory location.
- *asfResultPageNotAccessable*: the target process address space is available, the virtual address is mapped and available, but the controlling process does not have access to the storage because of key, page or segment protection.

__setHookEvents()

const asfTargetRef *asfTargetThreadRef

Specifies the address space identifier and virtual address of the target Language Environment environment anchor associated with a particular target thread in the target enclave. For AMODE 31 programs, this is the address of the CAA, which is loaded into register R12 while the thread is running. If the calling thread is also the target thread, *asfTargetThreadRef* can be a null pointer. If *asfCallbacks* is a null pointer, the *asid* member of **asfTargetThreadRef* is ignored. If *asfCallbacks* is not a null pointer, *asfTargetThreadRef* and *asfTargetThreadRef->addr* must also not be a null pointers.

const threadSpec *reservedForFutureUse

Specifies a null pointer. It is included to simplify future specifications of particular threads, rather than all threads in the target enclave.

Note:

1. **Restriction:** Because C and C++ linkage conventions are incompatible, *__setHookEvents()* cannot receive a C++ function pointer as one of the callback routine function pointers. If you attempt to pass a C++ function pointer to *__setHookEvents()*, the compiler flags it as an error. You can pass a C or C++ function to *__setHookEvents()* by declaring it as extern 'C'.
2. The bit value macros can be bit-wise ORed to calculate the *eventsMask* value.
3. If successful, *__setHookEvents()* returns 0.
4. If an error occurs, the execute hook event state of the target process is unchanged and a negative value is returned:
 - If any parameter is not valid, -1 is returned.
 - If the target process runtime environment does not support instruction hook events, -2 is returned.

CEE3CBTS — pass component broker connector parameters

Language Environment provides the following CWI service to a debugging tool, such as Debug Tool, to pass Component Broker Connector (CBC) debug context parameters.

By using the Attach Debug_Thread function code, the debugger can distinguish between being invoked for debugging all the threads in an environment or for a single specific thread.

Syntax

void CEE3CBTS (*function_code*, *trace_dbg_context_ptr*, *fc*)

```
INT4      *function_code;
POINTER   *trace_dbg_context_ptr;
FEED_BACK *fc;
```

CEE3CBTS

This CWI is callable only from C or C++. The reference to CEE3CBTS is resolved at link-edit time using the SCEELKED data set. Call this CWI interface as follows:

```
#pragma map(CEE3CBTS,"CEE3CBTS")
#pragma linkage(CEE3CBTS, OS)

#define attach_dbg      1
#define start_dbg      2
#define suspend_dbg    3
#define resume_dbg     4
#define stop_dbg       5
```



```

#define attach_dbg_thread 6

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <leawi.h>
#include <ceedcct.h>

struct sess_info {
    int tcpaddress;
    int portid;
    int client_pid;
    int client_tid;
    int client_tcpaddr;
    int debugflow;
} sess_cb;

    _FEEDBACK fc;
    void CEE3CBTS( int, struct sess_info *, struct _FEEDBACK *);

main()
{
    CEE3CBTS(resume_dbg, &(sess_cb), &(fc) );
    if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
        printf("CEE3CBTS failed with message number %d\n",
            fc.tok_msgno);
    }
}

```

function_code (input)

A fullword binary integer with one of the following values:

- 1 Attach Debug
- 2 Start Debug
- 3 Suspend Debug
- 4 Resume Debug
- 5 Stop Debug
- 6 Attach Debug_Thread

trace_dbg_context_ptr (input/output)

This pointer contains the address of the CBC trace/debug context structure. This structure should have the attribute of *inout*. The six elements of this structure are defined as follows:

TCP/IP address (int)

A fullword binary integer containing the TCP/IP address of the workstation debugger GUI.

Debugger Port ID

A fullword binary integer containing the Port ID of the debugger workstation daemon.

Client Process ID

A fullword binary integer containing the Process ID of the client.

Client Thread ID

A fullword binary integer containing the Thread ID of the client.

Client IP address

A fullword binary integer containing the TCP/IP address of the works client.

Debug Flow

A fullword binary integer describing debugger flow within CBC debug scenarios.

fc (output/optional)

The feedback code indicates the result of this service. The following symbolic conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE2F2	Severity	3
	Msg_No	2530
	Message	A debug tool was not available.
CEE2F7	Severity	3
	Msg_No	2535
	Message	Profiler loaded, debug tool was not available.

Note:

1. For Attach Debug, Attach Debug_Thread, or Start Debug, if a debug tool is not loaded, Language Environment loads and calls the debugger and passes the parameters in the call. If the debugger is already loaded, Language Environment calls it and passes parameters in the call. If a debug tool is not available, Symbolic Feedback Code CEE2F2 is returned.
2. For Suspend Debug, Resume Debug, or Stop Debug, if a debug tool is already loaded, Language Environment calls the debugger, passing the parameters in the call. If a debug tool is not available, Symbolic Feedback Code CEE2F2 is returned.

CEEBFBC — build feedback code routine

The CEEBFBC CWI constructs a Language Environment symbolic condition name given a Language Environment 12-byte Language Environment feedback condition name.

Syntax

```
void CEEBFBC (cond_token, cond_name, [fc])
```

```
FEED_BACK *cond_token;
VSTRING *cond_name;
FEED_BACK *fc;
```

CEEBFBC

Call this CWI interface as follows:

```
L    R12,A(CAA)           Get the address of CAA in R12
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,2972(,R15)
BALR R14,R15
```

cond_token (input)

A 12-byte condition token that is constructed from the Language Environment symbolic name. The L_S_Info field is ignored.

cond_name (output)

An 80-byte character string symbolic condition name. The condition name is left-justified and padded right with blanks. If the condition name is unknown, this field is undefined.

fc (output/optional)

The parameter in which the callable service feedback code is placed. The following conditions can result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3A9	Severity	1
	Msg_No	3401
	Message	The condition token was not recognized and the value of the <i>cond_name</i> is undefined.
CEE3AA	Severity	1
	Msg_No	3402
	Message	The condition token passed is invalid and the value of the <i>cond_name</i> is undefined.

Note:

- The following checks are used to determine the validity of the inbound token:
 - Validating the case, the case must be 1 or 2.
 - Validating the severity which must be 0 through 4, inclusive.
 - For case 1 tokens, the severity occurs twice, and they must be consistent.
- If the facility identifier is Language Environment while the IBM-assigned flag is not set, the condition CEEabc is returned or raised and the value of the *cond_name* is undefined.
- If the facility identifier is not Language Environment, the condition CEEabc is returned or raised and the value of the *cond_name* is undefined.
- Language Environment recognizes *cond_tokens* that have Language Environment as the facility identifier with the IBM-assigned flag on, and have a corresponding message within the Language Environment message set. If *cond_token* has a facility ID that is not CEE, Language Environment polls the members.

CEEKRGPM — register pattern match routine

Language Environment provides the following CWI service to a debugging tool, such as Debug Tool, to register a pattern match routine to enable deferred debugging.

Syntax

CEEKRGPM (*pm_addr*, *reserved*, *pm_user*, [*fc*])

```

POINTER *pm_addr;
INT4    *reserved;
POINTER *pm_user;
FEED_BACK *fc;

```

CEEKRGPM

Call this CWI interface as follows:

```

L      R15,CEECAACELV-CEECAA(,R12)
L      R15,68(,R15)
BALR  R14,R15

```

pm_addr (input)

The address of a pattern match routine that is to be registered or, zero, when no pattern match routine should be registered (de-registration). Registering a pattern match routine indicates that deferred debugging is requested.

When deferred debugging has been requested, the pattern match routine is used to compare the name of the routine that is about to be entered to the name of the routine that the user requested to be debugged. If the pattern match routine determines the routine that is about to be entered should be debugged, the pattern match routine can activate the debugger.

The pattern match routine must be a non-XPLink, AMODE 31 routine, with no writable static. The pattern match routine linkage is MVS-style (R1 is a pointer to pointers to the arguments).

The parameters to the pattern match routine are:

Parameter 1

Fullword function code; this value should be 177.

Parameter 2

Pointer to the program name.

Parameter 3

Fullword containing the length of program name field.

Parameter 4

Entry point address of the program that is about to be entered.

Parameter 5

Pointer to the work area provided when the pattern match routine was registered.

reserved (input)

A fullword reserved for future use; this must be set to zero.

pm_user (input)

The address of a work area that is to be passed to the pattern match routine each time it is called

fc (output/optional)

A feedback code that indicates the result of this call; possible values are:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE377	Severity	2
	Msg_No	3303
	Message	The callable service was passed reserved arguments that were not set to zero.

Note: Language Environment supports registration of a pattern match routine in CEEPIPI subroutine environments, with the following considerations:

1. Language Environment will drive the pattern match routine during the CEEPIPI call_sub function. The program name passed will be that of the routine, as stored in the CEEPIPI table. The length of the name will be a value between 1 and 8 and will not include any trailing blanks. The entry point passed will be that of the routine entry point, as stored in the CEEPIPI table.

Language Environment will not pass the function pointer address that may have been created to support routines with writable static.

2. Language Environment will not drive the pattern match routine for the CEEPIPI call_sub_addr, call_sub_addr_nochk, or call_sub_addr_nochk2 functions or when there is no routine name in the CEEPIPI table
3. This service is intended to be used in the assembler user exit (CEEBXITA) as part of enclave initialization. It is also intended for use from within a subroutine that is run in the environment at which time it would register the pattern match routine. Invocation of the pattern match routine begins on the next call into the environment.
4. The user of this service is responsible for loading the pattern match routine and ensuring it remains loaded across CEEPIPI subroutine calls. You should use one of the Language Environment process level load services, such as CEEZLOD, so that the pattern match routine also remains loaded across any enclave termination that may have been triggered by a subroutine. The CEERCB_PMADDR field (see Table 20 on page 74) can be checked for a non-zero value before loading and registering the pattern match routine. This can prevent an additional load and registration call after an enclave termination and subsequent enclave re-initialization in the subroutine environment.

CEEQFBC — query feedback code routine

The CEEQFBC CWI constructs a condition token given a Language Environment symbolic condition name.

Syntax

void CEEQFBC (*cond_name*, *cond_token*, [*fc*])

```
VSTRING *cond_name;
FEED_BACK *cond_token;
FEED_BACK *fc;
```

CEEQFBC

Call this CWI interface as follows:

```
L    R12,A(CAA)           Get the address of CAA in R12
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,2976(,R15)
BALR R14,R15
```

cond_name (input)

A halfword-prefixed character string symbolic condition name.

cond_token (output)

A 12-byte condition token that is constructed from the Language Environment symbolic name. The I_S_Info field is set to binary zero.

fc (output/optional)

The parameter in which the callable service feedback code is placed. The following conditions can result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.

Condition		
CEE3A8	Severity	1
	Msg_No	3400
	Message	The condition name was not recognized and the value of the <i>cond_token</i> is undefined.

Note:

1. If the condition token is unrecognized, the value of *cond_token* is undefined.
2. Language Environment recognizes *cond_name* values that start with CEE and have a corresponding message within the Language Environment message set. If the *cond_name* does not start with CEE, Language Environment polls the members.

CEEQLD — query modules loaded with enclave level load service

Language Environment provides the following CWI service to a debugging tool, such as Debug Tool, to query known modules currently loaded with the Language Environment enclave level load service.

Syntax

void CEEQLD (*function_code*, *load_list_pointer*, [*fc*])

```
INT4      *function_code;
POINTER   *load_list_pointer;
FEED_BACK *fc;
```

CEEQLD

Call this CWI interface as follows:

```
L    R12,A(CAA)           Get the address of CAA in R12
L    R15,CEECAACELV-CEECAA(,R12)
L    R15,2836(,R15)
BALR R14,R15
```

***function_code* (input)**

A fullword binary integer with one of the following values:

```
1      Get load list
2      Free load list
```

***load_list_pointer* (input/output)**

The address of a load list of module information of known modules currently loaded with the Language Environment enclave level load service. For a description of load list, see Figure 77 on page 349. For function code 1 (get load list), Language Environment sets this parameter to the address of a load list. For function code 2 (free load list), Language Environment receives this as an inbound parameter and frees the load list addressed by this pointer.

***fc* (output/optional)**

A 12-byte feedback code passed by reference. If specified as an argument, feedback information (a condition token) is returned to the calling routine. If not specified, and the requested operation was not successfully completed, the condition is signaled to the condition manager. The following symbolic conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3DN	Severity	2
	Msg_No	3511
	Message	Invalid function code.
CEE3DR	Severity	0
	Msg_No	3515
	Message	No known modules currently loaded via Language Environment load service.

Note:

1. The *get load list* function obtains storage for the load list and returns module information of all known modules currently loaded with the Language Environment load service. If no module has been loaded, the *load_list_pointer* returned is a -1.
2. The *free load list* function should be called prior to Language Environment termination.
3. The *free load list* function does not need to be called when the feedback code is CEE3DR. However, it is not invalid to do so.

```

1 qll Based ,                /* QUERY LOAD LIST */
3 qll_header,              /* QLL header */
5 qll_eye Char(4),         /* Eye catcher 'QLL' */
5 qll_version Bin(15),     /* qll version number*/
5 qll_size Bin(15),       /* qll length */
5 qll_num_lod Fixed,      /* qll number loads */
5 qll_span Fixed,        /* qll element size */
3 qll_elem(0:*),          /* load list element */
5 qll_elem_flags Bit(8),  /* ld list elem flags*/
7 * Bit(1),              /* reserved */
7 qll_elem_np Bit(1),    /* 1=pointer to name */
                          /* 0=name upto 8 char*/
7 * Bit(6),             /* reserved */
5 * Fixed(8),           /* reserved */
5 qll_elem_lodtype Fixed(15), /* load type */
                          /* 0 - reserved */
                          /* 1 - reserved */
                          /* 2 - reserved */
                          /* 3 - reserved */
                          /* 4 - UNIX file system load*/
5 qll_elem_lpa Ptr,      /* load point addr */
5 qll_elem_epa Ptr,     /* entry point addr */
5 qll_elem_modsz Fixed, /* module size */
5 qll_elem_name Char(8) /* load mod name */
7 * Ptr,               /* reserved */
9 qll_elem_name1 Fixed, /* name length */
7 qll_elem_namep Ptr,  /* name pointer */
5 * Ptr;              /* reserved */

```

Figure 77. Load list layout

Language Environment provides a CWI that informs a debug tool of the routines that have already been loaded prior to the debug tool's initialization.

CEETGCAA — get next CAA pointer

The CEETGCAA CWI, given a pointer to a CAA, returns a pointer to CAA of the next thread in the enclave.

Syntax

```
void CEECELVTGCAA (caaptr, [fc])
```

```
POINTER *caaptr;
FEED_BACK *fc;
```

CEETGCAA

Call this CWI interface as follows:

```
L R15,CEECAALEOV-CEECAA(,R12)
L R15,236(,R15)
BALR R14,R15
```

caaptr (input/output)

Given a *caaptr* as input, this CWI returns the next *caaptr* in the enclave.

fc (output/optional)

A 12-byte feedback code passed by reference. The following symbolic condition can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.

Note: Upon first call, the *caaptr* value most likely should be register 12 of the active thread. Because this is a loop, this service can then be repeatedly called until the original *caaptr* value is encountered again.

CEETSFB — translate standard feedback token

The CEETSFB CWI constructs a Language Environment standard feedback code from a 12-byte feedback token.

Syntax

```
void (*CEECELVTSFB) (fb_token, sym_fbcode, [fc])
```

```
FEED_BACK *fb_token;
VSTRING *sym_fbcode;
FEED_BACK *fc;
```

CEECELVTSFB

A field in the CEL LIBVEC that points to the Translate Standard Feedback Token CWI. Call this CWI interface as follows:

```
L R15,CEECAACELV-CEECAA(,R12) CAA address is in R12
L R15,3020(,R15)
BALR R14,R15
```

fb_token (input)

A 12-byte condition token that is constructed from the CEL symbolic feedback code. The I_S_Info field will be ignored.

sym_fbcode (output)

An 80-byte character string symbolic condition feedback code. If the condition token is unknown, this field will be undefined.

***fc* (output/optional)**

An optional parameter in which the callable service feedback code will be placed. The following conditions may result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3A9	Severity	1
	Msg_No	3401
	Message	The condition token was not recognized and the value of the <i>sym_fbcode</i> is undefined.
	Programmer Response	Contact your service representative.
	System Action	Value of the feedback token is undefined.
	Explanation	A condition token was not able to be translated into a corresponding condition name.
CEE3AA	Severity	2
	Msg_No	3402
	Message	The condition token passed is invalid and the value of the <i>sym_fbcode</i> is undefined.
	Programmer Response	Contact your service representative.
	System Action	Value of the symbolic feedback code is undefined.
	Explanation	A condition token was determined to be invalid and is not able to be translated into a corresponding condition name.

Note:

1. This CWI is usually called by a member event handler when processing the translate event-event 20.
2. A standard symbolic feedback code consists of the three letter facility ID catenated with message number expressed in base 32.

CEETSFC — translate standard feedback code

The CEETSFC CWI constructs a condition token from a symbolic feedback code in Language Environment standard form.

Syntax

void (*CEECELVTSFC) (*sym_fbcode*, *fb_token*, [*fc*])

```
VSTRING    *sym_fbcode;
FEED_BACK *fb_token;
FEED_BACK *fc;
```

CEECELVTSFC

A field in the CEL LIBVEC that points to the Translate Standard Feedback Code CWI. Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12) CAA address is in R12
L    R15,3024(,R15)
BALR R14,R15
```

***sym_fbcode* (input)**

A halfword-prefixed character string symbolic condition name.

***fb_token* (output)**

A 12-byte condition token that is constructed from the symbolic feedback code. The I_S_Info field will be set to binary zero.

***fc* (output/optional)**

An optional parameter in which the callable service feedback code will be placed. The following conditions may result from this service.

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE3A8	Severity	3
	Msg_No	3400
	Message	The condition feedback code was not recognized and the value of the <i>fb_token</i> is undefined.
	Explanation	The condition name was not able to be translated into a corresponding Language Environment condition code.
	Programmer Response	Contact your service representative.
	System Action	Value of the condition token is undefined.

Note:

1. This CWI is usually called by a member event handler when processing the translate event-event 20.
2. A standard symbolic feedback code consists of the three letter facility ID catenated with message number expressed in base 32.

Debug tool-provided event handlers

One of the most important things a debug tool must do to be called by Language Environment is provide two logical event handlers:

- An event handler to handle general Language Environment events. When the debugger initializes, it must place the address of this event handler in the member list slot that corresponds to the debugger's member identifier. If the debugger has no member identifier, it should not modify any slots in the member list. If that slot is already initialized, then two members are using the same member identifier, and debugger initialization should fail.
- An event handler to handle debug events. The address of this event handler is maintained by Language Environment in the PCB field, CEEPCBDBGEH. When Language Environment initializes, this field is initialized to zero; when Language Environment loads the debug event handler, it sets this field to the address of the debug event handler.

Debug tool event handler

The debug event handler is loadable by Language Environment with the following:

- The name CEEEVDBG
- The `__CEE_DEBUG_FILENAME31` environment variable

If the application is running with the POSIX runtime option set to 'ON', Language Environment checks for one of two new environment variables. If these variable exists, Language Environment uses the value specified as the name of the debug event handler and loads it from the z/OS UNIX file system. This name, combined with the path name (in the z/OS UNIX file system) that is specified in the LIBPATH environment variable, provides the fully qualified path name for the debug event handler. The attempt to load the debug event handler is performed from either the z/OS UNIX file system or a PDS, not both.

For additional information on invoking the debug event handler, see “Event code 16 — Debug Tool event” on page 502. Specification of which debug tool to be used is made at run time by exposing its name to the system for Language Environment to LOAD. A load failure indicates to Language Environment that a debug tool is not available while this program is running. The debug event handler is loaded and initialized when any one of the following occur:

- An initial command string or PROMPT is discovered and the TEST runtime option is in effect.
- The error condition is raised for the first time and the TEST runtime option is in effect with the ERROR suboption specified.
- Any condition is raised for the first time and the TEST runtime option is in effect with the ALL suboption specified.
- A call to CEETEST is made, regardless of the TEST runtime option setting.

Language Environment notifies the debugger of events through the address of the debug tool event handler contained in the CEEPCBDBGEH. The event handler interface is defined in Table 53 and the bit map descriptions are in Table 54 on page 355. The CWI CEE3CBTS event handler interface is defined in Table 55 on page 356.

Table 53. Debugger Language Environment event handler interface

Debug Tool Event	Debug Tool Event Code	Parm 2	Parm 3	Parm 4
Condition raised	101	CIB	result code	
Unhandled condition	103	CIB	result code	
User handler next	105	CIB	<ul style="list-style-type: none"> • 1 • 2 	<ul style="list-style-type: none"> • user handler address • member event handler address
Goto	111	DSA	DSA format	
PIPI Sub Initialization	115			
PIPI Sub Termination	116			
Enclave init	118	creator's EDB		
Enclave term	119			
Thread init	120	creator's CAA		
Debug tool term	121			
Thread term	122			
External entry	123	<ul style="list-style-type: none"> • Parm 2 = DSA (see note) • Parm 3 = cmd string • Parm 4 = INPL • Parm 5 = DSA format 		
Module load	124	DSA	module descriptor	DSA format

Debug Interfaces

Table 53. Debugger Language Environment event handler interface (continued)

Debug Tool Event	Debug Tool Event Code	Parm 2	Parm 3	Parm 4
Module delete	125	DSA	module name	DSA format
Storage free	126	storage	storage length	
Condition promote	127	CIB	result code	
Condition goto	128	DSA	DSA format	
Attention	129			
Debug tool program check	130	result code		
Message redirect	131	msg_text	ddname	
CALL CEETEST	132	DSA (see note 1)	cmd string	DSA format
Execute Hook invocation	133	<ul style="list-style-type: none"> • Parm 2 = DSA • Parm 3 = hook offset • Parm 4 = DSA format • Parm 5 = A buffer containing general purpose registers • Parm 6 = Return address to the routine that was interrupted • Parm 7 = Entry point to the routine that was interrupted 		
mutex_init	140	initializing thread_id	mutex	(for bit mask descriptions, see Table 54 on page 355)
mutex_destroy	141	destroying thread_id	mutex	
mutex_lock	142	owner thread_id	mutex	
mutex_unlock	143	thread_id releasing mutex	mutex	
mutex_wait	144	waiting thread_id	mutex	
mutex_unwait	145	posted thread_id	mutex	
mutex_relock	146	owner thread_id	mutex	
mutex_unrelock	147	owner thread_id	mutex	
cond_init	150	initializing thread_id	condition var	cv attr object
cond_destroy	151	destroying thread_id	condition var	
cond_wait	152	waiting thread_id	condition var	mutex
cond_unwait	153	posted thread_id	condition var	mutex
Initial thread create	160	initial thread_id	nil	stack_size
Initial thread exit	161	initial thread_id		
Pthread create	162	creating thread_id	created thread_id	stack_size
Pthread created	163	created thread_id	nil	stack_size
Pthread exit	164	created thread_id		
Pthread wait	165	joining thread_id	joined thread_id	
Pthread unwait	166	joining thread_id	joined thread_id	
Imminent CAA Chain Addition	167			
CAA Chain Addition Complete	168			

Table 53. Debugger Language Environment event handler interface (continued)

Debug Tool Event	Debug Tool Event Code	Parm 2	Parm 3	Parm 4
Imminent CAA Chain Deletion	169			
CAA Chain Deletion Complete	170			
POSIX fork() imminent	171	thread_id		
In child process	172			
POSIX exec() imminent	173			
Process clean up imminent	174			
Spawn is imminent	175			
UNIX file system load module	176	DSA	UNIX file system module descriptor	DSA format
Delete UNIX file system load module	177	DSA	UNIX file system module name	DSA format
In parent process	178			
After spawn	179			
CALL CEE3CBTS	180	(for parameter descriptions, see Table 55 on page 356)		
rwlock lock for read	181	thread_id	rwlock	
rwlock lock for write	182	thread_id	rwlock	
rwlock wait for read	183	thread_id	rwlock	
rwlock wait for write	184	thread_id	rwlock	
Multiple event Execute Hook invocation	189	<ul style="list-style-type: none"> • Parm 2 = DSA • Parm 3 = hook offset • Parm 4 = DSA format • Parm 5 = A buffer containing general purpose registers • Parm 6 = Return address to the routine that was interrupted • Parm 7 = Entry point to the routine that was interrupted • Parm 8 = Event mask 		

Note:

1. This is the requestor's DSA, which means an HLL library routine DSA is likely the requestor of the Language Environment service or user DSA.
2. If DSA format is 1 in a 64-bit environment, i.e. XPLink DSA, 64-bit address of 64-bit'ized DSA

Table 54. Debugger Language Environment event handler bit mask descriptions

Bit mask	Description
'00000000'X	The object is a private mutex with the non-recursive characteristic.
'00000001'X	The object is a private mutex with the recursive characteristic.
'00800000'X	The object is a shared mutex with the non-recursive characteristic.

Debug Interfaces

Table 54. Debugger Language Environment event handler bit mask descriptions (continued)

Bit mask	Description
'00800001'X	The object is a shared mutex with the recursive characteristic.
'08000001'X	The object is a private rwlock with the recursive characteristic.
'08800001'X	The object is a shared rwlock with the recursive characteristic.

Note:

- Indicators are available for objects that are shared and separate events for each type of lock. This information indicates the shared object has two copies of DBX that run in different address spaces for applications that use a shared mutex or rwlock. The first occurrence of a lock event, and the fact the object is shared, causes a new control structure for this object. That is, when the following unique events occur and the high order bit of the mutex_object content is ON, a control structure with a lock count of one will be created. This makes the view of a shared mutex or rwlock available in the using address space after the originating address space has initialized the shared object.
 - 142** mutex object
 - 181** rwlock object, locking for read
 - 183** rwlock object, locking for write
- Shared mutex and rwlock objects will always be presented even if the NODEBUG option is one of the object's attributes.
- If a shared object acquire event is reported and there is no entry for the lock object, an entry will be created for the object with a lock count of one. Then when an unlock event happens which sets the lock count to zero, the entry for the shared object will be removed.

Table 55. CWI CEE3CBTS event handler interface parameters

Number	Name	Description
1	Function Code	Integer values passed to CEE3CBTS by the invoker of the CWI. <ul style="list-style-type: none"> 1 Attach Debug 2 Start Debug 3 Suspend Debug 4 Resume Debug 5 Stop Debug 6 Attach Debug_Thread
2	TCP/IP address <i>inout</i>	A fullword binary integer containing the TCP/IP address of the debugger GUI.
3	Debugger port ID <i>inout</i>	A fullword binary integer containing the port ID of the debugger daemon.
4	Client Process ID <i>inout</i>	A fullword binary integer containing the Process ID of the client.
5	Client Thread ID <i>inout</i>	A fullword binary integer containing the Thread ID of the client.
6	Client IP address <i>inout</i>	A fullword binary integer containing the IP address of the client.
7	Debug Flow <i>inout</i>	A fullword binary integer containing debug flow information as provided by CBC.

CAA

A fullword binary integer that contains the address of the CAA.

CIB

A fullword binary integer that contains the address of the CIB.

DSA

A fullword binary integer that contains the address of the DSA.

DSA format

A fullword binary integer set to one of the following:

0 The format of the DSA is a standard OS linkage register save area (with/without Language Environment fields including NAB).

1 The format of the DSA is XPLINK style.

General purpose registers

A 64-byte buffer containing the general purpose registers stored in order 0 to 15 at the time the debug hook was executed. If the debugger changes these register values, the new values will be used when control is returned to the routine that executed the debug hook.

return_address

A fullword pointer containing the address of the instruction where control will be returned to the routine that executed the debug hook. If the debugger changes this address, control will be returned to the new location.

entry_ptr

A fullword pointer containing the address of the entry point of the routine that contains the debug hook.

EDB

A fullword binary integer that contains the address of the EDB.

module name

A halfword-prefixed string of the module name being deleted.

UNIX file system module name

A fullword-prefixed string of the module name being deleted.

module descriptor

A structure describing the module that was just loaded. The structure is as follows:

```
dcl 1 module descriptor,
    3 load point pointer,
    3 module size fixed,
    3 entry point pointer,
    3 name length fixed(15),
    3 module name char(255);
```

UNIX file system module descriptor

A structure describing the module that was just loaded. The structure is as follows:

```
dcl 1 UNIX file system module descriptor,
    3 load point pointer,
    3 module size fixed,
    3 entry point pointer,
    3 name length fixed(31),
    3 module name char(255);
```

result code

A fixed(31) binary value action for condition manager to take. The supported values are:

110 Resume at the resume cursor

120 Percolate to next condition handler

Debug Interfaces

storage length

A fixed(31) binary value containing the number of bytes of storage.

cmd string

A halfword-prefixed string containing the debug command.

msg_text

A halfword-prefixed string of the text that is transmitted by Language Environment message services.

ddname

An 8-byte character string, left-justified, padded right with blanks of the target ddname.

INPL

The Initialization Parameter List as passed to CEEINT. For the format of the INPL, see Figure 54 on page 151.

start_rtn

A function pointer to the start routine for the pthread.

thread_id

An 8-byte thread identifier.

mutex

A pointer to a mutex object.

recursive

A recursive type mutex.

nonrecurs

A nonrecursive type mutex.

condition var

A pointer to a condition variable object.

cv attr object

A pointer to a condition variable attributes object.

stack_size

A stack size attribute (in bytes) of initial or created thread.

nil

Unused; null pointer.

event mask

a fullword binary value in which each bit represents a different hook event. When the bit is '1'b, the event occurred. The values of the bits are:

Bit	Event
0-11	Not used
12	Multiple Event Hook
13	Allocate Descriptor Built
14	Block Entry
15	Not used
16	User label
17	Begin of statement
18	Call return
19-20	Not used
21	Start of loop
22	If evaluated TRUE
23	If evaluated FALSE
24	Switch/case/select choice start
25	Switch/case/select default start
26	Multiple flows join
27	Not used


```

28      Call begin
29      Goto
30      Procedure exit
31      Multiple exit

```

Note:

1. A message is issued if the load fails because the debug tool is not available.
2. All parameters are passed by reference.
3. Return codes (in decimal) are placed in R15
 - 00 Success
 - 16 Critical error in the debug tool; do not invoke again.
4. The debugger signals a CEE2F1 condition when it needs to quit from a nested enclave.

Language Environment actions for the interactive debug tool

This section discusses the actions Language Environment takes on behalf of a debug tool.

Language Environment parses the TEST runtime option on behalf of the debug tool and sets the appropriate flags within the Language Environment options control block. Language Environment sets the initial values for the test level and the debug tool event handler in the PCB. After its initial setting during the initialization of the first enclave within the process, this field is updated only by debug tool commands such as the SET TEST command. It is not influenced by nested enclave invocations. For every new enclave spawned and every thread being terminated, if the debug tool has been initialized, Language Environment thread initialization/termination calls the debug event with an enclave initialization or termination event code.

If the debug tool has been initialized, messages can be directed to the Language Environment message file are delivered to the debug tool by calling the debug event handler. In addition, the Language Environment error handler calls the debug event handler for all enabled conditions. The debug event handler is called after the enablement phase and prior to calling the condition handlers. It is also called when a condition is promoted. Upon the occurrence of an attention interrupt, Language Environment calls the debug event handler with an event code indicating an attention interrupt. The debug tool can set hooks, process the event within certain restrictions, and wait for a synchronization point.

Language Environment interactive debug data areas

Language Environment provides data areas for a debug tool's use. These areas are described in this section. The CAA fields are as follows:

- Initial command string address and length is contained within the Language Environment options control block.
- The TEST option's command file ddname is contained within the Language Environment options control block.
- Indication of ALL, ERROR, or NONE TEST suboption is contained within the Language Environment options control block.
- Any debug tool can provide an event handler. The address of this handler should be placed in the member list slot for its member identifier, during debugger initialization, allowing processing of normal events. Debug type events

are passed to one of the debug event handlers, CEEEVDBG or __CEE_DEBUG_FILENAME31. The two event handlers can be the same routine, if desired.

Execute hook support

Language Environment gives you the capability to establish an exit that gains control when a compiled execute hook (EX) is enabled and executed. The user-provided exit is identified by the HLL user exit (CEEBINT) that is invoked during initialization of the Language Environment environment. Language Environment owns the HLL user exit and provides support for the execute hook exit.

The compiled execute hook can be a single event hook or a multiple event hook. A multiple event hook represents the simultaneous of more than one execute hook event. The multiple event hook collapses multiple EX instructions into a single EX instruction, followed by a NOP instruction.

Language Environment initialization:

- Establishes the address of the hook handler entry point
- Sets the hook handler suffix
- Sets the hooks (CAA+X'01A8' thru CAA+X'01F0' for a length of X'48') to X'0700',S(CEECAAUDHOOK)
- Sets the hook handler prefix

Invoking the event handler:

- Single event hook:

If the debugger has been initialized when a single event hook is enabled and executed, the debugger event handler is invoked with the following interface:

1. Event code 133
2. A DSA that was in control when the hook was executed
3. The offset of the hooks within the hook set that was executed (a multiple of 4 ranging from 0 to 15 inclusive)
4. DSA format
5. A buffer containing general purpose registers
6. Return address to the routine that was interrupted
7. Entry point to the routine that was interrupted

- Multiple event hook:

If the debugger has been initialized when a multiple event hook is enabled and executed and the hook for at least one of the events is active, the debugger event handler is invoked with the following interface:

1. Event code 189
2. A DSA that was in control when the hook was executed
3. The offset of a multiple event hook is a specific number determined by the events
4. DSA format.
5. A buffer containing general purpose registers
6. Return address to the routine that was interrupted
7. Entry point to the routine that was interrupted
8. Event mask

In addition, R12 points to the CAA.

To enable a particular execute hook, set the first 2 bytes of the hook to X'45C0'. To disable a particular execute hook, set the first 2 bytes of the hook to X'0700'. No other values should be used for these first 2 bytes.

Performance analysis support

This section describes Language Environment's support for performance analysis, or profiler tools. You can use a profiler tool to determine the performance level of an application; for example, trace data from a profiler tool can reveal the areas of an application that require the most processing time.

The C/C++ Performance Analyzer is available with the IBM C/C++ Productivity Tools for z/OS product. Use the Performance Analyzer to help analyze, understand, and tune your C and C++ applications for improved performance.

Profile tool event handler

The profile event handler is loadable by Language Environment with the name CEEEVPRF. The profiler event handler is loaded and initialized if the PROFILE runtime option is in effect and the TEST runtime option is not specified.

Reminder: If the TEST runtime option is specified, the PROFILE runtime option is ignored and a profiler tool is not loaded. A load failure occurs if Language Environment cannot find the CEEEVPRF routine or if the routine is not available.

The CEEPCBPRFEH field of the PCB contains the address of the profiler event handler. Language Environment uses this address to notify the profiler tool of certain events. These events, which are described in Table 56, are a subset of the notifications and parameters that Language Environment passes to the debug tool event handler.

Table 56. Profile tool — Language Environment event handler interface

Profile Tool Event	Profile Tool Event Code	Parm 2	Parm 3	Parm 4
Condition raised	101	CIB	result code	
Unhandled condition	103	CIB	result code	
Enclave init	118	creator's EDB		
Enclave term	119			
Thread init	120	creator's CAA		
Profile tool term	121			
Thread term	122			
External entry	123	DSA address (see note)	profiler invocation string	<ul style="list-style-type: none"> • Parm 4 = INPL • Parm 5 = DSA format
Condition promote	127	CIB	result code	

Performance Analysis Support

Table 56. Profile tool — Language Environment event handler interface (continued)

Profile Tool Event	Profile Tool Event Code	Parm 2	Parm 3	Parm 4
Execution Hook invocation	133	<ul style="list-style-type: none"> • Parm 2 = DSA • Parm 3 = hook offset • Parm 4 = DSA format • Parm 5 = A buffer containing general purpose registers • Parm 6 = Return address to the routine that was interrupted • Parm 7 = Entry point to the routine that was interrupted • Parm 8 = Eight-byte clock value returned by the STORE Clock (STCK) instruction • Parm 9 = Eight-byte elapsed CPU time in microseconds returned by the TIMEUSED assembler service 		
Initial thread create	160	initial thread_id	nil	stack_size
Initial thread exit	161	initial thread_id		
Pthread create	162	creating thread_id	created thread_id	stack_size
Pthread created	163	created thread_id	nil	stack_size
Pthread exit	164	created thread_id		
POSIX fork() imminent	171	thread_id		
In child process	172			
POSIX exec() imminent	173			
Process clean up imminent	174			
Spawn is imminent	175			
In parent process	178			
After spawn()	179			
Multiple event Execute Hook invocation	189	<ul style="list-style-type: none"> • Parm 2 = DSA • Parm 3 = hook offset • Parm 4 = DSA format • Parm 5 = A buffer containing general purpose registers • Parm 6 = Return address to the routine that was interrupted • Parm 7 = Entry point to the routine that was interrupted • Parm 8 = eight-byte clock value returned by the STORE Clock (STCK) instruction • Parm 9 = eight-byte elapsed CPU time in microseconds returned by the TIMEUSED assembler service • Parm 10 = Event mask 		

Note: This is the requestor's DSA, which means an HLL library routine DSA is likely the requestor of the Language Environment service or user DSA.

CAA

A fullword binary integer that contains the address of the CAA.

CIB

A fullword binary integer that contains the address of the CIB.

DSA

A fullword binary integer that contains the address of the DSA.

EDB

A fullword binary integer that contains the address of the EDB.

Hook offset

A fullword binary integer that contains the offset of the hook that was executed within the hook set. (This value is a multiple of 4 ranging from 0 to 52 inclusive.)

DSA format

A fullword binary integer set to one of the following:

0 The format of the DSA is a standard OS linkage register save area (with/without Language Environment fields including NAB).

1 The format of the DSA is XPLINK style.

General purpose registers

A 64-byte buffer containing the general purpose registers stored in order 0 to 15 at the time the debug hook was executed. If the debugger changes these register values, the new values will be used when control is returned to the routine that executed the debug hook.

return_address

A fullword pointer containing the address of the instruction where control will be returned to the routine that executed the debug hook. If the debugger changes this address, control will be returned to the new location.

entry_ptr

A fullword pointer containing the address of the entry point of the routine that contains the debug hook.

result code

A fixed(31) binary value action for condition manager to take. The supported values are:

110 Resume at the resume cursor

120 Percolate to next condition handler

storage length

A fixed(31) binary value containing the number of bytes of storage.

profiler invocation string

A halfword-prefixed string that contains the invocation string of the profiler tool. This value, which is specified as the *string* parameter of the PROFILE runtime option, it is translated to upper case characters. For more information about the runtime option, see *z/OS Language Environment Programming Reference*.

INPL

The Initialization Parameter List as passed to CEEINT. For the format of the INPL, see Figure 54 on page 151.

thread_id

An 8-byte thread identifier.

stack_size

A stack size attribute (in bytes) of initial or created thread.

nil

Unused; null pointer.

Performance Analysis Support

event mask

a fullword binary value in which each bit represents a different hook event. When the bit is '1'b, the event occurred. The values of the bits are:

Bit	Event
0-11	Not used
12	Multiple Event Hook
13	Allocate Descriptor Built
14	Block Entry
15	Not used
16	User label
17	Begin of statement
18	Call return
19-20	Not used
21	Start of loop
22	If evaluated TRUE
23	If evaluated FALSE
24	Switch/case/select choice start
25	Switch/case/select default start
26	Multiple flows join
27	Not used
28	Call begin
29	Goto
30	Procedure exit
31	Multiple exit

Language Environment actions for profiler

Language Environment parses the PROFILE runtime option on behalf of the profile tool and sets the appropriate flags and profiler invocation string with the Options Control Block (OCB). If the TEST runtime option has also been specified, Language Environment issues a message to indicate that the TEST option will take precedence; that is, Language Environment will load the specified debug tool and will not load the specified profiler tool. If the NOTEST runtime option is specified, Language Environment loads module CEEVPRF and stores the entry point address in the PCB (field CEEPCBPRFEH).

Chapter 10. DFSORT interface

This chapter describes and discusses the DFSORT interface. Note that whenever DFSORT is mentioned, an equivalent sort product can be used.

DFSORT interface description

Typically, an implicit enclave boundary occurs when an application issues an SVC LINK. However, this is not the case when DFSORT is invoked directly; that is, Language Environment will create a new enclave for DFSORT. To simplify calls to DFSORT, Language Environment concentrates the logic of the DFSORT invocation into the Language Environment service CEE3SRT.

When CEE3SRT is invoked, the routine acquires a new stack frame and some flags are set to indicate DFSORT invocation. As a result, the path length is slightly longer than if your application used LINK SVC to invoke DFSORT directly. However, when you invoke CEE3SRT, the routine also establishes exit DSAs and calls DFSORT using defined interfaces, which are described in the *z/OS DFSORT Application Programming Guide*.

Language Environment supports the DFSORT extended parameter list, which allows parameters to be placed above the 16M line. DFSORT Version 1 Release 1.1 or later is required for extended parameter list support.

CEE3SRT — call DFSORT

Purpose

This CWI interface establishes an exit DSA and call DFSORT.

Syntax

```
void CEE3SRT (dfsort_extended_plist, ret_code)
STRUCT      *dfsort_extended_plist;
INT4       *ret_code;
```

CEE3SRT

Call this CWI interface as follows:

```
L      R12,A(CAA)           Get the address of CAA in R12
L      R15,CEECAACELV-CEECAA(,R12)
L      R15,2916(,R15)
BALR  R14,R15
```

dfsort_plist

The address of the extended parameter list that is passed to DFSORT. The DFSORT extended parameter list is shown in Figure 78 on page 366. Language Environment reserves the use of the address of the ESTAE area pointer (+X'14' into the extended parameter list). Language Environment gets the exit address to establish the environment for the member-specified exit, before this exit gets control. It is the caller's responsibility to adhere to the DFSORT interface, as described in the *z/OS DFSORT Application Programming Guide*.

SORTEPL	DSECT		
CONTROL	DS	A	Addr of control statements or zero
E15_E32	DS	A	Addr of user exit E15 or E32, or zero
E35	DS	A	Addr of user exit E35 or zero
USER	DS	A	User exit addr constant or zero
ALTSEQ	DS	A	Addr of ALTSEQ translation table or zero
ESTAE	DS	A	Addr of ESTAE area pointer or zero
E18	DS	A	Addr of user exit E18 or zero
E39	DS	A	Addr of user exit E39 or zero
END_MARK	DS	F	F'-1' to indicate the end

Figure 78. DFSORT's extended parameter list

ret_code

The return code from the DFSORT invocation which is contained within R15 upon return from DFSORT. Refer to the DFSORT library for detailed information on the return codes. It is the CEE3SRT caller's responsibility to manage the DFSORT return code. For example, COBOL would save it in the SORT-RETURN special register.

Usage notes

- Note the following restrictions:
 - DFSORT does not run under CICS. Language Environment calls DFSORT using EXEC CICS LOAD and BALR 14,15 while executing under CICS.
 - DFSORT is not supported in a POSIX(ON) environment.
 - Language Environment only supports E15, E35, and E32 exits.
- Identifying restrictions on DFSORT invocation on a per-HLL basis is the responsibility of the particular HLL.
- Language Environment calls DFSORT using SVC LINK while executing under z/OS.
- The caller of CEE3SRT must provide and manage the DFSORT exit addresses using the extended parameter list. Typically, the address of an exit identifies an HLL library routine which, in turn, calls a user routine. If no user exit routine is needed, a zero can be specified in the extended parameter list.

Language Environment gets the exit address in the DFSORT PLIST and replaces it with a Language Environment routine so that Language Environment can, among other things, establish R12 and R13 to point to the CAA and a DSA respectively prior to calling the caller's supplied exit.

- When a DFSORT user exit is called, the registers are as follows:

- R1** Address of a parameter list for the particular exit
- R12** Address of the CAA
- R13** Address of a standard DSA-formatted save area with a valid NAB established
- R14** The return address
- R15** Address of the exit's entry point

- Invocation of DFSORT from within a DFSORT user exit is restricted in Language Environment.
- R15 is used to pass return codes back to DFSORT from the user exit.
- The exit address that is passed to CEE3SRT is called, honoring the AMODE bit, and with R12 and R13 established as described above.

- Language Environment uses the ESTAE area pointer. For additional information, see “Error handling within SORT exits.”
- A new enclave is not created when calling DFSORT in z/OS even though an RB boundary is crossed.
- CEE3SRT restores the program mask from the value in the CAA upon return from the call. If the program mask is altered (using CEE3SPM or a dynamic call) in the DFSORT user exit, the effect persists upon return from DFSORT.

ILC within SORT exits

Inter-language communication (ILC) is allowed within the DFSORT user exit, as long as the ILC is performed within the same load module. ILC is not permitted in dynamically loaded routines.

Error handling within SORT exits

Language Environment terminates all routines up to the routine that called DFSORT (using CEE3SRT) for all abends. Neither HLL condition handlers nor user handlers established within the DFSORT exit is driven for abends occurring within DFSORT or the DFSORT exit.

When a condition is raised by an abend, the handle cursor and the resume cursor are set to the return point following the call to CEE3SRT. The information in the CIB contains the information on the condition that was raised in the sort exit and an indication that the condition occurred while in DFSORT (including the USER DFSORT exit). The current invocation of DFSORT is terminated and error handling starts with the stack frame of the caller of CEE3SRT.

Conditions raised either by CEESGL or by a program interrupt continues to operate in the same manner, independent of the CEE3SRT call. However, when the resume cursor is moved to a stack frame that precedes the CEE3SRT stack frame, Language Environment terminates the DFOSRT invocation.

Messages and conditions

The following conditions can arise during the invocation of CEE3SRT:

Condition		
CEE35L	Severity	4
	Msg_No	3253
	Message	Catastrophic exception raised within the CEE3SRT invocation.
CEE35M	Severity	4
	Msg_No	3254
	Message	Incorrect DFSORT PLIST passed to CEE3SRT.
CEE35N	Severity	4
	Msg_No	3255
	Message	Attempt to invoke CEE3SRT from within a DFSORT exit.

DFSORT Interface

Chapter 11. Math library

The interface conventions provided by the Language Environment math service library routines are:

- Scalar routines callable service call
- Scalar routines CWI for all HLLs

Language Environment math service library consists of two logical libraries. All routines in one library are called with the callable service interface. All routines in the other library are called using the CWI interface. When a condition is encountered in the callable service invocation, the condition token is constructed. If the caller specifies to receive the condition token, control is returned to the caller. If the caller does not specify to receive the condition token, it is presented to the condition handler for processing. When a condition is encountered in the CWI invocation the language-specific condition handlers are called to handle it. These condition handling rules are the same for software and hardware detected conditions.

Calling math services from an application

Math services can be called either from HLLs or from assembler, if Language Environment is initialized. There are two ways to call math services:

1. From individual HLLs as that language's own intrinsic math service (CWI or Register CWI)
2. From any HLL or assembler as a callable service

Math service condition handling requirements

Math services need to satisfy semantic condition handling requirements of all Language Environment members. A condition in a math service is treated differently than a condition elsewhere in either the generated code or in the other routines of the language library. This section lists what special functions are needed by condition handlers that support math services.

Individual programming languages vary widely in handling semantic action in regard to math services. When a condition occurs in a CWI, the condition handling mechanism needs to provide the following information to the condition handler:

- Indication that a condition occurred in a math service
- The name or other identification of the math service
- The type of condition

This information is communicated in the CIB (condition information block) constructed by the math service for software-detected conditions or by the ESTAE exit of the operating system provided by Language Environment for hardware-detected conditions.

Member-specific condition handling

Member language condition handlers needing to do special processing for conditions originated in math services are assisted as follows; if an execution interruption occurred in a math service, CIB_MRC will have one of the following values:

'1'B Indicates math service originated condition.

'0'B Indicates nonmath service condition.

Data types and their abbreviations

Table 57 shows the data types used in this section and their abbreviations.

Table 57. Data types and their abbreviations

Abbreviation	Data Type Explanation
R*S	32-bit single floating-point number (hexadecimal)
R*L	64-bit double floating-point number (hexadecimal)
R*E	128-bit extended floating-point number (hexadecimal)
C*S	complex number consisting of two 32-bit single floating-point numbers
C*L	complex number consisting of two 64-bit double floating-point numbers
C*E	complex number consisting of two 128-bit extended floating-point numbers
I*S	32-bit binary integer number
L*S	32-bit logical value
I*J	64-bit binary integer number
I*H	16-bit binary integer number
I*K	8-bit binary integer number
I*U	8-bit unsigned binary integer number

CWI conventions for scalar math services

The Language Environment math library scalar functions are accessed through a CWI entry point. HLLs invoke math routines at the CWI conventional interface or at the CWI register interface.

Table 58 on page 371 and Figure 79 on page 372 show supported formats. The first format uses the register interface, where the address of the parameter is placed in a register and the result is in a floating-point register. The second format uses the conventional interface where GPR1 contains the address of the parameter list and the result is returned in storage.

Upon entry, standard Language Environment linkage conventions are assumed:

GPR12

CAA

GPR13

Save area

GPR14
Return address

GPR15
Entry point

Register interface

This is a S/370 platform-specific extension. Upon entry, GPR1 contains the address of the first argument, GPR2 contains the address of the second argument, and GPR3 contains the address of the third argument for functions with three arguments as shown in the following figure. The result is returned in the register or registers as shown in Table 59.

Table 58. CWI register interface format

One Input Parameter	Two Input Parameters	Three Input Parameters
R1 = @(parm 1)	R1 = @(parm 1)	R1 = @(parm 1)
	R2 = @(parm 2)	R2 = @(parm 2)
		R3 = @(parm 3)

Upon return, registers 4-14 contain the same values as they did when the routine was entered. Registers 2 and 3 are preserved by routines of one input parameter. Register 3 is preserved by routines of two input parameters.

Table 59. Result registers for scalar routines (CWI register interface)

Result Data Type	Real Part	Imaginary Part
R*S	FPR0	
R*L	FPR0	
R*E	FPR0,2	
C*S	FPR0	FPR2
C*L	FPR0	FPR2
C*E	FPR0,2	FPR4,6
I*S	GPR0	

Conventional interface

Upon entry, GPR1 contains address of the parameter list as shown in Figure 79 on page 372.

Math Services

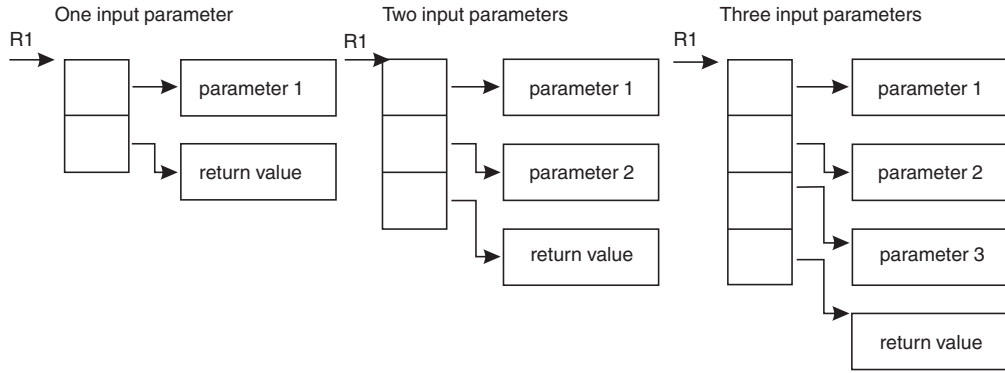


Figure 79. HLL CWI parameter list format

Upon return, registers 2-14 contain the same values as they did when the routine was entered.

Condition token values for math services

Figure 80 shows the condition token values.

Severity		Msg_No	C/S/C	Facility ID			
X'00'	X'00'	X'0000'	X'00'	X'00'	X'00'	X'00'	success
X'00'	X'02'		X'51'	C	E	E	failure

Facility ID: CEE

Msg_No: see Table 29 on page 222

C/S/C Byte has the following values:

X'51' for failure (severity 2)
X'00' for success

Case	Sev	Control
0 1	0 1	0 0 1 0
0 0	0 0	0 0 0 0

Figure 80. Condition token values for math services

Math services

The Language Environment math services library consists of 239 scalar routines. Math service entry point names have a specific format. All entry points are 8-character names.

The first three characters are always **CEE**.

The fourth character is one of the following values:

- S** Scalar routine, AWI callable service entry point
- T** Scalar routine, CWI callable service entry point
- 9** Scalar routine, register CWI entry point

The fifth character designates the data type of input parameter(s):

I → **I*S** 32-bit binary integer number

S → **R*S**

32-bit single floating-point number

D → R*L	64-bit double floating-point number
Q → R*E	128-bit extended floating-point number
T → C*S	32-bit single float-complex number
E → C*L	64-bit double float-complex number
R → C*E	128-bit extended float-complex number
J → I*J	64-bit binary integer number
H → I*H	16-bit binary integer number
K → I*K	8-bit binary integer number
U → I*U	8-bit unsigned binary integer number

The last three characters are a mnemonic designating the unique routine.

Scalar math services

Table 60 describes the scalar math services in Language Environment.

Note:

1. `Msg_No` is a decimal value identifying a given condition. Routines that do not raise conditions have an asterisk (*) in the `Msg_No` column.
2. The implementation of several math services involved calls to other math services. These called math services can generate conditions and messages.

Table 60. Language Environment Scalar math services

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
Absolute Function					
	CEE9HABS	I*2	I*2	AFBFABS	*
	CEESIABS CEETIABS CEE9IABS	I*S	I*S	AFBFABS	*
	CEE9JABS	I*L	I*L	AFBFABS	*
	CEESSABS CEETSABS CEE9SABS	R*S	R*S	AFBFABS	*
	CEESDABS CEETDABS CEE9DABS	R*L	R*L	AFBFABS	*
	CEESQABS CEETQABS CEE9QABS	R*E	R*E	AFBFABS	*
	CEESTABS CEETTABS CEE9TABS	C*S	R*S	VSFCSABS	*
	CEESEABS CEETEABS CEE9EABS	C*L	R*L	VSFCLABS	2025
	CEESRABS CEETRABS CEE9RABS	C*E	R*E	AFBCQABS	*
Arccosine					
	CEESSACS CEETSACS CEE9SACS	R*S	R*S	VSFACOS	2016

Math Services

Table 60. Language Environment Scalar math services (continued)

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
	CEESDACS CEETDACS CEE9DACS	R*L	R*L	VSFLACOS	2016
	CEESQACS CEETQACS CEE9QACS	R*E	R*E	AFBQASCN	2016
Arcsine					
	CEESSASN CEETSASN CEE9SASN	R*S	R*S	VSFSASIN	2016
	CEESDASN CEETDASN CEE9DASN	R*L	R*L	VSFLASIN	2016 2025
	CEESQASN CEETQASN CEE9QASN	R*E	R*E	AFBQASCN	2016
Arctangent					
	CEESSATN CEETSATN CEE9SATN	R*S	R*S	VSFSATAN	*
	CEESDATN CEETDATN CEE9DATN	R*L	R*L	VSFLATAN	2025
	CEESQATN CEETQATN CEE9QATN	R*E	R*E	AFBQATN2	*
	CEESTATN CEETTATN CEE9TATN	C*S	C*S	IBMBMKXA	2022
	CEESEATN CEETEATN CEE9EATN	C*L	C*L	IBMBMKYA	2022
	CEESRATN CEETRATN CEE9RATN	C*E	C*E	IBMBMKZA	2022
Arctangent2					
	CEESSAT2 CEETSAT2 CEE9SAT2	R*S R*S	R*S	VSFSATN2	2014
	CEESDAT2 CEETDAT2 CEE9DAT2	R*L R*L	R*L	VSFLATN2	2014 2025
	CEESQAT2 CEETQAT2 CEE9QAT2	R*E R*E	R*E	AFBQATN2	2014
Conjugate of Complex					
	CEESTCJG CEETTCJG CEE9TCJG	C*S	C*S	AFBFCONJ	*
	CEESECJG CEETECJG CEE9ECJG	C*L	C*L	AFBFCONJ	*
	CEESRCJG CEETRCJG CEE9RCJG	C*E	C*E	AFBFCONJ	*
Cosine					
	CEESSCOS CEETSCOS CEE9SCOS	R*S	R*S	VSFSCOS	2017
	CEESDCOS CEETDCOS CEE9DCOS	R*L	R*L	VSFLCOS	2017
	CEESQCOS CEETQCOS CEE9QCOS	R*E	R*E	AFBQSCN	2017
	CEESTCOS CEETTCOS CEE9TCOS	C*S	C*S	AFBCSSCN	2013 2019
	CEESECOS CEETECOS CEE9ECOS	C*L	C*L	AFBCLSCN	2013 2019
	CEESRCOS CEETRCOS CEE9RCOS	C*E	C*E	AFBCQSCN	2013 2019

Table 60. Language Environment Scalar math services (continued)

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
Cotangent					
	CEESCTN CEETSCTN CEE9SCTN	R*S	R*S	VSFSCOTN	2002 2017
	CEESDCTN CEETDCTN CEE9DCTN	R*L	R*L	VSFLCOTN	2002 2017
	CEESQCTN CEETQCTN CEE9QCTN	R*E	R*E	AFBQTNCT	2002 2017
Cube Root					
	CEETDCRT	R*L	R*L	new	*
Error Function					
	CEESSERC CEETSERC CEE9SERC	R*S	R*S	AFBSERF	*
	CEESDERC CEETDERC CEE9DERC	R*L	R*L	AFBLERF	*
	CEESQERC CEETQERC CEE9QERC	R*E	R*E	AFBQERF	*
	CEESSERF CEETSERF CEE9SERF	R*S	R*S	AFBSERF	*
	CEESDERF CEETDERF CEE9DERF	R*L	R*L	AFBLERF	*
	CEESQERF CEETQERF CEE9QERF	R*E	R*E	AFBQERF	*
Exponential (base e)					
	CEESSEXP CEETSEXP CEE9SEXP	R*S	R*S	VSFSEXP	2011
	CEESDEXP CEETDEXP CEE9DEXP	R*L	R*L	VSFLEXP	* 2011 2025
	CEESQEXP CEETQEXP CEE9QEXP	R*E	R*E	AFBFQXPQ	2011
	CEESTEXP CEETTEXP CEE9TEXP	C*S	C*S	AFBCSEXP	2009 2015
	CEESEEXP CEETEEXP CEE9EEXP	C*L	C*L	AFBCLEXP	2009 2015
	CEESREXP CEETREXP CEE9REXP	C*E	C*E	AFBCQEXP	2009 2013
Exponentiation (**)					
	CEESDXPD CEETDXPD CEE9DXPD	R*L R*L	R*L	VSFFDXPD	2006 2020 2025
	CEESXPE CEETXPE CEE9XPE	C*L C*L	C*L	AFBFCDCD	2008
	CEESIXPI CEETIXPI CEE9IXPI	I*S I*S	I*S	AFBFIXPI	2003
	CEESXPI CEETSXPI CEE9SXPI	R*S I*S	R*S	AFBFRXPI	2004
	CEESDXPI CEETDXPI CEE9DXPI	R*L I*S	R*L	AFBFDXPI	2004
	CEESQXPI CEETQXPI CEE9QXPI	R*E I*S	R*E	AFBFQXPI	2004
	CEESTXPI CEETTXPI CEE9TXPI	C*S I*S	C*S	AFBFCXPI	2008
	CEESXPI CEETXPI CEE9EXPI	C*L I*S	C*L	AFBFCDXI	2008
	CEESRXPI CEETRXPI CEE9RXPI	C*E I*S	C*E	AFBFCQXI	2008
	CEE9JXPI	I*L I*S	I*L	AFBF8XPI	2003
	CEE9IXPJ	I*S I*L	I*S	AFBFIXP8	2003
	CEE9JXPJ	I*L I*L	I*L	AFBF8XP8	2003
	CEE9SXPJ	R*S I*L	R*S	AFBFRXP8	2004

Math Services

Table 60. Language Environment Scalar math services (continued)

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
	CEE9DXPJ	R*L I*L	R*L	AFBFDXP8	2004
	CEE9QXPJ	R*E I*L	R*E	AFBFQXP8	2004
	CEE9TXPJ	C*S I*L	C*S	AFBFCXP8	2008
	CEE9EXPJ	C*L I*L	C*L	AFBFCDX8	2008
	CEE9RXPJ	C*E I*L	C*E	AFBFCQX8	2008
	CEESQXPQ CEETQXPQ CEE9QXPQ	R*E R*E	R*E	AFBFQXPQ	2020 2021
	CEESRXPR CEETRXPR CEE9RXPR	C*E C*E	C*E	AFBFCQCQ	2008
	CEESSXPS CEETSXPS CEE9SXPS	R*S R*S	R*S	VSFRRXPR	2006 2020
	CEESTXPT CEETTXPT CEE9TXPT	C*S C*S	C*S	AFBFCXPC	2008
	CEESQXP2 CEETQXP2 CEE9QXP2	R*E	R*E	AFBFQXPQ	2007
Exp(x)-1					
	CEETDEM1	R*L	R*L	new	2011
Floating Complex Divide					
	CEESTDVD CEETDVD CEE9TDVD	C*S C*S	C*S	VSFCSAD	*
	CEESDVD CEETDVD CEE9EDVD	C*L C*L	C*L	VSFCLAD	*
	CEESRDVD CEETRDVD CEE9RDVD	C*E C*E	C*E	AFBCQRIT	*
Floating Complex Multiply					
	CEESTMLT CEETMLT CEE9TMLT	C*S C*S	C*S	AFBCSAM	*
	CEESEMLT CEETEMLT CEE9EMLT	C*L C*L	C*L	AFBCLAM	*
	CEESRMLT CEETRMLT CEE9RMLT	C*E C*E	C*E	AFBCQRIT	*
Gamma Function					
	CEESSGMA CEETSGMA CEE9SGMA	R*S	R*S	AFBSGAMA	2005
	CEESDGMA CEETDGMA CEE9DGMA	R*L	R*L	AFBLGAMA	2005
Hyperbolic Arccosine					
	CEETDACH	R*L	R*L	new	2010
Hyperbolic Arcsine					
	CEETDASH	R*L	R*L	new	*
Hyperbolic Arctangent					
	CEESSATH CEETSATH CEE9SATH	R*S	R*S	IBMBMLSA	2017
	CEESDATH CEETDATH CEE9DATH	R*L	R*L	IBMBMLLA	2017
	CEESQATH CEETQATH CEE9QATH	R*E	R*E	IBMBMLEA	2017

Table 60. Language Environment Scalar math services (continued)

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
	CEESTATH CEETTATH CEE9TATH	C*S	C*S	IBMBMKXA	2022
	CEESEATH CEETEATH CEE9EATH	C*L	C*L	IBMBMKYA	2022
	CEESRATH CEETRATH CEE9RATH	C*E	C*E	IBMBMKZA	2022
Hyperbolic Cosine					
	CEESSCSH CEETSCSH CEE9SCSH	R*S	R*S	VSFSCOSH	2016
	CEESDCSH CEETDCSH CEE9DCSH	R*L	R*L	AFBLSCNH	2016
	CEESQCSH CEETQCSH CEE9QCSH	R*E	R*E	AFBQSCNH	2016
	CEESTCSH CEETTCSH CEE9TCSH	C*S	C*S	IBMBMGXA	*
	CEESECSH CEETECSH CEE9ECSH	C*L	C*L	IBMBMGYA	*
	CEESRCSH CEETRCSH CEE9RCSH	C*E	C*E	IBMBMGZA	*
Hyperbolic Sine					
	CEESSNH CEETSSNH CEE9SSNH	R*S	R*S	VSFSSINH	2016
	CEESDSNH CEETDSNH CEE9DSNH	R*L	R*L	AFBLSCNH	2016
	CEESQSNH CEETQSNH CEE9QSNH	R*E	R*E	AFBQSCNH	2016
	CEESTSNH CEETTSNH CEE9TSNH	C*S	C*S	IBMBMGXA	*
	CEEESNH CEETESNH CEE9ESNH	C*L	C*L	IBMBMGYA	*
	CEESRSNH CEETRSNH CEE9RSNH	C*E	C*E	IBMBMGZA	*
Hyperbolic Tangent					
	CEESSTNH CEETSTNH CEE9STNH	R*S	R*S	VSFSTANH	*
	CEESDTNH CEETDTNH CEE9DTNH	R*L	R*L	AFBLTANH	*
	CEESQTNH CEETQTNH CEE9QTNH	R*E	R*E	AFBQTANH	*
	CEESTTNH CEETTTNH CEE9TTNH	C*S	C*S	IBMBMHXA	*
	CEESETNH CEETETNH CEE9ETNH	C*L	C*L	IBMBMHYA	*
	CEESRTNH CEETRTNH CEE9RTNH	C*E	C*E	IBMBMHZA	*
Imaginary part of Complex					

Math Services

Table 60. Language Environment Scalar math services (continued)

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
	CEESTIMG CEETTIMG CEE9TIMG	C*S	R*S	AFBFIMAG	*
	CEESEIMG CEETEIMG CEE9EIMG	C*L	R*L	AFBFIMAG	*
	CEESRIMG CEETRIMG CEE9RIMG	C*E	R*E	AFBFIMAG	*
Load exponent					
	CEETDSCB	R*L I*S	R*L	new	2024 2025
Logarithm Base e					
	CEESSLOG CEETSLOG CEE9SLOG	R*S	R*S	VSFSLGN	2012
	CEESDLOG CEETDLOG CEE9DLOG	R*L	R*L	VSFLLGN	2012
	CEESQLOG CEETQLOG CEE9QLOG	R*E	R*E	AFBFQXPQ	2012
	CEESTLOG CEETTLOG CEE9TLOG	C*S	C*S	AFBCSLOG	2018
	CEESELOG CEETELOG CEE9ELOG	C*L	C*L	AFBCLLOG	2018
	CEESRLOG CEETRLOG CEE9RLOG	C*E	C*E	AFBCQLOG	2018
Logarithm Base 10					
	CEESLG1 CEETSLG1 CEE9SLG1	R*S	R*S	VSFSLGC	2012
	CEESDLG1 CEETDLG1 CEE9DLG1	R*L	R*L	VSFLLGC	2012
	CEESQLG1 CEETQLG1 CEE9QLG1	R*E	R*E	AFBFQXPQ	2012
Logarithm Base 2					
	CEESLG2 CEETSLG2 CEE9SLG2	R*S	R*S	IBMBMDSA	2012
	CEESDLG2 CEETDLG2 CEE9DLG2	R*L	R*L	IBMBMDLA	2012
	CEESQLG2 CEETQLG2 CEE9QLG2	R*E	R*E	IBMBMYEA	2012
$\text{Log}_e(1.0 + x)$					
	CEETDL1P	R*L	R*L	new	2012
Log Gamma Function					
	CEESLGM CEETSLGM CEE9SLGM	R*S	R*S	AFBSGAMA	2005
	CEESDLGM CEETDLGM CEE9DLGM	R*L	R*L	new	2005 2031
Modular Arithmetic					
	CEE9HMOD	I*2 I*2	I*2	AFBFMODI	*
	CEESIMOD CEETIMOD CEE9IMOD	I*S I*S	I*S	AFBFMODI	*

Table 60. Language Environment Scalar math services (continued)

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
	CEE9JMOD	I*L I*L	I*L	AFBFMODI	*
	CEESSMOD CEETSMOD CEE9SMOD	R*S R*S	R*S	VSFFMODR	*
	CEESDMOD CEETDMOD CEE9DMOD	R*L R*L	R*L	VSFFMODR	*
	CEESQMOD CEETQMOD CEE9QMOD	R*E R*E	R*E	VSFFMODR	*
Nearest Integer					
	CEESSNIN CEETSNIN CEE9SNIN	R*S	I*S	AFBFNINT	*
	CEESDNIN CEETDNIN CEE9DNIN	R*L	I*S	AFBFNINT	*
	CEE9QNIN	R*E	I*S	AFBFNINT	*
	CEE9SNJN	R*S	I*L	AFBFNINT	*
	CEE9DNJN	R*L	I*L	AFBFNINT	*
	CEE9QNJN	R*E	I*L	AFBFNINT	*
Nearest Whole Number					
	CEE9QNWN	R*E	R*E	AFBFNINT	*
	CEESSNWN CEETSNWN CEE9SNWN	R*S	R*S	AFBFNINT	*
	CEESDNWN CEETDNWN CEE9DNWN	R*L	R*L	AFBFNINT	*
Nextafter					
	CEETDNXA	R*L	R*L	new	*
Positive Difference					
	CEE9HDIM	I*2 I*2	I*2	AFBFDIM	*
	CEESIDIM CEETIDIM CEE9IDIM	I*S I*S	I*S	AFBFDIM	*
	CEE9JDIM	I*L I*L	I*L	AFBFDIM	*
	CEESDIM CEETSDIM CEE9SDIM	R*S R*S	R*S	AFBFDIM	*
	CEESDDIM CEETDDIM CEE9DDIM	R*L R*L	R*L	AFBFDIM	*
	CEESQDIM CEETQDIM CEE9QDIM	R*E R*E	R*E	AFBFDIM	*
Remainder					
	CEETDREM	R*L R*L	R*L	new	2030
Sine					
	CEESSIN CEETSSIN CEE9SSIN	R*S	R*S	VSFSSIN	2017
	CEESDSIN CEETDSIN CEE9DSIN	R*L	R*L	VSFLSIN	2017 2025
	CEESQSIN CEETQSIN CEE9QSIN	R*E	R*E	AFBQSCN	2017
	CEESTSIN CEETTSIN CEE9TSIN	C*S	C*S	AFBCSSCN	2013 2019
	CEESESIN CEETESIN CEE9ESIN	C*L	C*L	AFBCLSCN	2013 2019
	CEERSIN CEETRSIN CEE9RSIN	C*E	C*E	AFBCQSCN	2013 2019
Square Root					

Math Services

Table 60. Language Environment Scalar math services (continued)

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
	CEESSQT CEETSSQT CEE9SSQT	R*S	R*S	VSFSSQRT	2010
	CEESDSQT CEETDSQT CEE9DSQT	R*L	R*L	VSFLSQRT	2010
	CEESQSQT CEETQSQT CEE9QSQT	R*E	R*E	AFBQSQRT	2010
	CEESTSQT CEETTSQT CEE9TSQT	C*S	C*S	AFBCSSQT	*
	CEESESQT CEETESQT CEE9ESQT	C*L	C*L	AFBCLSQT	*
	CEERSQT CEETRSQT CEE9RSQT	C*E	C*E	AFBCQSQT	*
Tangent					
	CEESSTAN CEETSTAN CEE9STAN	R*S	R*S	VSFSTAN	2017
	CEESDTAN CEETDTAN CEE9DTAN	R*L	R*L	VSFLTAN	2017 2025
	CEESQTAN CEETQTAN CEE9QTAN	R*E	R*E	AFBQTNCT	2002 2017
	CEESTTAN CEETTTAN CEE9TTAN	C*S	C*S	IBMBMHXA	*
	CEESETAN CEETETAN CEE9ETAN	C*L	C*L	IBMBMHYA	*
	CEESRTAN CEETRTAN CEE9RTAN	C*E	C*E	IBMBMHZA	*
Transfer of Sign					
	CEE9HSGN	I*2 I*2	I*2	AFBFSIGN	*
	CEE9JSGN	I*L I*L	I*L	AFBFSIGN	*
	CEESISGN CEETISGN CEE9ISGN	I*S I*S	I*S	AFBFSIGN	*
	CEESSGN CEETSSGN CEE9SSGN	R*S R*S	R*S	AFBFSIGN	*
	CEESDSGN CEETDSGN CEE9DSGN	R*L R*L	R*L	AFBFSIGN	*
	CEESQSGN CEETQSGN CEE9QSGN	R*E R*E	R*E	AFBFSIGN	*
Truncation					
	CEESINT CEETSINT CEE9SINT	R*S	R*S	AFBFAINT	*
	CEESDINT CEETDINT CEE9DINT	R*L	R*L	AFBFAINT	*
	CEESQINT CEETQINT CEE9QINT	R*E	R*E	AFBFAINT	*
Unbiased exponent					
	CEETILGB	R*L	I*S	new	2029
	CEETDLGB	R*L	R*L	new	2029

Degree input/output trigonometry functions

Table 61 on page 381 lists the supported degree input/output trigonometry functions

Table 61. Degree input/output trigonometry functions

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
Sine					
	CEETSSND CEE9SSND	R*S	R*S	IBMRMGSB	*
	CEETDSND CEE9DSND	R*L	R*L	IBMRMGLB	*
	CEETQSND CEE9QSND	R*E	R*E	IBMRMGEB	*
Cosine					
	CEETSCSD CEE9SCSD	R*S	R*S	IBMRMGSD	*
	CEETDCSD CEE9DCSD	R*L	R*L	IBMRMGLD	*
	CEETQCSD CEE9QCSD	R*E	R*E	IBMRMGED	*
Tangent					
	CEETSTND CEE9STND	R*S	R*S	IBMRMHSB	*
	CEETDTND CEE9DTND	R*L	R*L	IBMRMHLB	*
	CEETQTND CEE9QTND	R*E	R*E	IBMRMHEB CEEIQTND	*
Arcsine					
	CEE9SASD	R*S	R*S	CEEISASN	2016
	CEE9DASD	R*L	R*L	CEEIDASN	2016 2025
	CEE9QASD	R*E	R*E	CEEIQASN	2016
Arccosine					
	CEE9SACD	R*S	R*S	CEEISACS	2016
	CEE9DACD	R*L	R*L	CEEIDACS	2016
	CEE9QACD	R*E	R*E	CEEIQACS	2016
Arctangent					
	CEETSATD CEE9SATD	R*S	R*S	IBMRMKSB	*
	CEETDATD CEE9DATD	R*L	R*L	IBMRMKLB	2025
	CEETQATD CEE9QATD	R*E	R*E	IBMRMKEB	*
Arctangent 2					
	CEETSA2D CEE9SA2D	R*S R*S	R*S	IBMRMKSD	2014
	CEETDA2D CEE9DA2D	R*L R*L	R*L	IBMRMKLD	2014 2025
	CEETQA2D CEE9QA2D	R*E R*E	R*E	IBMRMKED	2014

Entry point names for scalar bit manipulation routines

Table 62 summarizes the entry point names, the parameter types, and the result types for the scalar bit manipulation routines.

Table 62. Language Environment Scalar bit manipulation routines

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
Bit Shift					
	CEESISHF CEETISHF CEE9ISHF	I*S I*S	I*S	AFBBTSHS	2028
	CEE9JSHF	I*L I*L	I*L	AFBBTSH8	2028

Math Services

Table 62. Language Environment Scalar bit manipulation routines (continued)

Math Operation	Entry Name Callable Service, CWI, Register CWI	Arg Type(s)	Result Type	Algorithm Source	Msg_No
	CEE9KSHF	I*1 I*1	I*1	AFBBTSH1	2028
	CEE9HSHF	I*2 I*2	I*2	AFBBTSH2	2028
	CEE9USHF	U*1 U*1	U*1	AFBBTSH1	2028
Left Shift					
	CEE9ILSH	I*S I*S	I*S	AFBBTSHS	2028
	CEE9JLSH	I*L I*L	I*L	AFBBTSH8	2028
	CEE9KLSH	I*1 I*1	I*1	AFBBTSH1	2028
	CEE9HLSH	I*2 I*2	I*2	AFBBTSH2	2028
	CEE9KLSH	U*1 U*1	U*1	AFBBTSH1	2028
Right Shift					
	CEE9IRSH	I*S I*S	I*S	AFBBTSHS	2028
	CEE9JRSR	I*L I*L	I*L	AFBBTSH8	2028
	CEE9KRSR	I*1 I*1	I*1	AFBBTSH1	2028
	CEE9HRSR	I*2 I*2	I*2	AFBBTSH2	2028
	CEE9KRSR	U*1 U*1	U*1	AFBBTSH1	2028
Bit Clear					
	CEESICLR CEETICLR CEE9ICLR	I*S I*S	I*S	AFBBTSHS	2028
	CEE9JCLR	I*L I*L	I*L	AFBBTSH8	2028
	CEE9KCLR	I*1 I*1	I*1	AFBBTSH1	2028
	CEE9HCLR	I*2 I*2	I*2	AFBBTSH2	2028
	CEE9KCLR	U*1 U*1	U*1	AFBBTSH1	2028
Bit Set					
	CEESISSET CEETISET CEE9ISET	I*S I*S	I*S	AFBBTSHS	2028
	CEE9JSET	I*L I*L	I*L	AFBBTSH8	2028
	CEE9KSET	I*1 I*1	I*1	AFBBTSH1	2028
	CEE9HSET	I*2 I*2	I*2	AFBBTSH2	2028
	CEE9KSET	U*1 U*1	U*1	AFBBTSH1	2028
Bit Test					
	CEESITST CEETITST CEE9ITST	I*S I*S	I*S	AFBBTSHS	2028
	CEE9JTST	I*L I*L	I*S	AFBBTSH8	2028
	CEE9KTST	I*1 I*1	I*S	AFBBTSH1	2028
	CEE9HTST	I*2 I*2	I*S	AFBBTSH2	2028
	CEE9KTST	U*1 U*1	I*S	AFBBTSH1	2028
	CEE9ITJT	I*S I*S	I*L	AFBBTSHS	2028
	CEE9JTJT	I*L I*L	I*L	AFBBTSH8	2028
	CEE9KTJT	I*1 I*1	I*L	AFBBTSH1	2028
	CEE9HTJT	I*2 I*2	I*L	AFBBTSH2	2028
	CEE9KTJT	U*1 U*1	I*L	AFBBTSH1	2028

Message ID — message text for math library

The following symbolic parameters are used in Table 63.

rtn_name

The name of the routine that issued the message. Usually, there is one routine for each combination of input and output data types. Look up the routine to determine valid data types for a particular routine.

limit

Contains the limit value for a given routine; for details, see Table 64 on page 384.

range

Contains the range values (lower and upper limits) for a given routine; see Table 64 on page 384.

Table 63. Math message_IDs

Msg_No	Msg_ID	Explanation
2002	CEE1UI	The argument value is too close to one of the singularities (plus or minus $\pi/2$, plus or minus $3\pi/2$, ... for the tangent; or plus or minus π , plus or minus 2π , ... for the cotangent) in math service <i>rtn_name</i> .
2003	CEE1UJ	For an exponentiation operation ($I^{**}J$) where I and J are integers, I is equal to zero and J is less than or equal to zero in math service <i>rtn_name</i> .
2004	CEE1UK	For an exponentiation operation ($R^{**}I$) where R is real and I is integer, R is equal to zero and I is less than or equal to zero in math service <i>rtn_name</i> .
2005	CEE1UL	The value of the argument is outside the valid range <i>range</i> in math service <i>rtn_name</i> .
2006	CEE1UM	For an exponentiation operation ($R^{**}S$) where R and S are real values, R is equal to zero and S is less than or equal to zero in math service <i>rtn_name</i> .
2007	CEE1UN	The exponent exceeds <i>limit</i> in math service <i>rtn_name</i> .
2008	CEE1UO	For an exponentiation operation ($Z^{**}P$) where the complex base Z equals zero, the real part of the complex exponent P, or the integer exponent P, is less than or equal to zero in math service <i>rtn_name</i> .
2009	CEE1UP	The value of the real part of the argument is greater than <i>limit</i> in math service <i>rtn_name</i> .
2010	CEE1UQ	The argument is less than <i>limit</i> in math service <i>rtn_name</i> .
2011	CEE1UR	The argument is greater than <i>limit</i> in math service <i>rtn_name</i> .
2012	CEE1US	The argument is less than or equal to <i>limit</i> in math service <i>rtn_name</i> .
2013	CEE1UT	The absolute value of the imaginary part of the argument is greater than <i>limit</i> in math service <i>rtn_name</i> .
2014	CEE1UU	Both arguments are equal to <i>limit</i> in math service <i>rtn_name</i> .
2015	CEE1UV	The absolute value of the imaginary part of the argument is greater than or equal to <i>limit</i> in math service <i>rtn_name</i> .
2016	CEE1V0	The absolute value of the argument is greater than <i>limit</i> in math service <i>rtn_name</i> .
2017	CEE1V1	The absolute value of the argument is greater than or equal to <i>limit</i> in math service <i>rtn_name</i> .
2018	CEE1V2	The real and imaginary parts of the argument are equal to <i>limit</i> in math service <i>rtn_name</i> .
2019	CEE1V3	The absolute value of the real part of the argument is greater than or equal to <i>limit</i> in math service <i>rtn_name</i> .
2020	CEE1V4	For an exponentiation operation ($R^{**}S$) where R and S are real values, either R is equal to zero and S is negative or R is negative and S is not an integer whose absolute value is less than or equal to <i>limit</i> in math service <i>rtn_name</i> .

Math services

Table 63. Math message_IDs (continued)

Msg_No	Msg_ID	Explanation
2021	CEE1V5	For an exponentiation operation ($X^{**}Y$) the argument combination of $Y \cdot \log_2(X)$ generates a number greater than or equal to <i>limit</i> in math service <i>rtn_name</i> .
2022	CEE1V6	The value of the argument is plus or minus <i>limit</i> in math service <i>rtn_name</i> .
2024	CEE1V8	Overflow has occurred in the calculation in math routine <i>rtn_name</i> .
2025	CEE1V9	An underflow has occurred in math service <i>rtn_name</i> .
2028	CEE1VC	The value of the second argument was outside the valid range <i>limit</i> in math service <i>rtn_name</i> .
2029	CEE1VD	The value of the argument was equal to <i>limit</i> in math routine <i>rtn_name</i> .
2030	CEE1VE	The value of the second argument was equal to <i>limit</i> in math routine <i>rtn_name</i> .
2031	CEE1VF	The value of the argument was a nonpositive whole number in math routine <i>rtn_name</i> .
2040	CEE1VO	The value of the third argument was outside the valid range <i>limit</i> in math routine <i>rtn_name</i> .
2041	CEE1VP	The absolute value of the second argument was greater than either the value of the third argument or the number of bits in the first argument in math routine <i>rtn_name</i> .
2042	CEE1VQ	The sum of the second and the third arguments was greater than the number of bits in the first argument in math routine <i>rtn_name</i> .
2043	CEE1VR	The value of the second or third argument was less than 0 in math routine <i>rtn_name</i> .

Language Environment math services — value of inserts

Table 64 shows the value of inserts for the math services.

Table 64. Language Environment Math services - value of inserts

Msg_No	Callable Service or CWI	Value of Insert (Limit or Range)
2002	CEESSCTN CEESQTAN CEESDCTN CEESQCTN	NULL
2003	CEESIXPI	NULL
2004	CEESSXPI CEESDXPI CEESQXPI	NULL
2005	CEESSGMA CEESDGMA	$2^{*-252} < X < 57.5744$
2005	CEESLGM CEESDLGM	$0 < X < 4.2937 \cdot 10^{*73}$
2006	CEESXPS CEESDXPD	NULL
2007	CEESQXP2	252
2008	CEESTXPI CEESEXPI CEESRXPI CEESTXPT CEESEXPE CEESRXPR	NULL
2009	CEESTEXP CEESEEXP CEESREXP	174.673
2010	CEESSQT CEESDQT CEESQQT	0
2011	CEESSEXP CEESDEXP CEESQEXP	174.673
2012	CEESSLOG CEESDLOG CEESQLOG CEESLG1 CEESDLG1 CEESQLG1 CEESLG2 CEESDLG2 CEESQLG2	0
2013	CEESTSIN CEEESIN CEESRSIN CEESTCOS CEESECOS CEESRCOS	174.673
2013	CEESREXP	2^{*100}
2014	CEESAT2 CEESDAT2 CEESQAT2	0

Table 64. Language Environment Math services - value of inserts (continued)

Msg_No	Callable Service or CWI	Value of Insert (Limit or Range)
2015	CEESTEXP	$\pi^{(2^{**}18)}$ ($\pi^{(2^{**}18)} = .823\ 550\ E +06$)
2015	CEESEEXP	$\pi^{(2^{**}50)}$ ($\pi^{(2^{**}50)} = .353\ 711\ 887\ 601\ 422\ 01D +16$)
2016	CEESSASN CEESDASN CEESQASN CEESSACS CEESDACS CEESQACS	1
2016	CEESSSNH CEESDSNH CEESQSNH CEESSCSH CEESDCSH CEESQCSH	175.366
2017	CEESSSIN CEESSCOS CEESSTAN CEESSCTN	$\pi^{(2^{**}18)}$ ($\pi^{(2^{**}18)} = .823\ 550\ E +06$)
2017	CEESDSIN CEESDCOS CEESDTAN CEESDCTN	$\pi^{(2^{**}50)}$ ($\pi^{(2^{**}50)} = .353\ 711\ 887\ 601\ 422\ 01D +16$)
2017	CEESQSIN CEESQCOS CEESQTAN CEESQCTN	$2^{**}100$
2017	CEESSATH CEESDATH CEESQATH	1
2018	CEESTLOG CEESELOG CEESRLOG	0
2019	CEESTSIN CEESTCOS	$\pi^{(2^{**}18)}$ ($\pi^{(2^{**}18)} = .823\ 550\ E +06$)
2019	CEERSIN CEERSCOS	$2^{**}100$
2019	CEEESIN CEESECOS	$\pi^{(2^{**}50)}$ ($\pi^{(2^{**}50)} = .353\ 711\ 887\ 601\ 422\ 01D +16$)
2020	CEESSXPS	$16^{**}6 - 1$
2020	CEESDXPD	$16^{**}14 - 1$
2020	CEESQXPQ	$16^{**}28 - 1$
2021	CEESQXPQ	252
2022	CEESTATN CEESEATN CEESRATN	1i
2022	CEESTATH CEESEATH CEESRATH	1
2024	CEETDSCB	NULL
2025	CEESDASN CEESDATN CEESDAT2 CEESDEXP CEESDSIN CEESDTAN CEESDXPD CEESEABS	NULL
2028	CEESISHF CEETISHF CEE9ISHF CEE9JSHF CEE9KSHF CEE9HSHF CEE9USHF CEE9ILSH CEE9JLSH CEE9KLSH CEE9HLSH CEE9IRSH CEE9JRS9 CEE9KRS9 CEE9HRSH CEESICLR CEETICLR CEE9ICLR CEE9JCLR CEE9KCLR CEE9HCLR CEESISSET CEETISSET CEE9ISET CEE9JSET CEE9KSET CEE9HSET CEESITST CEETITST CEE9ITST CEE9JTST CEE9KTST CEE9HTST CEE9ITJT CEE9JTJT CEE9KTJT CEE9HTJT	$0 < = X < = 31$
2029	CEETILGB CEETDLBG	0
2030	CEETDREM	0
2031	CEETDLGM	NULL
2040	CEE9ISHC CEE9JSHC CEE9KSHC CEE9HSHC CEE9USHC	$0 < x_3 < = \text{number_of_bits_in_}x_1$ (where x_1 means the first input argument and x_3 means the third input argument)

Math services

Table 64. Language Environment Math services - value of inserts (continued)

Msg_No	Callable Service or CWI	Value of Insert (Limit or Range)
2041	CEE9ISHC CEE9JSHC CEE9KSHC CEE9HSHC CEE9USHC	NULL
2042	CEE9IBIT CEE9JBIT CEE9KBIT CEE9HBIT	NULL
2043	CEE9IBIT CEE9JBIT CEE9KBIT CEE9HBIT	NULL

Language Environment conversion services

Language Environment provides 3 conversion services to perform the most complex and numerically sensitive part of converting numeric data between decimal character and floating-point representations: the mathematics of the conversion between decimal and float, while leaving other activities, likely to be specific to the calling environment, to be handled by the calling routine. The most important feature of this conversion is *accuracy*; correctly rounded conversions provide the only guarantee of recoverable values.

Terminology

The following terms are used with these definitions:

user data

Numeric data in the forms recognized as syntactically valid in programs and products that call the conversion routines. The forms permitted for user data are varied, and not always syntactically consistent with one another across languages and other products. These conversion routines therefore provide only the most fundamental conversion capabilities, and rely on the caller to manage the details of creating and enforcing syntactic correctness in the calling environment.

input Conversion of decimal data in character form (external representation) to hexadecimal floating-point (internal representation).

output

Conversion of hexadecimal floating-point data (internal representation) to decimal data in character form (external representation).

digits In numeric data represented in character form, the decimal digits only (with no decimal point, no signs, no exponent) sometimes used to refer to hexadecimal or binary digits, which will be clear from the context.

value part

In numeric data represented in character form, the significant decimal digits and possibly, a decimal point.

exponent

An integer value indicating the power of ten by which the value part must be multiplied to obtain the actual value of a numeric datum.

ulp Unit in the Last Place

F-format

Character data in a form where only the *value part* is specified. Examples:

```
'12.345'   '12345'   '0'
'.12345'   '12345.'   '.0'
```

E-format

Character data in scientific notation, where a numeric *value part*, as defined above, is followed by an exponent indicator, usually the letter 'E', and a possibly signed integer that indicates a power of ten by which the numeric value should be multiplied.

```
'12.345E+00' '12345E-3'   '0E0'   '1.0E+000000001'
'.12345E2'   '12345.E-03' '.0E0'
```

In some languages, the exponent indicator may be omitted and an explicit exponent sign is used to indicate the presence of an exponent.

scale factor

Some languages, Fortran and PL/I, particularly, permit the user to specify in a FORMAT statement a power of ten by which the floating-point datum should be scaled during conversion. These scale factors are explicitly provided to the conversion interface only for F-format output conversions. For the other conversions, they are handled by the caller.

CEEYCVHE — E-format output conversion routine

E-format (scientific notation) output is the simplest and most natural decimal format for hex floating-point data, since both express a numeric value in terms of its most significant digits. The output of this conversion is the number of most significant digits requested by the caller, with the last digit correctly rounded. The caller can then create the *user data* by adding signs, decimal points, and formatting the exponent as desired.

Extra precision may be requested. For example, the caller may request 15 output decimal digits from a 4-byte hex float input, even though a 1-ulp change in the input value could cause all decimal digits from the 6th through the 15th to change.

Syntax

void CEEYCVHE (*float_input*, *float_len*, *char_len*, *dec_chars*, *dec_exponent*, [*fc*])

```
VFLOAT    *float_input;
INT4      *float_len;
INT4      *char_len;
CHARn     *dec_chars;
INT4      *dec_exponent;
FEED_BACK *fc;
```

CEEYCVHE

Call this CWI interface as follows:

```
L    R12,A(CAA)           address of CAA
L    R15,CEECAACELV-CEECAA(,R12) address of libvec
L    R15,3504(,R15)       address of routine
BALR R14,R15              invoke the CEEYCVHE
```

***float_input* (input)**

The floating-point value to be converted to decimal. A negative zero will be treated as a positive zero. If negative zeros are significant in the calling environment, it is the caller's responsibility to distinguish between negative and positive zeros.

Conversion services

An input floating-point value may be unnormalized. If the presence of unnormalized data is significant in the calling environment, it is the caller's responsibility to detect and accommodate the fact of unnormalization.

Note: VS Fortran currently produces unnormalized E-format decimal output for unnormalized hex inputs. The intent is to let the user see the loss of significance directly. However, it has been decided that this is a language-dependent facility, and should be separately handled by languages that require it.

float_len (input)

The length of the floating-point input argument. The allowed values would be 4, 8, and 16.

char_len (input)

The number of decimal characters (digits) in the output character string. There may be from 1 to 35 digits. If the value of the floating-point number *float_input* is zero, an implementation is not required to check the validity of this argument.

dec_chars (output)

A string of 1 to 35 decimal characters representing the pure decimal fraction of the converted result. These digits will contain the leading significant digits of the converted result:

- The leading digit is nonzero if the input floating-point value is not zero.
- The last output digit is correctly rounded.
- The (implied) decimal point lies immediately to the left of the first digit of the output string.
- The width of the output field may be from 1 to 35 characters.

dec_exponent (output)

A signed integer specifying the power of ten by which the decimal fraction must be multiplied to give the true value of the output number. This is the value that will normally be placed following an 'E' in the final result.

The value of this exponent may be adjusted by the caller to accommodate any scale factor that may have been provided in the language's format conversion-control specification. (VS Fortran and PL/I output formatting permit the specification of a scaling factor that shifts the position of the decimal point. For example, the value `' .12345E+67 '` can be displayed as `' 123.45E+64 '`.)

The actual placement of a decimal point, and subsequent adjustments to the external exponent and its conversion and formatting, are the responsibility of the caller.

fc (output/optional)

An 8-byte feedback code. The following conditions may result from this service:

Condition		
CEE03G	Severity	3
	Msg_No	0112
	Message	For data conversion from internal floating-point form to character form, the value specified for the length of the output character string is outside the acceptable range. The valid range for E-format conversion is 1 to 35, and for F-format conversion is 2 to 36.

Condition		
CEE2H8	Severity	0
	Msg_No	2600
	Message	Success with zero result. The conversion has been completed successfully, and the result is a true zero value.
CEE2H9	Severity	0
	Msg_No	2601
	Message	Success with positive result. The conversion has been completed successfully, and the result is strictly greater than zero.
CEE2HA	Severity	0
	Msg_No	2602
	Message	Success with a negative result. The conversion has been completed successfully, and the result is strictly less than zero.

Note: C runtime library (C RTL) can call this routine to perform the *ecvt* function with no extra formatting. The returned function value can be obtained from *dec_chars*. The returned *decpt* value can be obtained from *dec_exponent* and *sign* can be obtained from feedback code.

E-format output examples

For example, the following strings and exponents returned by CEEYCVHE could be converted by its caller to the external user data representations shown in Figure 81.

Output Digits	External Exponent	Examples of Possible Final User Data Representations (Formatted by Caller)			
'10'	+1	1.0	10.E-1	.1E+1	1
'10'	+2	10.0	10.E+0	.1E+2	10
'12'	-1	0.012	12.E-3	1.2E-2	
'501'	0	0.501	5.01E-1	501E-3	
'23'	+3	230.0	2.3E+2	.23E+3	230

Figure 81. Examples of E-format output conversions

The output string of decimal digits is treated as a pure fraction. The exponent and decimal point may be placed in any combination to represent the value in the desired *user data* representation.

CEEYCVHF — F-format output conversion routine

F-format output requires that data be converted to a fixed point format with a specified number of decimal places following the decimal point. As such, it is limited by two factors. First, producing a fixed number of decimal places may be in conflict with the property of floating-point data that it carry a fixed number of *significant* digits. Many values of such data can produce output with more digits than are actually significant. This is one reason for the second limitation: F-format output conversions tend to be applied to data with values in a known, and rather narrower, range than for data using E-format conversions.

Conversion services

F-format output produces fixed point output, with no decimal exponent. The floating-point value 10^{10} would be formatted as a '1' digit followed by ten '0' digits:

```
'10000000000.'
```

and not as (for example) the E-format result

```
'1.0E10'
```

The size of the output string is dictated by three factors:

- The mandatory presence of a decimal point.
- The desired number (including zero) of decimal digits following the decimal point.
- The magnitude of the input floating-point value, which may require the placement of digits preceding the decimal point.

The output value is always right-justified in the output field, with leading blanks filling any unneeded positions. If the output string is too short to hold the output characters, an appropriate return or feedback code will indicate the field overflow.

Visualizing F-format conversions

There is a simple way to visualize what happens in converting floating-point data using F-type formatting box.

1. First, imagine that the number is written in infinite-precision fixed point decimal format.
2. If there is any scaling, move the decimal point right or left appropriately.
3. Put a horizontal window of single-character panes (the window's length is the same as the length of the output character string) over the resulting string of digits, making sure the decimal point is always visible. Round the rightmost visible digit, using the first discarded digit. Further adjustments may be needed to the position of the decimal point after rounding, if a carry out of the high-order position occurs. In this case, the window may have to be shifted, and it is possible the result will no longer fit.
4. Detect error conditions such as:
 - a. No nonzero digits in the window
 - b. Overflow
 - c. No decimal point is visible

Syntax

void CEEYCVHF (*float_input*, *float_len*, *scale*, *frac_digits*, *char_len*, *dec_chars*, [*fc*])

```
VFLOAT    *float_input;  
INT4      *float_len;  
INT4      *scale;  
INT4      *frac_digits;  
INT4      *char_len;  
CHARn     *dec_chars;  
FEED_BACK *fc;
```

CEEYCVHF

Call this CWI interface as follows:

```
L    R12,A(CAA)           address of CAA  
L    R15,CEECAACELV-CEECAA(,R12) address of libvec  
L    R15,3508(,R15)       address of routine  
BALR R14,R15             invoke the CEEYCVHF
```


***float_input* (input)**

The floating-point value to be converted to decimal. A negative zero will be treated as a positive zero. If negative zeros are significant in the calling environment, it is the caller's responsibility to distinguish between negative and positive zeros.

An input floating-point value may be unnormalized. If the presence of unnormalized data is significant in the calling environment, it is the caller's responsibility to detect and accommodate the fact of unnormalization.

Note: VS Fortran currently produces unnormalized E-format decimal output for unnormalized hex inputs. The intent is to let the user see the loss of significance directly. However, it has been decided that this is a language-dependent facility, and should be separately handled by languages that require it.

***float_len* (input)**

The length of the floating-point input argument. The allowed values are 4, 8, and 16.

***scale* (input)**

A signed integer specifying the power of ten by which the decimal fraction must be multiplied to give the desired value of the output number. This value represents the number of places the decimal point should be shifted relative to the unscaled value.

***frac_digits* (input)**

A nonnegative integer specifying the number of digits to be placed following the decimal point.

***char_len* (input)**

The number of decimal characters (digits) in the output character string. There may be from 2 to 36 characters, containing a decimal point and 1 to 35 digits and/or blanks. If the value of the floating-point number *float_input* is zero, an implementation is not required to check the validity of this argument.

***dec_chars* (output)**

A string of 2 to 36 characters (a decimal point, one or more decimal digits, and possibly blanks) representing the converted result. A decimal point is always correctly placed in the *dec_chars* output string.

If the input floating-point value is identically zero, no characters are stored, and the return code will indicate the zero value. The caller is responsible for formatting the user data. This is done because zero fields are often blanked, or are formatted according to rules that may vary, depending on the language.

The nonblank output characters will be right-adjusted in the output string *dec_chars*. The last character position of the string will contain either the correctly rounded least significant decimal digit, or a decimal point, if no fraction digits are requested. Extra (unneeded) character positions at the left end of the string will be set to blanks.

It is possible for all output digits to be zero. If this condition arises, feedback/return codes will indicate that a nonzero input value has produced a correctly-rounded zero output. (Some languages do not permit attaching a minus sign to a zero. Thus, the conversion routines must be able to detect and manage this condition.)

Conversion services

At least one decimal digit is always produced, either immediately before the decimal point (if no fraction digits were requested) or immediately after the decimal point (if the rounded magnitude of the input floating-point number *float_input* is less than 1.0).

No leading zeros will appear to the left of the decimal point. Zero results will cause specific settings of the feedback/return code. A decimal digit will precede the decimal point only if the magnitude of the input floating-point number is at least 1.0.

The last (least significant) digit is always correctly rounded.

The width of the output field may be larger than needed, in the sense that more digits are requested than are truly significant. For example, if the input floating-point value were changed by one ULP, then one or more digits to the left of the lowest-order output digit would change. Alternatively, changing the low-order decimal digit by 1, or more, and converting back to hex would not yield a different hex float number from the value originally supplied for output conversion to decimal.

In all cases, the conversion routine will supply no more than 35 significant digits.

If the magnitude of the input argument would cause the size of the output character string to exceed the allotted string length, the conversion will be abandoned and a feedback/return code setting will indicate conversion failure. The contents of *dec_chars* is unpredictable.

fc (output/optional)

An 8-byte feedback code. The following conditions may result from this service:

Condition		
CEE03F	Severity	3
	Msg_No	0111
	Message	For data conversion from internal floating-point form to character form, the number of fraction digits specified was either negative or greater than the value specified for the length of the character string.
CEE03G	Severity	3
	Msg_No	0112
	Message	For data conversion from internal floating-point form to character form, the value specified for the length of the output character string is outside the acceptable range. The valid range for E-format conversion is 1 to 35, and for F-format conversion is 2 to 36.
CEE2H8	Severity	0
	Msg_No	2600
	Message	Success with zero result. The conversion has been completed successfully, and the result is a true zero value.
CEE2H9	Severity	0
	Msg_No	2601
	Message	Success with positive result. The conversion has been completed successfully, and the result is strictly greater than zero.

Condition		
CEE2HA	Severity	0
	Msg_No	2602
	Message	Success with negative result. The conversion has been completed successfully, and the result is strictly less than zero.
CEE2HB	Severity	0
	Msg_No	2603
	Message	Success with plus-rounded-to-zero result. The conversion has been completed successfully, and the result contains a zero result that was created by a strictly positive input value that rounded to zero.
CEE2HC	Severity	0
	Msg_No	2604
	Message	Success with minus-rounded-to-zero result. The conversion has been completed successfully, and the result contains a zero result that was created by a strictly negative input value that rounded to zero.
CEE2HE	Severity	2
	Msg_No	2606
	Message	Result overflows output field. The <i>float_input</i> argument is either too large, or the output string <i>dec_chars</i> is too small to contain the fixed-point representation of the input argument.

Note: C runtime library (C RTL) can call this routine to perform the basic conversion. It will then format the returned *fcvt* function value based on the returned values in *dec_chars*. The value *decpt* can be obtained by locating the position of the radix character in *dec_chars*. Also, the value of *sign* can be obtained from feedback code.

F-format output examples

In Figure 82 on page 394, the character 'b' represents a blank. Assume in each case that the output character string *dec_chars* is 7 characters long.

Input Float Value	Fraction Digits Requested	Character String	Feedback Code
0.0	5	<none>	Success_with_zero_result
-0.0	5	<none>	Success_with_zero_result
0.0077	0	'bbbbbb0.'	Success_with_plus-rounded-to-zero_result
0.0077	1	'bbbbbb.0'	Success_with_plus-rounded-to-zero_result
-0.0077	0	'bbbbbb0.'	Success_with_minus-rounded-to-zero_result
-0.0077	1	'bbbbbb.0'	Success_with_minus-rounded-to-zero_result
0.0077	2	'bbbb.01'	Success_with_positive_result
0.0077	3	'bbb.008'	Success_with_positive_result
0.0077	4	'bb.0077'	Success_with_positive_result
0.0077	5	'b.00770'	Success_with_positive_result
0.0077	6	'.007700'	Success_with_positive_result
0.5000	0	'bbbbbb1.'	Success_with_positive_result
0.5000	1	'bbbbbb.5'	Success_with_positive_result
3.4567	0	'bbbbbb3.'	Success_with_positive_result
3.4567	1	'bbbbb3.5'	Success_with_positive_result
3.4567	2	'bbb3.46'	Success_with_positive_result
3.4567	3	'bb3.457'	Success_with_positive_result
3.4567	4	'b3.4567'	Success_with_positive_result
3.4567	5	'3.45670'	Success_with_positive_result
3.4567	6	undefined	Result_overflows_output_field
-0.9876	0	'bbbbbb1.'	Success_with_negative_result
-0.9876	1	'bbbbbb1.0'	Success_with_negative_result
-0.9876	2	'bbbbbb.99'	Success_with_negative_result
-0.9876	6	'.987600'	Success_with_negative_result
34.5678	0	'bb1235.'	Success_with_positive_result
34.5678	1	'b1234.6'	Success_with_positive_result
34.5678	2	'1234.57'	Success_with_positive_result
34.5678	3	undefined	Result_overflows_output_field

Figure 82. Examples of F-format output conversions

CEEYCVHI — decimal to float input conversion routine

The input conversion routines take a string of decimal characters representing a pure fraction, and a binary integer representing a decimal exponent, and converts them to the best approximating floating-point value.

It is important to remember that the input datum is considered to be infinitely precise. That is, the conversion routine assumes an infinite number of trailing zeros following the last input digit. Actual real-world data does not behave this way. Such data is usually contaminated by estimation errors in at least the last digit. The ability of an application to capture information that could provide error estimates or error intervals requires techniques beyond the capabilities of these conversion interfaces. (The ACRITH package was designed specifically to provide this kind of information.)

The requirement that the decimal digits be treated as pure fractions may require that numeric user data in each product's favorite or traditional representations be pre-scanned by the caller (for syntactic validation, among other things) to remove embedded decimal points, sign characters, and to determine the value of any explicitly-specified decimal exponent.

For example, suppose the caller wishes to convert input user data from one of these character strings:

```

    ' 12.345 '
or   '.12345E+2 '
or   '12345.E-003'

```

Then, in all of these cases, the character data and exponent passed to the conversion routine would be:

```

exponent   =    +2
digit string = '12345'

```

Similarly, if an input user datum had the form '1.2345E+67', then the character data and exponent passed to the conversion routine would be:

```

exponent   =    +68
digit string = '12345'

```

Further examples are shown in "Input examples" on page 397 and in Figure 83. In Figure 83, floating-point values are shown as their equivalent decimal approximations. To illustrate, the following digit strings and exponents would be converted to the internal floating-point values as indicated.

Decimal Digit String	Decimal Exponent	Resulting Output Floating-point Value
'1'	+5	10000.0
'10'	-1	0.01
'10'	-2	0.001
'501'	-3	.000501
'23'	+1	2.3
'23'	+3	230.0

Figure 83. Examples of input conversions

The string of decimal digits is treated as a pure fraction. The decimal exponent is derived externally (by the caller) from a combination of known decimal point placement, any explicit exponent of a form like 'Enn', and the scale factor.

Syntax

void CEEYCVHI (*dec_chars*, *char_len*, *dec_exponent*, *float_len*, *float_result*, [*fc*])

```

CHARn      *dec_chars;
INT4       *char_len;
INT4       *dec_exponent;
INT4       *float_len;
VFLOAT     *float_result;
FEED_BACK  *fc;

```

CEEYCVHI

Call this CWI interface as follows:

```

L      R12,A(CAA)           address of CAA
L      R15,CEECAACELV-CEECAA(,R12) address of libvec
L      R15,3512(,R15)       address of routine
BALR   R14,R15              invoke the CEEYCVHI

```

dec_chars (input)

A string of 1 to 35 decimal characters representing the pure decimal fraction to

Conversion services

be converted. There may be no signs, decimal points, commas, exponents, etc. in the string. The leading digit may be zero. The input service will automatically handle unnormalized data, for example: '0025'. However, the total number of input digits processed is still limited to 35. The string of digits is treated by the conversion routine as though there is an implied decimal point at the left end of the string.

char_len (input)

The number of decimal characters (digits) in the input character string. There may be from 1 to 35 digits. The actual length of the input character string may be greater than the number of decimal digits, but only the number of digits specified will be used in the conversion.

The conversion routine will ignore excess decimal digits whose values cannot affect the value of the converted result. (35 decimal digits provide sufficient precision to separate all representable 16-byte hex float values.) Invalid data in those ignored digit positions may or may not cause unpredictable results or other error or exception conditions.

dec_exponent (input)

A signed integer specifying the power of ten by which the decimal fraction must be multiplied to give the true value of the input number to be converted to floating-point. The caller determines this value from a combination of:

- The position of any decimal point (if it was present) within the original input user data,
- Any exponent value specified in the original user data.
- Any scale factor specified in the HLL's format conversion-control specification.

float_len (input)

The length of the floating-point result. For System/370 hexadecimal floating-point, the allowed values are 4, 8, and 16; for IEEE floating-point, the allowed values are 4 and 8.

float_result (output)

The converted floating-point result. A result may or may not be stored here; see the discussion of *fc*.

fc (output/optional)

An 8-byte feedback code. The following conditions may result from this service:

Condition		
CEE03E	Severity	3
	Msg_No	0110
	Message	For data conversion from character form to internal floating-point form, an invalid character was specified in the input character string <i>character_string</i> .
CEE03H	Severity	3
	Msg_No	0113
	Message	For data conversion from character form to internal floating-point form, the value specified for the length of the input character string is outside the acceptable range. The valid range is 1 to 35.

Condition		
CEE2H8	Severity	0
	Msg_No	2600
	Message	Success with zero result. The conversion has been completed successfully, and the result is a true zero value.
CEE2H9	Severity	0
	Msg_No	2601
	Message	Success with positive result. The conversion has been completed successfully, and the result is strictly greater than zero.
CEE2HF	Severity	2
	Msg_No	2607
	Message	Result has underflowed. The conversion would have resulted in a number smaller than the underflow threshold for the floating-point representation. A true floating-point zero result has been returned in <i>float_result</i> .
CEE2HG	Severity	2
	Msg_No	2608
	Message	Result has overflowed. The conversion would have resulted in a number larger than the overflow threshold for the floating-point representation. The maximum possible floating-point magnitude has been returned in <i>float_result</i> .

Input examples

Figure 84 shows examples of input conversions, including zero and out-of range values. The example assumes that a System/370 4-byte hexadecimal *float_result* was requested.

Decimal Digits	Decimal Exponent	4-byte Hex Float_Result	Feedback Code
'100'	+4	X'433E8000'	Success_with_positive_result
'100'	+3	X'42640000'	Success_with_positive_result
'100'	+2	X'41A00000'	Success_with_positive_result
'100'	+1	X'41100000'	Success_with_positive_result
'100'	0	X'4019999A'	Success_with_positive_result
'100'	-1	X'3F28F5C3'	Success_with_positive_result
'000'	+4	X'00000000'	Success_with_zero_result
'12345'	-123	X'00000000'	Result_has_underflowed
'12345'	+123	X'7FFFFFFF'	Result_has_overflowed

Figure 84. Examples of input conversions with feedback indicated

Conversion services

Chapter 12. Dump and tracing services

This section covers the dump services available in Language Environment and includes information on the Language Environment tracing facilities.

Dump services

Language Environment provides the following dump services. The dumps generated by these services are designed to be easier to read than a system dump, which can often minimize the need to examine a system dump. The first service, CEE3DMP, is a callable service. The remaining are CWIs and are not intended to be called by the application writer.

CEE3DMP

CEE3DMP is a callable service. It dumps the runtime environment of Language Environment and the member language libraries in an easily understandable form. CEE3DMP is the only dump service that can be called directly by an application program. It produces a dump report that is formatted into pages for printing. When providing a multithread dump, Language Environment must quiesce all other threads within the application. When the dump option THREAD(CURRENT) is specified, only the current thread is dumped. When the dump option THREAD(ALL) is specified, the current thread is dumped first and then, starting with the IPT, all remaining threads are dumped one at a time. For more information see *z/OS Language Environment Programming Reference*.

CEESDMP

CEESDMP is a CWI. It symbolically dumps the variables of one routine. The format is similar to the symbolic dump of variables in the CEE3DMP report.

CEETRCB

CEETRCB is a CWI. This low-level service assists in tracing the call chain backwards. It identifies the language, program unit, entry point, current location, caller's DSA, and other information from the address of a DSA or save area for a program unit.

CEETBCK

CEETBCK is a CWI which will replace CEETRCB. It assists in tracing the call chain backwards. It identifies the language, program unit, entry point, current location, caller's DSA, and other information from the address of a DSA or save area for a program unit.

To perform language-specific processing for CEE3DMP, CEESDMP, and CEETRCB, each HLL must provide utility and dump exit routines in their libraries. These exit routines are called with the member event handler using event codes 6 and 7. For a description of the calling method, see "Language Environment member list and event handler" on page 83. The following CWI dump services can only be used in dump exits. They format information that is placed in the dump report or written to the message file:

CEELDMP

Writes a single line message into the dump report.

Dump and Tracing Services

CEEVDMP

Formats the name, type, value, and other information about a variable and writes it to the dump report.

CEEHDMP

Dumps a section of storage in hexadecimal and character.

CEEBDMP

Dumps a control block.

Figure 85 shows possible transfers of control among an application program, a member language library, and Language Environment for dump processing.

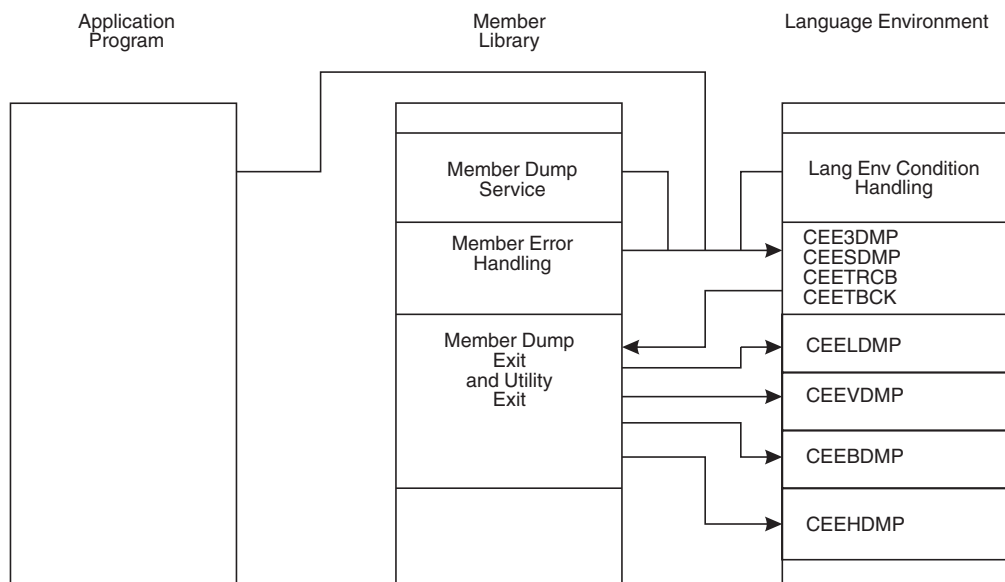


Figure 85. Transferring control between application program, member language library, and Language Environment

CEE3DMP, CEESDMP, CEETBCK or CEETRCB can be called:

- By a member language library as part of member language dump services
- By a member language library as part of language-specific error handling
- By Language Environment exception handling

In addition, CEE3DMP can be called directly by the application program. These dump utilities then call member language library dump exits and utility exits, which in turn call CEELDMP, CEEVDMP, CEEBDMP, and CEEHDMP.

Output from CEE3DMP is written to a file whose DDNAME is specified on the call to CEE3DMP using Language Environment message services. All output from CEESDMP is written to the message file.

The remainder of this section describes the dump services, and their linkage to a dump exit routine. Linkage to the utility exit is described in “Event code 6 — event handler utilities event” on page 487.

CEE3DMP — runtime environment dump service

This callable service generates a dump of the runtime environment. Sections of the dump are selectively included, depending on options specified with the *options* parameter. For more information see *z/OS Language Environment Programming Reference*.

CEESDMP — symbolic dump of a routine

This low-level service symbolically dumps all variables in a program unit to the message file. The format of this dump is similar to the symbolic dump of variables in the CEE3DMP report when the VARIABLES option is specified.

Syntax

void CEESDMP (*dsaptr, fc*)

```
INT4      *dsaptr;
FEED_BACK *fc;
```

CEESDMP

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L    R15,2892(,R15)
BALR R14,R15
```

dsaptr (input)

A fullword containing the DSA address of the routine whose variables are being dumped.

fc (output)

A 12-byte feedback code passed by reference. The following symbolic conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE30V	Severity	3
	Msg_No	3103
	Message	An error occurred in writing messages to the dump file. This could be caused by a bad file name specified with the FNAME option.

Note: Member language dump exits are called to interpret the values of their symbol tables and dump variables with calls to CEEVDMP. This is done primarily with function codes 4 and 5. This is identical to the processing of CEE3DMP with function codes 2 and 3 except that CEEVDMP formats the data for the terminal and sends it to the message file.

CEETRCB — traceback utility

Note: CEETRCB has been deprecated but remains for compatibility. CEETRCB does not provide information about the format of the DSA. It should be considered obsolete and calls to it should eventually be replaced with calls to the CWI CEETBCK.

This low-level service assists in tracing the call chain backwards. It identifies the language, program unit, entry point, current location, caller's DSA, and other information from the address of a DSA or save area for a program unit. This is essential for creating meaningful traceback messages.

Syntax

void CEETRCB (*dsaptr, caaptr, member_id, program_unit_name, program_unit_name_length, program_unit_address, entry_name, entry_name_length, entry_address, call_instruction_address, statement_id, statement_id_length, cibptr, main_program, callers_dsaptr, fc*)

```

POINTER *dsaptr;
POINTER *caaptr;
INT4 *member_id;
CHARn *program_unit_name;
INT4 *program_unit_name_length;
INT4 *program_unit_address;
CHARn *entry_name;
INT4 *entry_name_length;
INT4 *entry_address;
INT4 *call_instruction_address;
CHARn *statement_id;
INT4 *statement_id_length;
POINTER *cibptr;
INT4 *main_program;
POINTER *callers_dsaptr;
FEED_BACK *fc;

```

CEETRCB

Call this CWI interface as follows:

```

L    R15,CEECAACELV-CEECAA(,R12)   Address of CAA in R12
L    R15,0072(,R15)
BALR R14,R15

```

dsaptr (input)

A fullword pointer containing the address of the DSA for the current routine in the traceback. This can also be the address of a standard 18 fullword save area if Language Environment conventions were not used for the routine.

caaptr (input)

A fullword pointer containing the address of the CAA associated with the DSA or save area pointed to by *dsaptr*.

member_id (output)

A fullword binary integer to contain the member identifier for the routine associated with the DSA. If the member ID cannot be determined, this parameter is set to negative one.

program_unit_name (output)

A fixed-length character string of arbitrary length to contain the name of the program unit containing the routine associated with the DSA. If the program unit name cannot be determined, this parameter is set to all blanks. If the program unit name cannot fit within the supplied string, it is truncated. (Truncation of DBCS preserves even byte count and SI/SO pairing.)

program_unit_name_length (input/output)

A fullword binary integer containing the length of the program unit name string on entry and the actual length of the program unit name placed in the string on exit. If the program unit name cannot be determined, this parameter is set to zero. The maximum length a string can be is 256 bytes. Lengths less than zero are treated as zero. Lengths greater than 256 are treated as 256.

***program_unit_address* (output)**

A fullword binary integer containing the address of the start of the program unit for the routine associated with the DSA. If the program unit address cannot be determined, this parameter is set to zero.

***entry_name* (output)**

A fixed-length character string of arbitrary length to contain the name of the entry point into the routine associated with the DSA. If the entry point name cannot be determined, this parameter is set to all blanks. If the entry point name cannot fit within the supplied string, it is truncated. (Truncation of DBCS preserves even byte count and SI/SO pairing.)

***entry_name_length* (input/output)**

A fullword binary integer containing the length of the entry point name string on entry and the actual length of the entry point name placed in the string on exit. If the entry point name cannot be determined, this parameter is set to zero. The maximum length a string can be is 256 bytes. Lengths less than zero are treated as zero. Lengths greater than 256 are treated as 256.

***entry_address* (output)**

A fullword binary integer that contains the address of the entry point into the routine associated with the DSA. If the entry point address cannot be determined, this parameter is set to zero.

***call_instruction_address* (output)**

A fullword binary integer that contains the address of the instruction that caused transfer out of the routine. This is either the address of a BALR or BASSM instruction if transfer was made by subroutine call, or the address of the interrupted statement if transfer was caused by an exception. If the address cannot be determined, this parameter is set to zero.

***statement_id* (output)**

A fixed-length character string of arbitrary length that contains the identifier of the statement containing the instruction which caused transfer out of the routine. If the statement cannot be determined, this parameter is set to all blanks. If the statement ID cannot fit within the supplied string, it is truncated. (Truncation of DBCS preserves even byte count and SI/SO pairing.)

***statement_id_length* (input/output)**

A fullword binary integer containing the length of the statement ID string on entry and the actual length of the statement ID placed in the string on exit. If the statement ID cannot be determined, this parameter is set to zero. The maximum length a string can be is 256 bytes. Lengths less than zero are treated as zero. Lengths greater than 256 are treated as 256.

***cibptr* (output)**

A fullword pointer containing the address of the CEECIB associated with the DSA if an exception occurred. If no exception occurred, this parameter is set to zero. Note that if an exception caused transfer out of the routine, the state of the registers after the last instruction ran in the routine is saved in the CIB, rather than in the DSA.

***main_program* (output)**

A fullword binary integer set to one of the following:

- 0 The routine associated with the DSA is not the main program.
- 1 The routine associated with the DSA is the main program.

***callers_dsaptr* (output)**

A fullword pointer containing the address of the DSA or save area of the caller.

If the address of the caller's DSA cannot be determined or is not valid (points to inaccessible storage), then this parameter is set to zero.

fc (output)

A 12-byte feedback code passed by reference. The following symbolic conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE310	Severity	3
	Msg_No	3104
	Message	Information could not be successfully extracted for this DSA. It is likely that the <i>dsaptr</i> parameter does not point to an actual DSA or save area.

Note: CEETRCB uses member event handler utility exits, described in “Event code 6 — event handler utilities event” on page 487.

CEETBCK — traceback utility (replaces CEETRCB)

The CEETBCK CWI assists in tracing the call chain backwards. It identifies the language, program unit, entry point, current location, caller's DSA, and other information from the address of a DSA or save area for a program unit. This is essential for creating meaningful traceback messages. The CWI will handle both upward- and downward-growing stacks.

Note: There are several reasons for executing CEETBCK instead of just updating CEETRCB:

- For XPLINK, a routine's registers on entry are saved in the routine's own stack frame instead of its caller's stack frame.
- For XPLINK, the return address stored in a DSA is the caller's return address and not the return address to the stack frame owner.
- Additional parameters which indicate the stack frame format for both the input and the output (caller's) DSA are maintained.
- The function of the *call_instruction_address* parameters has changed and a new parameter *callers_call_instruction* has been added.

Syntax

```
void CEETBCK (dsaptr, dsa_format, caaptr, member_id, program_unit_name,
program_unit_name_length, program_unit_address, call_instruction_address, entry_name,
entry_name_length, entry_address, callers_call_instruction_address, callers_dsaptr,
callers_dsa_format, statement_id, statement_id_length, cibptr, main_program, fc)
```

```
POINTER *dsaptr;
INT4 *dsa_format;
POINTER *caaptr;
INT4 *member_id;
CHARn *program_unit_name;
INT4 *program_unit_name_length;
INT4 *program_unit_address;
INT4 *call_instruction_address;
CHARn *entry_name;
INT4 *entry_name_length;
```

```

INT4      *entry_address;
INT4      *callers_call_instruction_address;
POINTER   *callers_dsaptr;
INT4      *callers_dsa_format;
CHARn     *statement_id;
INT4      *statement_id_length;
POINTER   *cibptr;
INT4      *main_program;
FEED_BACK *fc;

```

CEETBCK

Call this CWI interface as follows:

```

L      R15,CEECAALEOV-CEECAA(,R12)   Address of CAA in R12
L      R15,304(,R15)
BALR   R14,R15

```

***dsaptr* (input)**

A fullword pointer containing the address of the DSA for the current routine in the traceback. This can also be the address of a standard 18 fullword save area if Language Environment conventions were not used for the routine.

***DSA_format* (input/output)**

A fullword binary integer set to one of the following:

- 0** The format of the DSA is a Standard OS linkage register save area (with or without Language Environment fields, including the next available byte).
- 1** The format of the DSA is XPLINK style.
- 1** The format of the DSA is unknown. When multiple calls are made to CEETBCK to scan the call chain, the *callers_dsa_format* returned from the previous call can be used here.

***caaptr* (input)**

A fullword pointer containing the address of the CAA associated with the DSA or save area pointed to by *dsaptr*.

***member_id* (output)**

A fullword binary integer containing the member identifier for the routine associated with the DSA. If the member ID cannot be determined, this parameter is set to negative one.

***program_unit_name* (output)**

A fixed-length character string of arbitrary length containing the name of the program unit containing the routine associated with the DSA. If the program unit name cannot be determined, this parameter is set to all blanks. If the program unit name cannot fit within the supplied string, it is truncated. (Truncation of DBCS preserves even byte count and SI/SO pairing.)

***program_unit_name_length* (input/output)**

A fullword binary integer containing the length of the program unit name string on entry, and the actual length of the program unit name placed in the string on exit. If the program unit name cannot be determined, this parameter is set to zero. The maximum length a string can be is 256 bytes. Lengths less than zero are treated as zero. Lengths greater than 256 are treated as 256.

***program_unit_address* (output)**

A fullword binary integer containing the address of the start of the program unit for the routine associated with the DSA. If the program unit address cannot be determined, this parameter is set to zero.

***call_instruction_address* (input/output)**

A fullword binary integer that contains the address of the instruction that

caused transfer out of the routine. This is either the address of a BASR, BALR or BASSM instruction if transfer was made by subroutine call, or the address of the interrupted statement if transfer was caused by an exception. When multiple calls are made to CEETBCK to scan the call chain, the *callers_call_instruction* returned from the previous call can be used here. If the address is not known, this parameter should be set to zero. When this parameter is zero on input and the address can be determined, it will be returned.

***entry_name* (output)**

A fixed-length character string of arbitrary length to contain the name of the entry point into the routine associated with the DSA. If the entry point name cannot be determined, this parameter is set to all blanks. If the entry point name cannot fit within the supplied string, it is truncated. (Truncation of DBCS preserves even byte count and SI/SO pairing.)

***entry_name_length* (input/output)**

A fullword binary integer containing the length of the entry point name string on entry, and the actual length of the entry point name placed in the string on exit. If the entry point name cannot be determined, this parameter is set to zero. The maximum length a string can be is 256 bytes. Lengths less than zero are treated as zero. Lengths greater than 256 are treated as 256.

***entry_address* (output)**

A fullword binary integer that contains the address of the entry point into the routine associated with the DSA. If the entry point address cannot be determined, this parameter is set to zero.

***callers_call_instruction_address* (output)**

A fullword binary integer that contains the address of the instruction that caused transfer out of the caller. This is either the address of a BASR, BALR or BASSM instruction if transfer was made by subroutine call, or the address of the interrupted statement if transfer was caused by an exception. If the address cannot be determined, this parameter is set to zero.

***callers_dsaptr* (output)**

A fullword pointer containing the address of the DSA or save area of the caller. If the address of the caller's DSA cannot be determined or is not valid (points to inaccessible storage), then this parameter is set to zero.

***callers_DSA_format* (output)**

A fullword binary integer set to one of the following:

- 0 The format of the DSA is a Standard OS linkage register save area (with or without Language Environment fields, including the next available byte.)
- 1 The format of the DSA is XPLINK style.

***statement_id* (output)**

A fixed-length character string of arbitrary length that contains the identifier of the statement containing the instruction which caused transfer out of the routine. If the statement cannot be determined, this parameter is set to all blanks. If the statement ID cannot fit within the supplied string, it is truncated. (Truncation of DBCS preserves even byte count and SI/SO pairing.)

***statement_id_length* (input/output)**

A fullword binary integer containing the length of the statement ID string on entry, and the actual length of the statement ID placed in the string on exit. If the statement ID cannot be determined, this parameter is set to zero. The

maximum length a string can be is 256 bytes. Lengths less than zero are treated as zero. Lengths greater than 256 are treated as 256.

***cibptr* (output)**

A fullword pointer containing the address of the CEECIB associated with the DSA if an exception occurred. If no exception occurred, this parameter is set to zero. Note that if an exception caused transfer out of the routine, the state of the registers after the last instruction ran in the routine is saved in the CIB, rather than in the DSA.

***main_program* (output)**

A fullword binary integer set to one of the following:

- 0 The routine associated with the DSA is not the main program.
- 1 The routine associated with the DSA is the main program.

***fc* (output)**

A 12-byte feedback code passed by reference. The following symbolic conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE310	Severity	3
	Msg_No	3104
	Message	Information could not be successfully extracted for this DSA. It is likely that the <i>dsaptr</i> parameter does not point to an actual DSA or save area.

Note: CEETBCK uses member event handler utility exits (Event Code 6).

Example

The code example below shows a sample program called CELEBC53, which uses the CEETBCK function.

```

/*****
*
* CELEBC53: This example uses the CEETBCK() function.
*
* Notes: Can be compiled C or C++, 31-bit XPLINK or non-XPLINK
*
*       For non-XPLINK C++, compile with DLL(CBA) compiler
*       option
*
*       For non-XPLINK C, compile with NODLL (default) or
*       DLL(CBA) compiler options.
*
*       Use the DEBUG(SYMBOL) compiler option to get non-blank
*       statement IDs.
*
* Flow:  - main() calls function1()
*         - function1() calls function2()
*         - function2() does divide by 0, causing SIGFPE, which drives
*         catch1()
*         - catch1() raise()s SIGUSR1, which drives catch2()
*         - catch2() calls CEE3CIB() to get a starting DSA for the
*         traceback, and then calls CEETBCK() in a loop
*
*****/
/* XPLINK(ON) and STACK(1K,1K,...) cause extra DSAs to appear */
#pragma runopts(POSIX(ON), XPLINK(ON), STACK(1K,1K,ANY,KEEP,1K,1K))

```

CEETBCK

```

#define _XOPEN_SOURCE_EXTENDED 1
#include <ceedcct.h>
#include <errno.h>
#include <leawi.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>          /* _FBCHECK() uses memcmp()          */

/* ----- */
/* Prototype for CEETBCK() function and MACRO to invoke it */
/* ----- */

/* non-XPLINK -- use Library vector, since there is no stub */
/* ===== */

#ifdef __XPLINK__

#ifdef __cplusplus
extern "C"
#endif
typedef
void CEETBCK_t( _POINTER * /* dsaptr          */
, _INT4 * /* dsa_format          */
, _POINTER * /* caaptr          */
, _INT4 * /* member_id          */
, char * /* program_unit_name    */
, _INT4 * /* program_unit_name_length */
, _INT4 * /* program_unit_address */
, _INT4 * /* call_instruction_address */
, char * /* entry_name          */
, _INT4 * /* entry_name_length    */
, _INT4 * /* entry_address        */
, _INT4 * /* callers_call_instruction_address */
, _POINTER * /* callers_dsaptr    */
, _INT4 * /* callers_dsa_format    */
, char * /* statement_id         */
, _INT4 * /* statement_id_length  */
, _POINTER * /* cibptr           */
, _INT4 * /* main_program         */
, _FEEDBACK * /* fc              */
);

typedef
struct ceeleov          /* partial mapping of library vector */
{
    char    pad__[304]; /* skip down to CEETBCK entry */
    void    *CEELEOVTBCK; /* pointer to CEETBCK in library vector */
} ceeleov_t;

struct caa          /* partial mapping of CAA */
{
    char    pad__[816]; /* skip down to library vector ptr */
    ceeleov_t *ceecaaleov; /* pointer to library vector */
};

/* MACRO to get address of CEETBCK() from library vector, and make */
/* sure high-order bit is off */

#define CEETBCK \
( \
*(CEETBCK_t *) \
( \
( \
(unsigned long) \
(((struct caa *)_gtca())->ceecaaleov)->CEELEOVTBCK) \
) \
& \
0x7FFFFFFFUL \
) \
)

#else

/* XPLINK -- use XPLINK side deck instead of library vector */
/* ===== */

#ifdef __cplusplus
extern "C"

```

```

#endif
void CEETBCK( _POINTER * /* dsaptr */
, _INT4 * /* dsa_format */
, _POINTER * /* caaptr */
, _INT4 * /* member_id */
, char * /* program_unit_name */
, _INT4 * /* program_unit_name_length */
, _INT4 * /* program_unit_address */
, _INT4 * /* call_instruction_address */
, char * /* entry_name */
, _INT4 * /* entry_name_length */
, _INT4 * /* entry_address */
, _INT4 * /* callers_call_instruction_address */
, _POINTER * /* callers_dsaptr */
, _INT4 * /* callers_dsa_format */
, char * /* statement_id */
, _INT4 * /* statement_id_length */
, _POINTER * /* cibptr */
, _INT4 * /* main_program */
, _FEEDBACK * /* fc */
);

#pragma map(CEETBCK, "CEEKTBCK")
#endif

/* ----- */
/* Signal catcher for SIGFPE */
/* ----- */
#ifdef __cplusplus
extern "C"
#endif

void catch1(int sig)
{
    #line 4444
    if (0 != raise(SIGUSR1))
    {
        printf("raise(SIGUSR1) failed, errno=%d\n", errno);
        exit(-1);
    }

    exit(0); /* normal exit -- can't return after divide by 0 */
}

/* ----- */
/* Signal catcher for SIGUSR1 */
/* ----- */

#ifdef __cplusplus
extern "C"
#endif
void catch2(int sig)
{
    int rc;
    int loop;
    _CEE3CIB *cib_ptr;
    _POINTER dsaptr;
    _INT4 dsa_format;
    _POINTER caaptr;
    _INT4 member_id;
    char program_unit_name[2000];
    _INT4 program_unit_name_length;
    _INT4 program_unit_address;
    _INT4 call_instruction_address;
    char entry_name[256];
    _INT4 entry_name_length;
    _INT4 entry_address;
    _INT4 callers_call_instruction_address;
    _POINTER callers_dsaptr;
    _INT4 callers_dsa_format;
    char statement_id[256];
    _INT4 statement_id_length;
    _POINTER cibptr;
    _INT4 main_program;
    _FEEDBACK fc;

    /* Find CIB to get a starting DSA for input to CEETBCK() */

    CEE3CIB( (_FEEDBACK *)NULL /* get most recent CIB */
, &cib_ptr /* pointer to most recent CIB */

```

CEETBCK

```

        , &fc                /* feedback code from CEE3CIB */
    );
if (0 != _FBCHECK(fc, CEE000))
{
    printf("CEE3CIB failed -- fc != CEE000\n");
    exit(-1);
}

if (cib_ptr == NULL)
{
    printf("No CIB pointer returned from CEE3CIB()\n");
    exit(-1);
}

/* Set up for first call to CEETBCK */
caaptr   = (_POINTER)gtca(); /* our CAA */
dsaptr   = (_POINTER)(cib_ptr->cib_svl); /* starting DSA */
dsa_format = -1; /* DSA format unknown */
call_instruction_address = 0; /* not yet known */

/* Main loop to call CEETBCK to display call chain traceback */
/* ----- */

loop = 1;
do
{
    program_unit_name_length = sizeof program_unit_name;
    entry_name_length        = sizeof entry_name;
    statement_id_length      = sizeof statement_id;

    /* Call CEETBCK to get information about one DSA */
    CEETBCK( &dsaptr
        , &dsa_format
        , &caaptr
        , &member_id
        , program_unit_name
        , &program_unit_name_length
        , &program_unit_address
        , &call_instruction_address
        , entry_name
        , &entry_name_length
        , &entry_address
        , &callers_call_instruction_address
        , &callers_dsaptr
        , &callers_dsa_format
        , statement_id
        , &statement_id_length
        , &cibptr
        , &main_program
        , &fc
    );

    if (0 != _FBCHECK(fc, CEE000))
    {
        printf("CEETBCK failed -- dsaptr = %08p\n", dsaptr);
        exit(-1);
    }
    printf("-----\n");
    printf("DSA ptr/fmt (input) = %08X/%d\n"
        , dsaptr
        , dsa_format
    );
    printf("DSA ptr/fmt (caller) = %08X/%d\n"
        , callers_dsaptr
        , callers_dsa_format
    );
    printf("CAA/CIB ptr = %08X/%08X %s\n"
        , caaptr
        , cibptr
        , cibptr == NULL ? "" : "<==== CIB"
    );

    printf("main-flag/member-ID = %d/%d %s\n"
        , main_program
        , member_id
        , main_program == 1 ? "(main)" : ""
    );
    printf("PU addr/entry addr = %08X/%08X\n"
        , program_unit_address
        , entry_address
    );
}

```

```

printf("CALL addr/caller's      = %08X/%08X\n"
      ,
      ,
      );
printf("PU name                  = \".*.s\" \n"
      ,
      ,
      );
printf("entry name              = \".*.s\" \n"
      ,
      ,
      );
printf("statement ID            = \".*.s\" \n"
      ,
      ,
      );

/* Set up to call CEETBCK for next DSA, or end loop          */

if (callers_dsaptr != 0)
{
    dsaptr          = callers_dsaptr;
    dsa_format      = callers_dsa_format;
    call_instruction_address = callers_call_instruction_address;
}
else
{
    printf("-----\n");
    loop = 0;          /* end of traceback -- end loop          */
}
} while (loop == 1);
return;
}
/* ----- */
/* function1() and function2() */
/* ----- */

void function2( unsigned long long ull
               , unsigned long   ul
               , long            l
               )
{
    /* Cause divide by zero, to raise SIGFPE and drive catch1() */

    #line 3333
    printf("!!! shouldn't see this line !!!\n", 1/ull, 1/ul, 1/l);
}

void function1(void)
{
    #line 2222
    function2(0ULL, 0UL, 0L);          /* should not return          */
}

/* ----- */
/* Main program */
/* ----- */

int main(void)
{
    if (SIG_ERR == sigset(SIGFPE, &catch1))
    {
        printf("sigset(SIGFPE, &catch1) failed, errno=%d\n", errno);
        return -1;
    }
    if (SIG_ERR == sigset(SIGUSR1, &catch2))
    {
        printf("sigset(SIGUSR1, &catch2) failed, errno=%d\n", errno);
        return -1;
    }
    #line 1111 function1();          /* should not return          */
    /* return -1;                  /* shouldn't get here          */
}

```

The code example below shows the output produced by sample program CELEBC53, shown above.

```

/*****
Sample output, when running CELEBC53 compiled non-XPLINK C++ :
-----
DSA ptr/fmt (input)   = 227E5B00/1
DSA ptr/fmt (caller) = 227E5BC0/1

```

CEETBCK

```

CAA/CIB ptr           = 219B2B48/227FD990 <==== CIB
main-flag/member-ID  = 0/3
PU addr/entry addr   = 223B33D8/223B33D8
CALL addr/caller's   = 223B3A54/21B6CB92
PU name              = ""
entry name           = "raise"
statement ID         = ""
-----
DSA ptr/fmt (input)  = 227E5BC0/1
DSA ptr/fmt (caller) = 227F8018/0
CAA/CIB ptr         = 219B2B48/00000000
main-flag/member-ID = 0/1
PU addr/entry addr   = 21B6B950/21B6B9A8
CALL addr/caller's   = 21B6CB92/0001FEF2
PU name              = ""
entry name           = "CEEVROND"
statement ID         = ""
-----
DSA ptr/fmt (input)  = 227F8018/0
DSA ptr/fmt (caller) = 227F33D8/0
CAA/CIB ptr         = 219B2B48/00000000
main-flag/member-ID = 0/1
PU addr/entry addr   = 0001FEE0/0001FEE0
CALL addr/caller's   = 0001FEF2/2199EC1A
PU name              = "CEEVSSFR"
entry name           = "CEEVSSFR"
statement ID         = ""
-----
DSA ptr/fmt (input)  = 227F33D8/0
DSA ptr/fmt (caller) = 227F3218/0
CAA/CIB ptr         = 219B2B48/00000000
main-flag/member-ID = 0/3
PU addr/entry addr   = 2199EB88/2199EB88
CALL addr/caller's   = 2199EC1A/21B6E830
PU name              = "'/'POSIX.ESAME.TESTCASE.C(CELEBC53)'"
entry name           = "catch1"
statement ID         = "4444"
-----
DSA ptr/fmt (input)  = 227F3218/0
DSA ptr/fmt (caller) = 227F9800/1
CAA/CIB ptr         = 219B2B48/00000000
main-flag/member-ID = 0/1
PU addr/entry addr   = 21B6D7E8/21B6D7E8
CALL addr/caller's   = 21B6E830/225976A4
PU name              = "CEEVRONU"
entry name           = "CEEVRONU"
statement ID         = ""
-----
DSA ptr/fmt (input)  = 227F9800/1
DSA ptr/fmt (caller) = 227FA360/1
CAA/CIB ptr         = 219B2B48/00000000
main-flag/member-ID = 0/3
PU addr/entry addr   = 22596A78/22596A78
CALL addr/caller's   = 225976A4/21B686B2
PU name              = ""
entry name           = "_zerro"
statement ID         = ""
-----
DSA ptr/fmt (input)  = 227FA360/1
DSA ptr/fmt (caller) = 227E3AA0/1
CAA/CIB ptr         = 219B2B48/00000000
main-flag/member-ID = 0/1
PU addr/entry addr   = 21B686A0/21B686B0
CALL addr/caller's   = 21B686B2/22596A36
PU name              = ""
entry name           = "CEEVHPR"
statement ID         = ""
-----
DSA ptr/fmt (input)  = 227E3AA0/1
DSA ptr/fmt (caller) = 227E3B20/1
CAA/CIB ptr         = 219B2B48/00000000
main-flag/member-ID = 0/3
PU addr/entry addr   = 22596860/22596860
CALL addr/caller's   = 22596A36/21B6CB92
PU name              = ""
entry name           = "_zerros"
statement ID         = ""

```

```

-----
DSA ptr/fmt (input) = 227E3B20/1
DSA ptr/fmt (caller) = 227F3018/0
CAA/CIB ptr = 219B2B48/00000000
main-flag/member-ID = 0/1
PU addr/entry addr = 21B6B950/21B6B9A8
CALL addr/caller's = 21B6CB92/0001FEF2
PU name = ""
entry name = "CEEVROND"
statement ID = ""
-----
DSA ptr/fmt (input) = 227F3018/0
DSA ptr/fmt (caller) = 227F4098/0
CAA/CIB ptr = 219B2B48/00000000
main-flag/member-ID = 0/1
PU addr/entry addr = 0001FEE0/0001FEE0
CALL addr/caller's = 0001FEF2/21A89254
PU name = "CEEVSSFR"
entry name = "CEEVSSFR"
statement ID = ""
-----
DSA ptr/fmt (input) = 227F4098/0
DSA ptr/fmt (caller) = 227EF158/0
CAA/CIB ptr = 219B2B48/00000000
main-flag/member-ID = 0/1
PU addr/entry addr = 21A88528/21A88528
CALL addr/caller's = 21A89254/2199EEB6
PU name = "CEEHDS"
entry name = "CEEHDS"
statement ID = ""
-----
DSA ptr/fmt (input) = 227EF158/0
DSA ptr/fmt (caller) = 227EF098/0
CAA/CIB ptr = 219B2B48/227F4990 <==== CIB
main-flag/member-ID = 0/3
PU addr/entry addr = 2199EE18/2199EE18
CALL addr/caller's = 2199EEB6/2199EFB8
PU name = "'/''POSIX.ESAME.TESTCASE.C(CELEBC53)'"
entry name = "function2(unsigned long long,unsigned long,long)"
statement ID = "3333"
-----
DSA ptr/fmt (input) = 227EF098/0
DSA ptr/fmt (caller) = 227EF018/0
CAA/CIB ptr = 219B2B48/00000000
main-flag/member-ID = 0/3
PU addr/entry addr = 2199EF40/2199EF40
CALL addr/caller's = 2199EFB8/0001FEF2
PU name = "'/''POSIX.ESAME.TESTCASE.C(CELEBC53)'"
entry name = "function1()"
statement ID = "2222"
-----
DSA ptr/fmt (input) = 227EF018/0
DSA ptr/fmt (caller) = 227D01B8/0
CAA/CIB ptr = 219B2B48/00000000
main-flag/member-ID = 0/1
PU addr/entry addr = 0001FEE0/0001FEE0
CALL addr/caller's = 0001FEF2/2199F28A
PU name = "CEEVSSFR"
entry name = "CEEVSSFR"
statement ID = ""
-----
DSA ptr/fmt (input) = 227D01B8/0
DSA ptr/fmt (caller) = 227D00F0/0
CAA/CIB ptr = 219B2B48/00000000
main-flag/member-ID = 0/3
PU addr/entry addr = 2199EFE0/2199EFE0
CALL addr/caller's = 2199F28A/22266AD0
PU name = "'/''POSIX.ESAME.TESTCASE.C(CELEBC53)'"
entry name = "main"
statement ID = "1111"
-----
DSA ptr/fmt (input) = 227D00F0/0
DSA ptr/fmt (caller) = 227D0030/0
CAA/CIB ptr = 219B2B48/00000000
main-flag/member-ID = 0/3
PU addr/entry addr = 222669F0/222669F0
CALL addr/caller's = 22266AD0/21A55660
PU name = "EDCZHINV"
entry name = "EDCZHINV"

```

```

statement ID      = ""
-----
DSA ptr/fmt (input) = 227D0030/0
DSA ptr/fmt (caller) = 00000000/0
CAA/CIB ptr      = 219B2B48/00000000
main-flag/member-ID = 1/1 (main)
PU addr/entry addr = 21A554A8/21A554A8
CALL addr/caller's = 21A55660/2199E3C4
PU name          = "CEEBBEXT"
entry name       = "CEEBBEXT"
statement ID      = ""
-----
*/

```

Member language dump exit

While dump services are running, all member-specific processing is performed through an exit to the member event handler with an event code of 7. For more information about establishing member event handlers, see “Language Environment member list and event handler” on page 83. Each member language is required to supply a dump exit routine.

CEELDMP — single line message dump service

The CEE3DMP low-level service allows member language dump exits to place a single-line message into the dump. These are usually informational messages, such as the attributes of a file. Member language library dump exits should always use CEELDMP to write messages into the dump. They should never write directly to CEE3DMP through Language Environment message services. Otherwise, Language Environment dump services cannot keep track of the number of lines in the dump to break pages correctly.

Syntax

void CEELDMP (*message*, *message_length*, *fc*)

```

CHARn    *message;
INT4     *message_length;
FEED_BACK *fc;

```

CEELDMP

Call this CWI interface as follows:

```

L    R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L    R15,0036(,R15)
BALR R14,R15

```

message (input)

A fixed-length character string of arbitrary length containing the message to be placed in the dump. It does not include printer control characters. Control characters and leading blanks are added by Language Environment dump services.

message_length (input)

A fullword binary integer containing the length of the message string. The string can be up to 120 bytes long. The string length is treated as zero if it is less than zero. String lengths greater than 120 are truncated to 120 bytes.

fc (output)

A 2-byte feedback code passed by reference. The following symbolic conditions can result from this service; if more than one error condition occurs, the feedback code reflects the last diagnosed condition:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE30S	Severity	2
	Msg_No	3100
	Message	The message was longer than 120 bytes. It has been truncated to 120.
CEE30V	Severity	3
	Msg_No	3103
	Message	An error occurred in writing messages to the dump file.

CEEVDMP — variable dump service

CEEVDMP is a low-level service that assists in formatting and dumping variables for member languages. This service promotes consistency in the display of variables among member languages.

Syntax

void CEEVDMP (*statement_id, statement_id_length, indent, level, name, name_length, type, type_length, value, value_length, value_division, array_continued, fc*)

```
CHARn    *statement_id;
INT4     *statement_id_length;
INT4     *indent;
INT4     *level;
CHARn    *name;
INT4     *name_length;
CHARn    *type;
INT4     *type_length;
CHARn    *value;
INT4     *value_length;
INT4     *value_division;
INT4     *array_continued;
FEED_BACK *fc;
```

CEEVDMP

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)   Address of CAA in R12
L    R15,0048(,R15)
BALR R14,R15
```

statement_id (input)

A fixed-length character string containing an identifier of the statement from which the dumped variable is declared. This is usually a statement number.

statement_id_length (input)

A fullword binary integer containing the length of the statement identifier. The length is zero if there is no statement identifier for the variable. The maximum length is 8 bytes. Values less than zero are treated as zero. Values greater than 8 bytes are truncated to 8.

indent (input)

A fullword binary integer containing the number of additional blanks to insert after the statement identifier and before the level of the variable. This feature indents fields of a structure or elements of an array. The maximum indent

allowed is 10 blanks. Indent values less than zero are regarded as zero. Values greater than 10 blanks are truncated to 10.

***level* (input)**

A fullword binary integer containing the level of the variable if the variable is a field in a record or structure. It can be in the range of 1 to 255. If the language does not use level numbers or the variable does not have a level number, the level value is zero. Level values less than zero are regarded as zero. Values greater than 255 are truncated to 255.

***name* (input)**

A fixed-length character string containing the name of the variable. The subscript is part of character string if the variable is an array element.

***name_length* (input)**

A fullword binary integer containing the length of the name. If the *name_length* is greater than 16 characters, the succeeding fields of the message are placed on the next line in the dump. A *name_length* less than zero is regarded as zero. The maximum *name_length* is 60 characters. Lengths greater than 60 characters are truncated to 60.

***type* (input)**

A fixed-length character string containing the variable data type. Other variable attributes can be placed in this string if they are known. The type character string should contain only blanks if the data type is not known. Trailing blanks are ignored. If the length of the string is more than 16 characters, the type and value fields of the message are placed on the next line in the dump.

***type_length* (input)**

A fullword binary integer containing the length of the data type string. The value field of the message is placed on the next line in the dump if the *type_length* is greater than 16 characters. The maximum *type_length* is 60 characters. A *type_length* that is less than zero is regarded as zero. Values greater than 60 characters are truncated to 60.

***value* (input)**

A fixed-length character string containing the value of the variable. For arrays, it is recommended that the value field show more than one element of the array to minimize the number of lines in the dump. Examples of array output are shown 2 on page 418.

***value_length* (input)**

A fullword binary integer containing the length of the value string. If the length of the string is more than 60 characters, it is divided and printed on one or more following lines in the dump as needed. The actual point of division is indicated by the *value_division* parameter. A *value_length* that is less than zero is regarded as zero.

***value_division* (input)**

A fullword binary integer indicating how a value string should be divided when it is more than 60 characters. It can contain one of the following values; values that are not valid are treated as 1.

- 1 Divide the string every 60 characters, without regard to the contents of the string.
- 2 Divide the string at blanks, if possible.

***array_continued* (input)**

A fullword binary integer indicating additional calls to CEEVDMP to dump other elements of the same array. This allows CEEVDMP to compress multiple

lines of the same array with the same values. This saves space if the array contains many elements with the same value. This parameter should be one of the following; values that are not valid are treated as 0.

- 0 This is not a dump of an array or this is the last call to CEEVDMP to dump an array.
- 1 This is the dump of an array and additional calls to CEEVDMP are made to dump additional elements of this array.

fc (output)

A 12-byte feedback code passed by reference. The following symbolic conditions can result from this service; if more than one error condition occurs, the feedback code reflects the last diagnosed condition:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE30V	Severity	3
	Msg_No	3103
	Message	An error occurred in writing messages to the dump file.
CEE312	Severity	3
	Msg_No	3106
	Message	An invalid parameter value was specified.

The dump output has the format shown in Figure 86.



Figure 86. CEEVDMP output format

- 1 The statement identifier. This field is removed if the identifier is all blanks or zero.
- 2 The indent, shown at its maximum of 10 blanks. There is always one more space between the statement identifier and the level than the indent number. So, for this example, there are 11 spaces.
- 3 The level number; if the level number is 0, this field is removed.
- 4 The name of the variable, array, or field.
- 5 The type information.
- 6 A character representation of the value of the data. It is the responsibility of the user of CEEVDMP to translate the data to character and to precede the value with the appropriate number of spaces if right justification is desired. Any characters with byte values between X'00' and X'3F' are displayed as periods, except for DBCS shift-out and shift-in codes. When the value wrapping to a new line causes DBCS data to be divided, even byte count and SI/SO pairing is preserved.

Usage Notes:

CEEVDMP

1. When elements in the dump of an array are removed because the elements have the same value, the following message is inserted (a and b are the names of the first and last element suppressed):
a to b elements same as above.
2. The following is an example of a variable and a record in COBOL:

```
00000024 77 BAD-FLAG      X          N
00000029 01 PRINT-DATE   AN-GR
00000030 02 FILLER          X(16)      TODAY'S DATE IS
00000032 02 PRINT-MONTH    X(9)        APRIL
00000033 02 FILLER          XX          *** Invalid data for this data type ***
                                         Hex 0000
00000034 02 PRINT-DAY     99          *** Invalid data for this data type ***
                                         Hex 0000
00000035 02 FILLER          XXX
00000036 02 PRINT-YEAR    99          ,19
                                         88
```

CEEHDMP — hexadecimal storage dump service

The CEEHDMP low-level service dumps a section of storage in both hex and character representations. It contains protection against addresses that are not valid.

Syntax

void CEEHDMP (*title, title_length, address, length, fc*)

```
CHARn    *title;
INT4     *title_length;
INT4     *address;
INT4     *length;
FEED_BACK *fc;
```

CEEHDMP

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L      R15,0044(,R15)
BALR   R14,R15
```

title (input)

A fixed-length character string that identifies the displayed storage section.

title_length (input)

A fullword binary integer containing the length of the title. The maximum length is 60 characters.

address (input)

A 31-bit address of the first byte of storage to be dumped.

length (input)

A fullword binary integer containing the length of the storage area.

fc (output)

A 12-byte feedback code passed by reference. The following symbolic conditions can result from this service; if more than one error condition occurs, the feedback code reflects the last diagnosed condition:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE30T	Severity	2
	Msg_No	3101
	Message	The title string was longer than 132 characters and was truncated.
CEE30V	Severity	3
	Msg_No	3103
	Message	An error occurred in writing messages to the dump file.
CEE313	Severity	3
	Msg_No	3107
	Message	Dump terminated before all storage could be dumped because inaccessible storage was encountered.

Lines in the dump contain the format shown in Figure 87.

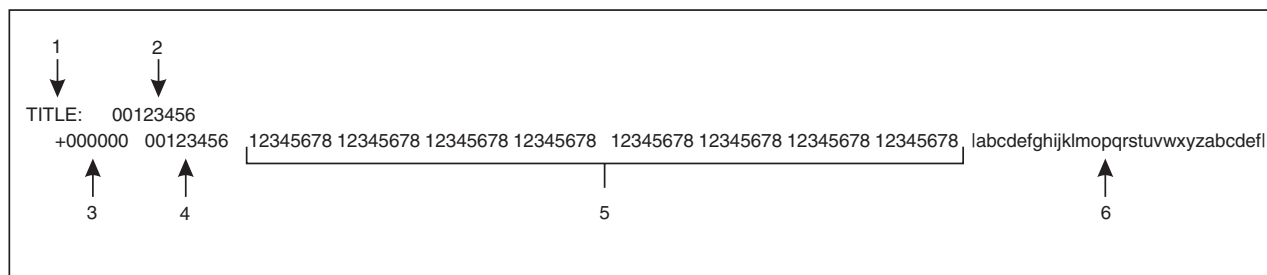


Figure 87. CEEHDMP output format

- 1 The string given on the *title* argument. The string is truncated if it is too long to fit on a single dump line.
- 2 The starting address of this section of storage.
- 3 The offset in hexadecimal from the first byte of the dump.
- 4 The hexadecimal address of the first byte dumped on the line.
- 5 32 bytes of storage dumped as 8 single hexadecimal numbers.
- 6 The same 32 bytes of storage dumped in character form. Any byte values between X'00' and X'3F' are displayed as periods, however.

Usage Notes:

1. If an address that is not valid is detected, the following message is displayed instead of the storage contents:
Inaccessible storage.
2. CEEHDMP suppresses multiple lines of identical data, as CEEVDMP does.

CEEBDMP — control block dump service

The CEEBDMP low-level service dumps a control block with field identifiers. The fields themselves can be displayed in binary, hexadecimal, or character. CEEBDMP provides a standard format for control block dumps by determining how many fields to display on each line of a dump.

Syntax

void CEEBDMP (*title, title_length, address, offset, nfields, field_ids, field_lengths, field_types, fc*)

```
CHARn    *title;
INT4     *title_length;
INT4     *address;
INT4     *offset;
INT4     *nfields;
CHAR8    *field_ids;
INT4     *field_lengths;
INT4     *field_types;
FEED_BACK *fc;
```

CEEBDMP

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)   Address of CAA in R12
L    R15,0052(,R15)
BALR R14,R15
```

title (input)

A fixed-length character string that identifies the control block.

title_length (input)

A fullword binary integer containing the length of the title. The maximum length is 60 characters.

address (input)

The 31-bit address of the control block.

offset (input)

A fullword binary integer containing the offset from the address of the control block to the first field of the control block.

nfields (input)

A fullword binary integer containing the number of fields in the control block.

field_ids (input)

An array of 8-character strings. Each element in the array contains an identifier for a field in the control block. The elements appear in the array in the same order in which the fields are arranged in storage. If a field identifier is less than 8 characters long, it should be left justified and padded on the right with blanks.

field_lengths (input)

An array of fullword binary integers containing the byte length of the fields in the control block. For example, a fullword pointer would be 4 bytes long and thus have a length of 4. This table parallels the *names* array.

field_types (input)

An array of fullword binary integers containing codes for field dump formats. The codes are defined as follows:

- 1 Display the field in hexadecimal.
- 2 Display the field in binary.
- 3 Display the field in character.

fc (output)

A 12-byte feedback code passed by reference. The following symbolic conditions can result from this service; if more than one error condition occurs, the feedback code reflects the last diagnosed condition:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE30T	Severity	2
	Msg_No	3101
	Message	The title string was longer than 132 characters and was truncated.
CEE30V	Severity	3
	Msg_No	3103
	Message	An error occurred in writing messages to the dump file.
CEE313	Severity	3
	Msg_No	3107
	Message	Dump terminated before entire control block could be dumped because inaccessible storage was encountered.

The control block dump has the format shown in Figure 88.

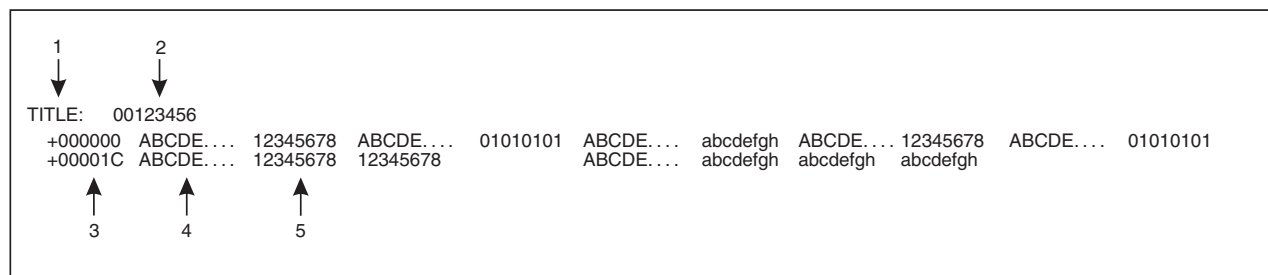


Figure 88. CEEBDMP output format

- 1 The name of the control block specified by the title argument.
- 2 The address of the control block.
- 3 The offset in the control block to the first field listed on the line. It is always preceded with a plus (+) or minus (-) sign.
- 4 The identifier for a field in the control block. The identifier can be up to 8 characters. It is padded with periods to make the field length 9 characters long.
- 5 The contents of the field. It is either a binary, hexadecimal, or character string. If the field is longer than 4 bytes and is displayed in hexadecimal, or longer than 1 byte and is displayed in binary, or longer than 8 bytes and displayed in character, then additional contents are displayed where the next field would normally be displayed, separated by two blanks. Examples of this form in hexadecimal and character are shown in the third line of the Output Format shown above. When displaying a field in character, any byte values between X'00' and X'3F' are displayed as periods.

Note: If an address that is not valid is detected, the following message is displayed instead of field identifiers and contents:

Inaccessible storage.

Other dump-related CWIs

The following CWIs, which are described in this section, provide additional dump-related services:

- CEE3CDO
- CEEKSNP
- CEEURTB

CEE3CDO — check dump options

CEE3CDO validates the options that could be passed to the Language Environment callable service CEE3DMP.

Syntax

void (*CEECELVKCD0) (*options, position, [fc]*)

```
VSTRING *options;
INT4     *position;
FEED_BACK *fc;
```

CEECELVKCD0

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L    R15,3380(,R15)
BALR R14,R15
```

options (input)

A halfword-prefixed character string containing options describing the type, format, and destination of dump information. Options are declared as a string of keywords separated by blanks or commas. Some options have suboptions which follow the option keyword and are contained in parentheses. These are the same options supported by the Language Environment callable service CEE3DMP. For more information, see *z/OS Language Environment Programming Guide*.

position (output)

A fixed-binary(31) integer that is the index (character offset within the string) where the first option or delimiter that is not valid was discovered. If no errors are discovered, this value is zero.

fc (output/optional)

A 12-byte feedback code passed by reference. If specified as an argument, feedback information (a condition token) is returned to the calling routine. If not specified, and the requested operation was not successfully completed, the condition is signaled to the condition manager. The following symbolic conditions can result from this service; if several simultaneous error conditions occur, the feedback code reflects the last diagnosed condition:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE314	Severity	2
	Msg_No	3108
	Message	An invalid option, suboption, or delimiter was found.

CEEKSNP — produce a SNAP dump

This CWI generates a system SNAP dump of the runtime environment. Once complete, execution continues.

Syntax

```
void (*CEECELVKSNP) (id, reserved, [fc]);
```

```
INT4      *id;
VSTRING   *reserved;
FEED_BACK *fc;
```

CEECELVKSNP

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L      R15,3392,R15
BALR   R14,R15
```

id (input)

An integer in the range 0 to 255 used in an identification string within the SNAP output.

reserved (input)

A halfword-prefixed character string reserved for future use. Its value must be a zero-length character string.

fc (output/optional)

A 12-byte feedback code passed by reference. If specified as an argument, feedback information (a condition token) is returned to the calling routine. If not specified, and the requested operation was not successfully completed, the condition is signaled to the condition manager. The following symbolic conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE33R	Severity	1
	Msg_No	3195
	Message	The SNAP dump file could not be opened.
	Explanation	The SNAP dump file could not be opened.
	Programmer Response	If a SNAP dump was desired, determine the reason the file could not be opened and correct the problem.
	System Action	No SNAP dump was taken.
CEE33S	Severity	1
	Msg_No	3196
	Message	The ID number was not in the allowed range.
	Explanation	The ID number must be in the range 0 to 255; it was not in that range.
	Programmer Response	This is an internal problem. Contact your service representative.
	System Action	The ID number 255 was used.

Condition		
CEE33T	Severity	1
	Msg_No	3197
	Message	An invalid value for <i>reserved</i> was passed.
	Explanation	An invalid value for the <i>reserved</i> argument was passed to the SNAP dump service.
	Programmer Response	This is an internal problem. Contact your service representative.
	System Action	The invalid value was ignored.
CEE33U	Severity	3
	Msg_No	3198
	Message	A SNAP dump was requested on an unsupported system.
	Explanation	The SNAP dump service was called to produce a SNAP dump on an unsupported system.
	Programmer Response	This is an internal problem. Contact your service representative.
	System Action	The SNAP dump was not produced.
CEE33V	Severity	3
	Msg_No	3199
	Message	An error was returned from the SNAP system function.
	Explanation	The SNAP system function returned an error. The SNAP dump service could not be completed.
	Programmer Response	This is an internal problem. Contact your service representative.
	System Action	The SNAP dump was not produced.

Usage Notes:

1. This service is not available under CICS. Calling it when executing under CICS results in feedback code CEE33U.
2. The ddname used is CEESNAP. If CEESNAP is not defined then no dump is produced and CEE33R is returned.
3. CEEKSNP uses the SDATA=(ALL) SNAP option, which dumps items such as the PSA, SQA, SWA, I/O supervisor control blocks, and the PDATA=(ALL) SNAP option, which dumps items such as the JPA, LPA, virtual storage subpools (0-127, 252).
4. The contents of the SNAP dump reflects the state of the registers and memory at the time the SNAP macro is called.

CEEURTB — produce a user routine traceback

The CEEURTB CWI generates a traceback of user routines from the point of the call to CEEURTB. The traceback consists of, where determinable, entry name, program unit name, statement number, offset and entry address. The output of the traceback is directed to the MSGFILE. When complete, execution continues.

Syntax

```
void (*CEECELMURTB) (levels, [fc]);
```

```
INT4      *levels;
FEED_BACK *fc;
```

CEECELMURTB

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)  Address of CAA in R12
L      R15,3368(,R15)
BALR   R14,R15
```

levels (input)

A fixed-binary (31) number representing the maximum number of levels of user routines to trace back. If *levels* is 0, then all user routines are traced back.

fc (output/optional)

A 12-byte feedback code passed by reference. If specified as an argument, feedback information (a condition token) is returned to the calling routine. If omitted, and the requested operation was not successfully completed, the condition is signaled to the condition manager.

The following symbolic conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE30Q	Severity	3
	Msg_No	3098
	Message	The user routine traceback could not be completed.
	Explanation	The user routine traceback could not be completed due to an error detected in tracing back through the DSA chain.
	Programmer Response	Attempt to perform problem determination through the use of a dump.
	System Action	The user routine traceback is not completed.

Usage Notes:

1. *levels* refers to the number of program unit level qualifiers within an application. For example, nested PL/I begin blocks are treated as one level.
2. If *levels* is 1, the format of the traceback is as follows. The traceback is in text; the traceback stops at the first program unit level qualifier. For example, nested PL/I begin blocks are treated as one level.

```
from entry BEGIN BLOCK at entry offset +00000082 at address 00020D70
from entry EXT_AB at entry offset +00000036 at address 00020D78
from entry EXT_01 at entry offset +00000040 at address 00020C58
from program unit DUMP03 at entry LABEL_E: BEGIN at statement 22 at offset
+00000082 at address 00020598
```

3. If *levels* is 0 (complete traceback) or greater than 1, the traceback is generated in a table format for the requested number of levels. The format of the traceback is as follows.

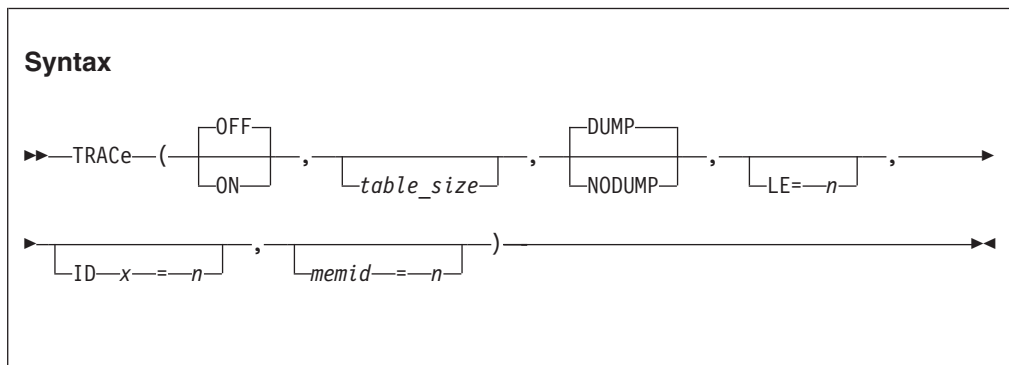
Tracing

```

traceback of user routines:
Program Unit  Entry          Statement PU Offset  Entry Offset  Address
              BEGIN BLOCK   +00000D70  +00000082  00020D70
              EXT_AB        +00000D28  +00000036  00020D28
              EXT_01        +00000C58  +00000040  00020C58
DUMP03       LABEL_E: BEGIN
              22 +00000598  +00000082  00020598
DUMP03       E              21 +000004F8  +00000082  000204F8
DUMP03       D              17 +00000454  +00000086  00020454
DUMP03       %BLOCK5        14 +000003AC  +0000008E  000203AC
DUMP03       C              13 +00000318  +0000007E  00020318
DUMP03       B              11 +0000020C  +000000EE  0002020C
DUMP03       A              6  +0000015C  +00000092  0002015C
DUMP03       DUMP03         4  +000000AC  +000000AC  000200AC
  
```

Tracing services

The TRACE runtime option controls whether tracing is active, the size of the trace buffer, the type of trace events to record, and whether a dump containing only the trace table should be taken at enclave termination. TRACE establishes the setting that indicates if the trace facility is active.



ON Indicates the tracing facility is active.

OFF

Indicates the tracing facility is inactive; this the default.

table_size

Determines the size of the trace table and is specified in bytes or as *nK*.

DUMP

Requests that a Language Environment formatted dump, containing at a minimum the trace table, be taken at program termination (normal or abnormal) independent of the setting of the TERMTHDACT runtime option.

LE=

Requests global trace for all Language Environment members to generate trace records into the trace table. All members include Language Environment as well as z/OS XL C/C++ and Berkeley sockets. The number that can be specified on this parameter is only the global trace levels (for a description of global and member-specific trace levels, see the description of *n* below). The value is limited to a range of 0–FF

IDx=

Identifies a specific Language Environment member ID that generates trace records for the trace table. This parameter is to be used only under the direction of IBM service.

x Specifies the member ID number, where *x* can be an integer from 1-17, inclusive. More than one $IDx = n$ can be specified at a time.

memid=

A symbolic name of the following specific Language Environment members:

CEL Language Environment member ID of 1

C370 Language Environment member ID of 3 (C/C++)

PLI Language Environment member ID of 10

Specific member tracing is specified by either the $IDx=$ or the *memid=* parameter. For example, C runtime library tracing can be specified either as $ID3=$ or $C370=$; CEL can be specified as $ID1=$ or $CEL=$. More than one *memid=* can be specified at a time. This parameter is to be used only under the direction of IBM service.

n A hex number that represents a 32-bit mask where each bit is associated with a specific trace type. The *n* value can be a maximum of 8 characters that represent a maximum of 8 hex digits. If less than 8 hex digits are specified, the value is padded on the left with zeroes (for example, 17 represents X'00000017'). The low-order eight bits are reserved for global trace events (those that apply to all Language Environment members).

Global and member-specific tracing

Every Language Environment member has a 32-bit field that contains its trace levels (or trace types). The first 24 bits are defined by each member; these bits are referred to as the member trace levels. Language Environment defines the last 8 bits to have one specific meaning across all Language Environment members; these bits are referred to as the global trace levels. In Language Environment, only the following global trace levels are defined:

0 No tracing
1 Library entry and exit trace
2 Locking trace
4 Monitor call
20 XPLINK/non-XPLINK transition trace for AMODE 31 only. If #pragma linkage (xxxxxxx, OS_UPSTACK) is specified, no transitions are recorded.

For a description of trace types 1 and 2, see *z/OS Language Environment Debugging Guide*. Trace type 4 enables and disables monitor call number 1351.

Global trace levels can be set in two ways. First, using the $LE=n$ option on the trace runtime option. You can activate either of the global trace options by specifying $LE=1$, $LE=2$, or both of the global traces by specifying $LE=3$. Second, using the low-order 8 bits of the 32-bit field for specific member using the *memid=* suboption or the $IDx=$ suboption. You can activate the library entry and exit global trace for callable service calls and return to Language Environment by specifying $CEL=1$. Note the X'01' is the low-order 8 bits of the 32-bit field.

A member can choose not to implement one of the global trace levels, but it must not redefine the meaning of the low-order 8 bits.

The member-specific trace suboption for the C/C++ library is called C370. For the TRACE option, the terms C/370, C370, and C/C++ are used interchangeably. In all cases they refer to the C/C++ language-specific runtime library component of Language Environment. Some examples of the trace setting are:

TRACE(ON,,,LE=1)

Set global trace type 1 (RTL entry/exit) tracing

Tracing

TRACE(ON,,,LE=2)

Set global trace type 2 user mutex and condition variable tracing

TRACE(ON,,,LE=3)

Set both trace types 1 and 2

TRACE(ON,,,CEL=1)

Set Language Environment callable service entry/exit tracing

TRACE(ON,,,C370=1)

Set C/C++ runtime library function entry/exit tracing

TRACE(ON,,,LE=1,CEL=100,C370=200)

Set the first Language Environment trace plus the second C/C++ trace and the first global trace level for all members in the application

TRACE(ON,,,LE=1,CEL=102,C370=200)

Set the first Language Environment trace plus the second global trace but only for Language Environment, plus the second C/C++ trace and the first global trace level for all members

TRACE(ON,,,LE=20)

Set global trace type 20 XPLINK/non-XPLINK transition tracing for AMODE 31 only. Transitions across OS_UPSTACK linkage are not recorded.

TRACE runtime options usage notes

- The IBM-supplied default is TRACE(ON, 4K, LE=0, CEL=700).
- The TRACE suboptions for member-specific tracing are not recommended for regular use, but are used by customers under the direction of IBM service.

CEEKCTRC — add a trace table entry

CEEKCTRC is a CWI callable service that adds a Trace Table Entry (TTE) to the single trace table.

Syntax

void CEEKCTRC (*trace_buffer*, [*trace_buffer_len*])

CHAR4 **trace_buffer*;
INT4 * [*trace_buffer_len*];

CEEKCTRC

Call this CWI interface as follows:

```
L R15,CEECAACELV-CEECAA(,R12) Address of CAA in R12
L R15,3480(,R15)
BALR R14,R15
```

trace_buffer (input)

The trace buffer to be included in the TTE when it is added to the trace table. It is the caller's responsibility to provide the trace buffer with the member ID and the member-specific type along with the member-specific information, if any.

trace_buffer_len (input)

The length of the trace buffer. The minimum length is 8 bytes (which includes the member ID and the member-specific Type), and the maximum length is 112. If omitted, 112 is assumed. The format of the first 8 bytes is as follows; the remaining 104 bytes are member-definable:

Byte	Usage
------	-------

- 0 Member ID
- 1-3 Member-defined flags
- 4-7 Member-defined trace record type

Usage Notes:

1. Callers of this service do not need to have acquired a new stack frame, because this service has a dedicated save area for its use.
2. The CWI adds the timestamp and the thread ID to the TTE.
3. Caller's of this CWI must first test the `CEECAA_TRACE_ACTIVE` flag before calling this CWI.
4. The trace CWI can be called:
 - From the start of member enclave initialization
 - To the end of member enclave termination
 - Excluding member dump processing

Calls to the CEEKCTRC CWI should be conditional. It is the responsibility of the caller to insure this. All trace points in the RTL would follow the basic structure shown in Figure 89.

At member enclave initialization, the member's unique trace levels and the low-order 8 bits of the global trace levels from the OCB should be merged and stored in a 32-bit field in the member's thread-level control block, such as:

```

struct {
    /* Trace levels:
    /* Combination of 24 member
    /* unique trace levels and 8
    /* global trace levels
    /*
    /* Member unique trace levels:
    int free_1    : 21; /* 'FFFFFF8..'x -
    int TraceType3 : 1; /* '....4..'x - Trace type 3
    int TraceType2 : 1; /* '....2..'x - Trace type 2
    int TraceType1 : 1; /* '....1..'x - Trace type 1
    /*
    /* Global trace levels:
    int free_3    : 6; /* '.....FC'x -
    int RtlLocks  : 1; /* '.....2'x - RTL/user locking
    int RtlFunc   : 1; /* '.....1'x - RTL function
    /*
    /* entry/exit
    } Trace;
    /*
  
```

At every trace point, the following code would be used to test for trace active and that this specific trace type has been requested.

```

#include "the file that contains your trace structure"
.
.
.
if ((ceecaa_trace) && /* Trace is active
    (Trace.TraceType1)) { /* and specific trace type was
    /* specified
    Format trace entry
    Invoke CEEKCTRC
  }
  
```

Figure 89. Example: calling the CEEKCTRC CWI from the C RTL

Chapter 13. Subsystem considerations

Language Environment provides support which, when used in conjunction with facilities provided in CICS Version 4 Release 1, gives programmers the ability to write and run Language Environment-enabled command level application programs (run units) in the CICS environment. When this support is used, CICS appears to the Language Environment-enabled program essentially as an operating system, and provides all job, task, and program management facilities. If CICS is being run in 31-bit mode under z/OS, this support permits Language Environment-enabled application programs to run in either 24-bit mode or 31-bit mode. Language Environment-enabled application programs also run on CICS under z/OS.

Communication between a Language Environment-enabled program (run unit) and a non-Language Environment-enabled program (run unit) can be accomplished with CICS facilities such as EXEC CICS LINK and XCTL commands.

Note: Fortran is not supported in this environment.

CICS and POSIX

Applications running with POSIX(ON) and not supported under CICS; if you try running an application with POSIX(ON) under CICS, you receive a warning message and execution continues.

Background information

The following sections provide some background information necessary before a detailed discussion of CICS.

Terminology

The following terminology is unique to the CICS environment. The Language Environment program model under batch environment defines some of the terms differently. "Language Environment-CICS and Language Environment-batch program models" on page 434 correlate these two program models.

CICS CICS is a licensed program product that runs on S/370, ESA/370, and ESA/390 architectures. It consists of a general purpose data communication or on-line transaction processing system, an on-line system controller, and some batch utilities capable of supporting a network of many thousands of terminals. It is used for commercial business transactions, rather than primarily scientific or engineering work. Throughout this section, the term "CICS" indicates CICS/ESA, CICS/MVS, or both; it does not indicate CICS/VM, unless an explicit reference is required.

Translator

A CICS Command Language Translator takes the application program source code and translates the CICS commands into the appropriate language statements. It also provides useful diagnostics.

Partition

A fixed-size subdivision of main storage allocated to a job step or system task. For example, a partition is established during a CICS initialization

(start-up job). Partition initialization is creation of an environment which is common to all transactions running in that environment.

Thread

A collection of (or a transaction consisting of) one or more run units (programs), each of which can be at a different language level. There can be multiple threads (transactions) running in parallel within a single CICS partition. The run units in a thread communicate with each other only by issuing EXEC CICS LINK or XCTL commands.

Transaction

A piece of processing initiated by a single request (transaction ID code), usually from a terminal. A single transaction consists of one or more application programs (run units) that, when run, carry out the processing needed.

Task

The CICS activity necessary to set up and run an application program on behalf of a user is called a task. A task is, in the simple case, an instance of a transaction. A task can read from and write to the terminal, read and write files, start other tasks, and do many other things. All these activities are controlled by and requested through CICS commands in the application program. CICS manages many tasks at the same time. The number of tasks running at any one instant depends on the characteristics of the processor.

Run unit

A statically and/or dynamically bound running set of one or more programs (defined below) that communicate with each other by CALL statement. In a CICS environment, a run unit is called at the start of a CICS task (triggered by entering a transaction ID at the terminal) or by issuing EXEC CICS LINK or EXEC CICS XCTL commands from another run unit. Each run unit has its own Language Environment environment.

Program

A running (link-edited load module) set of one or more object programs that communicate with each other by static CALL statements. Unlike run units, programs are called with dynamic CALLs (through use of EXEC CICS LOAD). Programs that call each other dynamically are part of the same run unit. They run in the same Language Environment environment as their caller. A program must be defined as a single entry in PPT. Notice that a single run unit can have multiple programs separately link-edited with separate PPT entries.

Language Environment-Enabled Program

A program with a special layout entry that contains Language Environment eye catcher (CEE) and references to Prolog Information Blocks. This is also referred to as a Fully Language Environment-Enabled program. Prolog Information Blocks contain information that is needed by Language Environment while the program is running. For more information on requirements of being a Language Environment-Enabled Program, see "Routine layout" on page 6.

Object Program

A set or group of machine language instructions that can be run, and other material designed to interact with data to provide problem solutions. Object programs are generated as a result of source program compilation (or assembly).

Compiler

A program that translates a program written in an HLL into a machine language object program.

Working Storage

Depends on the programming language of the application program, as follows:

ASM The storage defined in the current DFHEISTG DSECT.

COBOL

All data storage defined in the WORKING-STORAGE section of the program.

Except for COBOL programs, working storage starts with a standard format register save area; that is, R14 and R12 are stored at offset 12 and R13 at offset 4.

Token A group of language characters that logically belong together. Tokens such as keywords, symbols and storage addresses are used to identify a given environment. Language Environment adopts storage addresses as environment tokens. Partition, thread and run unit tokens are double-word. The first word is zero and the second word is a 31-bit address.

LIBVEC

A Language Environment vector transfer table, which is part of CEEPCOM and contains a series of slots, one for each Language Environment routine called. A slot is also provided for special CICS routines.

PCT CICS Program Control Table that defines the transactions known to the system.

PPT CICS Processing Program Control Table that defines all the application programs and maps in the system, and also various CICS modules and tables.

Running a program under CICS

It is useful to review some contextual information about running an application program under CICS before we proceed. Most importantly, the terms “run unit” and “program” require some clarification.

An event, generally receipt of an input message containing a transaction ID code, or possibly receipt of data identifying some other event which has been equated to a transaction ID code, (a 3270 terminal Program Function Key), triggers a CICS transaction.

CICS looks up the transaction ID code in the Program Control Table (DFHPCT) and extracts from that table the name of the program (or at least the first program) that is to process the transaction.

In preparation for running this program (and any other programs subsequently called as part of this transaction), the CICS Task Management Program attaches a CICS task to define the transaction as an item of work dispatchable by the CICS task dispatcher. This task is purely a CICS task, not an operating system task.

When the transaction processing task exists, the CICS Program Control Program looks up the identity of the required program in the Processing Program Table (DFHPPT) that contains information about programs (for example, language,

whether in storage or not, use count, and entry point address). Next, the CICS Program Control Program loads the program into storage if necessary and calls runtime language interface module (loaded during CICS initialization) to initialize the runtime environment and call the program.

This program need not perform all the processing associated with a transaction. It can request to call other programs with EXEC CICS LINK, EXEC CICS XCTL, or a language CALL construct. Each program identified in such a request is accessed and called with its entry in DFHPPT just as the first one was. From the CICS program management point of view, initiation of the first application program associated with this transaction task marks the beginning of a thread of programs.

This thread of programs can traverse only one program or many programs, all called in the service of the original transaction. Each such program must have an entry in the PPT and must be loadable by the CICS program loader. It must be a link-edited load module. It can contain a single object module or object modules produced by several compilations and combined by the linkage editor. Within the load module, the component modules can call each other with CALL statements, or function invocations, but CICS has no knowledge of this.

Each program (link-edited load module with a PPT entry) can be written in any language that is compatible with CICS (COBOL, C/C++, PL/I, or assembler). When a program is called through CICS facilities, it represents a run unit and has its own Language Environment environment. Statically linked ILC applications are supported under CICS between COBOL, C/C++, and assembler. Both static and dynamic ILC between PL/I, COBOL, and C are supported under CICS.

CICS discourages the coding of large or complex programs and encourages the implementation of complex transactions by the use of several programs called with, and communicating by means of, CICS facilities.

In summary, for the rest of this discussion, the term program in the Language Environment-CICS application environment means a link-edited load module with a PPT entry, consisting of at least one program's object module and perhaps other object modules. Such a program, when called with CICS facilities (EXEC CICS LINK or XCTL), represents a run unit and has its own Language Environment environment. The terms run unit and program can loosely be used unless an attention to a particular one is required.

Language Environment-CICS and Language Environment-batch program models

The following illustration and notes describe the correlation between the Language Environment-CICS program model and the Language Environment-batch program model. In general, the CICS Subsystem itself is not implemented as a Language Environment-enabled application. However, it is presented here in the form of a Language Environment-enabled application to illustrate the relationship of the two program models.

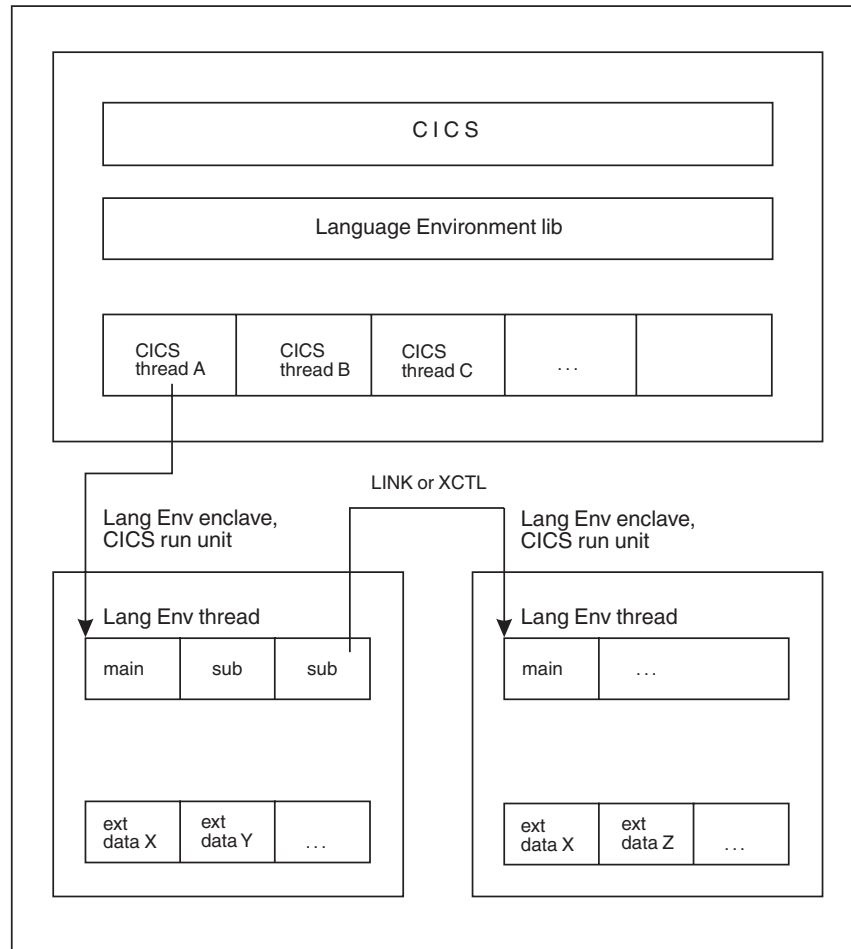


Figure 90. Language Environment-CICS and Language Environment-batch program model

Note:

1. Language Environment Process in the Language Environment-batch program model is the same as the CICS Partition in the Language Environment-CICS program model – an address space that consists of at least one enclave (CICS run unit), a collection of code and data. Unlike the Language Environment process, a character string argument (containing the runtime options) cannot be passed to the CICS partition. Also, default overrides cannot be passed to the CICS partition.
2. As in Language Environment process initialization, the anchor vector is set up at CICS partition initialization.
3. The Language Environment-batch program model does not provide an equivalent term to the CICS Thread in the Language Environment-CICS program model. The sequence of currently active CICS run units (Language Environment enclaves) for a single transaction is called a CICS Thread.
4. The CICS run unit in the Language Environment-CICS program model is an equivalent of the Language Environment Enclave in the Language Environment-batch program model.
The CICS run unit initialization in the Language Environment-CICS program model is same as the Language Environment Enclave creation in the Language Environment-batch program model.
5. Unlike the Language Environment-batch program model, the Language Environment-CICS program model only supports a single Language

Environment thread within an enclave (CICS run unit). Multiple Language Environment enclaves within a process (CICS partition) are supported.

Transfer of control within an enclave in the Language Environment-CICS program model is only accomplished with the CALL statement. Unlike the Language Environment-batch program model, transfer of control within an enclave with Language Environment thread creation service is not supported in the Language Environment-CICS program model.

The Language Environment-CICS program model does not support the multitask function of the Language Environment-batch program model. It supports the multithread function of CICS.

6. As in the Language Environment-batch program model, one enclave can transfer control to another enclave in the Language Environment-CICS program model. This happens as a result of running EXEC CICS LINK or EXEC CICS XCTL from the CICS run unit within the enclave.
7. In the Language Environment-CICS program model, abend propagation continues to be allowed only if the enclave (run unit) was created using the EXEC CICS LINK command.
8. All other definitions in the Language Environment-batch program model apply to the Language Environment-CICS program model.

Language Environment-CICS interface

Language Environment provides an environment (Language Environment-CICS interface) that supports application programs (transactions) written in HLLs under CICS. The Language Environment-CICS interface routine (CEECCICS) uses the API and ERTLI protocols provided by CICS extensively. In summary, the interface between Language Environment and CICS is accomplished through:

1. A CALL interface (ERTLI) from CICS to Language Environment with the Language Environment-CICS interface control routine CEECCICS.
2. EXEC CICS commands issued by Language Environment routines where appropriate to request CICS system services.

Languages supported

The primary languages that CICS provides specific command translators for, and which are supported within Language Environment-CICS environment include: COBOL, PL/I, C, C++, and assembler.

Language Environment provides the following support for member-language libraries and application programs (transactions) running under CICS:

- Language Environment-CICS interface control module
- Language Environment partition initialization/termination
- Language Environment thread initialization/termination
- Language Environment run unit (program) initialization/invocation/termination
- Program management
- Storage management
- Exception handling
- Message services
- Dump services
- Enclaves

Extended runtime language interface

The extended runtime language interface protocol is a set of special calls made from CICS to the Language Environment-CICS interface control module CEECCICS. Language Environment library routines call CICS services with the API protocol (command level interface). CICS makes the following runtime language interface calls to the Language Environment-CICS interface routine:

Interface call	Description
"Partition initialization (Language Environment enablement)" on page 441	Made for Language Environment enablement during CICS initialization
"Partition termination (Language Environment disablement)" on page 444	Made for Language Environment disablement during CICS shut-down.
"Establish ownership type call" on page 445	Made to identify the language of the application program. A Language Environment-enabled application program can be written in any Language Environment-enabled language.
"Thread initialization" on page 449	Made for thread (transaction) initialization.
"Thread termination" on page 451	Made for thread (transaction) termination
"Run unit (program) initialization" on page 451	Made for run unit (program) initialization.
"Run unit (program) termination" on page 456	Made for run unit (program) termination.
"Run unit (program) begin invocation" on page 457	Made to begin run unit (program) invocation.
"Run unit (program) end invocation" on page 459	Made to end run unit (program) invocation.
"Error recovery" on page 467	Enables the Language Environment-CICS interface routine to perform error recovery processing, if possible.
"Determine working storage and static storage" on page 468	Made for determining program working storage address to display the storage during program debugging using CICS EDF.
"Perform GOTO call" on page 469	Made for transferring control to a condition label specified by the EXEC CICS HANDLE CONDITION condition (label), or EXEC CICS HANDLE AID option (label), or EXEC CICS HANDLE ABEND LABEL (label) commands in the program.

ERTLI general call syntax

The following general syntax is used to describe each of the CICS calls to the Language Environment-CICS interface routine (CEECCICS). Note that this syntax is different than the Language Environment callable services syntax described elsewhere in this book.

Syntax

Call CEECCICS (*function*, *rsncode*, *args...*) **Retcode** (*rc*)

function

A fullword integer (a binary value) function code describing the function to be performed.

rsncode (output)

A fullword integer that contains the Language Environment or member language-specific reason code when the function is not performed successfully (*rc* = 16). CICS issues a message (to the operator's console) quoting this reason code returned by Language Environment and abends the task. The reason code format is *nnmffrr*:

nnm 3-digit member ID, 000-127 for IBM and 128-255 for non-IBM products (001 for Language Environment and 005 for COBOL).

ff 2-digit function code (10 for partition initialization).

rr 2-digit unique reason code within a function.

args...

Additional arguments based on the function being performed.

rc (output)

A fullword integer that contains the return code that is passed back in R15. It can contain one of the following values:

0 Function was performed successfully.

16 Function was not performed successfully. The *rsncode* parameter contains a code that describes the reason for the failure.

ERTLI conventions

To avoid confusion, the following conventions are used in describing the extended runtime language interface:

- SYSEIB translator option:

The Language Environment library routines that use the EXEC CICS commands must be translated with the SYSEIB translator option. As a result, a second EIB called system EIB (different from the user EIB) is used to contain information regarding the commands issued by Language Environment routines. There is no need to save and restore the user EIB around the commands issued by Language Environment routines. A routine translated with the SYSEIB option must:

- Run in AMODE(31) as the system EIB is located above 16M.
- Obtain the address of the system EIB using the EXEC CICS ADDRESS EIB command. This command returns system EIB address only if the routine is translated with SYSEIB option. Otherwise, it returns the user EIB address. Notice that for the Language Environment-CICS interface routine, CICS passes the system EIB address in the argument list of the ERTLI calls.
- Be aware that use of the SYSEIB translator option implies use of the NOHANDLE option on the commands.
- Copy all relevant fields from the system EIB to program instance local storage as soon as possible (this is because there is only one system EIB); for example, before running:
 - Another EXEC CICS command
 - A native language call or function or service that can run one or more EXEC CICS commands

- Register usage:

With all the above calls made by CICS to the Language Environment-CICS interface routine or made by the Language Environment-CICS interface routine to the member language-specific interface routines, the following standards apply:

R1 Address of the stored argument list

R13 Address of register save area

R14 Return address

R15 Entry point address on entry (for example, CEECCICS)

R15 Return code on return

- Argument list:

The argument list is shown in Figure 91.

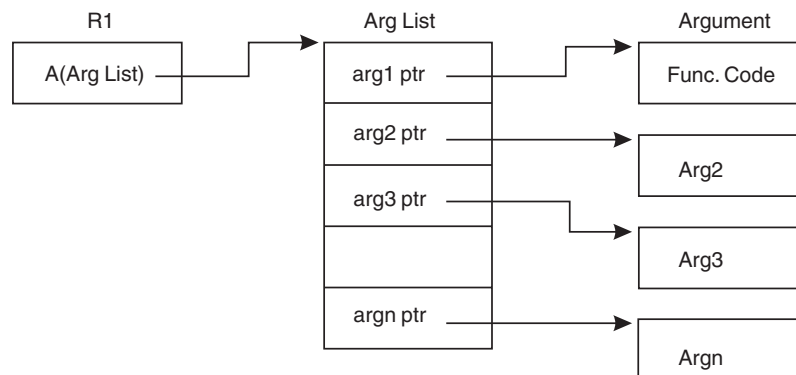


Figure 91. CICS call argument list

- Each item in the argument list passed by CICS is the address of the corresponding argument. The last argument does not have the high-order bit on for calls to member-specific ERTLI.
- The first argument is always a pointer pointing to a fullword binary value which identifies the function to be performed. The meaning of the remaining arguments depends on this function code. The actual argument list for each of the above ERTLI calls are described later in this document.
- When the n th argument is described as THING, it means that the n th word of the stored argument list is the address of THING (n starts at 1 for the first argument of the call).
- Where the callee (Language Environment-CICS interface control routine) sets a value in THING, THING is called a RECEIVER (output) argument. Notice that the special calls provided for COBOL to call CICS services have been replaced by receiver arguments on the CICS to Language Environment calls (ERTLI).
- Language Environment adopts storage addresses as environment tokens. Partition, thread, and run unit tokens are doublewords. The first word is zero and the second word is a 31-bit address.
- All addresses in the interfaces are assumed to be 31-bit addresses. Where a described address is the address of a routine, the top bit is set to indicate the addressing mode in which the routine is called. The bit is set ON to indicate AMODE(31). This mode is set by the caller before passing control to the routine.
- The program (run unit) entry point address passed by CICS is a doubleword entity. The first word is the actual entry point address and the second word is zero.
- If the Language Environment-CICS interface routines change the program mask, it must be restored before control is returned to CICS.
- On return from these calls, CICS expects a zero value in R15 for a successful call; for an abnormal return, R15 should contain a nonzero value (16). CICS issues a message quoting the reason code returned by Language Environment in the argument list of these calls. The possible reason codes are described under each call discussion.

Flowchart of activities

Figure 92 provides an overview of processing for Language Environment-CICS run unit initialization, invocation, and termination.

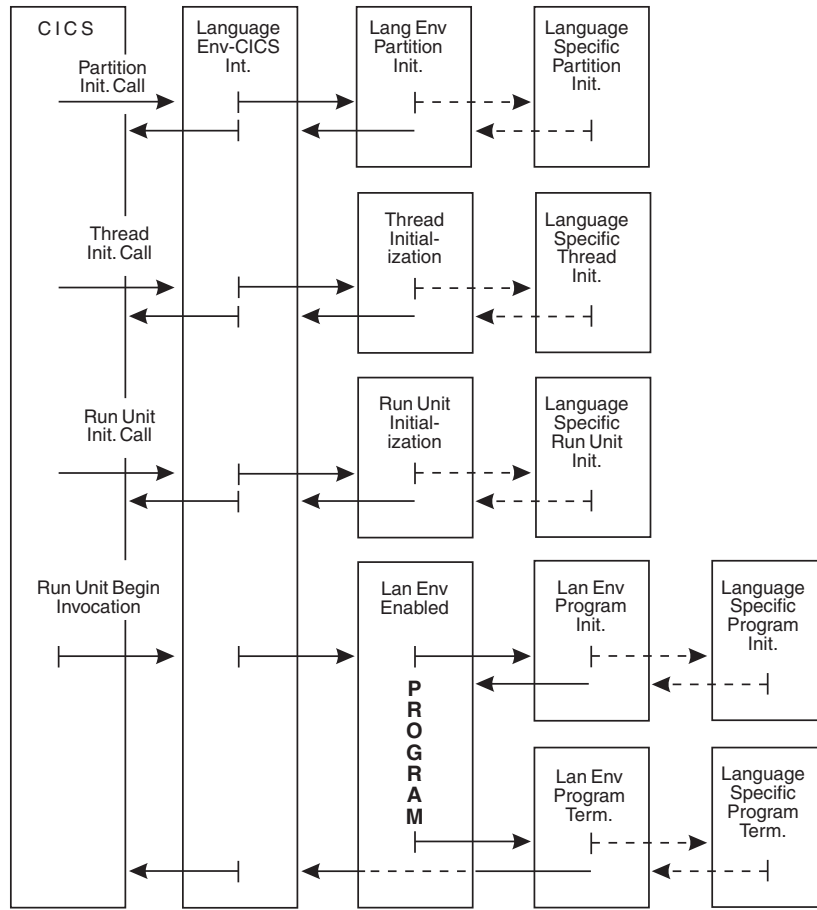


Figure 92. Language Environment-CICS run unit initialization, invocation, and termination

Similarly, Figure 93 on page 441 shows an overview of processing for Language Environment-CICS run unit termination.

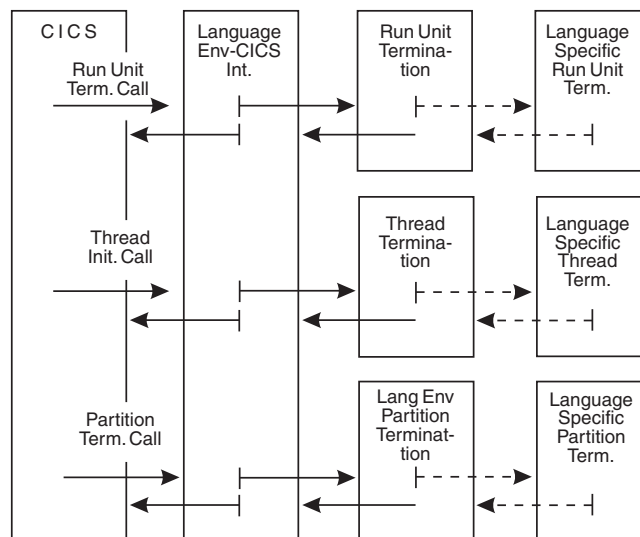


Figure 93. Language Environment-CICS run unit termination

Language Environment-CICS interface routines' DSA

To ensure reentrancy of the Language Environment-CICS interface routines and the language-specific Language Environment-CICS interface routines, a Dynamic Storage Area (DSA) is acquired at partition and thread initialization to be used for stack frames. Run unit initialization routines use the DSA acquired at thread initialization. This temporary stack mechanism is modeled after the Language Environment stack mechanism. The length of this stack storage is predetermined and should be greater than the sum of all possible active stacks used by Language Environment-CICS interface routines. The maximum length of the stack is 1024 bytes. If exceeded, an ABEND of 4093 is issued.

The Language Environment stack mechanism provides consistency with batch environment in terms of calling a common set of Language Environment routines such as option processing, storage management, and exception handling.

Partition initialization (Language Environment enablement)

Enablement of Language Environment is performed by a partition initialization call to the Language Environment-CICS interface module during CICS initialization. CICS loads (OS conditional load) and calls the Language Environment-CICS interface module (CEECCICS) during the system initialization. CEECCICS must be in an authorized data set for the OS LOAD to be successful.

If CICS can not load the Language Environment-CICS interface module no message is issued and processing continues without Language Environment. If the Language Environment partition initialization fails, CICS outputs a message quoting the reason code returned by Language Environment and continue processing as if Language Environment is not present.

The Language Environment-CICS interface module communicates with CICS with ERTLI and EXEC CICS commands and with Language Environment with standard CALLs to enable and maintain the Language Environment environment. Another similar interaction disables the Language Environment environment in response to CICS shut-down processing.

Partition Initialization

The Language Environment-CICS interface module remains in storage to support communication between CICS and Language Environment as long as Language Environment is enabled.

Syntax

Call CEECCICS (10, *rsncode*, *syseib*, *preasa*, *ptoken*, *eiblen*, *twalen*, *cellevel*, *getcaa*, *setcaa*, *partinit_flags*, *langavl*) **Retcode** (*rc*)

rsncode (output)

A fullword integer in *nnmffrr* format to contain the member language-specific partition initialization reason code or one of the following Language Environment reason codes:

- 11000 Invalid parameter was passed
- 11010 Storage was not available
- 11020 The library was not loaded
- 11030 Language-specific partition initialization was not done

syseib

The system EXEC interface block, as defined by CICS. This control block contains information about running the CICS commands issued by Language Environment. There is no need to save and restore the user EIB around the commands issued by Language Environment. The system EIB address is above 16M. For the information contained in the EIB, see .

preasa

A preallocated save area to be used by Language Environment to issue its first EXEC CICS GETMAIN command. The size of this save area is same as the size of DFHEISTG (248 bytes).

ptoken (output)

A doubleword value to contain a token representing the Language Environment partition environment.

eiblen

A fullword integer containing the system EIB length.

twalen (output)

A fullword integer that contains the length of the preallocated thread work area. This work area is allocated by CICS from the user (task local) RMODE(ANY) storage for each thread, and passed to the Language Environment-CICS interface at thread initialization. The work area address is above 16M. You can return a length equal to 0. In this case, you must acquire the thread work area (if one is required) during thread initialization using EXEC CICS GETMAIN command.

cellevel (output)

A fullword integer that contains the Language Environment-CICS interface level.

getcaa

The CICS specific GET_CAA routine address. This routine returns the CAA address of the current run unit. It runs in AMODE=ANY,RMODE=24.

setcaa

The CICS specific SET_CAA routine address. This routine is called to set the current run unit's CAA address in a CICS control block for a later retrieval with GET_CAA routine. It runs in AMODE=ANY,RMODE=24.

partinit_flags (input/output) * Formerly 'langdef' *

A 32 byte flag field used to communicate interface information between CICS and Language Environment. The structure is depicted in Figure 94.

```

DCL 1 PARTINIT_FLAGS,                /* CICS/Lang Env interface flags */
                                     /* */
      2 CICS_FLAGS BIT(128), /*+00 CICS interface level flags */
      3 CICS_PROG_OBJ BIT(1), /* .00 CICS supports program objs */
      3 Reserved BIT(1), /* .01 Reserved for CICS */
      3 CICS_OTE_PHASE1 BIT(1), /* .02 CICS is at Phase I of OTE */
      3 CICS_REUSE_RWA BIT(1), /* .03 CICS supports reusable RWAs */
      3 CICS_OTE_PHASE2 BIT(1), /* .04 CICS is at Phase II of OTE */
      3 CICS_LDMDNAME BIT(1), /* .05 CICS provides program name */
      3 CICS_AUTOTUNE BIT(1), /* .06 CICS supports automatic storage*/
                                     /* tuning. */
      3 CICS_AUTOTUNE_SET BIT(1), /* .07 CICS indicates automatic stg */
                                     /* tuning should be done. This bit*/
                                     /* valid only if AUTOTUNE is ON. */

      3 CICS_RE BIT(1), /* .08 CICS supports reusable enclaves*/
      3 Reserved BIT(1), /* .09 Reserved for CICS */
      3 CICS_TRAN BIT(1), /* .10 CICS indicates transaction dump*/
                                     /* service routines are available.*/

      3 * BIT(1), /* .11 Reserved */
      3 CICS_DBGINFO BIT(1), /* .12 CICS indicates it can pass */
                                     /* A(Debug info blk) in pgminfol */

      3 * BIT(1), /* .13 Reserved */
      3 CICS_EXT_REG BIT(1), /* .14 CICS supports extended register*/
                                     /* interface in Run Unit (program)*/
                                     /* End Invocation TERMINFO area */

      3 Reserved BIT(113), /* Reserved for future use */
                                     /* */

      2 LE_FLAGS BIT(128), /*+10 Lang Env interface level flags */
      3 LE_PROG_OBJ BIT(1), /* .00 Lang Env supports program objs */
      3 Reserved BIT(1), /* .01 Reserved for CICS */
      3 LE_OTE_PHASE1 BIT(1), /* .02 Lang Env is at Phase I of OTE */
      3 LE_REUSE_RWA BIT(1), /* .03 Lang Env supports reusable RWAs*/
      3 LE_OTE_PHASE2 BIT(1), /* .04 Lang Env is at Phase II of OTE */
      3 LE_LDMDNAME BIT(1), /* .05 Set by Lang Env. Indicates that*/
                                     /* needs the address of the load */
                                     /* module name in pgminfol (for */
                                     /* storage tuning exit). */

      3 LE_AUTOTUNE BIT(1), /* .06 Lang Env supports automatic */
                                     /* storage tuning. */
      3 LE_AUTOTUNE_SET BIT(1), /* .07 Lang Env indicates automatic */
                                     /* storage tuning will be done. */

      3 LE_RE BIT(1), /* .08 Lang Env supports reus enclaves*/
      3 Reserved BIT(1), /* .09 Reserved for CICS */
      3 LE_TRAN BIT(1), /* .10 Lang Env indicates the trans */
                                     /* dump service routines required */
                                     /* available. */

      3 * BIT(1), /* .11 Reserved */
      3 LE_DBGINFO BIT(1), /* .12 Lang Env indicates it supports */
                                     /* copy A(Debug info blk) to PCB */

      3 * BIT(1), /* .13 Reserved */
      3 LE_EXT_REG BIT(1), /* .14 Lang Env supports extended */
                                     /* register interface in Run Unit */
                                     /* (program) End Invocation */
                                     /* TERMINFO area */

      3 Reserved BIT(113); /* Reserved for future use */
                                     /* */

```

Figure 94. Structure of interface flags field

langavl (output)

A fullword binary value to contain bit settings for the languages that provide a Language Environment-CICS member event handler (CEEEVnnn) capable of handling both the existing (non-Language Environment-enabled) application

Partition Initialization

programs as well as the new (Language Environment-enabled) application programs running in CICS environment. CICS uses this information to decide whether to interface with Language Environment or continue to interface with the languages directly as before (prior to Language Environment). Bit definitions are:

```
Value of Bit0 (far left bit):
  1=Assembler member event handler (CEEEV015) is available
  0=Assembler member event handler (CEEEV015) is not available
Value of Bit1
  1=C or C++ member event handler (CEEEV003) is available
  0=C or C++ member event handler (CEEEV003) is not available
Value of Bit2
  1=COBOL member event handler (CEEEV005) is available
  0=COBOL member event handler (CEEEV005) is not available
Value of Bit3
  1=PL/I member event handler (CEEEV010) is available
  0=PL/I member event handler (CEEEV010) is not available
```

Usage Notes:

1. Bit 3 (fourth bit) of the CICS_FLAGS parameter on partition_initialization will be set by CICS to indicate that CICS supports reusable run unit work areas (RRWA). This bit maps to an existing structure in the RCB. It will be reserved and called 'CEERCB_CICS_RRWA_OK'. This bit can then be tested by Language Environment or its members to determine if this environment is supported.
2. Bit 5 (sixth bit) of CICS_FLAGS parameter on partition_initialization will be set by CICS to indicate that CICS will provide the address of the load module name in *pgminfo1*.
If the CICS support to provide the program name is not available, the address of the program name passed to the storage tuning user exit in the CEESTX CICS specific control block will be zero.
3. LE_FLAGS bit 3 will be set by CEECPINI (Partition initialization) to signify that Language Environment supports the interface changes including the reusable run unit work area.
4. If CICS is running at a higher level than Language Environment, then Language Environment will run and CICS will descend to match the Language Environment level.
5. If Language Environment detects CICS is running at a lower level than Language Environment, then Language Environment will deactivate all higher level functions in order to match the CICS level.
6. Bit 6 (TUNE_SUP) of the CICS_FLAGS parameter on partition_initialization will be set by CICS to indicate that CICS has the support for automatic storage tuning. When this bit is on:
 - The value in Bit7 (LE_AUTODST) will indicate the setting for system initialization parameter AUTODST.
 - CICS provides a pointer to a 96 byte area in *pgminfo2*.
 - CICS provides support for the new bit in *pgminfo2* that indicates to CICS to update its sizes for RUWA and the 96 byte area at rununit termination.

Partition termination (Language Environment disablement)

Disablement of Language Environment is performed by Partition Termination Call to the Language Environment-CICS interface module during CICS termination. CICS calls the Language Environment-CICS interface module (CEECCICS) for

partition termination during the normal system termination. This occurs after PLTSD (Program List Table of programs to run during Shut-Down) processing completes.

If partition termination fails, CICS outputs a message quoting the reason code returned by Language Environment. CICS does not issue the Partition Termination Call during the following situations:

- Immediate System Termination (CEMT PERFORM SHUTDOWN IMMEDIATE)
- Abnormal System Termination

Member event handlers are called for process termination prior to the Partition Termination Call.

Syntax

Call CEECCICS (*11, rsnocode, syseib, preasa, ptoken*) **Retcode** (*rc*)

rsnocode **(output)**

A fullword integer in *nnnffrr* format to contain the member language-specific partition termination reason code or one of the following Language Environment reason codes:

- 11100 Invalid parameter was passed
- 11110 The library was not released
- 11120 Storage was not freed
- 11130 Language-specific partition termination was not done

syseib

The system EXEC interface block, as defined by CICS. Its address is above 16M.

preasa

A preallocated save area supplied by CICS. The size of this save area is 248 (DFHEISTG length) bytes.

ptoken

A doubleword value containing the Language Environment partition token passed back to CICS at partition initialization.

Establish ownership type call

When CICS loads any program, the CICS Program Control Program calls Language Environment to determine if Language Environment is managing the program. It then looks up the program name in its Processing Program Table (DFHPPT) to get its language type, determines whether it is in storage, and if it is, where its entry point is. If the program is not in storage, it is loaded into storage and its entry point address is placed in its PPT entry.

If the program is, for example, a COBOL program, CICS needs to know which language library (OS/VS COBOL, VS COBOL II, or Language Environment) to interface with. For Language Environment-enabled programs, the PPT's LANG parameter is not required (or LANG=NOTAPPLIC). Language Environment-enabled programs are identifiable through the Language Environment eye catcher at their entry point and through the information provided in Program Prolog Areas PPA1 and PPA2. For more information on requirements of being a Language Environment-enabled program, see "Routine layout" on page 6.

Establish Ownership

CICS discovers the language type and the run unit work area length (if desired to be preallocated) with "Establish Ownership Type Call" to the Language Environment-CICS interface routine the first time it loads a program. This call is made after partition initialization and prior to Thread and/or run unit initialization call and is subject to certain rules:

- The call for programs defined as LANG=NOTAPPLIC is issued if the Language Environment-CICS interface is enabled. The routine CEECCICS must have been loaded by CICS and Language Environment partition initialization must have been completed successfully by Language Environment.
- The call for programs defined as LANG=C or COBOL or PL/I is issued if the Language Environment-CICS interface is enabled for those languages.

Syntax

Call CEECCICS (50, *rsncode*, *syseib*, *preasa*, *ptoken*, *reserved1*, *reserved2*, *pgminfo1*, *pgminfo2*) **Retcode** (*rc*)

rsncode (output)

A fullword integer in *nnmffrr* format to contain the member language-specific establish ownership reason code or one of the following Language Environment reason codes:

- 15000** Invalid parameter was passed
- 15020** Program ownership type and/or run unit work area length was not established
- 15030** Language-specific establish ownership failed
- 15060** The application provided a program object that cannot be supported with the current level of CICS.

syseib

The system EXEC interface block, as defined by CICS. Its address is above 16M.

preasa

A preallocated save area (above 16M) supplied by CICS. The size of this save area is 248 (DFHEISTG length) bytes.

ptoken

A doubleword value containing the Language Environment partition token passed back to CICS at partition initialization.

reserved1

A reserved argument; it is neither referred to, nor set by, Language Environment.

reserved2

A reserved argument; it is neither referred to, nor set by, Language Environment.

pgminfo1

The following structure of information supplied by CICS to Language Environment, as shown in Figure 95 on page 447.

```

DCL 1 PGMINFO1,                /* Data from CICS to Lang Env */
  2 STRUC_LENGTH  FIXED BIN(31), /*+00 pgminfo1 structure length */
  2 RULANG,                /*+04 Run unit's "main" program
                        language defined in PPT as */
    3 ASSEMBLER  BIT(1),      /*+04.0 LANG=ASSEMBLER */
    3 C370       BIT(1),      /*+04.1 LANG=C/370 */
    3 COBOL      BIT(1),      /*+04.2 LANG=COBOL II */
    3 PLI        BIT(1),      /*+04.3 LANG=PL/I */
    3 RPG        BIT(1),      /*+04.4 LANG=RPG */
    3 NOTAPPLIC BIT(1),      /*+04.5 LANG=NOTAPPLIC or blank */
    3 *          BIT(2),      /* Reserved */
  2 FLAGS,                /*+05 Additional Flags */
    3 OPEN_PROGRAM BIT(1),    /*+05.0 1 = Program runs only on
                        OTE TCB and can use
                        Open C functions
                        0 = Program may run on
                        OTE or QR TCB */
    3 *          BIT(7),      /* Reserved */
  2 RULOADMOD,
    3 RULOAD      POINTER,     /*+08 Run unit load module addr */
    3 RULOADL    FIXED BIN(31), /*+0C Run unit load module length */
  2 ENTRY_STATIC,
    3 RUENTRY     POINTER,     /*+10 Run unit Entry Point Addr */
    3 RUSTATIC    POINTER,     /*+14 Run unit Static Address */
  2 PREARWA_31    POINTER,     /*+18 Preallocated run unit work
                        /* area above 16Meg */
  2 PREARWA_24    POINTER,     /*+1C Preallocated run unit work
                        /* area below 16Meg */
  2 APAL          POINTER,     /*+20 Application Pgm Arg List */
  2 RTOPTS        POINTER,     /*+24 Runtime options string
                        /* specified during debugging */
  2 RTOPTSL       FIXED BIN(31), /*+28 Runtime opts string length */
  2 RULOAD_NAMEA  POINTER,     /*+2C Address of the run unit
                        /* load module name */
  2 *             POINTER,     /*+30 Reserved */
  2 RUDEBUGA      POINTER,     /*+34 Address of the run unit
                        /* Debug Info Block */

```

Figure 95. Structure of information supplied to CICS by Language Environment for PGMINFO1

struc_length

A fullword integer value containing the *pgminfo1* structure length.

rulang

A fullword binary value indicating the language type of the run unit (main program) as defined in PPT. In PPT, LANG=NOTAPPLIC can be used for Language Environment-enabled application programs. For bit definitions, see Figure 95.

ruloada

A fullword value containing the load module address of the run unit. This address in conjunction with the entry point address is used to access run unit prolog information (PPA1, PPA2) when necessary.

ruloadl

A fullword value containing the load module length of the run unit. This value is used to validate the addresses accessed through the run unit prolog information (PPA1, PPA2) when necessary.

ruentry

A fullword value containing the entry point address of the run unit. This address is given control at run unit begin invocation call for Language Environment-enabled programs. The run unit entry is established at link-edit time. The high-order bit indicates the AMODE of the run unit.

Establish Ownership

rustatic

A fullword. This is passed in R0 to the main program at run unit begin invocation call.

ruload_namea

A fullword address that points to the load module name. The load module name is 8-bytes long and is padded with blanks.

rudebuga

A fullword address that points to the run unit debug info block. The debug info block is 8-bytes long and is padded with blanks.

pgminfo2 (output)

The structure of information supplied to CICS by Language Environment, as shown in the code example below.

```
DCL 1 PGMINFO2,                               /* Pgm Info from Lang Env to */
                                           /* CICS                       */
      2 STRUC_LENGTH  FIXED BIN(31),          /* pgminfo2 structure length */
      2 RWALEN_31     FIXED BIN(31),          /* Run unit workarea length  */
                                           /* above 16 megabyte         */
      2 RWALEN_24     FIXED BIN(31),          /* Run unit workarea length  */
                                           /* below 16 megabyte         */
      2 PGMTYPE       BIT(32),                /* Program Type of the "Main" */
      3 CEEENABLE     BIT(2),                 /* Lang Env Enablement       */
                                           /* 11 - Fully Lang Env-      */
                                           /* enabled                    */
                                           /* programs (w/ PPAs)        */
                                           /* 10 - Partially Lang Env-  */
                                           /* enabled programs (old     */
                                           /* or new w/o PPAs)         */
                                           /* 01 - Not Lang Env-enabled */
                                           /* programs                  */
                                           /* 00 - Don't know: programs */
                                           /* which can't be identi-    */
                                           /* fied. eg. OS/VS COBOL    */
      3 MIXED         BIT(1),                 /* Mixed or Single language  */
                                           /* load module               */
                                           /* 1 - Mixed                  */
                                           /* 0 - Single                 */
      3 COMPAT        BIT(1),                 /* Compatibility Requirement  */
                                           /* 1 - Required               */
                                           /* 0 - Not Required          */
      3 EXECUTE       BIT(1),                 /* Program Execution          */
                                           /* 1 - Executable            */
                                           /* 0 - Not Executable        */
      3 ASSEMBLER     BIT(1),                 /* "Main" Program Language   */
                                           /* 1 - Assembler             */
                                           /* 0 - Not Assembler        */
      3 C370          BIT(1),                 /* "Main" Program Language   */
                                           /* 1 - C/370                 */
                                           /* 0 - Not C/370            */
      3 COBOLII       BIT(1),                 /* "Main" Program Language   */
                                           /* 1 - VS COBOL II          */
                                           /* 0 - Not VS COBOL II      */
      3 OSCOBOL       BIT(1),                 /* "Main" Program Language   */
                                           /* 1 - OS/VS COBOL          */
                                           /* 0 - Not OS/VS COBOL     */
      3 PLI           BIT(1),                 /* "Main" Program Language   */
                                           /* 1 - OS PL/I               */
                                           /* 0 - Not OS PL/I          */
      3 UPDATE_PGMINFO2 BIT(1),              /* Output parameter on rununit */
                                           /* termination call. ON= tells */
                                           /* CICS to update its control  */
                                           /* blocks with the following  */
                                           /* fields in PGMINFO2:       */
                                           /* - RWALEN_31               */
                                           /* - RWA;EM_24               */
                                           /* - STG_TUNE_AREA           */
                                           /* Reserved                  */
      3 *             BIT(21),                /* Reserved                   */
      2 * CHAR(4),    /* additional CEL defined info */
      3 EPTYPE        FIXED(8),              /* type of module entry point: */
```

```

/* 0 - old          */
/* 1 - ppal        */
/* 2 - ceestart    */
/* 3 - ppal w v1r2 ceestart */
/* 4 - v1r2 ceestart */
3 NEEDOPTP      BIT(1), /* Member language of main */
/* needs to be called for */
/* option processing event */
/* 31 bit        */
3 PGM_ALL31_ON  BIT(1), /* Load module is eligible for */
3 STX_LDMOD_ELIG BIT(1), /* the storage tuning user */
/* exit.        */
/* ON = enable the storage */
/* tuning user exit for this */
/* load module. */
/* OFF = disable the storage */
/* tuning user exit for this */
/* load module. */
3 *             BIT(13), /*
3 MEMID         FIXED(8), /* Member id of the language */
/* of the "main" program */
2 *             CHAR(8)  /* Est Ownership return fields */
3 dopt_ptr      POINTER, /* pointer to default OCB */
3 uopt_ptr      POINTER, /* pointer to user OCB */
2 AUTOTUNE_AREA@ POINTER, /* pointer to a 96 byte area */
/* for Lang Env to remember */
/* storage tuning values. */

```

struc_length

A fullword integer value that contains the *pgminfo2* structure length.

rwalen_31 (output)

A fullword value to contain the above 16M run unit work area length. This work area is preallocated by CICS and passed to Language Environment-CICS interface routine at run unit initialization call. The run unit work area length is the sum of the run unit work areas required by Language Environment and member languages.

rwalen_24 (output)

A fullword value to contain the below 16M run unit work area length. This work area is preallocated by CICS and passed to Language Environment-CICS interface routine at run unit initialization call. The run unit work area length is the sum of the run unit work areas required by Language Environment and member languages.

pgmtype (output)

A fullword value to contain information regarding the main program of the run unit:

- If the main program is Language Environment enablement.
- If the run unit is a single or mixed language load module.
- If compatibility is required at invocation.

memid (output)

A fullword value to contain the member ID of the language of the main program of the run unit. CICS can provide this information back to the system programmer if desired.

Thread initialization

Processing a transaction can involve a single run unit or several run units in the same or different languages. The sequence of currently active run units for a single transaction is called a CICS thread. Thread initialization is performed when the first transaction run unit is encountered in a thread, and CICS calls the Language

Thread Initialization

Environment-CICS interface module to request it. This request is made through the Thread Initialization Call. If the thread cannot be initialized then CICS abends the task.

Syntax

Call CEECCICS (20, *rsncode*, *syseib*, *preasa*, *ptoken*, *ttoken*, *preatwa*, *pgminfo1*, *pgminfo2*, *statusflags*) **Retcode** (*rc*)

rsncode (output)

A fullword integer in *nnnffrr* format to contain the member language-specific thread initialization reason code or one of the following Language Environment reason codes:

12000 Invalid parameter was passed

12020 Thread work area was not preallocated

12030 Language-specific thread initialization was not done

syseib

The system EXEC interface block, as defined by CICS. Its address is above 16M.

preasa

A preallocated save area supplied by CICS. The size of this save area is 248 (DFHEISTG length) bytes. It can be used for issuing CICS commands if necessary.

ptoken

A doubleword value containing the Language Environment partition token passed back to CICS at partition initialization.

ttoken (output)

A doubleword value to contain the Language Environment thread token.

preatwa

The address of the preallocated thread work area. The length of this work area was passed to CICS at Language Environment partition initialization.

pgminfo1

The same structure of information supplied by CICS to Language Environment in an establish ownership type call; see Figure 95 on page 447.

pgminfo2

The same structure of information supplied by CICS to Language Environment in an establish ownership type call; see *pgminfo2*.

statusflags

The same structure of information supplied by CICS to Language Environment, as shown in Figure 96.

```
1 CEECICS_STATUS_FLAGS    BIT(32) BASED(CEECICS_ARG11PTR),
3 CEECICS_CICS_AP_TRACE,    /* State of AP trace */
5 CEECICS_TRACE_LEVEL1 BIT(1), /* Level 1 trace is requested */
5 CEECICS_TRACE_LEVEL2 BIT(1), /* Level 2 trace is requested */
```

Figure 96. Structure of information supplied to CICS by Language Environment for STATUSFLAGS

If the initial program in a transaction is a reusable enclave, then the CICS thread (Language Environment process) is marked as a Language Environment/CICS reusable process.

Thread termination

Thread termination occurs when a transaction is completed or is being terminated. This is done through the thread termination call from CICS to the Language Environment-CICS interface module. All thread level clean-up, such as freeing the acquired storage, occurs at this stage. Language Environment resets the thread token back to 0 at completion of the thread termination. The language-specific thread termination also occurs at this stage.

Usage Notes:

1. CICS issues the thread termination call if, and only if, a nonzero thread token has been returned to CICS by Language Environment at thread initialization. The same applies to the member languages.
2. Language Environment processing assumes that there will never be a scenario needed where a process might 'switch' to non-reusable after being initialized reusable.

Syntax

Call CEECCICS (21, *rsncode*, *syseib*, *preasa*, *ptoken*, *ttoken*) **Retcode** (*rc*)

rsncode (output)

A fullword integer in *nnmffrr* format to contain the member language-specific thread termination reason code or one of the following Language Environment reason codes:

- 12100 Invalid parameter was passed
- 12110 Active run unit(s) were detected
- 12130 Language-specific thread termination was not done

syseib

The system EXEC interface block, as defined by CICS. Its address is above 16M.

preasa

A preallocated save area supplied by CICS. The size of this save area is 248 bytes. It can be used for issuing CICS commands, such as FREEMAIN, if necessary.

ptoken

A doubleword value containing the Language Environment partition token passed back to CICS at partition initialization.

ttoken (input/output)

A doubleword value containing the Language Environment thread token passed back to CICS at thread initialization.

Run unit (program) initialization

The next stage prior to run unit (program) invocation is the run unit initialization. This is requested by CICS through the run unit initialization call after loading the run unit (Language Environment-enabled PPT module) and prior to control being passed to it. Run unit load point, entry point, parameter list as well as the

Run Unit Init

Language Environment thread token are provided in this call. If the run unit can not be initialized, CICS abends the task. Run unit initialization occurs at:

- CICS task attachment
- EXEC CICS LINK or EXEC CICS XCTL commands

Syntax

Call CEECCICS (30, *rsncode*, *syseib*, *preasa*, *ptoken*, *ttoken*, *rtoken*, *pgminfo1*, *pgminfo2*, *ioinfo*, *runinfo*, *retoken*) **Retcode** (*rc*)

rsncode (output)

A fullword integer in *nnmffrr* format to contain one of the following Language Environment reason codes:

- 13000** Invalid parameter was passed
- 13010** No run unit work area was passed
- 13040** No application program argument list was passed
- 13200** Invalid parameter was passed when the direct invoke bit is set
- 13210** No run unit work area was passed when the direct invoke bit is set
- 13230** Language-specific run unit invocation failed when the direct invoke bit is set
- 13240** No application program argument list was passed when the direct invoke bit is set

syseib

The system EXEC interface block, as defined by CICS. This control block contains information about running the CICS commands issued by Language Environment. There is no need to save and restore the user EIB around the commands issued by Language Environment. The system EIB address is above 16M. For the information contained in the EIB, see .

Note: The user EIB, which is different than the system EIB, is passed to the application program in the application program argument list.

preasa

A preallocated save area passed to Language Environment by CICS. The size of this save area is 248 bytes.

ptoken

A doubleword value containing the Language Environment partition token established at partition initialization.

ttoken

A doubleword value containing the Language Environment thread token established at thread initialization.

rtoken (output)

A doubleword value to contain the Language Environment run unit token.

pgminfo1

The structure of information supplied by CICS to Language Environment is shown in Figure 97 on page 453.

```

DCL 1 PGMINFO1,          /* Data from CICS to Lang Env */
  2 STRUC_LENGTH  FIXED BIN(31), /*+00 pgminfo1 structure length */
  2 RULANG,          /*+04 Run unit's "main" program
                    language defined in PPT as */
    3 ASSEMBLER  BIT(1),      /*+04.0 LANG=ASSEMBLER */
    3 C370      BIT(1),      /*+04.1 LANG=C/370 */
    3 COBOL     BIT(1),      /*+04.2 LANG=COBOL II */
    3 PLI       BIT(1),      /*+04.3 LANG=PL/I */
    3 RPG       BIT(1),      /*+04.4 LANG=RPG */
    3 NOTAPPLIC BIT(1),      /*+04.5 LANG=NOTAPPLIC or blank */
    3 *         BIT(2),      /* Reserved */
  2 FLAGS,          /*+05 Additional Flags */
    3 OPEN_PROGRAM BIT(1),    /*+05.0 1 = Program runs only on
                    OTE TCB and can use
                    Open C functions
                    0 = Program may run on
                    OTE or QR TCB */
    3 *         BIT(7),      /* Reserved */
  2 RULOADMOD,
    3 RULOADA    POINTER,     /*+08 Run unit load module addr */
    3 RULOADL    FIXED BIN(31), /*+0C Run unit load module length */
  2 ENTRY_STATIC,
    3 RUENTRY    POINTER,     /*+10 Run unit Entry Point Addr */
    3 RUSTATIC   POINTER,     /*+14 Run unit Static Address */
  2 PREARWA_31   POINTER,     /*+18 Preallocated run unit work
                    /* area above 16Meg */
  2 PREARWA_24   POINTER,     /*+1C Preallocated run unit work
                    /* area below 16Meg */
  2 APAL         POINTER,     /*+20 Application Pgm Arg List */
  2 RTOPTS       POINTER,     /*+24 Runtime options string
                    /* specified during debugging */
  2 RTOPTSL      FIXED BIN(31), /*+28 Runtime opts string length */
  2 RULOAD_NAMEA POINTER,     /*+2C Address of the run unit
                    /* load module name */
  2 *           POINTER,     /*+30 Reserved */
  2 RUDEBUGA     POINTER,     /*+34 Address of the run unit
                    /* Debug Info Block */

```

Figure 97. Structure of information supplied to Language Environment by CICS

struc_length

A fullword value containing the *pgminfo1* structure length.

rulang

A fullword binary value indicating the language type of the run unit (main program) as defined in PPT. In PPT, LANG=NOTAPPLIC can be used for Language Environment-enabled application programs. For bit definitions, see above.

ruloada

A fullword value containing the load address of the run unit. This address in conjunction with the entry point address is used to access run unit prolog information (PPA1, PPA2) when necessary.

ruloadl

A fullword value containing the load module length of the run unit. This value is used to validate the addresses accessed through the run unit prolog information (PPA1, PPA2) when necessary.

ruentry

A fullword value containing the entry point address of the run unit. This address is given control at the run unit begin invocation call for Language Environment-enabled programs. The run unit entry is established at link-edit time. The high-order bit indicates the AMODE of the run unit.

Run Unit Init

rustatic

A full word passed in R0 to the main program at the run unit begin invocation call.

prearwa_31

A fullword value containing the address of the preallocated run unit work area above 16M. The length of this preallocated work area was passed back to CICS at establish ownership type call prior to run unit initialization call. This work area can be initialized to contain CAA, HEAP, ISA and other Language Environment control blocks.

prearwa_24

A fullword value containing the address of the preallocated run unit work area below 16M. The length of this preallocated work area was passed back to CICS at establish ownership type call prior to run unit initialization call. This work area can be initialized to contain control blocks that have to be below 16M.

apal

A fullword value containing the address of the application program argument list. This argument list contains the address of the user EIB and the address of a COMMAREA.

rtopts

A fullword value containing the address of the runtime options string passed to CICS during debugging with CICS EXEC Debugging Facility (EDF). For example, CEDF terminal-ID,ON,'runtime options'.

rtopts1

A fullword value containing the length of the runtime options string.

When EDF is invoked in the following fashion:

```
CEDF term,ON,INSPECT
```

a special character string is passed to Language Environment during run unit initialization in the *rtopts* parameter. The following string is passed:

```
TEST,TERM=xxxx
```

where xxxx is the terminal identifier for the terminal where debugging information should be communicated. (This can either be information for a 3270-type terminal or communication to/from a workstation.) Language Environment detects this string, and internally initializes as if the options string TEST was passed. Also, Language Environment passes the terminal identifier to Debug Tool as a new, fifth parameter of the external entries debugger event.

rload_namea

A fullword address that points to the load module name. The load module name is 8-bytes long and is padded with blanks.

rdebuga

A fullword address that points to the run unit debug info block. The debug info block is 8-bytes long and is padded with blanks.

pgminfo2

The structure of information described at establish ownership type call.

ioinfo (output)

A structure that contains the standard input/output and error file information (transient data queue names or spool file classes). The structure declaration is

in Figure 98.

```

DCL 1 IOINFO,                                /* I/O Information Structure */
                                        /*                               */
    2 STD_IN,                                /* Standard input file       */
      3 QORS_IN CHAR(1),                    /* - either                  */
                                        /* 'Q', transient data queue */
                                        /* 'S', spoolfile           */
      3 TDQN_IN CHAR(4),                    /* - queue name              */
      3 SPOC_IN CHAR(1),                    /* - spool class              */
                                        /*                               */
    2 STD_OUT,                                /* Standard output file      */
      3 QORS_OUT CHAR(1),                   /* - either                  */
                                        /* 'Q', transient data queue */
                                        /* 'S', spoolfile           */
      3 TDQN_OUT CHAR(4),                   /* - queue name              */
      3 SPOC_OUT CHAR(1),                   /* - spool class              */
                                        /*                               */
    2 STD_ERR,                                /* Standard error file       */
      3 QORS_ERR CHAR(1),                   /* - either                  */
                                        /* 'Q', transient data queue */
                                        /* 'S', spoolfile           */
      3 TDQN_ERR CHAR(4),                   /* - queue name              */
      3 SPOC_ERR CHAR(1),                   /* - spool class              */
                                        /*                               */

```

Figure 98. Structure of standard I/O information provided to Language Environment by CICS

runinfo (input)

Address of a fullword containing information about how the run unit should be initialized and whether it should be immediately invoked after initialization. See Figure 99.

```

DCL 1 RUNINFO BIT(32),                      /* Run Information           */
    2 INVOKE BIT(1),                        /* Call Lan Env Begin Invocation*
                                        /* after initialization?    */
                                        /* 0 - Return after init   */
                                        /* 1 - Call Begin Invocation */
    2 RWA_REUSE BIT(1),                     /* RWA candidate for reuse? */
                                        /* 0 - No                   */
                                        /* 1 - Yes                   */
    2 * BIT(30),                             /* Reserved                  */

```

Figure 99. Run information supplied to Language Environment by CICS

retoken (input)

A doubleword value to contain the run-unit token of the most recently invoked JVM in the chain (or zero if none exists). The presence of a n address tells Language Environment there is a JVM reusable enclave as a parent.

Note: The run unit initialization process operates as follows, resulting in shortened path lengths and improved performance:

1. The INVOKE bit is tested and if on, run unit initialization calls run unit begin invocation directly using the registers/parmlist passed by CICS for the run unit initialization call.
2. The RWA_REUSE bit is tested in run unit initialization and a new bit, CEEEDB_CICS_RUNREUSE, is turned on to indicate that the run work area may be reused.

Run unit (program) termination

After the Language Environment-CICS interface routine returns from the run unit end invocation call to CICS, CICS can drive Language Environment-CICS interface routine with the run unit termination call. Run unit termination call can occur as frequent as the run unit end invocation call or it might only occur when CICS is under stress.

Syntax

Call CEECCICS (31, *rsncode*, *syseib*, *preasa*, *ptoken*, *ttoken*, *rtoken*, *terminfo*) **Retcode** (*rc*)

rsncode (output)

A fullword integer that contains one of the following Language Environment reason codes:

- 13100 Invalid parameter was passed
- 13110 Wrong thread token was passed
- 13140 No PTB was passed

syseib

The system EXEC interface block, as defined by CICS for use by Language Environment and language-specific interface routines. The system EIB address is above 16M. For the information contained in the EIB, see .

preasa

A fullword value containing the preallocated save area address to be used by Language Environment to issue the last EXEC CICS FREEMAIN command. The size of this save area is 248 bytes.

ptoken

A doubleword value containing the Language Environment partition token established at partition initialization.

ttoken

A doubleword value containing the Language Environment thread token established at thread initialization.

rtoken

A doubleword value containing the Language Environment run unit token established at run unit initialization.

terminfo

A 32 bit structure that communicates termination information from CICS to Language Environment. See Figure 100.

```
DCL 1 TERMINFO    BIT(32),          /* CICS termination information */
      2 TCB_SWITCH BIT(1),          /* Indicates TCB has been
                                     /* switched and clean-up should
                                     /* be bypassed
                                     /* For OTE and reusable enclaves*/
                                     /* CICS may or may not call Lang*/
                                     /* Env for the proper TCB that
                                     /* original task (TCB) was
                                     /* dubbed on
```

Figure 100. Termination information supplied from CICS to Language Environment

Run unit (program) begin invocation

After a successful Language Environment run unit initialization, CICS calls the Language Environment-CICS interface routine for the run unit begin invocation call. Language Environment-CICS interface routine then calls the main Language Environment-enabled program at its entry point established at link-edit time. If the program is not a Language Environment-enabled program, the language-specific interface routine is called to do the run unit invocation instead. A single link-edited load module under CICS can have only one functional entry point which is established at link-edit time. The application program argument list is passed to the program.

Before control is transferred to the main program, the initialization assembler user exit and the HLL initialization user exit is called and spool files if appropriate is opened. These spool files are closed at run unit end invocation.

If the run unit abends or terminates with a CICS request (EXEC CICS ABEND, EXEC CICS XCTL, or EXEC CICS RETURN), control is not returned to CICS from the run unit begin invocation. Then run unit end invocation call is made by CICS.

Syntax

Call CEECCICS (*32*, *rsncode*, *syseib*, *preasa*, *ptoken*, *ttoken*, *rtoken*, *pgminfo1*, *pgminfo2*, *ioinfo*) **Retcode** (*rc*)

rsncode (output)

A fullword integer in *nnnffrr* format to contain the member language-specific run unit invocation reason code or one of the following Language Environment reason codes:

- 13200** Invalid parameter was passed
- 13210** No run unit work area was passed
- 13230** Language-specific run unit invocation failed
- 13240** No application program argument list was passed

syseib

The system EXEC interface block, as defined by CICS. This control block contains information about running the CICS commands issued by Language Environment. There is no need to save and restore the user EIB around the commands issued by Language Environment. The system EIB address is above 16M. For the information contained in the EIB, see .

Note: The user EIB, which is different than the system EIB, is passed to the application program in the application program argument list.

preasa

A preallocated save area passed to Language Environment by CICS. The size of this save area is 248 bytes.

ptoken

A doubleword value containing the Language Environment partition token established at partition initialization.

ttoken

A doubleword value containing the Language Environment thread token established at thread initialization.

Begin Invocation

rtoken

A doubleword value containing the Language Environment run unit token established at run unit initialization.

pgminfo1

The structure of information supplied by CICS to Language Environment as shown in Figure 97 on page 453; this is the same structure as in the run unit initialization call.

pgminfo2

The structure of information supplied by CICS to Language Environment as shown in the example here; this is the same structure as in the run unit initialization call.

ioinfo

A structure that contains the standard input/output and error file information (transient data queue names or spool file classes). The structure declaration follows:

```
DCL 1 IOINFO,                /* I/O Information Structure */
      2 STD_IN,              /* Standard input file */
      3 QORS_IN CHAR(1),    /* - either */
                          /* 'Q', transient data queue, or */
                          /* 'S', spoolfile */
      3 TDQN_IN CHAR(4),    /* - queue name */
      3 SPOC_IN CHAR(1),    /* - spool class */
      2 STD_OUT,           /* Standard output file */
      3 QORS_OUT CHAR(1), /* - either */
                          /* 'Q', transient data queue, or */
                          /* 'S', spoolfile */
      3 TDQN_OUT CHAR(4), /* - queue name */
      3 SPOC_OUT CHAR(1), /* - spool class */
      2 STD_ERR,          /* Standard error file */
      3 QORS_ERR CHAR(1), /* - either */
                          /* 'Q', transient data queue, or */
                          /* 'S', spoolfile */
      3 TDQN_ERR CHAR(4), /* - queue name */
      3 SPOC_ERR CHAR(1), /* - spool class */
```

Figure 101. Structure of information supplied to Language Environment by CICS

std_in

A sub-structure that contains information regarding the standard input file. CICS acquires this information from the user through a user exit prior to the run unit begin invocation call to Language Environment.

qors_in

A character ('Q' or 'S') flag to indicate whether the file is a transient data queue or a spoolfile.

tdqn_in

A 4-character standard input transient data queue name when *qors_in* flag is set to 'Q'.

sproc_in

A character identifying the standard input spool file class when *qors_in* flag is set to 'S'.

std_out

A sub-structure that contains information regarding the standard output

file. CICS acquires this information from the user through a user exit prior to the run unit begin invocation call to Language Environment.

qors_out

A character ('Q' or 'S') flag to indicate whether the file is a transient data queue or a spoolfile.

tdqn_out

A 4-character standard output transient data queue name when *qors_in* flag is set to 'Q'.

spoc_out

A character identifying the standard output spool file class when *qors_in* flag is set to 'S'.

std_err

A sub-structure that contains information regarding the standard error file. CICS acquires this information from the user through a user exit prior to the run unit begin invocation call to Language Environment.

qors_err

A character ('Q' or 'S') flag to indicate whether the file is a transient data queue or a spoolfile.

tdqn_err

A 4-character standard error transient data queue name when *qors_in* flag is set to 'Q'.

spoc_err

A character identifying the standard error spool file class when *qors_in* flag is set to 'S'.

The Language Environment Initialization routine (CEEINT), which is usually called from the application program or a language library (COBOL for z/OS and VM and VM's bootstrap library routine IGZCBS0) after the invocation of the application program, checks the Language Environment anchor first (with CEEARLU). Since under CICS, the Language Environment environment is initialized prior to the run unit (program) invocation at partition, thread and run unit initialization, no further action is required. Control is returned to the caller of the CEEINT with an appropriate return code indicating that the Language Environment environment is up. The program continues to run. For a detail information regarding the CEEINT and the possible return codes, see "CEEINT interface" on page 153.

Run unit (program) end invocation

The Language Environment-CICS interface routine is called for run unit end invocation as a result of a normal return (with language terminating statements) from the run unit begin invocation or as a result of a CICS terminating command from the application program or Language Environment.

If the program undergoes normal termination through the language statements such as COBOL's STOP RUN, PL/I's END or RETURN or SIGNAL FINISH, Language Environment termination imminent condition is raised by the program load module. After this condition is handled by the Language Environment exception handler, control is returned to the Language Environment run unit begin invocation routine with the return address (R14) originally stored in the first save area of the program work area. Language Environment run unit begin invocation routine then returns to CICS. Next, the Language Environment-CICS interface routine expects a call for run unit end invocation to do the invocation-based clean ups.

End Invocation

If the program undergoes termination (normal or abnormal) as a result of running the EXEC CICS XCTL, RETURN, or SEND PAGE RELEASE commands or from an abend (program check or EXEC CICS ABEND command), the Language Environment-CICS interface module is driven by CICS. The run unit end invocation call receives the run unit token to identify the Language Environment run unit that is being terminated. A program termination block (PTB, the TERMINFO structure) is also provided by CICS to describe the termination situation (normal or abnormal). This call notifies Language Environment that one of the resources for which it is responsible is terminating due to some reason that might not yet be known.

Restrictions on the Language Environment run unit end invocation routine are:

- It must always return to the address in R14
- It must not issue EXEC CICS ABEND (it can issue any other EXEC CICS command that does not raise a subsequent error)

Syntax

Call CEECCICS (*33*, *rsncode*, *syseib*, *preasa*, *ptoken*, *ttoken*, *rtoken*, *pgminfo1*, *pgminfo2*, *ptb*) **Retcode** (*rc*)

rsncode **(output)**

A fullword integer that contains one of the following Language Environment reason codes:

13300 Invalid parameter was passed

13310 Wrong thread token was passed

13320 No PTB was passed

syseib

The system EXEC interface block, as defined by CICS for use by Language Environment and language-specific interface routines. The system EIB address is above 16M. For the information contained in the EIB, see .

preasa

A fullword value containing the preallocated save area address to be used by Language Environment to issue the last EXEC CICS FREEMAIN command. The size of this save area is 248 bytes.

ptoken

A doubleword value containing the Language Environment partition token established at partition initialization.

ttoken

A doubleword value containing the Language Environment thread token established at thread initialization.

rtoken

A doubleword value containing the Language Environment run unit token established at run unit initialization.

pgminfo1

The structure of information supplied by CICS to Language Environment, as shown in Figure 97 on page 453; this is the same structure as in the run unit begin call.

pgminfo2

The structure of information supplied by CICS to Language Environment, as shown in the example here; this is the same structure as in the run unit begin call.

ptb

A fullword value containing the address of the program termination block, the TERMINFO structure. This structure contains information regarding the normal and abnormal termination of a run unit. For abnormal termination, information such as PSW, general purpose registers, floating-point registers and access registers at the time of interrupt are provided. A retry mechanism is also provided. A declaration of the PTB is shown in Figure 102 on page 462.

End Invocation

```

DCL 1 TERMINFO,                /* Program Termination Block */
  2 TERMCODE BIT(32),          /* Termination Code - see below */
  3 ABNORM BIT(1),            /* .00 Abnormal Termination */
  3 NORMCEL BIT(1),           /* .01 Normal Termination (LE) */
  3 NORMRET BIT(1),          /* .02 Normal Termination (return) */
  3 NORMASM BIT(1),          /* .03 Normal Termination (ASM) */
  3 ABNPCHK BIT(1),          /* .04 Abnormal Term (PGM Check) */
  3 ABNOTHER BIT(1),         /* .05 Abnormal Term (ABEND) */
  3 ABNLINK BIT(1),          /* .06 Abnormal Term (lower level) */
  3 HANDELAB BIT(1),         /* .07 User HANDLE ABEND pending */
  3 PTBINUSE BIT(1),         /* .08 PTB is busy */
  3 PSWCICS BIT(1),          /* .09 PSW is in CICS code */
  3 NODUMP BIT(1),           /* .10 CICS specified nodump */
  3 CANCEL BIT(1),           /* .11 CICS specified cancel */
  3 PCHKGR64 BIT(1),         /* .12 CICS supplying 64-bit regs */
  3 PCHKAR BIT(1),           /* .13 CICS supplying access regs */
  3 PCHKFR16 BIT(1),         /* .14 CICS supplying 16 FPRs, FPC */
  3 * BIT(17),               /* reserved */
  2 ABCODE CHAR(4),          /* CICS Abend Code */
  2 PCHK,                     /* Program Check Information */
  3 PCHK_PSW CHAR(16),       /* - PSW */
  3 PCHK_INT,                 /* - Interrupt Data */
  4 PCHK_ILC CHAR(2),        /* - Instruction Length Code */
  4 PCHK_INC CHAR(2),        /* - Interruption Code */
  4 PCHK_EAD CHAR(4),        /* - Exception Address */
  3 PCHK_GR POINTER,         /* - General Registers (32-bit or
                             /* 64-bit registers) */
  3 PCHK_FR POINTER,         /* - floating-point Registers
                             /* (4 FPRs or 16 FPRs and FPC
                             /* register) */
  3 PCHK_AR POINTER,         /* - Access Registers (if present) */
  2 COMMREGS POINTER,        /* Registers at last CICS command
                             /* (Appl or library -- 32-bit or
                             /* 64-bit registers) */
  2 CONTCODE BIT(32),        /* Continuation Code (receiver) */
  3 TERM BIT(1),             /* .00 Continue RU termination */
  3 EXEC BIT(1),             /* .01 Resume at RTRY_GR R15 */
  3 RTRY BIT(1),             /* .02 Resume at RTRY_AD */
  3 TERMOTETCB BIT(1),      /* .03 Terminate the OTE TCB */
  3 * BIT(1),                /* reserved */
  3 RTRYGR64 BIT(1),         /* .05 Retry with 64-bit registers */
  3 RTRYAR BIT(1),          /* .06 Retry with access registers */
  3 RTRYFR16 BIT(1),        /* .07 Retry with 16 FPRs and FPC */
  3 * BIT(24),               /* reserved */
  2 RTRY,                     /* Retry Information (receiver) */
  3 RTRY_AD POINTER,         /* - Resume Address */
  3 RTRY_PM POINTER,         /* - Program Mask */
  3 RTRY_GR POINTER,         /* - General Registers (32-bit or
                             /* 64-bit registers) */
  3 RTRY_FR POINTER,         /* - floating-point Registers
                             /* (4 FPRs or 16 FPRs and FPC
                             /* register) */
  3 RTRY_AR POINTER,         /* - Access Registers (if present) */
  2 * POINTER,               /* Reserved */

```

Figure 102. Program termination block (PTB) declaration

termcode

A fullword binary value that contains bit settings to indicate the nature, the cause and the location of the run unit end invocation. The following table describes each of the bit settings:

Bit setting	Description
Bit0 (abnorm)	Indicates an abnormal termination. It indicates a normal termination when it is off.

Bit setting	Description
Bit1 (normcel)	Indicates that a normal termination is caused by a CICS command in a Language Environment-enabled run unit. Notice that the caller does not get control back after an EXEC CICS RETURN in a called (link-edited) routine.
Bit2 (normret)	Indicates that a normal termination is caused by a BR 14 from the run unit begin invocation (program has ended with a language statement).
Bit3 (normasm)	Indicates that a normal termination is caused by a CICS command in a called (link-edited) assembler routine.
Bit4 (abnpchk)	Indicates that an abnormal termination is caused by a program check (abend ASRA) in a run unit.
Bit5 (abnother)	Indicates that an abnormal termination is caused by an abend. See the ABCODE.
Bit6 (abnlink)	Indicates that the lower level (linked to) run unit has terminated abnormally.
Bit7 (handleab)	Indicates that a user HANDLE ABEND routine is pending. The Language Environment exception manager should not handle the exception.
Bit8 (input/output) (ptbinuse)	Indicates that this PTB is busy. It is set by Language Environment during the exception handling to avoid recursion. It is turned off by Language Environment when the exception handling completes.
Bit9 (pswcics)	Indicates that the PSW is in the CICS code.
Bit10 (nodump)	Indicates that CICS specified nodump.
Bit11 (cancel)	Indicates that CICS specified cancel.
Bit12 (pchgr64)	Indicates that CICS is supplying 64-bit registers 0-15 rather than 32-bit registers 0-15 in the areas pointed to by PCHK_GR and COMMREGS. This bit is valid only if both the CICS_EXT_REG and LE_EXT_REG flags in the Partition Init flags are 1. If CICS_EXT_REG or LE_EXT_REG is 0, Bit12 is not valid and is assumed to be 0.
Bit13 (pchkar)	Indicates that CICS is supplying the access registers in the area pointed to by <i>pchk_ar</i> . If Bit13 is 0, no access registers are supplied. This bit is valid only if both CICS_EXT_REG and LE_EXT_REG flags in the Partition Init flags are 1. If CICS_EXT_REG or LE_EXT_REG is 0, Bit13 is not valid and is assumed to be 0.
Bit14 (pchkfr16)	Indicates that CICS is supplying all 16 floating-point registers and the floating-point control register in the area pointed to by <i>pchk_fr</i> , rather than just the 4 floating-point registers (0, 2, 4, 6). This bit is valid only if both CICS_EXT_REG and LE_EXT_REG flags in the Partition Init flags are 1. If CICS_EXT_REG or LE_EXT_REG is 0, Bit14 is not valid, and is assumed to be 0.

abcode

A 4-byte character string that identifies the abend code for the abnormal termination.

pchk

A structure that contains the program check information for the abnormal termination.

pchk_psw

An 8-byte field containing the Program Status Word (PSW). Notice that the first word of the PSW contains information such as the condition code and program mask at the time of interrupt and the second word

End Invocation

of the PSW contains the address and the AMODE of the instruction after the instruction which caused the program check.

pchk_int

An 8-byte field containing the instruction length code (2 bytes), interruption code (2 bytes) and the exception address (4 bytes).

pchk_gr

If TERMCODE Bit12 is 0, *pchk_gr* points to a 64-byte storage area that contains the 32-bit registers (R0-R15) at the time of the program check. If TERMCODE Bit12 is 1, and both CICS_EXT_REG and LE_EXT_REG in the Partition Init flags are 1, *pchk_gr* points to a 128-byte storage area that contains the 64-bit registers (R0-R15) at the time of the program check.

pchk_fr

If TERMCODE Bit14 is 0, *pchk_fr* points to a 32-byte storage area that contains the floating-point registers F0, F2, F4, and F6 at the time of the program check. If TERMCODE Bit14 is 1, and both CICS_EXT_REG and LE_EXT_REG in the Partition Init flags are 1, *pchk_fr* points to a 132-byte storage area that contains all 16 floating-point registers and the floating-point control (FPC) register at the time of the program check. The register values are saved in the order: F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15, FPC register. *pchk_fr* might point to the same area as *rtry_fr*.

pchk_ar

If TERMCODE Bit13 is 1, and both CICS_EXT_REG and LE_EXT_REG in the Partition Init flags are 1, *pchk_ar* points to a 64-byte storage area that contains access registers (AR0-AR15) at the time of the program check. If TERMCODE Bit13 is 0, *pchk_ar* is not valid. *pchk_ar* might point to the same area as *rtry_ar*.

commregs

If TERMCODE Bit12 is 0, *commregs* points to a 64-byte storage area that contains the 32-bit registers (R0-R15) at the time of the last CICS command. If TERMCODE Bit12 is 1, and both CICS_EXT_REG and LE_EXT_REG in the Partition Init flags are 1, *commregs* points to a 128-byte storage area that contains the 64-bit registers (R0-R15) at the time of the last CICS command. In either case, the CICS command could have been issued from an application program or a library routine.

contcode (output)

A fullword binary value to contain bit settings to indicate if the run unit end invocation should continue.

Bit setting	Description
Bit0 (term)	Set by Language Environment when normal or abnormal run unit end invocation is considered complete. CICS continues its run unit end invocation process without calling Language Environment for the run unit end invocation of the same run unit again.

Bit setting	Description
Bit1 (exec)	Set by Language Environment to have CICS continue running the run unit at the retry address in R15 in the <i>rtry_gr</i> field. If contcode bit 5 is set, only the lower 31 bits of the 64-bit R15 value are used as the retry address. CICS sets up all the 32-bit or 64-bit registers from the retry structure before resuming the program with a branch on R15. The retry PSW (RTRY_AD) is ignored. Language Environment is recalled for the run unit end invocation later for this run unit. Note: The termination routine can change a normal termination into an abnormal termination by setting Bit1 of CONTCODE and making the retry point the address of an EXEC CICS ABEND.
Bit2 (rtry)	Set by Language Environment to have CICS continue running the run unit at the retry PSW (RTRY_AD) with the retry registers (RTRY_GR, RTRY_FR, and RTRY_AR if contcode Bit6 is on) set by Language Environment. CICS resumes using a method that allows all resume registers and the resume PSW to be set to the requested values as control is passed to the resume point. Language Environment is called for the run unit end call for this run unit later.
Bit3 (termotetcb)	CICS must terminate the OTE TCB.
Bit5 (rtrygr64)	Indicates that Language Environment is returning 64-bit registers 0-15 rather than 32-bit registers 0-15 in the area pointed to by <i>rtry_gr</i> . This bit is valid only if both CICS_EXT_REG and LE_EXT_REG flags in the Partition Init flags are 1. If CICS_EXT_REG is 0 or LE_EXT_REG is 0, Bit5 is not valid and is assumed to be 0.
Bit6 (rtryar)	Indicates that Language Environment is returning access registers AR0-AR15 in the area pointed to by <i>pchk_ar</i> . If Bit5 is 0, no access registers are being returned. This bit is valid only if both CICS_EXT_REG and LE_EXT_REG flag in the Partition Init flags are 1. If CICS_EXT_REG is 0 or LE_EXT_REG is 0, Bit6 is not valid and is assumed to be 0.
Bit7 (rtryfr16)	Indicates that Language Environment is returning all 16 floating-point registers (F0-F15) and the floating-point control register (FPC) in the area pointed to by <i>rtry_fr</i> , rather than just the 4 floating-point registers (F0, F2, F4, F6). This bit is valid only if both CICS_EXT_REG and LE_EXT_REG flag in the Partition Init flags are 1. If CICS_EXT_REG is 0 or LE_EXT_REG is 0, Bit7 is not valid, and is assumed to be 0.

rtry (input/output)

A pointer structure. The retry pointers address areas into which Language Environment places the retry information when the run unit might be continued at some point in the code. Before calling Language Environment, CICS sets these pointers to address the same areas as are used to hold the registers at program check or abend (PCHK) details. When a retry has been requested, Language Environment will reset these variables to the value in the resume cursor.

rtry_ad

The retry address desired by Language Environment. CICS uses this address to build the retry PSW or load it into R15 (depending upon the CONTCODE) before continuing to run.

rtry_pm

A pointer to a one byte field to contain the retry condition code (Bits 2 and 3) and program mask (Bits 4-7). Bits 0 and 1 are not used.

rtry_gr

If CICS_EXT_REG in the Partition Init flags is 0, *rtry_gr* points to a

End Invocation

64-byte storage area. If CICS_EXT_REG in the Partition Init flags is 1, *rtry_gr* points to a 128-byte storage area. When Language Environment returns to CICS, contcode Bit5 determines the contents of the area pointed to by *rtry_gr*. When contcode Bit5 is 0, this area contains 32-bit general registers R0-R15. When contcode Bit5 and LE_EXT_REG are 1, this area contains 64-bit general registers R0-R15.

rtry_fr

If CICS_EXT_REG in the Partition Init flags is 0, *rtry_fr* points to a 32-byte storage area. If CICS_EXT_REG in the Partition Init flags is 1, *rtry_fr* points to a 132-byte area. When Language Environment returns to CICS, contcode Bit7 determines the contents of the area pointed to by *rtry_fr*. When contcode Bit7 is 0, this area contains floating-point registers 0, 2, 4, and 6 for the retry. When contcode Bit7 and LE_EXT_REG are 1, this area contains floating-point registers 0-15 and the floating-point control (FPC) register. The register values are saved in the order: F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15, FPC register. *rtry_fr* might point to the same area as *pchk_fr*.

rtry_ar

If CICS_EXT_REG in the Partition Init flags is 1, *rtry_ar* points to a 64-byte storage area. If CICS_EXT_REG in the Partition Init flags is 0, *rtry_ar* is not valid. When Language Environment returns to CICS, contcode Bit6 determines the contents of the area pointed to by *rtry_fr*. When contcode Bit6 is 0, the contents of this area is undefined. When contcode Bit6 and LE_EXT_REG are 1, the area pointed to by *rtry_ar* contains access registers AR0-AR15 for the retry. *rtry_ar* might point to the same area as *pchk_ar*.

If the program is terminated with the following commands, CICS requests the Language Environment-CICS interface routine through the run unit end invocation call to terminate the run unit invocation:

- EXEC CICS XCTL
- EXEC CICS RETURN
- EXEC CICS SEND PAGE RELEASE

Bit1 or Bit3 (normal termination due to a CICS command) of TERMCODE are ON at this call. The Language Environment-CICS interface routine performs run unit end invocation activities, set Bit0 of CONTCODE to indicate that CICS should continue run unit end invocation without reentering the Language Environment-CICS interface routine and return to CICS. Notice that the termination imminent condition is not raised in this case. The termination imminent condition is raised if Language Environment is not terminating for cleanup purposes and one or more of the following conditions are true:

- A user handler is active
- A PL/I finish on_unit is active
- The debug handler is active
- Runtime option TEST(ALL,,) is specified

For any other case, there is no handler that may take action, so termination imminent is not raised to improve performance.

If the program is terminated with one of the following methods, CICS requests the Language Environment-CICS interface routine through the run unit end invocation call to terminate the run unit invocation:

- EXEC CICS ABEND command
- Program Check (abend ASRA by CICS)

Bit0 (abnormal termination) of TERMCODE is ON at this call. If TRAP runtime option is in effect, the Language Environment-CICS interface routine calls the Language Environment exception handler. Otherwise, it returns to CICS.

When Language Environment is called for run unit end invocation again after the exception handling, it performs all the run unit end invocation activities and return to CICS. After control is returned to CICS by Language Environment run unit end invocation, CICS can drive Language Environment run unit termination to do the run unit clean up.

Due to the nature of the activities in the run unit end invocation (for example, storages are not freed and loaded routines are not released), there is no requirement for language-specific run unit end invocation.

Error recovery

When CICS first finds a program check, it calls the Language Environment-CICS interface routine to examine the exception and perform error recovery, if possible. Because this interface routine is called before CICS calls its own internal recovery procedures, console messages or dumps may not have been issued.

This call allows Language Environment to perform some of the low level error functions. This includes “shunt” routines and the Language Environment application resume function. This routine is not called for CICS ABEND conditions.

The following restrictions apply to this routine:

- It must always return to the address in R14
- It must not issue any EXEC CICS commands

Syntax

Call CEECCICS (34, *rsncode*, *ptb*, *preasa*) **Retcode** (*rc*)

***rsncode* (output)**

A fullword integer; it is initially set to zero.

***ptb* (input, output)**

A fullword value containing the address of the program termination block, the TERMINFO structure. This structure contains information regarding the normal and abnormal termination of a run unit. For abnormal termination, information such as PSW, general purpose registers, floating-point registers and access registers at the time of interrupt are provided. A retry mechanism is also provided. Figure 102 on page 462 shows a declaration of the PTB.

***preasa* (input, output)**

A preallocated save area passed to Language Environment by CICS. The size of this save area is 248 bytes.

***rc* (output)**

The following values should also be placed in R15 on exit. In addition to the return code, the resume code in the PTB must be set.

- 0 Language Environment can resolve the error; the PSW, general purpose registers, and floating-point registers have been updated.

- 4 Language Environment could not resolve the error; CICS processing continues.

Determine working storage and static storage

In order for CICS Execution Diagnostic Facility (EDF) to be able to display the program's working storage (DSA stack) and static storage, CICS makes the determine working storage address call to the Language Environment-CICS interface module. Through the program register save area (*pgmrsa*) argument passed by CICS, Language Environment examines the program entry point (R15 saved in the caller's DSA) to determine if the program is a Language Environment-enabled program. If so, the member ID of the program's language is remembered. Otherwise, the member ID is determined through the *lang* argument passed by CICS.

In either case (Language Environment-enabled or not), the member-specific interface routine associated with the member ID is called to provide the working storage address (most probably same as the *pgmrsa*) and the static storage address and their length. This is because the working storage address could potentially be different than the DSA (as in COBOL) and the static storage could potentially be separate from the program load module.

Syntax

Call CEECCICS (*60, rsnocode, syseib, preasa, ptoken, ttoken, rtoken, lang, pgmrsa, wsa, wsl, ssa, ssl, pgmep*) **Retcode** (*rc*)

rsnocode (output)

A fullword integer that contains one of the following Language Environment reason codes:

- 16000** Invalid parameter was passed
- 16030** Language-specific determine working storage address failed
- 16040** Working storage address and length was not determined

syseib

The system EXEC interface block, as defined by CICS for use by Language Environment and language-specific interface routines. The system EIB address is above 16M. For the information contained in the EIB, see .

preasa

A preallocated save area passed to Language Environment by CICS. The size of this save area is 248 bytes.

ptoken

A doubleword value containing the Language Environment partition token established at partition initialization.

ttoken

A doubleword value containing the Language Environment thread token established at thread initialization.

rtoken

A doubleword value containing the Language Environment run unit token established at run unit initialization.

lang

A fullword value identifying the language of the program which issued a CICS

command. Bit definitions are shown in Figure 103.

```

DCL 1 LANG,                               /* Language of program issuing */
                                     /* HANDLE CONDITION or AID cmd */
      2 ASSEMBLER BIT(1),                 /* LANG=ASSEMBLER             */
      2 C370      BIT(1),                 /* LANG=C/370                 */
      2 COBOL     BIT(1),                 /* LANG=COBOL II              */
      2 PLI       BIT(1),                 /* LANG=PL/I                   */
      2 RPG       BIT(1),                 /* LANG=RPG                    */
      2 NOTAPPLIC BIT(1),                 /* LANG=NOTAPPLIC or blank    */
      3 *         BIT(26),                /* Reserved                    */

```

Figure 103. Lang bit definition for CEECICS (60)

pgmrsa

A fullword value containing the address of the register save area (DSA) of the program which is issuing the CICS command.

wsa (output)

A fullword value to contain the working storage address (DSA) of the program issuing the command.

wsl (output)

A fullword value to contain the length of the working storage of the program issuing the command (DSA length).

ssa (output)

A fullword value to contain the static storage address of the program issuing the command.

ssl (output)

A fullword value to contain the length of the static storage of the program issuing the command.

pgmep

A fullword value to contain the address of the program entry point of the program that is issuing the CICS command.

Perform GOTO call

When a HANDLE CONDITION condition (label) or a HANDLE AID option (label) or a HANDLE ABEND LABEL (label) is in effect in the user program and CICS raises that condition, the need for transferring control to the specified label arises. This need is satisfied through the perform GOTO call from CICS to the Language Environment-CICS interface module.

Syntax

Call CEECCICS (70, *rsncode*, *syseib*, *preasa*, *ptoken*, *ttoken*, *rtoken*, *lang*, *label*, *invkdsa*, *gotoflgs*) **Retcode** (*rc*)

rsncode (output)

A fullword integer that contains one of the following Language Environment reason codes:

- 17000 Invalid parameter was passed
- 17040 GOTO Out-Of-Block was not performed
- 17060 Invalid DSA chain

Perform GOTO call

syseib

The system EXEC interface block, as defined by CICS for use by Language Environment and language-specific interface routines. The system EIB address is above 16M. For the information contained in the EIB, see .

preasa

A preallocated save area passed to Language Environment by CICS. The size of this save area is 248 bytes.

ptoken

A doubleword value containing the Language Environment partition token established at partition initialization.

ttoken

A doubleword value containing the Language Environment thread token established at thread initialization.

rtoken

A doubleword value containing the Language Environment run unit token established at run unit initialization.

lang

A fullword value identifying the language of the program which issued EXEC CICS HANDLE CONDITION, AID or ABEND command. Bit definitions are shown in Figure 104.

DCL 1 LANG,		/* Language of program issuing */
		/* HANDLE CONDITION or AID cmd */
2 ASSEMBLER	BIT(1),	/* LANG=ASSEMBLER */
2 C370	BIT(1),	/* LANG=C/370 */
2 COBOL	BIT(1),	/* LANG=COBOL II */
2 PLI	BIT(1),	/* LANG=PL/I */
2 RPG	BIT(1),	/* LANG=RPG */
2 *	BIT(27),	/* Reserved */

Figure 104. Lang bit definition for CEECICS (70)

label

A fullword value containing the label argument information passed to CICS by the member language-generated code (the CALL statement) for EXEC CICS HANDLE CONDITION or AID or ABEND commands. For COBOL programs, this is the address of an area (64 bytes) that contains all 16 general registers (in the order of R14, R15, R0-R13) at the time of the HANDLE command. For assembler programs, a resume point and registers 14 and 15 are passed.

invkdsa

A fullword value containing the address of the DSA of the program that caused the condition.

gotoflgs (output)

A fullword value that is used to indicate if the GOTO is allowed. Bit definitions are shown in Figure 105 on page 471.

```

DCL 1 GOTOFLGS,          /* Perform GOTO Flags      */
     2 GOTONOCDBIT(1),  /* ON=GOTO cannot be performed */
                               /* because cross program    */
                               /* branching is not supported */
                               /* by a member language.    */
     2 *                  /* Reserved                 */
     BIT(31),

```

Figure 105. Lang bit definition for GOTOFLGS

CEEECTCB — set TCB+X'144' routine

The sole function of this routine is to set TCB+X'144' to a storage area that is nonfetch protected and set that storage to zero. CEEECTCB is always invoked by the CICS AP-BIND independent of the storage protect feature being on or available. CEEECTCB is called by loading the executable named CEEECTCB (using the LOAD SVC service and BALR to the entry point) or by linking directly to CEEECTCB (using the LINK SVC service). The CEEECTCB executable resides in the SCEERUN data set.

Syntax

CEEECTCB

CEEECTCB

The load name of the Language Environment routine that alters the storage of TCB+'144' to point to a nonfetch protected, key 8, block of storage.

Entry Conditions:

1. Execution mode is supervisor state
2. A standard save area is not provided
3. The PSW key is the TCB key (TCBPKF)
4. Extraction authority is needed for IVSK (control reg, bit 4)
5. R1 must be zero
6. R14 is the return address
7. AMODE(31)/RMODE(ANY)
8. ASCMODE is primary

Exit Conditions:

1. Execution mode remains supervisor state
2. R15 contains the following values
 - X'00' Success
 - X'04' The area pointed to by TCB+X'144' was not TCBPKF/fetch protected
 - X'08' GETMAIN failure
 - X'0C' Setting the value at TCB+X'144' failed (CS failure?)
 - X'10' TCB+X'144' was zero
 - X'14' Entry into CEEECTCB was not in TCBPKF key
 - X'18' R1 was not zero upon entry
 - X'1C' (TCB+X'144' -> Anchorword) was not zero.

CEEECTCB

3. The PSW key is same as on entry
4. R2-R14 remain unchanged
5. ARs, FPRs remain unaltered
6. The condition code does not contain any useful information
7. It is CICS' responsibility to delete CEEECTCB
8. AMODE and ASCMODE are unchanged

CEECCICS — partition initialization changes

The reason code value of X'11050' denotes partition initialization failure, due to a problem with a CEEECTCB failure. CEECCICS checks the value of the field pointed to by TCB+144, for a the eyecatcher CEEECTCB value set by CEEECTCB. If the eyecatcher is not found, Language Environment partition initialization fails and sets the return code and reason code passed back to CICS appropriately.

If partition initialization was successful, R15 contains 0; otherwise, R15 contains 16. The reason code parameter, *rsncode*, that is passed back to CICS identifies one of the following failures:

rsncode (output)

A fullword integer in *mmffrr* format to contain the member language-specific partition initialization reason code or one of the following Language Environment reason codes:

- | | |
|--------------|--|
| 11000 | Invalid parameters were passed |
| 11010 | Storage was not available |
| 11020 | LIBVEC was not loaded |
| 11030 | Language-specific partition initialization was not done |
| 11040 | Language Environment process initialization failed, due to an internal abend |
| 11050 | Language Environment anchor vector setup failed |

IMS considerations

IMS supports PL/I, C, COBOL, and assembler applications.

IMS to Language Environment

The new interface from IMS to Language Environment supports all Language Environment-enabled languages. ILC capabilities are enhanced. IMS constructs a parameter list as shown in Figure 106 on page 473.

XPLINK applications are supported under IMS. For more information about XPLINK, see "Extra Performance Linkage (XPLINK) CALL linkage conventions" on page 114.

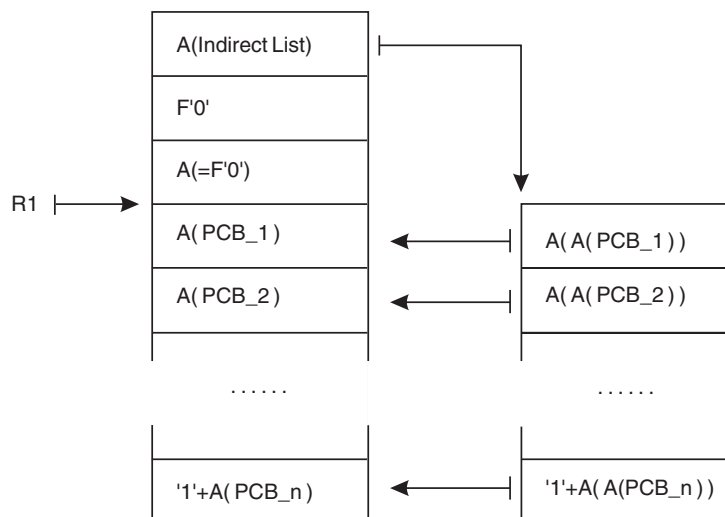


Figure 106. IMS parameter list format

Usage notes:

1. R1 contains the address of a parameter address list. Each word in the list contains the address of a PCB.
2. R1 has the high-order bit on indicating this particular parameter list format.
3. This interface can be manufactured regardless of the mechanism of application invocation (for example, BALR or SVC LINK).
4. IMS always constructs this parameter list format regardless of the LANG option on the PSB.
5. The last word in each of the above boxes has the high order bit set on, as indicated by the '1'.
6. When the indirect list is passed to the application, the high order bit should be turned off in the last A(PCB_n).

Compatibility concerns

Current PL/I, COBOL, and assembler support continues to work. Having the high-order bit on in R1 does not alter the use of R1 as an address.

Note: PL/I no longer needs to intercept calls to PLI2DLI. The Language Environment-to-IMS interface enables Language Environment to ask if IMS wants to process any errors that occur.

Also, the LANG option has no effect on the format of the parameter list built by IMS. Thus, any PSB can be used with any HLL application. Any adjustment to the contents of R1 to accommodate the various HLL requirements is performed by the Language Environment or the HLL-specific library, **not** by IMS.

Language Environment to IMS — CEETDLI

Language Environment is providing a callable service, CEETDLI, that is callable from any HLL that provides R12 pointing to the CAA and R13 pointing to a DSA. It has the same programming requirements (parameters) as the pre-Language Environment entry names: CTDLI, ASMTDLI, and CBLTDLI.

The Language Environment callable service calls IMS through a new entry point in module DFSLI000: DFSLICEL. The parameter list and the caller's save area can reside above or below the 16M line.

DFSLI000 is linked with the user's application code, and DFSLICEL is entered in the AMODE of the application.

The *parmcount* parameter is optional for CEETDLI.

CEETDLI uses a 2-byte length field (LL) indicating the total length of an IMS message or Scratch Pad Area (SPA). CEETDLI employs a flat parameter list. That is, each entry in the parameter address list points directly to the argument. The parameters described in the should be used with the new name CEETDLI.

The ABTERMENC(ABEND) runtime option or the CEEBXITA assembler user exit can be used by the installation to force an abend at application termination. These methods force data base rollbacks for applications terminating abnormally.

The PLITDLI, CBLTDLI, ASMTDLI, and CTDLI interfaces continue to function in their current capacity.

Implementation

IMS always constructs the parameter list as shown in Figure 106 on page 473, regardless of the LANG option in the PSB. IMS also adds a new entry point to DFSLI000 called DFSLICEL. This entry supports any of the member languages running in the Language Environment environment. Only Language Environment-enabled code should call CEETDLI.

On entry to CEETDLI, Language Environment takes the appropriate steps to insure exceptions and abends that occur while executing in IMS code are percolated.

Chapter 14. Anchor support

The Language Environment anchor is the address of Language Environment's main control block, the CAA. There is one CAA per thread, and the CAA is created during thread initialization. When the address of the anchor is unknown to the executing routine, it must be obtained. This usually occurs when an old routine calls a newly compiled, Language Environment-enabled routine. Upon entry into the new routine, the Language Environment anchor service must be able to return the Language Environment anchor using the same sequence of code regardless of the environment under which the application is executing.

Under CICS, one CAA exists per CICS run unit. Because multiple CICS run units can exist under one z/OS TCB, the TCB cannot directly hold the address of the CAA. However, the TCB can point to a code sequence that obtains the anchor.

Anchor service

Figure 107 shows the anchor service, which is based upon the z/OS control block structure. In particular, it is based upon the CVT and the TCB. A fullword field has been reserved in the z/OS TCB for Language Environment's use at TCB+X'144'. This TCB fullword points to a doubleword field that can be altered by Language Environment initialization/termination routines. Language Environment uses the first fullword to point to a control block known as the Language Environment anchor vector. The Language Environment anchor vector contains the addresses of two routines, the fetch anchor routine and the set anchor routine. At offset X'08' is a pointer to the fullword where the CAA address is saved. When this field is zero, the CAA address must be obtained using the get anchor routine.

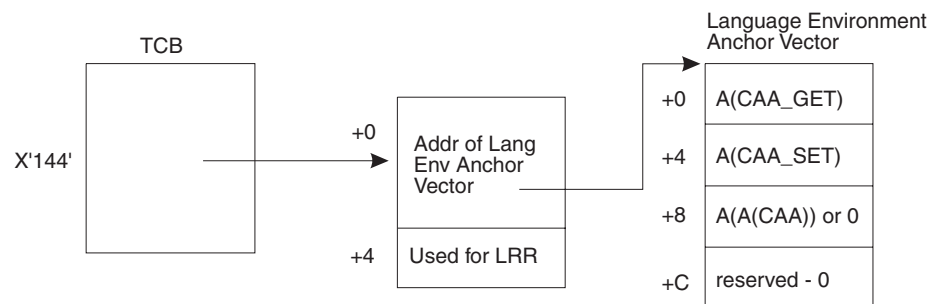


Figure 107. Structure of the Language Environment anchor vector

Fetch the anchor routine

The fetch anchor routine retrieves the current Language Environment anchor and returns it to its caller. A standard code sequence is employed to allow programs to obtain the anchor in a similar manner without incurring the overhead of specialized code for each operating system.

Description of the fetch anchor routine	
Input	<ul style="list-style-type: none"> R12 has the entry point address. R14 has the return address.
Output	R12 has the returned CAA.

Anchor Introduction

Description of the fetch anchor routine			
Code sequence to call this routine	L	R12,16	Get the CVT address
	L	R12,0(R12)	A(TCB old/new)
	L	R12,4(R12)	A(TCB New)
	L	R12,X'144'(R12)	A(A(Language Environment Anchor Vector)) from TCB
	L	R12,0(R12)	A(Language Environment Anchor Vector)
	L	R12,0(R12)	A('Get Anchor' Routine)
	BALR	R14,R12	Go get the anchor.

The fetch (get) anchor routine has the following requirements:

- R0 and R12 are destroyed across this call. All other registers are preserved.
- The Language Environment anchor is returned in R12.
- R14 is used as the return register.
- The only available work registers are R12 and R0.
- A save area is not provided by the caller.
- AMODE switching is not performed.
- This code sequence assumes Language Environment is active.

Set the anchor routine

The set anchor routine saves the token (for example, the address of the CAA) for future retrieval requests. This can also be used to reset the anchor to zero. A standard code sequence is employed to allow programs to set the anchor in a similar manner without incurring the overhead of specialized code for each operating system.

Description of the set anchor routine			
Input	<ul style="list-style-type: none"> • R12 has the CAA address or zero. • R14 has the return address. • R15 has the entry point address. 		
Output	No output is generated for this routine.		
Code sequence to call this routine	L	R15,16	Get the CVT address
	L	R15,0(R15)	A(TCB old/new)
	L	R15,4(R15)	A(TCB New)
	L	R15,X'144'(R15)	A(A(Language Environment Anchor Vector)) from TCB
	L	R15,0(R15)	A(Language Environment Anchor Vector)
	L	R15,4(R15)	A('Set Anchor' Routine)
	BALR	R14,R15	Go set the anchor.

The set anchor routine has the following requirements:

- The available registers are R0, R12, R14, and R15. All other registers are preserved.
- A save area is not provided by the caller.
- AMODE switching is not performed.
- This code sequence assumes the Language Environment environment is active.

CEEARLU — anchor lookup

Purpose

This routine sets register 12 to the Language Environment anchor. The anchor points to the CAA if Language Environment has been initialized.

Format

Call CEEARLU

CEEARLU

Call this CWI interface as follows:

```
L      R15,=V(CEEARLU)
BALR   R14,R15
```

Usage

- Upon return, R12 contains the address of the CAA, or zero if Language Environment has not been initialized.
- R14 and R15 are used as linkage registers. R0 is destroyed across the call. R13 is **not** used.
- This routine is meant to be a fast way of retrieving the anchor. The address of the CAA is not validated by this routine.
- AMODE switching is not performed across this call.
- For additional information on the anchor lookup, see Chapter 14, “Anchor support,” on page 475.
- If the Language Environment environment is not initialized, a value of zero is returned. This routine does not cause the Language Environment environment to be established.
- It must be linked with the program. It is not meant to be called from an HLL program.

Anchor considerations

ATTACH processing obtains a doubleword containing the address of the anchor vector in writable storage and initializes it to zero. In addition, the field at `TCB+X'144'` is initialized to point to this doubleword.

z/OS extends the size of the GETMAINed storage area by a doubleword used for the program's initial register save area. Note that the fullword for the service routine vector address is contained within the user's private space.

During Language Environment initialization, a Language Environment anchor vector is GETMAINed and initialized. The address of the GETMAINed anchor vector is placed into the first fullword provided by z/OS. The code shown in Figure 108 on page 478 is an example of the Language Environment anchor vector code used to set and obtain the Language Environment anchor. Language Environment saves the anchor value directly within the Language Environment anchor vector. At offset `X'08'` is a pointer to the fullword where the CAA address is saved. This does **not** occur under other systems/subsystems.

Anchor considerations

```
ANCHOR  CSECT
        DC  A(GETCAA)      A(Obtain the Current CAA Addr)
        DC  A(SETCAA)     A(Set the current CAA Addr)
        DC  A(CAA)        Pointer to the saved CAA address
        DC  A(0)           Reserved
*****
*   Obtain the current CAA Pointer   *
*****
GETCAA  DS  0H
        L   12,CAA-GETCAA(12)  Get the anchor into R12
        BR  14                 Return to the caller
*****
*   Set the current CAA Pointer     *
*****
SETCAA  DS  0D
        ST  12,CAA-SETCAA(15)  Save/clear the anchor
        BR  14                 Return to the caller
*****
*   Misc writable storage         *
*****
CAA     DC  A(0)           Spot to save the anchor
        DS  0D
SIZE   EQU *-ANCHOR
END
```

Figure 108. Example of code to set and obtain the Language Environment anchor

Bypassing anchor lookup, set, or reset

The CEEPIPI(call_sub_addr_nochk) call invokes a subroutine without causing Language Environment to perform anchor look-up, set, or reset. For more information, including the call syntax, see “CEEPIPI — invocation for subroutine by address” on page 193.

Chapter 15. Member language information

This section addresses the compiler and library programmers of those HLLs that run in Language Environment. It contains a list of restricted host system services, an overview of the structure of the executable program, and how the member languages interact with Language Environment.

OS services — restricted use

Language Environment provides a set of services that are used to insulate the HLL library routines from the underlying host system or subsystem. In providing such functions, the HLL library routines should use the Language Environment-provided services even though a similar service is provided by the underlying system. If the underlying system service is used, Language Environment cannot guarantee some of the consequences that might occur. Listed below is a set of those services which should be avoided because they might interfere with the Language Environment operations. The use of the services should be avoided by members.

Service

Language Environment Equivalent Service

GETMAIN

Language Environment heap storage services.

FREEMAIN

Language Environment heap storage services.

LINK The LINK is recognized as an enclave boundary for compatibility. A call of DFSORT requires a LINK SVC to be issued. Special action needs to be taken to support DFSORT calls.

XCTL Language Environment does not provide an exact equivalent substitute function. The use of CEEPLOD and a BALR entry (with additional logic) should be sufficient to support XCTL.

LOAD

Language Environment program management CEEPLOD.

DELETE

Language Environment program management CEEPDEL.

ABEND

Language Environment CEE3ABND service is an alternative. Signaling (through CEESGL) a condition is a preferred method.

SPIE The entire Language Environment condition management scheme minimizes the need to issue a SPIE. Issuing a SPIE directly interferes with Language Environment's operation. Many aspects of Language Environment's condition management can be used to advantage, such as the shunt service, the enablement phase, and default handling.

STAE The entire Language Environment condition management scheme minimizes the need to issue a STAE. Issuing a STAE directly interferes with Language Environment's operation. Many aspects of Language Environment's condition management can be used to advantage, such as the shunt service, the enablement phase, and default handling.

STAX The entire Language Environment condition management scheme minimizes the need to issue a STAX. However, issuing a STAX might prove to be useful in some instances.

Structure of executable programs

An executable program that is fully Language Environment-enabled contains Language Environment-generated compiled code and the following additional constructs:

CEESTART

This CSECT, pointed to by PPA2, contains an external adcon for an externals table for various pieces of information concerning the executable program. See “CEESTART” on page 140 for details.

CEEBETBL

This CSECT, pointed to by CEESTART, contains various external adcons for the executable program. Two adcons are especially important: the address of the assembler user exit and the language list. See “CEEBETBL — Language Environment externals table” on page 148 for details.

CEEBLLST

This CSECT contains a series of weak external adcons for the signature CSECTs the HLLs produce. If a particular slot in this vector of adcons is nonzero, a member has been identified as being present in the executable program. See “CEEBLLST — language list” on page 149 for details.

CEEINT

This is the Language Environment bootstrap routine. It's main function of this routine is to dynamically load the Language Environment initialization routines and then transfer control over to the routine that does the work of bringing up Language Environment. See “CEEINT interface” on page 153 for details.

Central control blocks

The Language Environment program model describes four levels: region, process, enclave, and thread. A control block is established to manage each of these levels:

RCB Region Control Block

PCB Process Control Block

EDB Enclave Data Block

CAA Thread level resource (Common Anchor Area)

Each HLL is enrolled as a member within the Language Environment environment. The list of members is known as the *member list*. This is an array of structures, indexed by the member ID, that contains member-specific information, and the address of an event handler. Two member lists are maintained within Language Environment: one at the process level, the other member list is found at the enclave level. Each slot within either member list for any given member has the same format.

The event handler for each HLL that is represented in the executable program is loaded during Language Environment-initialization, and its address saved in the appropriate slot in the member list. When certain events occur, the event handler is called to either notify the member that the event has occurred, or to solicit some information from the member.

To identify those member languages that are represented in the executable program, each member language has to generate a *signature CSECT* whose name is unique to that particular member language. Checking for the presence of this uniquely named CSECT, one can determine if the member language is represented. Language Environment performs this check during initialization. Language Environment maintains a list of weak external address constants, which can be found through the CEESTART CSECT.

Event handler

The *event handler* is a member-supplied routine that is called at various times as a program runs when a significant event has occurred, or when the environment needs some information that is held by the member. During environment initialization, Language Environment determines the set of members present in the application and loads the event handler for each member language. The name of the event handler is manufactured by concatenating a fixed prefix and the member ID. The name constructed is CEEEV mmm , where mmm is the member number. The address of the event handler is saved for later retrieval. The values for mmm are in Figure 15 on page 19.

Linkage to the member event handler is through BALR 14,15, and R1 contains a standard parameter address list. The first parameter always indicates the type of event for which the event handler has been called. Additional parameters are dependent upon the specific event.

The event handler places the following return codes (in decimal) in R15:

- 4 The event handler does not want to process the event.
- 0 The event handler was successful.
- 16 The event handler encountered an unrecoverable error.

All environment services are available during the handling of the event, including the stack and heap, except for the options event and the main-opts event. This is Event Code 4, which is called by environment initialization to allow compatibility processing of runtime options.

With the exception of the CAA, PCB, process member list (MEML), and anchor vector, Language Environment can allocate control blocks above the line. Any member code that accesses a Language Environment control block must run in AMODE(31) to have addressability to the control blocks.

The event handler is called with the dump Event Code, 7, while processing various dump services. The dump event code has a function code that describes which dump service is to be performed. The remaining parameters for the dump event vary according to the specific sub-function code. See “Event code 7 — dump event handler event” on page 492 for more information.

Language utilities function 6 has a sub-function code that describes what information is being requested. The remaining parameters for the utilities event vary according to the specific function code. For details, see “Event code 6 — event handler utilities event” on page 487.

Event handler calls

The following sections describe event handler calls. The calls are used in all environments, unless otherwise noted.

Event code 1 — handle condition represented by the CIB event

Syntax

Call CEEEVnnn (1, *ceecib*, *results*, *new_condition*)

```
void      *ceecib;
INT4     *results;
FEEDBACK *new_condition;
```

ceecib (input)

The CEECIB for which the condition handler is being called. This value is passed by reference. Part of the CEECIB is the *condition_token* and the machine environment for the procedure in which the condition occurred. (For more details, see “Language Environment condition information block” on page 283.)

results (output)

Contains the instructions indicating the actions that the language-specific handler wants the Language Environment condition manager to take as a result of processing the condition. This field is passed by reference. The following are valid responses:

Response	<i>results</i> value	Description
resume	10	Resume at the resume cursor (condition has been handled).
percolate	20	Percolate to the next condition handler.
	21	Percolate to the first user condition handler for the next stack frame. (This can skip a language-specific exception handler for this stack frame as well as the remaining user condition handlers in the queue at this stack frame.)
promote	30	Promote to the next condition handler.
	31	Promote to the next stack frame. (This can skip a language-specific exception handler for this stack frame as well as any remaining user condition handlers in the queue at this stack frame.)
	32	Promote and restart condition handling with the first condition handler for the stack frame that is denoted by the handler cursor location.
	33	Promote and restart condition handling with the first condition handler for the stack frame that is denoted by the resume cursor location.
enablement	40	Ignore the condition; the thread is resumed where interrupted.
	41	Enable the condition for condition handling.
	42	Enable the condition and transform the condition (using the <i>new_condition</i> parameter).
percolate enablement	50	Percolate the enablement to the calling stack frame.
	51	Transform the condition (using the <i>new_condition</i> parameter) and percolate the enablement to the calling stack frame.

***new_condition* (output)**

The new condition token representing the promoted condition. This field is used only for *result* values that denote *promote*.

Usage notes

- For a description of the calling method, see “Language Environment member list and event handler” on page 83.
- It is not valid to promote a condition without returning a new condition token. If the original condition is returned in *new_condition*, the condition manager acts as if 20 had been specified as the *results* value.
- Prior to a condition being promoted, the Message Insert Block (MIB) must be populated with the new inserts for the promoted condition if necessary.
- The language-specific handlers are automatically established by stack frame. The Language Environment condition manager determines the language associated with a given stack frame, and then calls the event handler with the appropriate event code for enablement, condition handling, or condition handling for stack frame zero.
- The language-specific handlers are automatically disestablished when the stack frame is popped off the stack either using a return, a GOTO out of block, or moving the resume cursor.
- If a *resume* is requested, the member that owns the target stack frame is called immediately prior to passing control to the target stack frame. For details, see “Event code 10 — resume from a condition handler event” on page 497.

Event code 2 — perform enablement for this stack frame event**Syntax**

Call CEEEVnnn (2, *ceecib*, *results*, *new_condition*)

```
void      *ceecib;
INT4     *results;
FEEDBACK *new_condition;
```

***ceecib* (input)**

The CEECIB for which the condition handler is being called. This value is passed by reference. Part of the CEECIB is the *condition_token* and the machine environment for the procedure in which the condition occurred. (For more details, see “Language Environment condition information block” on page 283.)

***results* (output)**

Contains the instructions indicating the actions the language-specific handler wants the Language Environment condition manager to take as a result of processing the condition. This field is passed by reference. The following are valid responses:

Response	<i>results</i> value	Description
resume	10	Resume at the resume cursor (condition has been handled).

Event Code 2

Response	results value	Description
percolate	20	Percolate to the next condition handler.
	21	Percolate to the first user condition handler for the next stack frame. (This can skip a language-specific exception handler for this stack frame as well as the remaining user condition handlers in the queue at this stack frame.)
promote	30	Promote to the next condition handler.
	31	Promote to the next stack frame. (This can skip a language-specific exception handler for this stack frame as well as any remaining user condition handlers in the queue at this stack frame.)
enablement	40	Ignore the condition; the thread is resumed where interrupted.
	41	Enable the condition for condition handling.
	42	Enable the condition and transform the condition (using the <i>new_condition</i> parameter).
percolate enablement	50	Percolate the enablement to the calling stack frame.
	51	Transform the condition (using the <i>new_condition</i> parameter) and percolate the enablement to the calling stack frame.

new_condition (output)

The new condition token representing the promoted condition. This field is used only for *result* values that denote *promote*.

Usage notes

- For a description of the calling method, see “Language Environment member list and event handler” on page 83.
- It is invalid to promote a condition without returning a new condition token. If the original condition is returned in *new_condition*, the condition manager acts as if a *result* of 20 had been specified.
- Prior to a condition being promoted, the MIB must be populated with the new inserts for the promoted condition if necessary.
- The language-specific handlers are automatically established by stack frame. The Language Environment condition manager determines the language associated with a given stack frame, and then calls the event handler with the appropriate event code for enablement, condition handling, or condition handling for stack frame zero.
- The language-specific handlers are automatically disestablished when the stack frame is popped off the stack either using a return, a GOTO out of block, or moving the resume cursor.
- If a *resume* is requested, the member that owns the target stack frame is called immediately prior to passing control to the target stack frame. For details, see “Event code 10 — resume from a condition handler event” on page 497.

Event code 3 — handle condition according to language defaults event

Syntax

Call CEEEVnnn (3, *ceecib*, *results*, *new_condition*)

```
void      *ceecib;
INT4     *results;
FEEDBACK *new_condition;
```

ceecib (input)

The CEECIB for which the condition handler is being called. This value is passed by reference. Part of the CEECIB is the *condition_token* and the machine environment for the procedure in which the condition occurred. (For more details, see “Language Environment condition information block” on page 283.)

results (output)

Contains the instructions indicating the actions the language-specific handler wants the Language Environment condition manager to take as a result of processing the condition. This field is passed by reference. The following are valid responses:

Response	<i>results</i> value	Description
resume	10	Resume at the resume cursor (condition has been handled).
percolate	20	Percolate to the next condition handler.
	21	Percolate to the first user condition handler for the next stack frame. (This can skip a language-specific exception handler for this stack frame as well as the remaining user condition handlers in the queue at this stack frame.)
promote	30	Promote to the next condition handler.
	31	Promote to the next stack frame. (This can skip a language-specific exception handler for this stack frame as well as any remaining user condition handlers in the queue at this stack frame.)
	33	Promote and restart condition handling for the first condition handler for the stack frame that is denoted by the resume cursor location.
enablement	40	Ignore the condition; the thread is resumed where interrupted.
	41	Enable the condition for condition handling.
	42	Enable the condition and transform the condition (using the <i>new_condition</i> parameter).
percolate enablement	50	Percolate the enablement to the calling stack frame.
	51	Transform the condition (using the <i>new_condition</i> parameter) and percolate the enablement to the calling stack frame.

new_condition (output)

The new condition token representing the promoted condition. This field is used only for *result* values that denote *promote*.

Usage notes

- For a description of the calling method, see “Language Environment member list and event handler” on page 83.
- It is invalid to promote a condition without returning a new condition token. If the original condition is returned in *new_condition*, the condition manager acts as if a *result* of 20 had been specified.
- Prior to a condition being promoted, the MIB must be populated with the new inserts for the promoted condition if necessary.
- The language-specific handlers are automatically established by stack frame. The Language Environment condition manager determines the language associated with a given stack frame, and then calls the event handler with the appropriate event code for enablement, condition handling, or condition handling for stack frame zero.
- The language-specific handlers are automatically disestablished when the stack frame is popped off the stack either using a return, a GOTO out of block, or moving the resume cursor.
- If a *resume* is requested, the member that owns the target stack frame is called immediately prior to passing control to the target stack frame. For details, see “Event code 10 — resume from a condition handler event” on page 497.

Event code 4 — runtime options event**Purpose**

This event has limited capabilities; no Language Environment callable services are available. The purpose is to allow the members to handle runtime options in a compatible fashion.

Syntax

```
Call CEEEVnnn (4, ocb_addr, ceestart_addr, inpl_addr, work_area)
```

```
POINTER *ocb_addr;
POINTER *ceestart_addr;
POINTER *inpl_addr;
POINTER *work_area;
```

ocb_addr (input)

The address of an OCB

ceestart_addr (input)

The address of CEESTART

inpl_addr (input)

The address of the main entry point

work_area (input)

The address of a 512-byte work area

Usage notes

- This event is not called if a CEEUOPT CSECT is found in the load module, or if more than one member is present in the load module.
- Only the member identified by the member ID in the INPL is called.

Event code 5 — main-opts event

Purpose

If INPL is control level 0 and the number of words indicate 6, or the control level is 1 and the `invoke_mainopts` flag is set, then the event handler whose member ID is found in the INPL is called requesting the main-opts word to be dynamically completed. If the event does not produce a main-opts word (indicated by returning a -4 in R15 or leaving the main-opts word unaltered), the following characteristics are assumed for the main-opts word: PLIST(HOST) and EXECOPS.

Syntax

```
Call CEEEVnnn (5, inpl_addr, R13_addr, R0_addr, R1_addr, main_opt_addr)
POINTER *inpl_addr;
POINTER *R13_addr;
POINTER *R0_addr;
POINTER *R1_addr;
POINTER *main_opt_addr;
```

inpl_addr (input)

The INPL passed to CEEINT or CEEP#INT or the INPL generated by CEEPIPI.

R13_addr (input)

A fullword containing the R13 value passed into CEEINT or CEEP#INT or CEEPIPI (`call_main`).

R0_addr (input)

A fullword containing the R0 value passed into CEEINT or zero when CEEP#INT or CEEPIPI (`init_main`) are the enclave initialization method.

R1_addr (input)

A fullword containing the R1 value passed into CEEINT or zero when CEEP#INT or CEEPIPI (`init_main`) are the enclave initialization method.

main_opt_addr (input)

The main-opts word.

Usage notes

- This event is invoked while processing a CEEPIPI (`call_main`) and calling a program with the any of the following:
 - The entry point style is not CEESTART
 - The entry point style is CEESTART and CEEMAIN is old format
 - The entry point style is CEESTART,CEEMAIN is new format, the INPL is new format, and the main-opts word is not valid
- A fixed size stack is available for use during this event.

Event code 6 — event handler utilities event

Purpose

Various Language Environment services, including exception handling, perform language-specific functions. To perform these functions, Language Environment receives information through the member language utility exit. The utility exit passes Language Environment the information it needs to perform the required processing. It is a part of a member event handler, using Event Code 6. Language Environment Dump Processing makes calls to member event handler 6 (Utilities).

Event Code 6

Many of these calls provide a DSA address as input and expect the member to provide information about the routine that owns the stack frame. The description and linkage to the event handler for each of these exits is shown below. All linkages have the event code of 6, followed by a unique function code, followed by parameters specific to the utility.

This event code performs several functions based upon the *function_code* passed as the second parameter. There are four *function_codes* for which the DSA address is passed as input:

- 1 DSA Ownership
- 2 Entry Point and Compile Unit Identification
- 3 Statement Identification
- 4 DSA Classification

Syntax

For the **DSA Ownership** exit, a member language specifies if a DSA is associated with a routine that it owns, or a routine that is written in that language. Language Environment uses this exit to determine the owner of code that does not have a PPA-style entry. Language Environment first checks to see if the code contains a PPA-style entry. The eye catcher of the saved R15 in the caller's DSA is checked to determine if it points to a Language Environment entry point. If this is not true, Language Environment calls member language exits for DSA ownership until a language claims ownership.

```
Call CEEEVnnn (6, 1, dsaptr, ownership, dsa_format)
POINTER *dsaptr;
INT4    *ownership;
INT4    *dsa_format;
```

dsaptr (input)

A fullword pointer to an active DSA or save area.

ownership (output)

A fullword binary integer set to contain:

- 0 The source code corresponding to the DSA is not in the member language.
- 1 The source code corresponding to the DSA is in the member language.

dsa_format (input)

A fullword binary integer set to one of the following:

- 0 The format of the DSA is a Standard OS linkage register save area (with/without Language Environment fields including NAB).
- 1 The format of the DSA is XPLINK style.

For the **Entry Point and Compile Unit Identification** exit, a member language identifies the entry point name, entry address, compile unit name, compile unit address, and current instruction address for a routine, given the DSA, CAA, and CIB associated with the routine. This exit is called only if a routine does not have a PPA-style entry.

Call CEEEVnnn (6, 2, *dsaptr*, *cibptr*, *compile_unit_name*, *compile_unit_name_length*, *compile_unit_address*, *entry_name*, *entry_name_length*, *entry_address*, *call_instruction_address*, *caaptr*, *dsa_format*)

```

POINTER *dsaptr;
POINTER *cibptr;
CHAR    *compile_unit_name;
INT4    *compile_unit_name_length;
POINTER *compile_unit_address;
CHAR    *entry_name;
INT     *entry_name_length;
POINTER *entry_address;
POINTER *call_instruction_address;
POINTER *caaptr;
INT4    *dsa_format;

```

***dsaptr* (input)**

A fullword pointer to an active DSA or save area.

***cibptr* (input)**

A fullword pointer to the CIB for the current condition, if one exists. Otherwise, this parameter is zero.

***compile_unit_name* (output)**

A fixed-length character string of arbitrary length to contain the name of the compile unit containing the routine associated with the DSA. If the compile unit name cannot be determined, this parameter should be set to all blanks. If the compile unit name cannot fit within the supplied string, it should be truncated. (Truncation of DBCS should preserve even byte count and SI/SO pairing.)

***compile_unit_name_length* (output)**

A fullword binary integer containing the length of the compile unit name string on entry and to contain the actual length of the compile unit name placed in the string on exit. If the compile unit name cannot be determined, this parameter should be set to zero. The maximum length a string can be is 256 bytes.

***compile_unit_address* (output)**

A fullword binary integer to contain the address of the start of the compile unit. If the compile unit address cannot be determined, this parameter should be set to zero.

***entry_name* (output)**

A fixed-length character string of arbitrary length to contain the name of the entry point into the routine associated with the DSA. If the entry point name cannot be determined, this parameter should be set to all blanks. If the entry point name cannot fit within the supplied string, it should be truncated. (Truncation of DBCS should preserve even byte count and SI/SO pairing.)

***entry_name_length* (output)**

A fullword binary integer containing the length of the entry point name string on entry and to contain the actual length of the entry point name placed in the string on exit. If the entry point name cannot be determined, this parameter should be set to zero. The maximum length a string can be is 256 bytes.

***entry_address* (output)**

A fullword binary integer to contain the address of the entry point. If the entry address cannot be determined, this parameter should be set to zero.

Event Code 6

call_instruction_address (output)

A fullword binary integer to contain the address of the instruction which transferred control out of the routine. This should either be the address of a calling instruction, such as BALR or BASSM, or the address of an interrupted instruction if control was transferred due to an exception. If the address cannot be determined, this parameter should be set to zero.

caaptr (input)

A fullword pointer to the CAA for the enclave associated with the DSA.

dsa_format (input)

A fullword binary integer set to one of the following:

- 0 The format of the DSA is a Standard OS linkage register save area (with or without Language Environment fields including NAB).
- 1 The format of the DSA is XPLINK style.

For this exit, **Statement Identification**, a member language identifies the statement number given an instruction address and the entry address into a routine. Also, the address of the DSA for the routine and the address of the CIB for the routine are passed, in case current register contents are also needed to determine the statement number.

```
Call CEEEVnnn (6, 3, entry_address, call_instruction_address, dsaptr, cibptr, statement_id,
statement_id_length, dsa_format)
POINTER *entry_address;
POINTER *call_instruction_address;
POINTER *dsaptr;
POINTER *cibptr;
CHAR *statement_id;
INT4 *statement_id_length;
INT4 *dsa_format;
```

entry_address (input)

A fullword binary integer containing the address of an entry point into the routine.

call_instruction_address (input)

A fullword binary integer containing the address of an instruction in the statement to be identified. Note that this can also be the address of an instruction in a small routine that does not have its own DSA (for example, fetch glue code). In such cases, the small routine is considered an extension of the code for the statement which called the routine. In these cases, the member language should pass back the statement number of the caller of the small routine.

dsaptr (input)

A fullword pointer containing the address of the DSA for the routine.

cibptr (input)

A fullword pointer containing the address of the CIB for the current condition. If there is no CIB, this parameter is zero.

statement_id (output)

A fixed-length character string of arbitrary length to contain the statement identifier of the instruction pointed to by *call_instruction_address*. If the statement cannot be determined, this parameter should be set to all blanks. If the statement ID cannot fit within the supplied string, it should be truncated. (Truncation of DBCS must preserve even byte count and SI/SO pairing.)

statement_id_length (output)

A fullword binary integer containing the length of the statement id string on entry and the actual length of the statement id placed in the string on exit. If the statement ID cannot be determined, this parameter should be set to zero. The maximum length a string can be is 256 bytes.

dsa_format (input)

A fullword binary integer set to one of the following:

- 0 The format of the DSA is a Standard OS linkage register save area (with/without Language Environment fields including NAB).
- 1 The format of the DSA is XPLINK style.

For this exit, **DSA Classification**, a member language identifies the type of DSA that is associated with the procedure.

Call CEEEVnnn (6, 4, dsaptr, class, dsa_format)

```
POINTER *dsaptr;
INT4    *class;
INT4    *dsa_format;
```

dsaptr (input)

A fullword pointer containing the address of the DSA or save area.

class (output)

A fixed-binary(31) fullword passed by reference indicating the classification of the passed DSA. The following is the format of the returned fullword. It can be quickly checked to distinguish library code from compiled code, identify the member, and allow the members to qualify the type of compiled/library if needed.

X'abcd yzzz'

zz - is the member ID with a max of X'FF'
yy - is used by the member to qualify the compiled code type or the library code type
d - 1 if library code; 2 if compiled code

dsa_format (input)

A fullword binary integer set to one of the following:

- 0 The format of the DSA is a Standard OS linkage register save area (with or without Language Environment fields including NAB).
- 1 The format of the DSA is XPLINK style.

The following list is the set of acceptable return values.

Return value		
X'0001 0001'	Library routine	Language Environment
X'0001 0003'	Library routine	C/C++
X'0001 0005'	Library routine	COBOL
X'0001 0006'	Library routine	Debug Tool
X'0002 0003'	Compiled code	C/370 or C/C++
X'0002 0005'	Compiled code	COBOL
X'0002 0006'	Compiled code	Debug Tool

Event code 7 — dump event handler event

Purpose

CEL Dump Processing makes calls to member event handlers for Event Code 7 (Dump). Many of these calls provide a DSA address as input and expect the member to provide information about the routine that owns the stack frame. This event handler code performs several CEEDUMP related functions based upon the *function_code* passed as the second parameter. There are five *function_codes* for which the DSA address is passed as input:

- 2 Dump arguments of a routine.
- 3 Dump variables of a routine.
- 4 Dump control blocks associated with a routine.
- 5 Dump storage for a routine.
- 18 Dump condition information for DSA/CIB.

Syntax

Calls to dump event handler are made with parameters shown in the following sample procedure statement

```
Call CEEEVnnn (7, function_code, additional_parms, fc, entry_ptr, dsa_format, call_addr)
```

```
INT4      *function_code;
POINTER   *dsaptr;
POINTER   *cibptr;
POINTER   *caaptr;
POINTER   *edbptr;
POINTER   *pcbptr;
FEEDBACK  *fc;
POINTER   *entry_ptr;
INT4      *dsa_format;
POINTER   *call_addr;
```

function_code (input)

A fullword binary integer that specifies the dump function to be performed. It must contain one of the following values:

- 1 Dump an informational message to explain why the dump is being taken. This *function_code* specifies that the exit of the language library that called CEE3DMP print the error message that resulted from the dump being taken in the first place. The informational messages would normally be a copy of the error messages sent to MSGFILE for the error. These messages could contain an ABEND code, the PSW, and register contents at time of the error. If CEE3DMP was not called by a member language library, member language libraries would normally not print any messages in this exit.
- 2 Dump the arguments of a routine. If the member language cannot distinguish between arguments and local variables for a routine, it should dump the arguments at the same time it is called by dump services to dump variables.
- 3 Dump the variables of a routine. This includes all local variables and any shared external variables used by the routine. Member language libraries should dump only those variables used or set by the routine if this can be determined.
- 4 Dump control blocks associated with a routine. This includes the DSA

mapped by the member language and any other control blocks associated with the routine that are useful for debugging. This includes compile information, symbol tables, and statement tables.

- 5 Dump storage for a routine. This includes automatic stack frame storage and static local variable storage. Static data storage shared between this routine and another routine should also be dumped. Only one copy of a shared storage area should be dumped though.
- 6 Dump control blocks associated with a thread. The CAA for the thread is dumped by Language Environment.
- 7 Dump storage associated with a thread. Language Environment dumps all stack storage associated with the thread. Member languages can dump any other stack storage that is associated with the thread using this exit. Any stack storage used by the thread is dumped even though it can not be associated with it. Only data storage should be dumped. Storage containing code should not be dumped if possible.
- 8 Dump control blocks associated with an enclave. The EDB for the enclave is dumped by Language Environment as well as the member list. Member languages should dump communications areas that are linked off of the member list. These are usually the static library communications regions that are part of the application load module.
- 9 Dump storage associated with an enclave. Language Environment dumps all heap storage associated with the enclave. Member languages can dump any other storage that is associated with the enclave using this exit. This usually includes storage obtained through direct calls to the operating system storage management. Only data storage should be dumped. Storage containing code should not be dumped if possible.
- 10 Dump status and attributes of files. Language Environment dumps the status and attributes of files used by message services. Member languages should dump status and attributes of their own files. This includes all currently open files as well as any previously open files in the course of running an application.
- 11 Dump control blocks associated with files. Control blocks and other language-specific control blocks that keep file status are dumped.
- 12 Dump storage buffers associated with files. These buffers are allocated by the operating system and typically do not use Language Environment heap services. Buffer storage allocated by Language Environment heap services can be dumped.
- 13 Dump control blocks associated with the process. The PCB for the process is dumped by Language Environment.
- 14 Dump storage associated with the process. Only data storage should be dumped. Storage containing code should not be dumped.
- 15 Dump any additional global information. This information appears at the end of the dump report. A list of loaded library modules is an example of additional global information.
- 16 Dump the variables of the enclave. This includes all static external variables used by the enclave.
- 17 End of dump call. This indicates that there are no additional calls to the event handler for this instance of dump.
- 18 Dump condition information associated with the passed DSA/CIB

Event Code 7

pointer. For example, PL/I displays ONCHAR, ONSOURCE, ONKEY, and ONCOUNT values when applicable.

additional_parms (input)

Parameters specific to a certain function code. Notice that the `dump_event_code 7` always precedes the function code. See Figure 109 to view the syntax.

```

Call CEEEVnnn (7, 1, fc)
Call CEEEVnnn (7, 2, dsaptr, cibptr, caaptr, fc, entry_ptr, dsa_format, call_addr)
Call CEEEVnnn (7, 3, dsaptr, cibptr, caaptr, fc, entry_ptr, dsa_format, call_addr)
Call CEEEVnnn (7, 4, dsaptr, cibptr, caaptr, fc, entry_ptr, dsa_format, call_addr)
Call CEEEVnnn (7, 5, dsaptr, cibptr, caaptr, fc, entry_ptr, dsa_format, call_addr)
Call CEEEVnnn (7, 6, caaptr, fc)
Call CEEEVnnn (7, 7, caaptr, fc)
Call CEEEVnnn (7, 8, edbptr, fc)
Call CEEEVnnn (7, 9, edbptr, fc)
Call CEEEVnnn (7, 10, edbptr, fc, caaptr)
Call CEEEVnnn (7, 11, edbptr, fc, caaptr)
Call CEEEVnnn (7, 12, edbptr, fc, caaptr)
Call CEEEVnnn (7, 13, pcbptr, fc)
Call CEEEVnnn (7, 14, pcbptr, fc)
Call CEEEVnnn (7, 15, edbptr, fc)
Call CEEEVnnn (7, 16, edbptr, fc)
Call CEEEVnnn (7, 17, fc)
Call CEEEVnnn (7, 18, dsaptr, cibptr, caaptr, fc, entry_ptr, dsa_format, call_addr)

```

Figure 109. Syntax by function_code

dsaptr (input)

A fullword binary integer containing the address of a DSA.

cibptr (input)

A fullword binary integer containing the address of the CIB for the routine. This parameter is zero if the routine does not have a CIB.

caaptr (input)

A fullword binary integer containing the address of a CAA.

edbptr (input)

A fullword binary integer containing the address of an EDB.

pcbptr (input)

A fullword binary integer containing the address of a PCB.

fc (output)

A 12-byte feedback code passed by reference. The following symbolic conditions might result from this exit:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE30V	Severity	3
	Msg_No	3103
	Message	An error occurred in writing messages to the dump file.

Condition		
CEE311	Severity	3
	Msg_No	3105
	Message	Member language dump exit was unsuccessful.

***entry_ptr* (input)**

A fullword pointer containing the address of the entry point for the routine that owns the stack frame.

***dsa_format* (input)**

A fullword binary integer set to one of the following:

- 0** The format of the DSA is a Standard OS linkage register save area (with/without Language Environment fields including NAB).
- 1** The format of the DSA is XPLINK style.

***call_addr* (input)**

A fullword pointer containing the address of the instruction that caused transfer out of the routine that owns the stack frame. If the calling instruction cannot be determined, then the value is zero. This is either the address of the BASR, BALR or BASSM instruction if transfer was made by a subroutine call or the address of the interrupted statement if the transfer was caused by an exception.

Event code 8 — new load module event

Purpose

This event notifies the members that a new executable program (load module or program object) was introduced to the enclave. This function is intended to be used when a high level language performs a dynamic call. It is also used when a DLL is first loaded, either by an explicit reference (for example, by using the C `dllload()` function) or by an implicit reference.

Syntax

Call CEEEVnnn (8, *entry_ptr*, *ceestart_ptr*, *idinfo*, *wsa*, *loadinfo*)

```
POINTER *entry_ptr;
POINTER *ceestart_ptr;
INT4 *idinfo;
POINTER *wsa;
POINTER *loadinfo;
```

***entry_ptr* (input)**

Language Environment only recognizes the following *entry_point* styles:

- C/370-style PPA
- Language Environment routine entry layouts (see “Routine layout” on page 6.)
- Language Environment-format CEESTART
- Language Environment callable service stubs

***ceestart_ptr* (input)**

CEESTART, if Language Environment recognized the executable program, or zero if Language Environment did not recognize the executable program.

Event Code 8

idinfo (input)

A fullword that gives the member language additional information about the calling environment. A new executable program is introduced into the enclave by a COBOL dynamic call, PL/I or C/C++ fetch, CEEPIPI services, and DLL implicit or explicit load. The following bits are defined:

0–23 Reserved

24–31 Load_reason. The following values indicate the reason the executable program was loaded:

- 1 The program was loaded because of a dynamic call, fetch, or CEEPIPI service. In this case, Language Environment does not preserve the writable static area (WSA) address because each of the members has defined ILC interfaces to support usage of the WSA, as required by the member languages. For example, a C/C++ fetch() call will obtain a WSA for every request, while a COBOL dynamic call will obtain a WSA area only for the first request.
- 2 The program was loaded due to the explicit or implicit reference of a DLL. The member should be obtained and fully-initialize the WSA. Language Environment will preserve the WSA address that is returned by the member to complete the initialization required before making a DLL available for use by the requesting DLL application.

wsa (input/output)

A fullword that contains the address of the WSA that was obtained and initialized by the member. If the *idinfo* indicates that the executable unit is a DLL, Language Environment will save the WSA and provide it to the DLL when the DLL is invoked. This parameter is initially set to zero when it is passed to a member language's event handler. It will contain the WSA address on subsequent calls when the WSA address is return by a member language.

loadinfo (output)

A fullword that will be passed as input to the DLL Initialization (22) and Static Constructor (25) events. It can be used to pass information to these events about the module that was just loaded.

Event code 9 — new condition event

Purpose

This event notifies all members that a new condition is about to be processed. This can be used by members to keep sets of condition handling related control areas concurrently with Language Environment condition handling.

Syntax

```
Call CEEEVnnn (9, function_code, cib_ptr)
```

```
INT4      *function_code;  
POINTER   *cib_ptr;
```

function_code (input)

A 31-bit fixed-binary field that describes the action that is being reported.

Value	Meaning
1	A new CIB is ready for processing.
2	An old CIB is collapsing.

***cib_ptr* (input)**

A pointer to the new or old CIB.

Usage notes

- The notification of the event handlers occurs after the condition has been enabled. This is done prior to the debug or any other event handler being called to process the new condition.
- The CIBs are treated as a stack. Even if CIBs go away due to move resume cursor this event notifies the members of the collapsing of the CIBs in LIFO order.
- If the event handler indicates an unrecoverable error with a 16 return code, condition management issues ABEND U4091-14.
- The CIB address can only be used as a token for the purpose of identifying which condition is ready for processing or is collapsing.

Event code 10 — resume from a condition handler event**Purpose**

This event code identifies that a resumption from a condition handler occurs within the *target_dsa*.

Syntax

```
Call CEEEVnnn (10, target_dsa, target_dsa_fmt, ph_callee_dsa, ph_callee_dsa_fmt)
```

```
void      *target_dsa;
INT4      *target_dsa_fmt;
void      *ph_callee_dsa;
INT4      *ph_callee_dsa_fmt;
```

***target_dsa* (input)**

The DSA that is the target for the resume.

***target_dsa_fmt* (input)**

The format of the DSA pointed to by *target_dsa*. Possible values are:

```
0      non-XPLINK
1      XPLINK
```

***ph_callee_dsa* (input)**

A pointer to the DSA of the routine called by the routine owning the DSA pointed to by *target_dsa*.

***ph_callee_dsa_fmt* (input)**

The format of the DSA pointed to by *ph_callee_dsa*. Possible values are:

```
0      non-XPLINK
1      XPLINK
```

Usage notes

- The Language Environment condition manager determines the member that owns the stack frame that is the target of the resume. Once determined,

Event Code 10

Language Environment condition manager calls the particular member's event handler just prior to performing the resume operation into the stack frame.

- It is the member's responsibility to perform the necessary actions to allow the resume to occur within the *target_dsa*.
- The *ph_callee_dsa* parameter is provided in case the event handler needs to extract registers from the DSA pointed to by *target_dsa*. Registers which are saved in the DSA pointed to by *target_dsa* for non-XPLINK are mostly saved in the DSA pointed to by *ph_callee_dsa*, if *target_dsa_fmt* is XPLINK. Note that *ph_callee_dsa_fmt* might not be the same as *target_dsa_fmt*. Also, the DSA pointed to by *ph_callee_dsa* may belong to a Language Environment transition or Language Environment overflow routine.

Event code 11 — DSA exit routines event

Purpose

An exit routine can be used to perform activities on behalf of a stack frame when the stack is being collapsed as the result of a return from a main, an immediate STOP request, a GOTO out of block, or a move resume cursor request. Exit routines allow for activities such as the closing of files and releasing of system resources that are held.

Members not requiring Exit DSAs may, for performance reasons, request that this processing be disabled. This applies to normal, or non-abend, enclave terminations initiated by a call to the CEETREN or CEETREC services. This is implemented with a parameter used on the Enclave Initialization Event, Event Code 18. Refer to this event for more information on enabling this feature. When this feature is on, the traverse of the stack for exit DSA routines is not executed and the DSA Exit event call is skipped. If multiple language members are present in an enclave, all must indicate that the DSA Exit scan may be skipped. Stack traverse and DSA Exit processing continues to occur for terminations with an abend pending or a GOTO out of block or move resume cursor request if the feature is enabled or not.

An exit routine is established in one of two mechanisms, as described below. The exit routine has two different interfaces depending upon the mechanism used to establish the exit.

- The stack frame (DSA) is marked as requiring DSA exit processing by flags set within the DSA.
- The PPA1 has the exit DSA flag on.

Syntax

```
Call CEEEVnnn (11, dsa_addr, dsa_fmt, ph_callee_dsa_addr, ph_callee_dsa_fmt)
```

```
POINTER    *dsa_addr;  
INT4       *dsa_fmt;  
POINTER    *ph_callee_dsa_addr;  
INT4       *ph_callee_dsa_fmt;
```

dsa_addr (input)

The address of the DSA that is being abnormally collapsed (nonreturn style).

dsa_fmt (input)

The format of the DSA pointed to by *dsa_addr*. Possible values are:

```
0          non-XPLINK  
1          XPLINK
```

***ph_callee_dsa_addr* (input)**

A pointer to the DSA of the routine called by the routine owning the DSA pointed to by *dsa_addr*.

***ph_callee_dsa_fmt* (input)**

The format of the DSA pointed to by *ph_callee_dsa_addr*. Possible values are:

0 non-XPLINK
1 XPLINK

Usage notes

- The HLL event handler is called with the Event Code 11 and the address of the stack frame that is being popped off the stack.
- If conditions arise while running they should be signaled to the Language Environment condition manager.
- No condition token is provided to the exit routine. It is assumed that the exit routine completed without error whenever it returns to the Language Environment condition manager.
- The *ph_callee_dsa_addr* parameter is provided in case the event handler needs to extract registers from the DSA pointed to by *dsa_addr*. Registers which are saved in the DSA pointed to by *dsa_addr* for non-XPLINK are mostly saved in the DSA pointed to by *ph_callee_dsa_addr*, if *dsa_fmt* is XPLINK. Note that *ph_callee_dsa_fmt* might not be the same as *dsa_fmt*. Also, the DSA pointed to by *ph_callee_dsa_addr* may belong to a Language Environment transition or Language Environment overflow routine.

Event code 12 — national language change event**Purpose**

This event notifies all members that the national language has been changed using Language Environment callable services. The members are notified before the options control block is updated.

Syntax

<p>Call CEEEVnnn (12, <i>nat_lang</i>) INT *<i>nat_lang</i>;</p>
--

***nat_lang* (input)**

The new national language (character(3)). For a list of supported national languages, see *z/OS Language Environment Programming Reference*.

Event code 13 — country code change event**Purpose**

This event notifies all members that the country code has been changed using Language Environment callable services. The members are notified before the options control block is updated.

Syntax

```
Call CEEEVnnn (13, country_code)
INT      *country_code;
```

country_code (input)

The new country code (character(2)). For a list of supported country codes, see *z/OS Language Environment Programming Reference*.

Event code 14 — main routine invocation event

Purpose

This event is called in both CICS and non-CICS environments to allow the member language to invoke the main program and handle normal return from the main program. This event allows member languages to apply language-specific semantics for main programs in cases where the language library does not gain control before Language Environment during initialization. This occurs in non-CICS environments when the environment is initialized by CEESTART, and is always the case in CICS environments.

Syntax

```
Call CEEEVnnn (14, mainaddr, amode, mainopts, apal, altentryaddr, XPLINKenvaddr)
POINTER  *mainaddr;
INT      *amode;
INT      *mainopts;
POINTER  *apal;
POINTER  *altentryaddr;
POINTER  *XPLINKenvaddr;
```

mainaddr (input)

The address of the main routine found either in CEEMAIN/PLIMAIN when CEESTART entered directly, or the PIPI table address when initialized through CEEPIPI (init_main) or the main address from the INPL or the alternate main in the EPL when being invoked through CEEP#CAL.

amode (input)

An AMODE indicator for the main program. This is a fullword with the amode in the least significant bit.

mainopts (input)

The main options word from the INPL.

apal (input)

The R1 value to be passed to the main routine. When CEESTART is executed directly, this is determined by either the call to CEECELVBPLST, the PLIST manipulation CWI, or CEEEDBDEFPLPTR field in the EDB. When starting execution from a CEEPIPI (call_main) function, this is the parm_pointer in the CEEPIPI parameter list. When starting execution from a CEEP#CAL function, this is the PLIST from the EPL.

altentryaddr (input)

An alternative entry address branched to if available. No assumptions can be made about the format of this entry point. Any initialization the event handler

needs to accomplish before branching to this address must still be performed using the `mainaddr` parameter. If an alternative entry address is not available, this parameter is NULL.

XPLINKenvaddr (input)

The address of the XPLINK-compiled main program's environment. If the main program is non-XPLINK, or doesn't have an environment, this parameter is NULL.

Usage notes

- This event is called in both CICS and non-CICS environments.
- Language Environment allocates a DSA in order to call the MAIN routine.
- The member event handler must perform any AMODE switching required to invoke the MAIN routine.
- When control returns from this event, Language Environment performs termination activities similar CEETREN.
- This call is made for CICS when the *invoke* parameter of the Enclave Initialization event (see page “Event code 18 — enclave initialization event” on page 503) was set to 1 by the language event handler for the language of the main, or when the *maininv_on* flag in INPL word 7 is set.
- This call is made for non-CICS when the *maininv_on* flag in INPL word 7 is set.
- Under CICS, the return code from the application program should be placed in CEECAACICSRSN before returning from this event.
- CICS SPF: Language Environment calls languages in the application key for this event. This key can be key 8 or key 9, depending on the EXECKEY setting for the application program in the PPT.
- CICS SPF: The parameters and storage areas pointed to by the parameters can potentially be in key 8 storage, with the exception of the *apal* parameter, which is in the application key.

Event code 15 — atterm event

Purpose

The atterm event is called during termination of an enclave. It is called after all user stack frames have been removed from the stack and prior to calling the members for the enclave termination event. Only the members that have been explicitly registered using the CWI CEEATTRM are called.

Syntax

Call CEEEVnnn (15)

The parameter list that is passed to this event consists of a single parameter, the Event Code 15.

Usage notes

- For more information on Language Environment return codes, reason codes, existing language semantics, processing, and conventions, see *z/OS Language Environment Debugging Guide*.

Event code 16 — Debug Tool event

Purpose

The Debug Tool event code is reserved for calls from member event handlers to the debugger. For more information, see the documentation supplied with the debugger.

Event code 17 — process initialization event

Purpose

Perform language-specific process initialization. This event is driven during preinitialization for main routines when the environment is being brought up during an INIT request. Application-specific initialization is left until a main routine is about to be called at a CALL request (see page “Event code 18 — enclave initialization event” on page 503). The members that are called with this event code are found by looking into the load modules that are passed in the PIPI table.

When library reuse is active, the specified *reuse_state* value indicates if this is the first process event call to a given member during the current reuse environment. In the reuse environment, Language Environment does not free the initial storage allocated for Language Environment control blocks or delete Language Environment modules or member event handlers between invocations of Language Environment-enabled programs. Library reuse will be active if a program uses LRR (Library Retention Routine) or if it is a medium-weight POSIX process.

At termination, all resources obtained through the service routine vector during process initialization **must** be released explicitly. Language Environment does not implicitly release any resource obtained during the process initialization event.

A combination of Event 17 and Event 18 should initialize the HLL-specific aspects of the environment for a given application. The counterpart for this event is Event 21.

Syntax

```
Call CEEEVnnn (17, reuse_state)
INT      *reuse_state;
```

reuse_state (input)

One of the following codes, which indicate if library reuse is active and if this is the first time in the current reuse environment that the event handler is called for process initialization.

- 0 Reuse is not in effect.
- 1 Reuse is in effect; this is the first call for process initialization.
- 2 Reuse is in effect; this is not the first call for process initialization.

Upon entry into the member event handler, the following is available:

- R13 points to a DSA into which the event handler is able to store its caller's registers.
- R12 is pointing to a simulated CAA allowing stack frame acquisition.
- A fixed size stack is available for use by the HLLs when called for process initialization. The stack size is 1024 bytes. There is no stack overflow support.

- The simulated CAA has a pointer to the PCB. The simulated CAA has a zero pointer to the EDB.
- R1 contains the address of a standard O/S style PLIST with a single parameter of Event Code 17.
- The addresses of LOAD and DELETE services and GETMAIN/FREEMAIN services are held in the PCB. It is the caller's responsibility to relinquish resources obtained at the process level.
- The format of the member list at the process level is of the same format as the member list at the enclave level.

Usage notes

- This event is called in both CICS and non-CICS environments.
- This event is called at most once during the execution of a CICS transaction. Member languages should initialize for the transaction during this call.
- This event is always called before enclave initialization for a member language. However, enclave initialization for other languages can precede process initialization for a language, if a subordinate enclave introduces a new language into the process.
- CICS SPF: Language Environment calls languages in key 8 for this event. Storage for parameters can be in key 8.

Event code 18 — enclave initialization event

Purpose

Perform language-specific enclave initialization. All language-specific initialization for the CICS run unit should be performed during this call.

Syntax

Call CEEEVnnn (18, *pgmmask*, *inpl*, *invoke*, *ioinfo*, *tolerate_newstk*, *idinfo*, *wsa*, *skippedsa*)

```

INT4      *pgmmask;
POINTER   *inpl;
INT4      invoke;
STRUCT    ioinfo;
INT4      *tolerate_newstk;
INT4      *idinfo;
POINTER   *wsa;
INT4      *skippedsa;

```

pgmmask (input/output)

A fullword containing the program mask in the right-most bits. This output program mask is ignored, when event 18 is called to initialize a member that appears only in the dependent member list of a signature CSECT in the language list."

inpl (input)

The initialization parameter list for the enclave.

invoke (output/CICS only)

A fullword that is set to indicate that Language Environment should call the member language to invoke the main procedure:

- 0 Language Environment should invoke the main procedure directly.
- 1 Language Environment should give control to the member language to invoke the main procedure.

Event Code 18

This parameter is initially set to zero and is used only under CICS. This parameter is only recognized for the member language whose main procedure is written in that language.

ioinfo (input/CICS only)

A structure describing the standard input, output, and error streams as defined by CICS. This parameter is only valid under CICS.

tolerate_newstk (input)

A fullword that indicates if the member language can support the performance enhancements to the stack extension routines. This parameter is initially set to zero when passed to the member language event handler. If the member language can tolerate the high-performance stack behavior, it should set this word to a nonzero value. If not, it should leave the value as zero. On return from the member event handler, Language Environment queries the value of the parameter and uses the appropriate stack handling code.

idinfo (input)

A fullword that indicates to the member language additional information that identifies the calling environment. Language Environment issues the enclave initialization event when a new common runtime environment is created for the set of members represented in an executable program and when an established environment needs to be augmented by adding additional members represented in a newly-loaded executable program. The following bits are defined:

- 0-7 Init_reason. The following values indicate the reason for the enclave initialization event.
 - 1 The initial build of the Language Environment. The reasons for this include: batch initialization, initialization for CEEPIPI, creation of nested enclave, and CICS run-unit initialization.
 - 2 The Language Environment was previously built and additional members need to be added to the existing environment. The reasons for this include: the dynamic call, fetch, adding routines to the CEEPIPI environment, or DLL load module which caused a load of an executable program that contains members that are new to the environment. In this case, Event Code 8 (see "Event code 8 — new load module event" on page 495) will follow to allow the member to obtain and initialize the WSA. Because Event Code 8 is always provided and Event Code 18 is only provided when new members are introduced into the environment, the WSA should be obtained once using Event Code 8.
- 8-15 dll_type. This value indicates if the executable program is a DLL; the values are defined as follows:
 - 0 The executable program is **not** a DLL; it is either a load module or a program object.
 - 1 The executable program is a DLL. This means it can export variables, functions, or both; optionally, the DLL can also import variables or functions.
- 17-31 Reserved; must be zero

wsa (input/output)

A fullword that contains the address of the member-obtained and initialized

WSA. If *idinfo* indicates that the executable program is a DLL, Language Environment will save the WSA address and provide it to the DLL when the DLL is invoked.

skippedsa (output)

A fullword that indicates if DSA Exit processing may be bypassed at normal, non-abend pending, enclave termination initiated by a call from the CEETREN or CEETREC services. The default is zero, which indicates DSA Exit processing should occur as previously at enclave termination. The member sets this fullword to a non-zero value to indicate it has no requirement for Exit DSA processing at normal enclave termination.

This event is used to initialize HLL portions at the enclave level. The order in which the member event handlers are driven is first based on the ascending order of the member ID. However, if the member ID is identified by a numerically lower ID in the dependencies part of the signature CSECT, then it could be called prior to a lower ID.

All Language Environment services are available at the time of this event. The member can influence the program mask setting by placing its requirements of the program mask in the second parameter as described below.

Upon entry into the member event handler for the enclave initialization event, the following is available:

- R1 contains the address of a standard O/S style PLIST (all of the parameters are passed by reference) with the following PLIST:
 1. Event code 18.
 2. Fullword field in which the program mask is held in the right-most bits; upon input, this field is zero.
 3. Initialization PLIST (INPL) passed to CEEINT.
 4. Fullword indicating how Language Environment should call the member language to invoke a main procedure; this parameter is initially set to zero.
 5. Structure describing the standard input, output, and error streams as defined by CICS.
 6. Fullword indicating if the member language can support performance enhancements to the stack extension routines. This parameter is initially set to zero.
 7. Fullword that indicates to the member language additional information to identify the calling environment.
 8. Fullword that contains the address of the WSA that was obtained and initialized by the member.
 9. Fullword indicating if the member language wishes to skip the Exit DSA scan at normal termination. This parameter is initially set to zero.
- R12 addresses the CAA
- R13 addresses a DSA
- R14, R15 are linkage registers

In the preinitialized interface, this event is driven for main routines to complete initialization for a specific application running within an enclave. This event occurs during the CALL request for main routines to allow HLLs complete their initialization for a particular application or for a particular run of an application.

Event Code 18

The combination of Event 17 and Event 18 should initialize the HLL specific aspects of the environment for a given application. The counterpart for this event is Event 19.

All callable services except CEE3CRE are available during Event 18. Stack storage is available.

Usage notes

- This event is called in CICS and non-CICS environments.
- CICS SPF: Language Environment calls languages in key 8 for this event. Storage for parameters can be in key 8.

Event code 19 — enclave termination event

Purpose

Perform language-specific enclave termination. This call allows the HLL to semantically terminate the application by enforcing the language semantics of a terminating enclave. Enclave-related resources should be released. This event is the counterpart of Event 18.

In the preinitialization interface, this event is driven for applications that run as main routines for the CALL request.

Syntax

```
Call CEEEVnnn (19, inpl)  
POINTER *inpl;
```

inpl (input)

The initialization parameter list for the enclave. Because the member ERTLI run unit termination call is no longer being made, the member languages should terminate for the run unit during this call. This event is used to terminate HLL portions at the enclave level. The order in which the member event handlers are called is in the reverse order of initialization. The dependencies are determined from the signature CSECTs. For more information, see “Signature CSECT” on page 147. Upon entry into the member event handler, the following is available:

- R1 contains the address of a standard O/S style PLIST (all of the parameters are passed by reference) with the PLIST consisting of the following:
 - An event code indicating enclave termination 19
 - The initialization parameter list that was passed to CEEINT during Language Environment initialization. The initialization parameter list is described here. It is assumed to be a **read-only** parameter list. Also, the member-defined field which directly follows the owning member ID, must be used only by the owning member.
- R12 addresses the CAA
- R13 addresses a DSA
- R14, R15 are linkage registers

Usage notes

- This event is called in both CICS and non-CICS environments.

- CICS SPF: Language Environment calls languages in key 8 for this event. Storage for parameters can be in key 8.

Event code 20 — query/build feedback code event

Purpose

The Query/Build event handler is used to convert 12-byte character strings to condition tokens and condition tokens to 12-byte character strings.

Syntax

Call CEEEVnnn (20, *function_code*, *additional_parms*, *ownership*)

```
INT4      *function_code;
INT2      *cond_name;
CHARn     *cond_token;
INT4      *ownership;
```

function_code (input)

Defines if this event is a query or build function. The functions are defined as follows:

- 1 Fixed-binary(31) indicating query feedback token event
- 2 Fixed-binary(31) indicating build feedback token event

additional_parms (input/output)

Parameters specific to a certain function code. The following parameters are for each function code:

Call CEEEVnnn (20, 1, *cond_name*, *cond_token*, *ownership*)

Call CEEEVnnn (20, 2, *cond_token*, *cond_name*, *ownership*)

Figure 110. Syntax by *function_code*

cond_name (input/output)

A halfword-prefixed character string symbolic condition name.

cond_token (input/output)

A 12-byte condition token that is constructed from the symbolic name. The I_S_Info field is set to binary zero.

ownership (input)

Fixed-binary (31) set to contain

- 0 This member does not recognize this *cond_name*
- 1 For query, this member recognizes this *cond_name* and has filled in the *cond_token*. For build, this member recognizes this *cond_token* and has filled in the *cond_name*.

Usage notes

- If the condition token is unrecognized, the value of *cond_token* is undefined.
- Language Environment recognizes only those *cond_names* that start with cel; and have a corresponding message within the Language Environment message set. If Language Environment does not recognize the *cond_name*, then all of the active members are invoked by the event handlers polling each member until one

Event Code 20

claims the *cond_name* returning the *cond_token*. Each member can validate if the condition token exists within their message set by the CEEGETFB CWI. If the *cond_name* remains unclaimed, the appropriate feedback code is returned.

Event code 21 — process termination event

Purpose

Event code 21 performs language-specific process termination. This event is used to terminate HLL portions at the process level. The order in which the member event handlers are called is undefined. In particular, the dependency list is not honored. Upon entry into the member event handler, the following is available:

- R13 points to a DSA into which the event handler is able to store its caller's registers.
- R12 is pointing to a simulated CAA allowing stack frame acquisition.
- A fixed size stack is available for use by the HLLs when called for process initialization. The stack size is 1024 bytes. There is no stack overflow support.
- The simulated CAA has a pointer to the PCB. The simulated CAA has a zero pointer to the EDB.
- The addresses of LOAD and DELETE services and GETMAIN/FREEMAIN services are held in the PCB. It is the caller's responsibility to relinquish resources obtained at the process level.
- The format of the member list at the process level is of the same format as the member list at the enclave level.
- The CEERCB_REUSE_STATE field, which indicates the state of library reuse and had one of the following values:
 - 0 Reuse is not in effect
 - 1 or 2 Reuse is in effect.
 - 3 The reuse environment is terminating.

The PLIST is an OS-style PLIST containing the single parameter of the event code for process termination.

At termination, all resources obtained at the process level **MUST** be released explicitly. Language Environment does not implicitly release any resource obtained at the process level. (Do not depend upon the resource persisting, even if the resource was not explicitly released.)

During preinitialization, this event indicates that the HLL should relinquish all resources maintained at the process level. Note all HLL semantics for a terminating application has already been accomplished by event 20 enclave termination event. This event is driven for a preinitialization TERM request for a main application.

The counterpart for this event is "Event code 17 — process initialization event" on page 502.

Syntax

```
Call CEEEVnnn (21, reuse_participant)
INT      *reuse_participant;
```

reuse_participant

Indicates if the member participates in library reuse; a value of 1 indicates participation.

Usage notes

- This event is called in both CICS and non-CICS environments.
- This event is called only if process initialization was called.
- In CICS, this event is called during transaction termination. Member languages should terminate for the transaction during this call.
- CICS SPF: Language Environment calls languages in key 8 for this event. Storage for parameters can be in key 8.
- Members must set the *reuse_participant* parameter to 1 if they participate in library reuse and need to be called for final process termination when the reuse environment terminates.

Event code 22 — DLL initialization event**Purpose**

This event is designed to be used by languages with Dynamic Link Libraries (DLLs) to perform initialization specific to the use of those DLLs. The event is driven during Language Environment enclave initialization, after the debugger initialization events but prior to the invocation of the main routine. The event is also driven by Language Environment whenever a new module has been loaded, immediately following the invocation of the New Load Module Event (8). In all cases, this event will be followed by a call to the Static Constructor Event (25).

Syntax

Call CEEEVnnn (22, *idinfo*, *loadinfo*)

```
INT4      *idinfo;
INT4      *loadinfo;
```

***idinfo* (input)**

A fullword that indicates to the member language additional information identifying the calling environment. A new executable unit (load module or program object) is introduced to the enclave by COBOL dynamic call, PL/1 or C fetch, CEEPIPI services, or DLL implicit or explicit load. The following bits are defined:

0 - 23 reserved

24 - 31 The value indicates the *load_reason*. The values are defined as follows:

- 0** The load was due to main Language Environment initialization.
- 1** The load was due to dynamic call, fetch, or ceepipi service.
- 2** The load was due to the explicit or implicit reference of a DLL.

***loadinfo* (input/output)**

A fullword returned from the New Load Module (8) event containing information about the module that was just loaded. If this event is being called as part of main Language Environment initialization flow (*load_reason* is zero), then the New Load Module event was not called and *loadinfo* is zero. It can optionally be modified by this event for use by the subsequent call to the Static Object Constructor event.

Event Code 23

A return code is placed in R15 by the Event Handler. The following return codes (in decimal) are defined:

- 4 The Event Handler does not want to process the event.
- 0 The Event Handler was successful.
- 16 The Event Handler encountered an unrecoverable error.

Event code 23 — stack frame zero processing event

Purpose

Calls the condition handler identified by the CEEHDHDL CWI. For information on registering a stack frame zero condition handler, see “CEEHDHDL — register an event handler for stack frame zero processing” on page 265.

Syntax

Call CEEEVnnn (23, *ceecib*, *results*, *new_condition*)

```
void      *ceecib;  
INT4     *results;  
FEEDBACK *new_condition;
```

ceecib (input)

The CEECIB for which the condition handler is being called. This value is passed by reference. Part of the CEECIB is the condition_token and the machine environment for the procedure in which the condition occurred. (For more details, see “Language Environment condition information block” on page 283.)

results (output)

Contains the instructions indicating the actions the language-specific handler wants the Language Environment condition manager to take as a result of processing the condition. This field is passed by reference. The following are valid responses:

Response	<i>results</i> value	Description
resume	10	Resume at the resume cursor (condition has been handled).
percolate	20	Percolate to the next condition handler.
	21	Percolate to the first user condition handler for the next stack frame. (This can skip a language-specific exception handler for this stack frame as well as the remaining user condition handlers in the queue at this stack frame.)
	23	To force CEL default condition handling for the unhandled condition when condition was signaled from CEESGL callable service with a feedback code.
promote	30	Promote to the next condition handler.
	31	Promote to the next stack frame. (This can skip a language-specific exception handler for this stack frame as well as any remaining user condition handlers in the queue at this stack frame.)
	33	Promote and restart condition handling for the first condition handler for the stack frame denoted by the resume cursor location.

Response	results value	Description
enablement	40	Ignore the condition; the thread is resumed where interrupted.
	41	Enable the condition for condition handling.
	42	Enable the condition and transform the condition (using the <i>new_condition</i> parameter).
percolate enablement	50	Percolate the enablement to the calling stack frame.
	51	Transform the condition (using the <i>new_condition</i> parameter) and percolate the enablement to the calling stack frame.

***new_condition* (output)**

The new condition token representing the promoted condition. This field is used only for *result* values that denote *promote*.

Usage notes

- For a description of the calling method, see “Language Environment member list and event handler” on page 83.
- It is invalid to promote a condition without returning a new condition token. If the original condition is returned in *new_condition*, the condition manager acts as if a *result* of 20 had been specified.
- Prior to a condition being promoted, the MIB must be populated with the new inserts for the promoted condition if necessary.
- The language-specific handlers are automatically established by stack frame. The Language Environment condition manager determines the language associated with a given stack frame, and then calls the event handler with the appropriate event code for enablement, condition handling, or condition handling for stack frame zero.
- The language-specific handlers are automatically disestablished when the stack frame is popped off the stack either using a return, a GOTO out of block, or moving the resume cursor.
- If a *resume* is requested, the member that owns the target stack frame is called immediately prior to passing control to the target stack frame. For details, see “Event code 10 — resume from a condition handler event” on page 497.
- CICS SPF: Language Environment calls languages in key 8 for this event.

Event code 24 — POSIX events event**Purpose**

The event handler is a member supplied routine that is invoked at various times throughout the execution of a program when an event had occurred. The address of each member's event handler is held in the Language Environment member list, in the third word of the appropriate member's block.

During Language Environment initialization, Language Environment loads CEEEVxxx, where xxx is the member number, if there is a corresponding signature CSECT in the load module. Language Environment saves this address in the appropriate slot in the member list.

Linkage to the member event handler is by BALR 14,15, and R1 contains the address of a standard parameter address list. The first parameter always indicates

Event Code 24

the type of event for which the event handler has been called. Additional parameters are dependent upon the specific event.

With the introduction of POSIX support, a new event code has been added to the existing set of event codes to identify various POSIX-related events that occur during the execution of the application. An accompanying function code, or event sub-code, uniquely identifies the POSIX event.

Syntax

Call CEEEVnnn (24, *function_code*, *additional_parms*)

```
INT4      *function_code;
POINTER   *ppsd_addr;
INT       *dsa_fmt;
POINTER   *valid_interrupt_dsa;
POINTER   *caa_copy_addr;
INT       *interrupt_flags
```

function_code (input)

Each of the POSIX-related events are discussed here:

- 1 POSIX fork() notification. This event is invoked before requesting the kernel to fork a new process when the calling process is not multi-threaded. It allows the members to indicate if they can tolerate a fork() request. Toleration is indicated by setting R15 to zero. If the member cannot tolerate a fork(), R15 is set to -4. If any member in the application cannot tolerate the fork(), the request to fork is denied. In a multi-threaded environment, function code 9 is used.
- 2 POSIX fork() in child. This event allows the members in the newly-forked child process to refresh their control blocks before the application code gains control. It is called when the process is not multi-threaded. In a multi-threaded environment, function code 12 is used.
- 3 POSIX asynchronous signal. This event is invoked when an asynchronous signal is received on a particular thread. The BPXYPPSD contains the information regarding the action to take for the specific signal. It is the responsibility of the member to either terminate the application or to resume at the next sequential instruction following the point of interrupt.
- 4 POSIX thread initialization. This event is driven on the newly created thread with a new CAA. A **copy** of the parent thread's CAA is passed to the event handler. This allows selective inheriting or copying of fields from the parent's CAA into the new CAA addressed by R12. There is no guarantee that the parent thread exists at the time of this event. It is the member's responsibility to access only those pointers that do not cause a reference to freed storage.
- 5 POSIX thread termination. This event offers the members the opportunity to clean up any thread-related resource that was allocated.
- 6 POSIX process initialization. This event is driven for POSIX(ON) applications under the Initial Process Thread (IPT). The POSIX environment has been initialized and all POSIX services are available. This event is driven after the Language Environment process initialization and after Language Environment enclave initialization.

- 7 POSIX process termination. This event is driven on the thread that requested termination, and not necessarily on the IPT. All threads have been terminated except the one driving this event. All POSIX functions are available. However, the use `pthread_create()` is restricted. This event is driven before the Language Environment enclave termination event. The intent of this event is to allow the cleanup of POSIX-related resources for the POSIX process.

By contrast, the Language Environment enclave termination event is always driven on the IPT to allow z/OS-related resources to be released. The POSIX environment has been terminated when this event is invoked and the POSIX flag in the EDB has been turned off.

- 9 POSIX multi-threaded `fork()` notification. This event is invoked before requesting the kernel to fork a new process in a multi-threaded environment. It allows the members to indicate if they can tolerate a multi-threaded `fork()` request. Toleration is indicated by setting R15 to zero. If the member cannot tolerate a `fork()` from a multi-threaded environment, R15 is set to -4. If any member in the application cannot tolerate the `fork()`, the request to fork is denied.
- 10 POSIX multi-threaded `fork()` lock. If the member tolerates the `fork()` request, any locking needed to prepare for the `fork()` is done.
- 11 POSIX multi-threaded `fork()` in parent after `fork()`. This event allows the members to undo any locking that occurred for the POSIX multi-threaded `fork()` notification. Any member that returned a zero return code for the POSIX multi-threaded `fork()` notification event is called for this event.
- 12 POSIX multi-threaded `fork()` in child. This event allows the members in the newly-forked child process to refresh their control blocks before the application code gains control.
- 13 POSIX process cleanup. This event is driven just prior to Language Environment requesting cleanup of the POSIX process.

additional_parms (**input**)

Parameters specific to a certain function code. The following diagram shows the parameters for each event.

Call CEEEVnnn	(24, 1)
Call CEEEVnnn	(24, 2)
Call CEEEVnnn	(24, 3, <i>ppsd_ptr, dsa_fmt, valid_interrupt_dsa, interrupt_flags</i>)
Call CEEEVnnn	(24, 4, <i>caa_copy_addr</i>)
Call CEEEVnnn	(24, 5, <i>last_thread</i>)
Call CEEEVnnn	(24, 6)
Call CEEEVnnn	(24, 7)
Call CEEEVnnn	(24, 9)
Call CEEEVnnn	(24, 10)
Call CEEEVnnn	(24, 11)
Call CEEEVnnn	(24, 12)
Call CEEEVnnn	(24, 13)

ppsd_addr (**input**)

A fullword binary integer containing the address of the BPXYPPSD, which

Event Code 24

is a z/OS UNIX control block. For a description of the fields in the BPXYPPSD, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

dsa_fmt (input)

The format of the active DSA when the signal was received. This DSA is pointed to by register 4 or 13 saved in the PPSD pointed to by *ppsd_ptr*. Possible values for *dsa_fmt* are:

- 0 non-XPLINK
- 1 XPLINK

valid_interrupt_dsa (input)

This is a pointer to the valid DSA that was used to expand the Language Environment stack from when the member event 24 handler was called. This may differ from the value in PPSD register 4 or 13. This value should be passed through to CEE3RSUM (in the *valid_interrupt_dsa* field in the CEE3RSUM resume information area) when resuming the user application after handling the signal.

caa_copy_addr (input)

A fullword binary integer containing the address of the copy of the parent's CAA.

last_thread (input)

Flag to indicate if this event is being called by the last thread in a terminating process. Possible values for *last_thread* are:

- 0 It is not the last thread
- 1 It is the last thread

interrupt_flags (input)

A fullword flag area. Bit 4 in the flags is ON if the *valid_interrupt_dsa* was saved in the CEECAA_SAVSTACK field at the time of interrupt. If the application is resumed where it was interrupted, the *valid_interrupt_dsa* must be restored to the CEECAA_SAVSTACK field.

Bit 5 in the flags is ON if *valid_interrupt_dsa* was saved in the field pointed to by the CEECAA_SAVSTACK_ASYNC field at the time of interrupt. If the application is resumed where it was interrupted, the *valid_interrupt_dsa* must be restored to the field pointed to by the CEECAA_SAVSTACK_ASYNC field. The remaining bits are reserved for future use and must be zero.

A return code is placed in R15 by the event handler. The following return codes (in decimal) are defined:

- 4 Event handler does not want to process the event
- 0 Event handler was successful
- 16 Event handler encountered an unrecoverable error

Typically, all Language Environment services are available during the handling of the event, including the stack and heap.

Event code 25 — static object constructor event

Purpose

Constructors and destructors are not resources, but are routines that are executed during the creation and destruction of application variables. Application variables can be static, automatic, or dynamic. Automatic variables are thread resources. The

invocation of constructors and destructors for those variables is performed by each thread. Static variables and dynamic variables are enclave resources, so the constructors and destructors are executed once for the creation/destruction of each static and dynamic variable in the enclave.

Constructors and destructors for automatic and dynamic variables are driven by the languages libraries or compiled code, without specific support from Language Environment. Constructors and destructors for static variables are driven by language libraries from within the static constructor and static destructor events.

The static object constructor lets a member gain control to perform constructor initialization prior to the invocation of the main routine. CEECONST is a CWI, called by member languages from their the enclave initialization event logic, to register the member to gain control, by the member event handler, for two events:

1. Static constructor event (Event Code 25)
2. Static destructor event (Event Code 36)

By requiring member languages to register for these events, the overhead of the event calls is avoided for member languages that do not need the event.

Syntax

```
void CEECONST (fc)
FEED_BACK *fc;
```

fc (output)

A 12-byte feedback code that indicates the result of this service. This parameter must be specified. The following symbolic conditions can result from this service:

Condition		
CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE38U	Severity	4
	Msg_No	3358
	Message	The service was invoked outside of the member enclave initialization. No action was taken.

The Static Constructor Event is designed to be used by languages with object oriented features to drive constructor functions (initialization methods) for all statically allocated objects. The event is driven during Language Environment enclave initialization, after the debugger initialization events but prior to the invocation of the main routine. The event is also driven by Language Environment whenever a new module has been loaded, immediately following the invocation of the DLL Initialization Event (22).

Call CEEEVnnn (25, *idinfo*, *loadinfo*)

INT4 *idinfo;
 INT4 *loadinfo;

idinfo (input)

A fullword that indicates to the member language additional information identifying the calling environment. A new executable unit (load module or program object) is introduced into the enclave by COBOL dynamic call, PL/1 or C fetch, CEEPIPI services, or DLL implicit or explicit load. The following bits are defined:

0 - 23 reserved

24 - 31 The value indicates the load_reason. The values are defined as follows:

- 0 The load was due to main Language Environment initialization.
- 1 The load was due to dynamic call, fetch, or ceepipi service. In this case, static constructors are run immediately.
- 2 The load was due to the explicit or implicit reference of a DLL. Static constructors will only be run if they are at the level represented by the initial DLL load (for example, all DLL initialization has been completed).

loadinfo (input)

A fullword returned from the New Load Module (8) or DLL Initialization (22) event containing information about the module that was just loaded.

A return code is placed in R15 by the Event Handler. The following return codes (in decimal) are defined:

- 4 The Event Handler does not want to process the event.
- 0 The Event Handler was successful.
- 16 The Event Handler encountered an unrecoverable error.

Usage notes

- This event is driven only if the member language has registered for static constructor/destructor events by calling the CEECONST CWI during the enclave initialization event.
- All services of Language Environment are available during this event.
- Application code may be driven during this event.

Event code 26 — region initialization event

Purpose

Perform language-specific initialization that can be shared among all processes in an address space.

Syntax

Call CEEEVnnn (26, *rcbptr*, *process_permstglen*)

POINTER *rcbptr;
 INT *process_permstglen;

***rcbptr* (input)**

The address of the Region Control Block (RCB) for the region. The member languages can reference fields of the RCB and reference/set the MEMLDEF field of the region member list anchored off the RCB.

***process_permstglen* (output)**

Set by the member language to the amount of permanent process storage that the language requests during process initialization (using the Get Permanent Process Storage macro, CEEXGPPS. This parameter is initially set to zero.

Usage notes

- This event is called in both CICS and non-CICS environments.
- For CICS, the CICS region defines the address space. Each running Language Environment-enabled transaction in the partition is a process in the region. Other S/370 environments do not support multiple processes in a single address space. However, member language init/term should still be structured as if that were a possibility, to maximize common code between CICS and other environments.
- Storage for parameters can be in key 8.

Event code 27 — region termination event**Purpose**

Perform language-specific termination for the region.

Syntax

<p>Call CEEEVnnn (27, <i>rcbptr</i>) POINTER *<i>rcbptr</i>;</p>

***rcbptr* (input)**

The address of the RCB for the region. The member languages can reference fields of the RCB and reference/set the MEMLDEF field of the region member list anchored off the RCB.

Usage notes

- This event is called in both CICS and non-CICS environments.
- CICS SPF: Language Environment calls languages in key 8 for this event. Storage for parameters can be in key 8.

Event code 28 — identify module entry point event**Purpose**

This event is used to determine the language of the procedure identified as the entry point of the module. Also, if the entry point is a main procedure, then return an INPL, as defined on the CEEINT CWI call. This INPL is used to initialize an enclave in order to invoke the main procedure.

Syntax

Call CEEEVnnn (28, <i>loadmodptr</i> , <i>entryptr</i> , <i>identified</i> , <i>main</i> , <i>inplptr</i> , <i>loadmodlen</i>)	
POINTER	*loadmodptr;
POINTER	*entryptr;
INT4	*identified;
INT4	*main;
POINTER	*inplptr;
INT4	*loadmodlen;

loadmodptr (input)

The address of the start of the load module.

entryptr (input)

The address of the entry point of the load module.

identified (output)

A fullword set to one of two values:

- 0 The procedure is not of the member's language.
- 1 The procedure is of the member's language.

main (output)

A fullword set to one of two values:

- 0 The procedure is not a main procedure.
- 1 The procedure is a main procedure.

inplptr (input/output)

The address of the INPL to be used to initialize the enclave. Enough storage is provided so that the member can build the INPL within the provided storage, or the member can set the *inplptr* parameter to point to other storage containing the INPL.

loadmodlen (input)

A fullword set to the length of the load module.

Usage notes

- This event is called only when running under CICS.
- If a member event handler detects an error during this event, it should return with return code 16, and place the reason code for the error in CEECAACICSRSN field of the CAA. Language Environment passes this reason code to CICS.
- If a member event handler detects a non-terminating condition (for example, the INPL cannot be built due to missing csects in the module), it should return with return code 4, and place the reason code for the error in CEECAACICSRSN field of the CAA. Language Environment passes this reason code to CICS and returns control to CICS without further processing.
- CICS SPF: Language Environment calls languages in key 8 for this event. Storage for parameters can be in key 8.

Event code 29 — determine enclave work area lengths event

Purpose

This event is used to determine the amount of permanent enclave storage that a member language requests during enclave initialization for a particular application program. Permanent enclave storage is allocated using the CEEXGPES macro.

Syntax

```
Call CEEEVnnn (29, inpl, enclave_permstglen_31, enclave_permstglen_24)
POINTER  *inpl;
INT4     *enclave_permstglen_31;
INT4     *enclave_permstglen_24;
```

inpl (input)

The INPL for the enclave. The INPL was either obtained by Language Environment from examining the code at the entry point of the load module or was obtained by the member language of the main procedure using the identify module entry point Event Code 28.

enclave_permstglen_31 (output)

A fullword integer to contain the amount of AMODE 31 storage, in bytes, that the member language needs at the enclave level. If no storage is needed, this parameter should be set to zero.

enclave_permstglen_24 (output)

A fullword integer to contain the amount of AMODE 24 storage, in bytes, that the member language needs at the enclave level. If no storage is needed, this parameter should be set to zero.

Usage notes

- This event is called only when running under CICS.
- If a member event handler detects an error during this event, it should return with return code 16, and place the reason code for the error in CEECAACICSRSN field of the CAA. Language Environment passes this reason code to CICS.
- CICS SPF: Language Environment calls languages in key 8 for this event. Storage for parameters can be in key 8.

Event code 31 — determine working storage (CICS only) event Purpose

This event is called to determine the address and length of the storage containing local variables for an executing routine. This information is returned to CICS EDF utility.

Syntax

```
Call CEEEVnnn (31, pgmrsa, memwsa, memwsl)
POINTER  *pgmrsa;
POINTER  *memwsa;
INT4     *memwsl;
```

pgmrsa (input)

The address of the save area of a routine.

memwsa (output)

A fullword to be set to the address of the routine's working storage or DSA. If this cannot be determined, the field should be set to zero.

Event Code 31

memwsl (output)

A fullword to be set to the length of the routine's working storage or DSA. If this cannot be determined, the field should be set to zero.

Usage notes

- This event is called only when running under CICS.
- If a member event handler detects an error during this event, it should return with return code 16, and place the reason code for the error in CEECAACICSRSN field of the CAA. Language Environment passes this reason code to CICS.
- CICS SPF: Language Environment calls languages in key 8 for this event. Storage for parameters can be in key 8.

Event code 32 — perform GOTO validation (CICS only) event

Purpose

This event is used to verify with the member language that a GOTO-out-of-block can be performed which transfers control to a location specified on an EXEC CICS HANDLE CONDITION condition (label).

Syntax

```
Call CEEEVnnn (32, pgmrsa, xpgmind)
```

```
POINTER *pgmrsa;  
INT4 *xpgmind;
```

pgmrsa (input)

The address of the save area of a routine that is being exited by a GOTO-out-of-block in order to transfer control to the EXEC CICS HANDLE CONDITION condition (label).

xpgmind (output)

A fullword set to indicate if the GOTO-out-of-block is restricted for this routine:

- 0 The GOTO-out-of-block is allowed.
- 1 The GOTO-out-of-block is not allowed.

Usage notes

- This event is called only when running under CICS.
- If a member event handler detects an error during this event, it should return with return code 16, and place the reason code for the error in CEECAACICSRSN field of the CAA. Language Environment passes this reason code to CICS.
- CICS SPF: Language Environment calls languages in key 8 for this event. Storage for parameters can be in key 8.

Event code 33 — member needs options processing event

Purpose

This event polls all members to see if quick options can be processed.

Syntax

```
Call CEEEVnnn (33, need_opts_processing)
INT4      *need_opts_processing;
```

need_opts_processing (input)

A fullword set to indicate if quick options processing can be done.

0 Quick options processing cannot be done.

1 Quick options processing can be done.

Event code 34 — command line equivalent event

Purpose

This event allows a member language to process a command line equivalent string. Runtime options can be changed and the parameter list to pass to the main program can be changed.

Syntax

```
Call CEEEVnnn (34, ocb_addr, R1_at_entry, plist)
POINTER *ocb_addr;
POINTER *R1_at_entry;
POINTER *plist;
```

ocb_addr (input)

The address of an OCB.

R1_at_entry (input)

The R1 value passed to CEEINT.

plist (output)

The address of the argument list to be interpreted as the inbound parameter.

Usage notes

- The OCB passed to this event will contain the IBM-supplied defaults, system-level defaults, region-level defaults, and programmer defaults merged.
- Only the member identified by the member ID in the INPL is called when the **reqcmdequ** flag in the main options word of the INPL is ON.
- This event has limited capabilities. There is a fixed stack available and a partially initialized CAA. No Language Environment callable services can be used from this event.
- Members which change a runtime option should change the corresponding OCB where_set field to PROGRAM_INVOCATION, which will cause the options report to show "Invocation command" for that option.
- Members which support main programs being called with a nonsupported parameter list can use this event to do their own command-line equivalent processing.

Event code 35 — default options event

Purpose

The purpose of this event is to allow the members to set default runtime options in a compatible fashion.

Syntax

```
Call CEEEVnnn (35, ocb_addr, ceestart_addr, inpl_addr, work_area)
POINTER *ocb_addr;
POINTER *ceestart_addr;
POINTER *inpl_addr;
POINTER *work_area;
```

ocb_addr (input)

The address of an OCB. The OCB passed to this event will contain the IBM-supplied defaults, system-level defaults, and region-level defaults merged.

ceestart_addr (input)

The address of CEESTART.

inpl_addr (input)

The address of the initialization parameter list (INPL).

work_area (input)

The address of a 512-byte work area.

Usage notes

- Only the member identified by the member ID in the INPL is called when the **defoptreq** flag in the main options word of the INPL is ON.
- This event has limited capabilities. There is a fixed stack available and a partially-nitialized CAA. No Language Environment callable services can be used from this event.
- Members that set a default option should change the corresponding OCB where_set field to DEFAULT_SETTING, which will cause the options report to show "Default setting" for that option.
- Members that recognize that the application being run needs a specialized set of runtime options can use this event to tailor the default options appropriately.

Event code 36 — static destructor event

Purpose

This event is designed to be used by languages with object oriented features to drive destructor functions (uninitialization methods) for all statically allocated objects. This event is driven during Language Environment enclave termination, after stack collapse, but prior to debugger termination events. This event occurs after the atterm event.

Syntax

```
void CEEEVnnn (event_code)
INT4 *event_code = 36;
```

***event_code* (input)**

Is a fullword integer with value 36 indicating that this is the static destructor event call.

Usage notes

- This event is driven only if the member language has registered for static constructor/destructor events by calling the CEECONST CWI during the enclave initialization event.
- All services of Language Environment are available during this event.
- Application code may be driven during this event.

Event code 37 — preallocated storage event**Purpose**

This event allows user-supplied storage to be used as the initial segment of user stack or user heap storage.

Syntax

Call CEEEVnnn (37, *ocb_addr*, *R1_at_entry*, *init_stack_addr*, *init_stack_len*, *init_heap_addr*, *init_heap_len*)

```
POINTER *ocb_addr;
POINTER *R1_at_entry;
POINTER *init_stack_addr;
POINTER *init_stack_len;
POINTER *init_heap_addr;
POINTER *init_heap_len;
```

***ocb_addr* (input)**

The address of an OCB.

***R1_at_entry* (input)**

The R1 value passed to CEEINT.

***init_stack_addr* (output)**

The address of the initial segment of stack storage.

***init_stack_len* (output)**

The length of the initial stack segment.

***init_heap_addr* (output)**

The address of the initial segment of heap storage.

***init_heap_len* (output)**

The length of the initial heap segment.

Usage notes

- Only the member identified by the member ID in the INPL is called when the prealloc flag in the main options word of the INPL is ON.
- This event has limited capabilities. There is a fixed stack available and a partially initialized CAA. No Language Environment callable services can be used from this event.
- The output of this event is used for the initial segment only. For the increment segments, location, and disposition of the user stack and user heap storage, the corresponding suboption specifications in the STACK and HEAP runtime options continue to be used.

Event Code 37

Note: If the location specification is BELOW, but the user-supplied storage is above the 16M line, the member is responsible for diagnosis and a return code of 16 must be returned by this event.

- The user-supplied storage must be located at a valid address and be on a doubleword boundary. The length must also be a multiple of 8. Otherwise, a return code of 16 must be returned by this event. If there is no user-supplied storage, a zero length must be returned as the output.
- The OCB passed to this event contains options merged through the Assembler user exit level.
- The user-supplied storage is not freed by Language Environment at termination.
- The user-supplied user heap storage is subject to the AMODE of the application that requests storage. The user-supplied storage is ignored if the following occurs:
 - The user-supplied storage is above the 16M line, and
 - The ANYWHERE suboption of the HEAP option is in effect, and
 - The application that requests storage is in AMODE(24)

Language Environment allocates below the line storage using the *initsz24* and *incrsz24* suboptions from the HEAP runtime option. In all other cases, the preallocated storage is used.

Event code 38 — normal resume in DSA event

Purpose

This event code identifies that a normal (non-condition handler) resumption occurs within the *target_dsa*.

Syntax

```
Call CEEEVnnn (38, target_dsa, target_dsa_fmt, ph_callee_dsa, ph_callee_dsa_fmt)
```

```
void      *target_dsa;  
INT       *target_dsa_fmt;  
void      *ph_callee_dsa;  
INT       *ph_callee_dsa_fmt;
```

target_dsa (input)

The DSA that is the target for the resume.

target_dsa_fmt (input)

The format of the DSA pointed to by *target_dsa*. Possible values are:

```
0      non-XPLINK  
1      XPLINK
```

ph_callee_dsa (input)

A pointer to the DSA of the routine called by the routine owning the DSA pointed to by *target_dsa*.

ph_callee_dsa_fmt (input)

The format of the DSA pointed to by *ph_callee_dsa*. Possible values are:

```
0      non-XPLINK  
1      XPLINK
```

Usage notes

- The Language Environment condition manager determines the member that owns the stack frame that is the target of the resume. Once determined,

Language Environment condition manager calls the particular member's event handler just prior to performing the resume operation into the stack frame.

- It is the member's responsibility to perform the necessary actions to allow the resume to occur within the *target_dsa*.
- The *ph_callee_dsa* parameter is provided in case the event handler needs to extract registers from the DSA pointed to by *target_dsa*. Registers which are saved in the DSA pointed to by *target_dsa* for non-XPLINK are mostly saved in the DSA pointed to by *ph_callee_dsa*, if *target_dsa_fmt* is XPLINK. Note that *ph_callee_dsa_fmt* might not be the same as *target_dsa_fmt*. Also, the DSA pointed to by *ph_callee_dsa* may belong to a Language Environment transition or Language Environment overflow routine.

Event code 39 — interrupt received event

Purpose

This event identifies special processing to determine if it is safe to accept the interrupt. This is a special interface between Language Environment's signal interface routine and the PL/I multitasking library and COBOL with multi-threading toleration.

Syntax

```
Call CEEEVnnn (39, function_code, module_pointer, ppsd_pointer, return_value, dsa_pointer,
Retcode(return_code))
INT4          *function_code;
CEE_ENTRY     *module_pointer;
CEE_TOKEN     *ppsd_pointer;
INT4          *return_value;
CEE_DSA       *dsa_pointer;
INT4          *return_code;
```

function_code (input)

The functions are defined as follows:

- 1 Determine if the module pointed to by *module_pointer* can accept the interrupt.
- 2 Language Environment has determined that the interrupt must be put back to the kernel. Determine how the interrupt can be resolicited.

module_pointer (input)

This argument is a pointer to one of the modules pointed to from the saved register 15 in one of the DSAs on the stack.

ppsd_pointer (input)

Is the address of a control structure received by the Language Environment signal interface routine from the kernel (defined in BPXYPPSD). This structure, referred to as the PPSD contains PSW and register information that can be used to determine where the interrupt occurred and how to handle it.

return_value (output)

If the *function_code* is 1, COBOL or PL/I returns one of the following values:

- 1 Accept the interrupt. For COBOL or PL/I this means the module pointed to by *module_pointer* is a COBOL or PL/I user module and it can accept the interrupt. That is, it is safe to accept the interrupt.
- 2 Do not accept the interrupt. For COBOL or PL/I this means it is a

COBOL or PL/I user module, but cannot accept the interrupt. That is, it is not safe to accept the interrupt. Therefore, the interrupt must be put back.

- 3 Do not know what to do. For COBOL or PL/I this means the module pointed to by *module_pointer* is not a COBOL or PL/I user module, and it is up to Language Environment to take a proper action.

If the *function_code* is 2, COBOL or PL/I returns one of the following values:

- 1 Insert CEEOSIGR into stack at the DSA pointed to by *dsa_pointer*.
- 2 Do not insert CEEOSIGR; instead swap the LIBVEC pointers with the 'signal glue versions' from CEELVTL.
- 3 Insert CEEOSIGR into stack at the DSA pointed to by *dsa_pointer* and swap the LIBVEC pointers with the 'signal glue versions' from CEELVTL.
- 4 Just put back the interrupt.
- 5 Do not know what to do. It is up to Language Environment to take a proper action.

dsa_pointer (input/output)

When used as input, this value is the DSA address of the *module_pointer*. For COBOL or PL/I, if it is a synchronous delivery, the DSA can be used along with the module prologue code to ensure the module is a COBOL or PL/I module.

When used as output, this value applies to function code 2 only. If *function_code* is 2 and the *return_value* is 1 or 3, this parameter is returned pointing to the DSA whose saved register 14 should be replaced with the address of CEEOSIGR. Otherwise, this parameter is ignored.

return_code (input)

Standard event handler return code (-4, 0, 16)

Usage notes

- R12 points to a valid CAA.
- R13 contains the address of a valid DSA with a register save area that can be used to save the caller's registers.
- The NAB in DSA pointed to by R13 cannot be used. If dynamic storage is required, it must be acquired using GETMAIN or it must be preallocated and the address saved in a control block whose address is accessible through the CAA. Calls to any routines including Language Environment services that require dynamic storage are strictly prohibited.
- The COBOL or PL/I event handler will be called with this function whenever Language Environment has to decide what to do with the interrupt and PL/I Multitasking is active (CEEEDBPLITASKING = 1) or COBOL has been initialized. COBOL or PL/I must then determine and tell CEEOSIGH/I/J/P what to do with the interrupt.
- COBOL or PL/I event handler should register a shunt routine in CEECAADMC when storage access could result in a program check. Because a shunt could have already been registered, the current value in CEECAADMC must be saved before registering a shunt and restored before returning to Language Environment.
- If the event handler returns an undefined disposition value, the action will take the default; that is, the interrupt will be put back and do nothing.

- The event handler must pass back a return code in R15. These codes are the same as other events. If a nonzero return code is passed back by the event handler, the action will take the default, which is to put back the interrupt and do nothing.
- CEEOSIGH/I/J/P will put out a trace entry to indicate the disposition result of the Event 39 invocation. If the disposition is to accept the interrupt, the trace entry will indicate the interrupt is accepted by a COBOL or PL/I user module. If the disposition is to put back the interrupt, the trace entry will indicate the signal put back codes.

Along with the existing signal put back codes defined in Language Environment, the following lists signal put back codes specific for the Event 39 support:

	Code	Description
Signal_return codes	06	Signal_Return1, set reg 14 in the user DSA to point to CEEOSIGR. It is corresponding to Event 39 function code 2 return value 1.
	08	Signal_Return1, do nothing but put back. It is corresponding to Event 39 function code 2 return value 4.
Signal_Return2 codes	09	Signal_Return1, swapped LIBVEC pointers to the signal glue code versions. It is corresponding to Event 39 function code 2 return value 2.
	10	Signal_Return1, swapped LIBVEC pointers to the signal glue code versions and set reg 14 in the user's stack frame to point to CEEOSIGR. It is corresponding to Event 39 function code 2 return value 3.

Event code 40 — get/release function pointer event

Purpose

The Get/Release Function Pointer event is used to obtain or release a function pointer for a function that resides in a separate load module.

Syntax

```
Call CEEEVnnn (40, function_code, func_pointer, entry_pointer, ceestart_ptr)
```

```
INT4      *function_code;
POINTER   *func_pointer;
POINTER   *entry_pointer;
POINTER   *ceestart_ptr;
```

func_code (input)

Defines if this event is a Get or Release request. The functions are defined as follows:

- 1 Fixed binary(31), indicating Get Function Pointer event
- 2 Fixed binary(31), indicating Release Function Pointer event

func_pointer (output)

For the Get Function Pointer event, contains the returned function pointer. For the Release Function Pointer event, this value contains the function pointer to release.

entry_pointer (input)

Language Environment recognizes the following *func_addr* style; Language Environment does not recognize any other entry styles:

Event Code 40

- C/370-style PPA
- Language Environment routine entry layout
- Language Environment-format CEESTART
- Language Environment AWI stubs

ceestart_ptr (input)

CEESTART of the load module; the load module must be recognized by Language Environment.

Usage notes

- All function pointers obtained must be released before deleting the load module which contains the associated functions.
- The CEE3ADDM service must be called prior to calling this event handler, to augment the set of currently active members and to notify members that a new load module has been introduced into the enclave.
- C and C++ are the only target languages that support the Get Function Pointer service.
- The function pointer is returned with the high-order bit indicating the AMODE of the routine. You must provide the necessary AMODE switching code when passing control to the function pointer.
- Event 40, function code 2 must be called to release each function pointer obtained, before deleting the load module containing the associated function.
- If the load module contains any ILC or the loading and loaded modules are written in different languages, the load module should not be deleted.
- A C function that is called using a pointer returned by Event 40 will have access to the writable static area, if it exists.
- To use Event 40 to obtain a function pointer for a C function, the C function must either:
 - Be compiled with the pragma linkage(...,fetchable) directive, or
 - Have the function name specified as the entry point when the module is linked.

In addition, C++ routines must be compiled as extern "C".

- Event 40 cannot be used to obtain a function pointer for a C main() routine.
- If you use Event 40 to obtain a function pointer for a C or C++ function, calling the function pointer will give control to a glue routine. This routine will perform AMODE switching, if needed, before passing control to the C/C++ routine.
- If you use Event 40 to obtain a function pointer for a C or C++ routine that is compiled as a DLL, the routine cannot export any functions or variables.

Event code 41 — cancel/release load module event

Purpose

This event notifies a member language that an executable program (load module or program object) is about to be released and to perform any necessary cleanup related to the executable program.

Syntax

Call CEEEVnnn (41, <i>entry_point</i> , <i>ceestart_ptr</i> , <i>load_point</i> , <i>module_length</i> , <i>idinfo</i>)	
POINTER	* <i>entry_point</i> ;
POINTER	* <i>ceestart_ptr</i> ;
POINTER	* <i>load_point</i> ;
POINTER	* <i>module_length</i> ;
INT4	* <i>idinfo</i> ;

entry_point (input)

Entry point of the module

ceestart_ptr (input)

CEESTART address, if the executable program was recognized, or zero, if it was not recognized

load_point (input)

Beginning address of the module

module_length (input)

Number of bytes in the module

idinfo (input)

A fullword that tells the member language additional information about the calling environment. A new executable program is introduced into the enclave by a COBOL dynamic call, PL/I or C fetch, CEEPIPI services, and DLL implicit and explicit load. The following bits are defined:

0–23 Reserved

24–31 The value indicates the *load_reason* provided on the Event Code 8 (see “Event code 8 — new load module event” on page 495). The following values are defined:

1 The load was due to a dynamic call, fetch, or CEEPIPI service.

2 The load was due to the explicit or implicit reference of a DLL.

Usage notes

- The member should do any cleanup required related to the module.
- CEEPIPI(*call_sub_addr_nochk*), which is described “CEEPIPI — invocation for subroutine by address” on page 193, calls this event after the target has returned from the call function.

Event code 42 — automatic destructor event

Purpose

This event enables languages with object-oriented features to drive destructor routines (uninitialization methods) for automatic objects on the stack. This event is driven only for C++. This event is driven for each remaining stack frame that needs destructors to be run on a thread that is terminated using `pthread_exit()` or on a thread that is being cancelled as the result of a `pthread_cancel()` issued by another thread.

Syntax

```

Call CEEEVnnn (42, stack_frame_ptr, stack_frame_fmt, ph_callee_stack_frame_ptr,
ph_callee_stack_frame_fmt)
POINTER    *stack_frame_ptr;
INT        *stack_frame_fmt;
POINTER    *ph_callee_stack_frame_ptr;
INT        *ph_callee_stack_frame_fmt;

```

stack_frame_ptr

Pointer to the stack frame for which destructors need to be run.

stack_frame_fmt (**input**)

The format of the DSA pointed to by *stack_frame_ptr*. Possible values are:

```

0        non-XPLINK
1        XPLINK

```

ph_callee_stack_frame_ptr (**input**)

A pointer to the DSA of the routine called by the routine owning the DSA pointed to by *stack_frame_ptr*.

ph_callee_stack_frame_fmt (**input**)

The format of the DSA pointed to by *ph_callee_stack_frame_ptr*. Possible values are:

```

0        non-XPLINK
1        XPLINK

```

Event code 44 — member program mask event

Purpose

The event allows a member language to report back the program mask requirements for that member language. Language Environment adds the bits in the member's output program mask to the program mask used while the enclave is active. This event is called when a member is added to an existing enclave and in the following situations:

- The member was previously added as a dependent member, that appeared only in the dependent member list of a signature CSECT in the language list. Event 18 (enclave Initialization event) was called at that time, but the output program mask from Event 18 was ignored
- The member now being added has a signature CSECT in the language list of the new module being added to the enclave. The output program mask from Event 44 will now be honored.

All callable services except CEE3CRE are available during event 44. Stack storage is also available

Syntax

```

Call CEEEVnnn (44, pgmmask)
INT4       *pgmmask

```

pgmmask (input / output)

A fullword containing the program mask in the rightmost bits. The bits in this output mask are added to the program mask that is in effect when the enclave is running.

All Language Environment services are available at the time of this event. The member can influence the program mask setting by placing its requirements of the program mask in the second parameter as described below.

Upon entry into the member event handler for the member program mask event, the following is available:

- R1** Contains the address of a standard O/S style PLIST (all of the parameters are passed by reference) with the following PLIST:
- Event code 44
 - Fullword field in which the program mask is held in the right-most bits; upon input, this field is zero.
- R12** Addresses the CAA
- R13** Addresses the DSA
- R14, R15**
Linkage registers

Usage notes

- This event might be called in CICS and non-CICS environments.
- CICS SPF: Language Environment might call languages in key 8 for this event. Storage for parameters can be in key 8.

Event Code 44

Chapter 16. z/OS UNIX System Services support

This section describes the support provided by Language Environment services for z/OS UNIX System Services (z/OS UNIX). All of the interfaces described in this section are intended for applications that include a C environment. The threading interfaces that are provided as CWIs support the POSIX 1003.4a (draft 6) specification. These threading functions cannot be dynamically fetched. For more information about individual CWIs, see the corresponding C functions in *z/OS XL C/C++ Runtime Library Reference*.

Note:

1. Functions that end with the characters `_np` are extensions to the POSIX standard.
2. Unless otherwise is specified, access to the CWIs in the following sections requires that the runtime option POSIX be set to ON.
3. Fortran is not supported in this environment.
4. The CWI arguments for the thread attributes object, the mutex attributes object, the condition variable attributes object, and the rwlock attributes object must be declared in the calling routine. These data types correspond to typedefs defined in the C/C++ runtime library header `sys/types.h`. For example, the thread attributes object is defined by `pthread_attr_t`. The size of the object in the calling routine must match the C definition.

Thread management functions

The following sections describe the various thread management functions.

CEEOPAI

C library interface: `pthread_attr_init()`

CEEOPAI initializes a thread attribute object, *attr*. The resulting thread attribute object (possibly modified by subsequent assignment to its members), when used by the create thread function, defines the attributes for the thread to be created.

A single thread attribute object can be used multiple times, thus creating a number of threads with the same characteristics. It is the user's responsibility to serialize changes to the thread attribute object.

Syntax

void CEEOPAI (*attr*, [*fc*])

```
CEE_PTAT    *attr;  
FEED_BACK  *fc;
```

CEEOPAI

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12  
L    R15,0124(,R15)  
BALR R14,R15
```

attr (input)

The user-supplied thread attribute object to be initialized. The thread attribute object is defined by the C/C++ typedef of `pthread_attr_t` in the `sys/types.h` header.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in initializing the new thread attribute object.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.

CEEOPAD

C library interface: `pthread_attr_destroy()`

CEEOPAD makes the thread attribute object, which is referred to by *attr*, unusable. An error occurs if the attribute object is used after it has been destroyed

Syntax

void CEEOPAD (*attr*, [*fc*])

```
CEE_PTAT    *attr;
FEED_BACK   *fc;
```

CEEOPAD

Call this CWI interface as follows:

```
L   R15,CEECAALE0V-CEECAA(,R12)    CAA address is in R12
L   R15,0128(,R15)
BALR R14,R15
```

attr (input)

The initialized thread attribute object.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in destroying the thread attribute object.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5F1	Severity	3
	Msg_No	5601
	Message	Thread attribute object that is not valid was passed.

CEEOPAGD

C library interface: *pthread_attr_getdetachstate()*

CEEOPAGD obtains the thread's *detachstate* attribute from the specified thread attribute object.

Syntax

void CEEOPAGD (*attr, detachstate, [fc]*)

```
CEE_PTAT    *attr;
INT4        *detachstate;
FEED_BACK   *fc;
```

CEEOPAGD

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0132(,R15)
BALR R14,R15
```

attr (input)

The initialized thread attribute object.

detachstate (output)

- 0 the thread remains in an undetached state after termination of the thread
- 1 the thread is detached on completion

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in getting the thread stack size attribute.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5F1	Severity	3
	Msg_No	5601
	Message	Thread attribute object that is not valid was specified.

CEEOPAGS

C library interface: *pthread_attr_getstacksize()*

CEEOPAGS obtains the thread's *stack_size* attribute from the specified thread attribute object.

Syntax

void CEEOPAGS (*attr, stack_size, [fc]*)

```
CEE_PTAT    *attr;
INT4        *stack_size;
FEED_BACK   *fc;
```

CEEOPAGS CWI

CEEOPAGS

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0140(,R15)
BALR R14,R15
```

attr (input)

The initialized thread attribute object.

stack_size (output)

The non-negative *stack_size* attribute value (in bytes).

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in getting the thread stack size attribute.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5F1	Severity	3
	Msg_No	5601
	Message	Thread attribute object that is not valid was specified.

CEEOPAGW

C library interface: *pthread_attr_getweight_np()*

CEEOPAGW obtains the thread's *threadweight* attribute from the specified thread attribute object.

Syntax

void CEEOPAGW (*attr*, *threadweight*, [*fc*])

```
CEE_PTAT    *attr;
INT4        *threadweight;
FEED_BACK   *fc;
```

CEEOPAGW

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0148(,R15)
BALR R14,R15
```

attr (input)

The initialized thread attribute object.

threadweight (output)

- 0 - indicates a heavy weight thread.
- 1 - indicates a medium weight thread.

Note: Light weight threads are not supported.

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in getting the thread's threadweight attribute.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5F1	Severity	3
	Msg_No	5601
	Message	Thread attribute object that is not valid was specified.

CEEOPASD

C library interface: *pthread_attr_setdetachstate()*

The *detachstate* of the thread attribute object indicates if a thread should either be detached immediately upon completion or remain nondetached. CEEOPASD sets the appropriate value in the thread attribute object.

Syntax

void CEEOPASD (*attr*, *detachstate*, [*fc*])

```
CEE_PTAT    *attr;
INT4       *detachstate;
FEED_BACK  *fc;
```

CEEOPASD

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0136(,R15)
BALR R14,R15
```

***attr* (input)**

The initialized thread attribute object.

***detachstate* (input)**

- 0 - indicates that the thread remains in an undetached state after termination of the thread.
- 1 - indicates that the thread is detached on completion.
-

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in setting the thread detachstate attribute.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5F1	Severity	3
	Msg_No	5601
	Message	Thread attribute object that is not valid was specified.
CEE5F2	Severity	3
	Msg_No	5602
	Message	The value of <i>detachstate</i> is not 0 or 1.

CEEOPASS

C library interface: *pthread_attr_setstacksize()*

CEEOPASS sets the thread's *stack_size* attribute in the specified thread attribute object, *attr*. Note the thread's *stack_size* attribute is initialized to the size specified by the STACK runtime option when the thread attribute object is initialized.

Syntax

void CEEOPASS (*attr*, *stack_size*, [*fc*])

```
CEE_PTAT   *attr;
INT4      *stack_size;
FEED_BACK *fc;
```

CEEOPASS

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0144(,R15)
BALR R14,R15
```

attr (input)

The initialized thread attribute object.

stack_size (input)

The non-negative initial size (in bytes) of the runtime stack for a thread.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in setting the thread stack size attribute.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5F1	Severity	3
	Msg_No	5601
	Message	Thread attribute object that is not valid was specified.

Condition		
CEE5FC	Severity	3
	Msg_No	5612
	Message	The <i>stack_size</i> attribute did not contain a valid value.

CEEOPASW

C library interface: *pthread_attr_setweight_np()*

CEEOPASW sets the *threadweight* property in the specified thread attribute object.

Syntax

void CEEOPASW (*attr*, *threadweight*, [*fc*])

```
CEE_PTAT  *attr;
INT4      *threadweight;
FEED_BACK *fc;
```

CEEOPASW

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L   R15,0152(,R15)
BALR R14,R15
```

attr (input)

The initialized thread attribute object.

threadweight (input)

- 0 - indicates a heavy weight thread.
- 1 - indicates a medium weight thread.

Note: Light weight threads are not supported.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in setting the thread weight attribute.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5F1	Severity	3
	Msg_No	5601
	Message	Thread attribute object that is not valid was specified.
CEE5F3	Severity	3
	Msg_No	5603
	Message	Value for threadweight is not valid.

CEEOPC

C library interface: *pthread_create()*

CEEOPC creates a new thread in the caller's enclave and in the context of the current enclave with the specified attribute, *attr*. The new thread starts executing the routine at the entry point referred to by *routine_addr* with *arg* as its sole argument. When the routine returns, thread is implicitly terminated using the return value of the program as the termination status. The thread is detached according to the *detachstate* setting of the thread attribute specified at thread creation.

Upon successful completion of this function, the thread identifier of the newly created thread is returned in the location referred to by *thread_id*. Other threading functions may use *thread_id* as a token in their parameter lists to refer to the new thread.

Syntax

void CEEOPC (*routine_addr*, [*arg*], [*attr*], *thread_id*, [*fc*])

```
CEE_ENTRY  *routine_addr;
CEE_TOKEN  *arg;
CEE_PTAT   *attr;
CEE_THDID  *thread_id;
FEED_BACK  *fc;
```

CEEOPC

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0048(,R15)
BALR R14,R15
```

routine_addr (input)

The entry point of an external routine (not a nested procedure) that the new thread starts executing.

arg (input/optional)

An argument to be passed to the routine at its entry point. Its type is determined by the requirements of the routine called. This is the R1 value that is inbound to the target routine.

attr (input/optional)

The thread attributes object to be used for the new thread. When *attr* is omitted, the default attributes are used.

thread_id (output)

The unique thread identifier generated by Language Environment. It is used to refer to the new thread in other services. The thread identifier occupies a double word. The exact content of the thread identifier is not externalized.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in creating the new thread.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5F1	Severity	3
	Msg_No	5601
	Message	The attributes object parameter did not contain a valid initialized attributes object (POSIX PTAT).
CEE5F4	Severity	3
	Msg_No	5604
	Message	A new thread could not be created due to some system-detected error with error code <i><err_code></i> and reason code <i><rsn_code></i> .
CEE5F5	Severity	3
	Msg_No	5605
	Message	There was not enough storage available to create the new thread.

Usage Notes:

1. It is assumed that *routine_addr* is currently available and does not require an explicit LOAD performed.
2. The new thread starts execution at the external procedure given in *routine_addr* and shares the context of the current enclave.
3. The thread shares all resources of the enclave.
4. The new thread has access to a new, independent stack. In particular, a new stack frame zero is provided.
5. The new thread inherits the execution priority from its creator. The size of the stack is determined by the stack size thread attribute.
6. The user must serialize use of shared resources, for instance, external data or arguments.
7. Arguments can be passed to the routine to be executed if the routine is declared to accept them. Output arguments and in/out arguments can be passed. Since the thread runs asynchronously with the creating thread, arguments passed by reference become shared variables and their use should be serialized, if necessary. Since the thread doesn't return to its creator, output arguments returned by value could be lost if the storage referred to by the arguments no longer exists. This might occur if the caller provided automatic storage for the arguments to the new thread.
8. POSIX provides a per-process signal vector and a per-thread signal mask.
9. The *thread_id* is used to refer to the thread as input to other services. No other use of *thread_id* is allowed.
10. Success of thread creation is reported by the *fc*. This does not report on success of Language Environment initialization in the new thread nor the successful execution of the code on the thread. If the *fc* is nonzero, *thread_id* is not valid. If the *fc* is zero, the *thread_id* is valid and can be used in functions that require thread identifiers.
11. The new thread's state is *runnable*.

CEEOPC

C library interface: *pthread_exit()*

CEEOPPE terminates the calling thread within the current enclave. A thread termination *status* can be specified so that it becomes available to a thread waiting for the terminating thread. The status remains available until the thread is detached. Thread termination does not release any application-visible enclave (or process) resources such as mutexes.

This function does not return to its caller.

Syntax

void CEEOPPE (*status*)

INT4 *status;

CEEOPPE

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0072(,R15)
BALR R14,R15
```

status (input)

The value of thread termination status to become available to a thread waiting for the current thread to terminate. The value is user-defined.

CEEOPPEQ

C library interface: *pthread_equal()*

CEEOPPEQ compares the specified thread identifiers. Upon successful completion of this function, a nonzero value is returned in the *result* argument if the specified thread identifiers are equal. Otherwise, a zero value is returned.

Syntax

void CEEOPPEQ (*thread_id1*, *thread_id2*, *result*, [*fc*])

```
CEE_THDID *thread_id1;
CEE_THDID *thread_id2;
INT2 *result;
FEED_BACK *fc;
```

CEEOPPEQ

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0088(,R15)
BALR R14,R15
```

thread_id1 (input)

The thread identifier of the first thread.

thread_id2 (input)

The thread identifier of the second thread.

result (output)

The result of the *thread_id* comparison. A nonzero value indicates the two thread identifiers are equal and a zero value indicates otherwise.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in comparing the thread identifiers.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.

CEEOPJ

C library interface: *pthread_join()*

CEEOPJ suspends execution of the calling thread until the target thread specified by *thread_id* terminates. The calling thread is thus placed into the *blocked* state. When the target thread completes, the calling thread is placed into the *runnable* state. The target thread cannot be the calling thread. If the target thread is already terminated, the call returns without the calling thread being blocked.

Upon successful completion of this function:

1. The termination status of the target thread is returned in the location referred to by *status*. This is set for normal return using *pthread_exit()*.
2. If the detach parameter of CEEOPJ, *WithDetach*, is set to one (1), CEEOPJ detaches the target thread before returning. Otherwise, CEEOPJ does not detach the target thread.

Syntax

void CEEOPJ (*thread_id*, *WithDetach*, *status*, [*fc*])

```
CEE_THDID  *thread_id;
INT4       *WithDetach;
POINTER    *status;
FEED_BACK  *fc;
```

CEEOPJ

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0068(,R15)
BALR R14,R15
```

thread_id (input)

The unique thread identifier of the target thread.

WithDetach (input)

The indicator of whether the target thread should be detached before CEEOPJ returns.

```
0    Do not detach
1    Detach
```

status (input/output)

The location in which the value passed to the thread termination function by the terminating (target) thread is returned.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in waiting for the thread termination.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5F6	Severity	3
	Msg_No	5606
	Message	The value specified by <i>thread_id</i> is not a valid thread identifier.
CEE5F7	Severity	3
	Msg_No	5607
	Message	The value specified by <i>thread_id</i> is the thread identifier of the currently executing thread.
CEE5F8	Severity	3
	Msg_No	5608
	Message	The z/OS UNIX BPX1PTJ system call by CEEOPJ failed.
CEE5F9	Severity	3
	Msg_No	5609
	Message	The thread specified by <i>thread_id</i> is not in an undetached state, is currently joined by another thread, or does not exist.

CEEOP0

C library interface: *pthread_once()*

CEEOP0 insures the routine passed is executed only once during the execution of a POSIX process (based upon the *once_ctl* that is passed).

Syntax

void CEEOP0 (*once_ctl*, *init_rtn*, [*fc*])

```
INT4          *once_ctl;
ENTRY        *init_rtn;
FEED_BACK    *fc;
```

CEEOP0

Call this CWI interface as follows:

```
L    R15,CEECAALE0V-CEECAA(,R12)    CAA address is in R12
L    R15,0084(,R15)
BALR R14,R15
```

once_ctl (input)

Determines whether the *init_rtn* has been called for the POSIX process. This variable must be initialized to the value of the PTHREAD_ONCE_INIT constant defined in the C/C++ library pthread.h header.

init_rtn (input)

The user routine that is executed on behalf of the pthread_once call. The user routine is invoked without any parameters.

fc (output/optional)

The feedback code returned by the service.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5GJ	Severity	3
	Msg_No	5651
	Message	The <i>once_control</i> parameter did not contain a valid value.

Usage Notes:

1. Nested CEEOP0 invocations are allowed.
2. Although `longjmp()` can be used in an *init_rtn*, be aware that if `longjmp()` prevents the *init_rtn* from completing, CEEOP0 will not terminate; any threads that are in a wait for the *once_ctl* will remain in a wait.

CEEOPS

C library interface: *pthread_self()*

CEEOPS obtains the identifier of the calling thread. This is useful since the thread creation call does not provide the thread identifier to the created thread.

Upon successful completion of this function, the thread identifier of the calling thread is returned in the specified argument, *thread_id*.

Syntax

void CEEOPS (*thread_id*, [*fc*])

```
CEE_THDID  *thread_id;
FEED_BACK  *fc;
```

CEEOPS

Call this CWI interface as follows:

```
L    R15,CEECAALE0V-CEECAA(,R12)    CAA address is in R12
L    R15,0080(,R15)
BALR R14,R15
```

thread_id (output)

The thread identifier of the calling thread.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in getting the calling thread identifier.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.

Signal handling CWIs

This section describes the CWIs for signal handling.

CEEOKILL

C library interfaces: *kill()*, *pthread_kill()*, *raise()*, *sigqueue()*

This CWI supports the C/C++ *kill()*, *pthread_kill()*, *raise()*, and *sigqueue()* functions. The specific mapping is as follows:

<i>kill(pid, sig):</i>	CEEOKILL('1', pid, '0', Cond-Token, '0', fc)
<i>pthread_kill(tid, sig):</i>	CEEOKILL('2', '0', tid, Cond-Token, '0', fc)
<i>sigqueue(pid, sig_val):</i>	CEEOKILL('3', pid, '0', Cond-Token, 'sig_val', '0', fc)
<i>raise(sig):</i>	CEEOKILL('1', getpid(), '0', Cond-Token, '0', fc)

The function value is set based on the severity code in the feedback token (*fc*).

Syntax

void CEEOKILL (*function, process_id, thread_id, cond_rep, sig_val, [q_data_token], [fc]*)

```
INT4      *function;
CEE_TOKEN *process_id;
CEE_THDID *thread_id;
FEED_BACK *cond_rep;
CEE_TOKEN *sig_val;
CEE_TOKEN *q_data_token;
FEED_BACK *fc;
```

CEEOKILL

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0028(,R15)
BALR R14,R15
```

function (input)

Indicate the origin of this CWI request. It is 1 for *kill()*, 2 for *pthread_kill()* and 3 for *sigqueue()*. *raise()* is mapped to *kill()* to self (for example, function value = 1).

process_id (input)

Ignored unless the function value is 1. It indicates the process identifier to which the signal is to be sent. If that identifier is zero, the signal is sent to all processes (excluding an unspecified set of system processes) whose group identifier is equal to the process group identifier of the sender and for which the process has permission to send a signal. If the process identifier is negative (but not -1), the signal is sent to all of the processes (excluding an unspecified set of system processes) whose process group identifier is equal to the absolute value of this argument and for which the process has permission to send a signal.

thread_id (input)

Ignored unless the function value is 2. If the thread identifier is nonzero, the signal is sent to the identified thread. If the thread identifier is zero, the signal is sent to the process(es) based on the process identifier setting.

cond_rep (input)

The condition token defining the signal to be raised. The valid conditions are CEE5201 through CEE5222. CEE5223 and CEE5234 are supported in z/OS

UNIX System Services. Additionally, CEE5200 represents the signal number value of zero and indicates the request for the validation the arguments but causes no signal to be sent. For a list of condition tokens that map to signals, see *z/OS Language Environment Programming Guide*.

sig_val (input)

Ignored unless the function value is 3. It indicates the value to be supplied with the signal when it is delivered to the process identified by the pid.

q_data_token (input/optional)

32-bit data to be placed in the ISI for use in accessing the qualifying data associated with the given instance of the signal.

fc (output/optional)

A condition token returned by the service, indicating the degree of success of the service. Note that the module returning the code is in parentheses, but to the caller, it appears that CEEOKILL can return any of these.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The function failed due to POSIX(OFF) in effect (CEEOKILL).
CEE55L	Severity	3
	Msg_No	5301
	Message	The value specified by cond_rep represents a condition that is not valid or a condition that is not a POSIX signal as defined for this product (CEEOKILL).
CEE55M	Severity	3
	Msg_No	5302
	Message	The service was unsuccessful due to a z/OS Environmental or Internal error (CEEOSIGG). Consult the Reason_Code returned to determine the exact reason the error occurred. The following reason code can accompany this error: JRPTCANCELERROR.

Note that Return_Code and Reason_Code are returned as part of the qualifying data information of the *fc*, as shown in Figure 111.

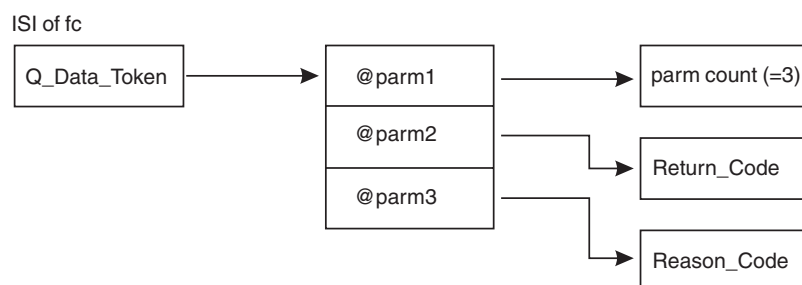


Figure 111. Condition qualifying data returned by CEEOKILL CWI

Thread keyed data CWIs

The CWIs in this section support POSIX thread keyed data functions.

CEEOPGS

C library interface: *pthread_getspecific()*

CEEOPGS obtains the thread-specific value associated with a key that was obtained from a previous call to CEEOPKC. Different threads can have different values bound to the same key.

When successful, CEEOPGS stores the value currently bound to the specified *key* to the storage location referred to by the storage location which is in turn referred to by *value*. Language Environment manages the storage associated with the key/value bindings.

Syntax

void CEEOPGS (*key*, *value*, [*fc*])

```
CEE_THDKEY *key;
POINTER    *value;
FEED_BACK  *fc;
```

CEEOPGS

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0096(,R15)
BALR R14,R15
```

key (input)

The identifier for which the value is to be obtained. The key is generated by a previous call to CEEOPKC.

value (output)

The address of the address of the location to store the value currently associated with the key identifier. The value binding for the key is specific to the thread. The value typically is the address of a storage area to be unallocated during thread termination by the destructor function.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in creating the new key.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	POSIX services not available.

Condition		
CEE5CQ	Severity	3
	Msg_No	5530
	Message	The key value is not valid. That is, the key identifier is not one of the keys previously defined by CEEOPKC.
CEE5CS	Severity	3
	Msg_No	5532
	Message	Thread termination is in progress. This operation is not allowed. A key get operation is not permitted during thread termination.
CEE5CT	Severity	3
	Msg_No	5533
	Message	Program interrupt referring to user parameters.

Usage Notes:

1. Different threads can bind different values to the same key.
2. This function cannot be called during thread termination.

CEEOPKC

C library interface: *pthread_key_create()*

CEEOPKC creates a new unique key in the enclave of the caller and in the context of the current enclave. The *destructor* is a pointer to a function to be executed upon thread termination. The CEEOPKC service assigns a key identifier and returns it in the location referred to by *key*. Key identifiers and their associated destructor functions are common to all threads in the enclave.

Syntax

void CEEOPKC (*key*, [*destructor*], [*fc*])

```
CEE_THDKEY *key;
CEE_ENTRY *destructor;
FEED_BACK *fc;
```

CEEOPKC

Call this CWI interface as follows:

```
L    R15,CEECAALE0V-CEECAA(,R12)    CAA address is in R12
L    R15,0092(,R15)
BALR R14,R15
```

key (output)

The unique key identifier generated by Language Environment. Any thread within the enclave can refer to this key.

destructor (input/optional)

The function pointer which is the user routine to gain control during thread termination. This routine must be an external routine (not a nested procedure). This parameter can be omitted.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in creating the new key.

CEEOPKC CWI

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	POSIX is not initialized.
CEE5CM	Severity	3
	Msg_No	5526
	Message	There was not enough storage available to create the new key.
CEE5CN	Severity	3
	Msg_No	5527
	Message	The key name space is exhausted. The key creation would have resulted in more than the system imposed limit for the maximum number of data keys which can be created per enclave.
CEE5CO	Severity	3
	Msg_No	5528
	Message	Thread termination is in progress. This operation is not allowed. Key creation is not permitted during thread termination.
CEE5CT	Severity	3
	Msg_No	5533
	Message	The key pointer passed is not valid.

Usage Notes:

1. It is assumed that *destructor* is currently available and does not require an explicit LOAD performed.
2. The key identifier returned can be used by all threads within the enclave that uses the CEEOPSS and CEEOPGS services.

CEEOPKD

C library interface: *pthread_key_delete()*

CEEOPKD deletes a thread-specific data key in the caller's enclave and in the context of the current enclave.

Syntax

void CEEOPKD (*key*, [*fc*])

```
CEE_THDKEY *key;  
FEED_BACK *fc;
```

CEEOPKD

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12  
L    R15,0564(,R15)  
BALR R14,R15
```


key (input)

A key identifier returned by a previous invocation of CEEOPKC.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in creating the new key.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	POSIX is not initialized.
CEE5CT	Severity	3
	Msg_No	5533
	Message	The key pointer passed is not valid.

CEEOPSS

C library interface: *pthread_setspecific()*

CEEOPSS establishes a thread-specific value to a key obtained by a previous call to CEEOPKC. Different threads can bind different values to the same key.

When successful, CEEOPSS obtains the value from the location referred to by *value* and assigns it to a Language Environment-managed storage location associated with the *key*.

Syntax

void CEEOPSS (*key, value, [fc]*)

```
CEE_THDKEY *key;
CEE_TOKEN  *value;
FEED_BACK  *fc;
```

CEEOPSS

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0100(,R15)
BALR R14,R15
```

key (input)

The identifier to associate with the value. The identifier is generated by a previous call to CEEOPKC.

value (input)

The value to be associated with the key identifier. The value binding for the key is specific to the thread. The value typically is the address of a storage area to be unallocated during thread termination by the destructor function.

CEEOPSS CWI

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in creating the new key.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE5CP	Severity	3
	Msg_No	5529
	Message	There was not enough storage available to bind the value to the key.
CEE5CQ	Severity	3
	Msg_No	5530
	Message	The key value is not valid. The key identifier is not one of the keys previously defined by CEEOPKC.
CEE5CR	Severity	3
	Msg_No	5531
	Message	Thread termination is in progress. This operation is not allowed. A key set operation is not permitted during thread termination.
CEE5CT	Severity	3
	Msg_No	5533
	Message	Incorrect user parameter caused a program exception.

Usage Notes:

1. Different threads can bind different values to the same key.
2. Calling this function during thread termination can result in undefined behavior.

Thread cancellation CWIs

The CWIs in this section support POSIX thread cancellation functions.

Usage Notes:

1. The *routine* to be executed was previously established by the CEEOPCPU service.
2. It is assumed that *routine* is currently available and does not require an explicit LOAD performed.

CEEOPCPO

C library interface: *pthread_cleanup_pop()*

CEEOPCPO removes the routine at the top of the cleanup stack of the calling thread and optionally executes it, if *execute* is nonzero. The cleanup stack is what is specific to cleanup routines registered for the thread by the CEEOPCPU service. The processing of the registered cleanup routines also takes place at thread termination.

Syntax

void CEEOPCPO (*execute*, [*fc*])

```
INT4      *execute;
FEED_BACK *fc;
```

CEEOPCPO

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L   R15,0060(,R15)
BALR R14,R15
```

execute (**input**)

An indicator to execute the cleanup routine that is being popped. If *execute* is nonzero, the routine that was previously registered through the CEEOPCPU service is executed.

fc (**output/optional**)

The feedback code returned by the service. It indicates the degree of success in popping and optionally executing the routine.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	POSIX services not available.
CEE5FS	Severity	3
	Msg_No	5628
	Message	Thread termination is in progress. This operation is not allowed. Calls to cleanup routine functions are not permitted during thread termination.
CEE5FT	Severity	3
	Msg_No	5629
	Message	No routine to execute (stack is empty).

CEEOPCPU

C library interface: *pthread_cleanup_push()*

CEEOPCPU registers a new thread-specific cleanup routine. The *routine* is a pointer to a function to be executed as a result of thread termination, and optionally as part of the processing of CEEOPCPO. The *arg* value refers to an optional argument that is passed to the cleanup routine when it is called. The registration of a cleanup routine is on a per-thread basis at a given stack frame.

Syntax

void CEEOPCPU (*routine*, [*arg*], [*fc*])

CEEOPCPU CWI

```
CEE_ENTRY *routine;  
CEE_TOKEN *arg;  
FEED_BACK *fc;
```

CEEOPCPU

Call this CWI interface as follows:

```
L   R15,CEECAALE0V-CEECAA(,R12)   CAA address is in R12  
L   R15,0064(,R15)  
BALR R14,R15
```

routine (input)

The entry point of a routine which is to be executed at thread termination and (optionally) upon a call to CEEOPCPO. An optional value can be passed to this routine.

arg (input/optional)

An argument that is passed to *routine* when it is executed.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in pushing the cleanup routine.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	POSIX services not available.
CEE5FQ	Severity	3
	Msg_No	5626
	Message	There was not enough storage available to register the cleanup handler.
CEE5FR	Severity	3
	Msg_No	5627
	Message	Thread termination is in progress. This operation is not allowed. Cleanup routine registration is not permitted during thread termination.

Usage Notes:

1. It is assumed that *routine* is currently available and does not require an explicit LOAD performed.
2. This routine does not do any validation of the routine address.
3. Cleanup handlers that are pushed in a destructor routine but not popped explicitly, are not executed. According to POSIX, the order of execution at thread termination is first cleanup handlers, then destructor routines.
4. If a `longjmp()` is executed that exits the cleanup handler and returns into a point of the executing code, remaining cleanup handlers that have not yet been popped remain pending. If POSIX is ON, a `longjmp()` out of a cleanup handler is defined as an undefined behavior (such as an error).

Thread synchronization — mutex and read-write locks

The CWIs in this section support POSIX mutex and read-write locks thread synchronization.

CEEOPMD

C library interfaces: *pthread_mutex_destroy()*, *pthread_rwlock_destroy()*

CEEOPMD destroys the mutex or read-write lock referred to by *lock_object*. Attempting to destroy a locked mutex or read-write lock results in an error condition.

Syntax

void CEEOPMD (*lock_object*, [*fc*])

```
CEE_MUTEX  *lock_object;
FEED_BACK  *fc;
```

CEEOPMD

Call this CWI interface as follows:

```
L   R15,CEECAALE0V-CEECAA(,R12)    CAA address is in R12
L   R15,0284(,R15)
BALR R14,R15
```

lock_object (**input**)

The mutex or read-write lock to be destroyed.

fc (**output/optional**)

The feedback code returned by the service. It indicates the degree of success in destroying the mutex or read-write lock.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5I5	Severity	3
	Msg_No	5701
	Message	The pthread_mutex_t object specified by <i>lock_object</i> is not valid (not initialized).
CEE5I7	Severity	3
	Msg_No	5703
	Message	Address exception accessing pthread_mutex_t object specified by <i>lock_object</i> .

CEEOPMD CWI

Condition		
CEE5I8	Severity	4
	Msg_No	5704
	Message	Address exception while referencing storage allocated by mutex initialization for pthread_mutex_t object specified by <i>lock_object</i> .
CEE5I9	Severity	3
	Msg_No	5705
	Message	Pthread_mutex_t object specified by <i>lock_object</i> is damaged.
CEE5II	Severity	3
	Msg_No	5714
	Message	The pthread_mutex_t object specified by <i>lock_object</i> is busy.
CEE5IK	Severity	4
	Msg_No	5716
	Message	Unable to free storage allocated by mutex initialization for pthread_mutex_t object specified by <i>lock_object</i> .
CEE5K4	Severity	3
	Msg_No	5764
	Message	The lock object specified by <i>lock_object</i> was not initialized.
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	Address exception while referencing system storage allocated by lock object initialization.
CEE5K8	Severity	3
	Msg_No	5768
	Message	The lock object specified by <i>lock_object</i> has been changed since it was initialized.
CEE5KH	Severity	3
	Msg_No	5777
	Message	The lock object specified by <i>lock_object</i> was busy.
CEE5KJ	Severity	4
	Msg_No	5779
	Message	System lock storage could not be freed.
CEE5KQ	Severity	3
	Msg_No	5786
	Message	The callable service BPX1SLK failed during shared lock processing. The system return code was <i>return_code</i> , the reason code was <i>reason_code</i> , X'00'.

Condition		
CEE5L4	Severity	3
	Msg_No	5796
	Message	The callable service, BPX1SMC, failed during shared lock processing. The system return code was <i>return_code</i> . The reason code was <i>return_code</i> X'00'.

CEEOPMI

C library interfaces: *pthread_mutex_init()*, *pthread_rwlock_init()*

CEEOPMI initializes the mutex or read-write lock referred to by *lock_object* with the attributes identified by *attr_object*. If this function fails, the mutex or read-write lock is not initialized and the contents of *lock_object* is undefined.

Syntax

void CEEOPMI (*lock_object*, *attr_object*, *lock_type*, [*fc*])

```
CEE_MUTEX      *lock_object;
CEE_LOCKATTR  *attr_object;
INT4          *lock_type;
FEED_BACK     *fc;
```

CEEOPMI

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L   R15,0280(,R15)
BALR R14,R15
```

***lock_object* (input/output)**

The mutex or read-write lock to be initialized.

***attr_object* (input)**

The attributes object used to initialize the mutex or read-write lock.

***lock_type* (input)**

A full word integer with the following defined values:

X'00000000'
mutex (without `_OPEN_SYS_MUTEX_EXT` feature)

X'00000001'
read-write lock

X'00000002'
extended mutex (with `_OPEN_SYS_MUTEX_EXT` feature)

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in initializing the *lock_object*.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.

Condition		
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5I7	Severity	3
	Msg_No	5703
	Message	Address exception accessing object specified by <i>lock_object</i> or <i>attr_object</i> .
CEE5I8	Severity	4
	Msg_No	5704
	Message	Address exception while referencing storage allocated by mutex initialization for pthread_mutex_t object specified by <i>lock_object</i> .
CEE5IC	Severity	3
	Msg_No	5708
	Message	The pthread_mutex_t object specified by <i>lock_object</i> was already initialized.
CEE5ID	Severity	3
	Msg_No	5709
	Message	The pthread_mutexattr_t object specified by <i>attr_object</i> is not valid (not initialized).
CEE5IE	Severity	3
	Msg_No	5710
	Message	Insufficient storage to initialize the pthread_mutex_t object specified by <i>lock_object</i> .
CEE5IK	Severity	4
	Msg_No	5716
	Message	Unable to free storage allocated by mutex initialization for pthread_mutex_t object specified by <i>lock_object</i> .
CEE5IP	Severity	0
	Msg_No	5721
	Message	Insufficient resource to initialize mutex specified by <i>lock_object</i> .
CEE5IQ	Severity	0
	Msg_No	5722
	Message	Insufficient privilege to initialize mutex specified by <i>lock_object</i> .
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	Address exception while referencing system storage allocated by lock object initialization.

Condition		
CEE5KB	Severity	3
	Msg_No	5771
	Message	The pthread_rwlock_t object specified by <i>lock_object</i> was already initialized.
CEE5KC	Severity	3
	Msg_No	5772
	Message	The lock attribute object specified by <i>attr_object</i> was not initialized.
CEE5KD	Severity	3
	Msg_No	5773
	Message	Insufficient storage to initialize the pthread_rwlock_t object specified by <i>lock_object</i> .
CEE5KJ	Severity	4
	Msg_No	5779
	Message	System lock storage could not be freed.
CEE5KO	Severity	0
	Msg_No	5784
	Message	Insufficient resource to initialize another read-write lock specified by <i>lock_object</i> .
CEE5KP	Severity	0
	Msg_No	5785
	Message	Insufficient privilege to initialize the read-write lock specified by <i>lock_object</i> .
CEE5KQ	Severity	3
	Msg_No	5786
	Message	The callable service BPX1SLK failed during shared lock processing. The system return code was <i>return_code</i> , the reason code was <i>reason_code</i> , X'00'.
CEE5L4	Severity	3
	Msg_No	5796
	Message	The callable service, BPX1SMC, failed during shared lock processing. The system return code was <i>return_code</i> . The reason code was <i>return_code</i> , X'00'.

CEEOPML

C library interface: *pthread_mutex_lock()*

CEEOPML acquires (locks) the mutex referred to by *mutex*. If the mutex is already locked by another thread, the calling thread blocks until the mutex becomes available. This function returns with the mutex in the locked state with the calling thread as its owner.

Syntax

void CEEOPML (*mutex*, [*fc*])

```
CEE_MUTEX  *mutex;
FEED_BACK  *fc;
```

CEEOPML

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)   CAA address is in R12
L   R15,0288(,R15)
BALR R14,R15
```

***mutex* (input)**

The mutex to be locked.

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in locking the mutex.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5I6	Severity	3
	Msg_No	5702
	Message	The current thread already owns the pthread_mutex_t object specified by <i>mutex</i> .
CEE5IO	Severity	4
	Msg_No	5720
	Message	Mutex specified by <i>mutex</i> was not locked because thread was forced to terminate.
CEE5IS	Severity	0
	Msg_No	5724
	Message	Not enough resource (other than memory).
CEE5K4	Severity	3
	Msg_No	5764
	Message	The mutex specified by <i>mutex</i> was not initialized.
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	Address exception while referencing system storage allocated by lock object initialization.

Condition		
CEE5KQ	Severity	3
	Msg_No	5786
	Message	The callable service BPX1SLK failed during shared lock processing. The system return code was <i>return_code</i> , the reason code was <i>reason_code</i> , X'00'.
CEE5L4	Severity	3
	Msg_No	5796
	Message	The callable service, BPX1SMC, failed during shared lock processing. The system return code was <i>return_code</i> . The reason code was <i>return_code</i> X'00'.

Usage Notes:

1. An attempt by the current owner of a mutex to relock the mutex is allowed if the CEEOPXS service was used to give the mutex the attribute RECURSIVE before the mutex was initialized with the CEEOPMI service. Otherwise, the mutex has (by default) the attribute NONRECURSIVE, and any request to relock it fails.
2. A recursive mutex must be unlocked as many times as it has been locked and relocked to relinquish ownership.
3. Only the owning thread (the thread which acquired a mutex) can unlock it.

CEEOPML2

C library interface: *pthread_mutex_lock()*

CEEOPML2 is the C-callable Language Environment interface to the *pthread_mutex_lock()* function.

Syntax

```
int CEEOPML2 (mutex)
```

```
pthread_mutex_t *mutex;
```

CEEOPML2

Call this CWI interface as follows:

```
L    R15,CEECAALE0V-CEECAA(,R12)
```

```
L    R15,316(,R15)
```

```
BALR R14,R15
```

***mutex* (input)**

Identifies the mutex to be locked.

CEEOPML2 can return the following function values:

- 0 Function completed successfully.
- 1 An error occurred; *errno* can be one of the following values:
 - EINVAL**
The specified mutex is not valid.
 - EDEADLK**
The specified mutex is already locked.
 - EAGAIN**
The specified mutex could not be acquired because the maximum number of recursive locks for the mutex has been exceeded.

CEEOPMT

C library interface: *pthread_mutex_trylock()*

CEEOPMT conditionally acquires (locks) the mutex referred to by *mutex*. Conditionally means the call always returns immediately, whether or not the lock is acquired. If the mutex is already locked, the mutex is not acquired. When successful, this function returns with the mutex in the locked state with the calling thread as its owner.

Syntax

void CEEOPMT (*mutex*, [*fc*])

```
CEE_MUTEX *mutex;
FEED_BACK *fc;
```

CEEOPMT

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0296(,R15)
BALR R14,R15
```

mutex (input)

The mutex to be locked conditionally.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in locking the mutex conditionally. The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5I5	Severity	3
	Msg_No	5701
	Message	The pthread_mutex_t object specified by <i>mutex</i> is not valid (not initialized).
CEE5I8	Severity	4
	Msg_No	5704
	Message	Address exception while referencing storage allocated by mutex initialization for pthread_mutex_t object specified by <i>mutex</i> .
CEE5IB	Severity	0
	Msg_No	5707
	Message	The pthread_mutex_t object specified by <i>mutex</i> is busy.
CEE5IO	Severity	4
	Msg_No	5720
	Message	Thread forced by quiesce.

Condition		
CEE5IS	Severity	0
	Msg_No	5724
	Message	Not enough resource (other than memory).
CEE5K4	Severity	3
	Msg_No	5764
	Message	The mutex specified by <i>mutex</i> was not initialized.
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	Address exception while referencing system storage allocated by lock object initialization.
CEE5KQ	Severity	3
	Msg_No	5786
	Message	The callable service BPX1SLK failed during shared lock processing. The system return code was <i>return_code</i> , the reason code was <i>reason_code</i> , X'00'.

Usage Notes:

1. If the CEEOPXS service was used to give the mutex the attribute RECURSIVE before the mutex was initialized with the CEEOPMI service, the thread which locked the mutex can relock it with lock service, CEEOPML, or trylock service, CEEOPMT. In other words, trylock returns with success rather than busy feedback code, if the thread already owns a recursive mutex. Only when the mutex is nonrecursive, which is the default attribute for mutexes, does trylock return the busy feedback code if the mutex is already locked.
2. A recursive mutex must be unlocked as many times as it has been locked and relocked to relinquish ownership.
3. Only the owning thread (the thread that acquired a mutex) can unlock it.

CEEOPMU

C library interface: *pthread_mutex_unlock()*

CEEOPMU releases a mutex held by the calling thread. A recursive mutex must be unlocked as many times as it has been locked and relocked to relinquish ownership. Only the owning thread (the thread that acquired a mutex) can unlock it.

Syntax

void CEEOPMU (*mutex*, [*fc*])

```
CEE_MUTEX  *mutex;
FEED_BACK  *fc;
```

CEEOPMU

Call this CWI interface as follows:

CEEOPMU CWI

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0292(,R15)
BALR R14,R15
```

mutex (input)

The mutex to be unlocked.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in unlocking the mutex.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5IA	Severity	3
	Msg_No	5706
	Message	The current thread does not own the pthread_mutex_t object specified by <i>mutex</i> .
CEE5K4	Severity	3
	Msg_No	5764
	Message	The mutex specified by <i>mutex</i> was not initialized.
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	Address exception while referencing system storage allocated by lock object initialization.
CEE5KQ	Severity	3
	Msg_No	5786
	Message	The callable service BPX1SLK failed during shared lock processing. The system return code was <i>return_code</i> , the reason code was <i>reason_code</i> , X'00'.
CEE5L4	Severity	3
	Msg_No	5796
	Message	The callable service, BPX1SMC, failed during shared lock processing. The system return code was <i>return_code</i> . The reason code was <i>return_code</i> X'00'.

CEEOPMU2

C library interface: *pthread_mutex_unlock()*

CEEOPMU2 is the C-callable Language Environment interface to the `pthread_mutex_unlock()` function.

Syntax

```
int CEEOPMU2 (mutex)
pthread_mutex_t *mutex;
```

CEEOPMU2

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)
L    R15,320(,R15)
BALR R14,R15
```

mutex (input)

Identifies the mutex to be locked.

CEEOPMU2 can return the following function values:

- 0 Function completed successfully.
- 1 An error occurred; `errno` can be one of the following values:
 - EINVAL**
 The specified mutex is not valid.
 - EPERM**
 The current thread does not own the mutex.

CEEOPRL

C library interface: `pthread_rwlock_rdlock()`

CEEOPRL acquires (locks) the read-write lock for read referred to by *rwlock*. If the read-write lock is already locked for write by another thread, or if there are threads waiting for the read-write lock for write, the calling thread blocks until the read-write lock becomes available. This function returns with the read-write lock in the locked state with the calling thread as its holder.

Syntax

```
void CEEOPRL (rwlock, [fc])
```

```
CEE_RWLOCK *rwlock;
FEED_BACK *fc;
```

CEEOPRL

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)      CAA address is in R12
L    R15,500(,R15)
BALR R14,R15
```

rwlock (input)

The read-write lock to be locked for read.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in locking the read-write lock for read.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5K4	Severity	3
	Msg_No	5764
	Message	The read-write lock specified by the <i>rwlock</i> was not initialized.
CEE5K5	Severity	3
	Msg_No	5765
	Message	The read-write lock specified by <i>rwlock</i> had already been locked by the same thread for writing. A read-write lock can only be locked for reading multiple times by the same thread.
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	Address exception while referencing system storage allocated by lock object initialization.
CEE5KN	Severity	4
	Msg_No	5783
	Message	Read-write lock specified by <i>rwlock</i> was not locked because thread was forced to terminate.
CEE5KQ	Severity	3
	Msg_No	5786
	Message	The callable service BPX1SLK failed during shared lock processing. The system return code was <i>return_code</i> , the reason code was <i>reason_code</i> , X'00'.
CEE5KR	Severity	3
	Msg_No	5787
	Message	Not enough resource (other than memory).
CEE5KU	Severity	3
	Msg_No	5790
	Message	Insufficient storage to lock the read-write lock object specified by <i>rwlock</i> .
CEE5KV	Severity	4
	Msg_No	5791
	Message	System read-write lock storage could not be freed.

Usage Notes:

1. Multiple threads will be granted the read-write lock for read if no thread holds the read-write lock for write and no thread is blocked waiting for the read-write lock for write.
2. A read-write lock locked for read must be unlocked as many times as it has been locked and relocked to relinquish ownership.

CEEOPRL2

C library interface: *pthread_rwlock_rdlock()*

CEEOPRL2 is the C-callable Language Environment interface to the *pthread_rwlock_rdlock()* function for a read lock.

Syntax

int CEEOPRL2 (*rwlock*)

pthread_rwlock_t **rwlock*;

CEEOPRL2

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)
L   R15,520(,R15)
BALR R14,R15
```

rwlock (**input**)

Identifies the read-write lock to be locked for read.

CEEOPRL2 can return the following function values:

- 0 Function completed successfully.
- 1 An error occurred; *errno* can be one of the following values:

EINVAL

The specified read-write lock is not valid.

EDEADLK

The specified read-write lock is already locked.

EAGAIN

The specified read-write lock could not be acquired because the maximum number of recursive locks for the read-write lock has been exceeded.

ENOMEM

Not enough memory to acquire a lock.

CEEOPRT

C library interface: *pthread_rwlock_tryrdlock()*

CEEOPRT conditionally acquires (locks) the read-write lock for read referred to by *rwlock*. Conditionally means the call always returns immediately, whether or not the lock is acquired. If the read-write lock is already locked for write, or if there are threads waiting for the read-write lock for write, the read-write lock is not acquired. The exception is when the calling thread has already locked the read-write lock for read, in which case it will still acquire the lock. When successful, this function returns with the read-write lock in the locked state with the calling thread as its owner.

Note: Only the owning thread (the thread that acquired a read-write lock) can unlock it.

Syntax

void CEEOPRT (*rwlock*, [*fc*])

CEE_RWLOCK **rwlock*;

FEED_BACK **fc*;

CEEOPRT

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
```

```
L    R15,504(,R15)
```

```
BALR R14,R15
```

rwlock (input)

The read-write lock to be locked for read conditionally.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in locking the read-write lock for read conditionally.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5K4	Severity	3
	Msg_No	5764
	Message	The read-write lock specified by the <i>rwlock</i> was not initialized.
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	Address exception while referencing system storage allocated by lock object initialization.
CEE5KN	Severity	4
	Msg_No	5783
	Message	Read-write lock specified by <i>rwlock</i> was not locked because thread was forced to terminate.

Condition		
CEE5KQ	Severity	3
	Msg_No	5786
	Message	The callable service BPX1SLK failed during shared lock processing. The system return code was <i>return_code</i> , the reason code was <i>reason_code</i> , X'00".
CEE5KR	Severity	3
	Msg_No	5787
	Message	Not enough resource (other than memory).
CEE5KS	Severity	3
	Msg_No	5788
	Message	The read-write lock specified by <i>rwlock</i> was busy.
CEE5KU	Severity	3
	Msg_No	5790
	Message	Insufficient storage to lock the read-write lock object specified by <i>rwlock</i> .
CEE5KV	Severity	4
	Msg_No	5791
	Message	System read-write lock storage could not be freed.

Usage Notes:

1. It is assumed that *routine* is currently available and does not require an explicit LOAD performed.

CEEOPRU

C library interface: *pthread_rwlock_unlock()*

CEEOPRU releases a read-write lock held by the calling thread. A read-write lock must be unlocked as many times as it has been locked, and relocked to relinquish ownership. Only the holding thread (the thread that acquired a read-write lock) can unlock it.

Syntax

void CEEOPRU (*rwlock*, [*fc*])

```
CEE_RWLOCK *rwlock;
FEED_BACK *fc;
```

CEEOPRU

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,516(,R15)
BALR R14,R15
```

***rwlock* (input)**

The read-write lock to be unlocked.

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in unlocking the read-write lock.

CEEOPRU CWI

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5K4	Severity	3
	Msg_No	5764
	Message	The read-write lock specified by the <i>rwlock</i> was not initialized.
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	Address exception while referencing system storage allocated by lock object initialization.
CEE5KN	Severity	4
	Msg_No	5783
	Message	Read-write lock specified by <i>rwlock</i> was not locked because thread was forced to terminate.
CEE5KQ	Severity	3
	Msg_No	5786
	Message	The callable service BPX1SLK failed during shared lock processing. The system return code was <i>return_code</i> , the reason code was <i>reason_code</i> , X'00'.
CEE5KU	Severity	3
	Msg_No	5790
	Message	Insufficient storage to lock the read-write lock object specified by <i>rwlock</i> .
CEE5KV	Severity	4
	Msg_No	5791
	Message	System read-write lock storage could not be freed.

CEEOPRU2

C library interface: *pthread_rwlock_unlock()*

CEEOPRU2 is the C-callable Language Environment interface to the *pthread_rwlock_unlock()* function.

Syntax

```
int CEEOPRU2 (rwlock)
pthread_rwlock_t *rwlock;
```

CEEOPRU2

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)
L    R15,528(,R15)
BALR R14,R15
```

rwlock (input)

The read-write lock to be unlocked.

CEEOPRU2 can return the following function values:

- 0 Function completed successfully.
- 1 An error occurred; *errno* can be one of the following values:
 - EINVAL**
The specified read-write lock is not valid.
 - EPERM**
The current thread does not own the read-write lock object.
 - ENOMEM**
There is not enough memory during the unlock process.

CEEOPWL

C library interface: *pthread_rwlock_wrlock()*

CEEOPWL acquires (locks) the read-write lock for write referred to by *rwlock*. If the read-write lock is already locked by another thread or threads, the calling thread blocks until the read-write lock becomes available. This function returns with the read-write lock in the locked state with the calling thread as a holder.

Syntax

```
void CEEOPWL (rwlock, [fc])
```

```
CEE_RWLOCK *rwlock;  
FEED_BACK *fc;
```

CEEOPWL

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)      CAA address is in R12
L    R15,508(,R15)
BALR R14,R15
```

rwlock (input)

The read-write lock to be locked for write.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in locking the read-write lock for write.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5K4	Severity	3
	Msg_No	5764
	Message	The read-write lock specified by the <i>rwlock</i> was not initialized.
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	Address exception while referencing system storage allocated by lock object initialization.
CEE5KA	Severity	3
	Msg_No	5770
	Message	The read-write lock specified by <i>rwlock</i> has already been locked by the thread. A read-write lock can only be locked for reading multiple times by the same thread.
CEE5KN	Severity	4
	Msg_No	5783
	Message	Read-write lock specified by <i>rwlock</i> was not locked because thread was forced to terminate.
CEE5KQ	Severity	3
	Msg_No	5786
	Message	The callable service BPX1SLK failed during shared lock processing. The system return code was <i>return_code</i> , the reason code was <i>reason_code</i> , X'00'.
CEE5KU	Severity	3
	Msg_No	5790
	Message	Insufficient storage to lock the read-write lock object specified by <i>rwlock</i> .
CEE5KV	Severity	4
	Msg_No	5791
	Message	System read-write lock storage could not be freed.

Usage Notes:

1. An attempt by the current holder of a read-write lock for write or read to relock the read-write lock for write will cause a deadlock.
2. A read-write lock must be unlocked as many times as it has been locked, and relocked to relinquish ownership.

- Only the holding thread (the thread which acquired a read-write lock for write) can unlock it.

CEEOPWL2

C library interface: *pthread_rwlock_wrlock()*

CEEOPWL2 is the C-callable Language Environment interface to the *pthread_rwlock_wrlock()* function for a write lock.

Syntax

```
int CEEOPWL2 (rwlock)
pthread_rwlock_t *rwlock;
```

CEEOPWL2

Call this CWI interface as follows:

```
L    R15,CEECAALE0V-CEECAA(,R12)
L    R15,524(,R15)
BALR R14,R15
```

rwlock (input)

Identifies the read-write lock to be locked for write.

CEEOPWL2 can return the following function values:

- 0 Function completed successfully.
- 1 An error occurred; *errno* is set to one of the following values:
 - EINVAL**
The specified read-write lock is not valid.
 - EDEADLK**
The specified read-write lock is already locked for write.
 - ENOMEM**
Not enough memory to acquire a lock.

CEEOPWT

C library interface: *pthread_rwlock_trywrlock()*

CEEOPWT conditionally acquires (locks) the read-write lock for write referred to by *rwlock*. Conditionally means the call always returns immediately, whether or not the lock is acquired. If the read-write lock is already locked, the read-write lock is not acquired. When successful, this function returns with the read-write lock in the locked state with the calling thread as its owner.

Note: Only the owning thread (the thread that acquired a read-write lock) can unlock it.

Syntax

```
void CEEOPWT (rwlock, [fc])
CEE_RWLOCK *rwlock;
```

CEEOPWT

Call this CWI interface as follows:

```
L    R15,CEECAALE0V-CEECAA(,R12)       CAA address is in R12
L    R15,512(,R15)
BALR R14,R15
```

***rwlock* (input)**

The read-write lock to be locked for write conditionally.

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in locking the read-write lock for write conditionally.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5K4	Severity	3
	Msg_No	5764
	Message	The read-write lock specified by the <i>rwlock</i> was not initialized.
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	Address exception while referencing system storage allocated by lock object initialization.
CEE5KN	Severity	4
	Msg_No	5783
	Message	Read-write lock specified by <i>rwlock</i> was not locked because thread was forced to terminate.
CEE5KQ	Severity	3
	Msg_No	5786
	Message	The callable service BPX1SLK failed during shared lock processing. The system return code was <i>return_code</i> , the reason code was <i>reason_code</i> , X'00'.
CEE5KT	Severity	1
	Msg_No	5785
	Message	The read-write lock specified by <i>rwlock</i> was busy.
CEE5KU	Severity	3
	Msg_No	5790
	Message	Insufficient storage to lock the read-write lock object specified by <i>rwlock</i> .
CEE5KV	Severity	4
	Msg_No	5791
	Message	System read-write lock storage could not be freed.

Usage notes:

1. The *routine* to be executed was previously established by the `pthread_cleanup_push()` (CEEOPCPO) function.
2. It is assumed that *routine* is currently available and does not require an explicit LOAD performed.

CEEOPXD

C library interface: `pthread_mutexattr_destroy()`, `pthread_rwlockattr_destroy()`

CEEOPXD destroys a mutex attributes object or read-write lock attributes object.

Syntax

void CEEOPXD (*attr_object*, [*fc*])

```
CEE_LOCKATTR  *attr_object;
FEED_BACK     *fc;
```

CEEOPXD

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L   R15,0328(,R15)
BALR R14,R15
```

***attr_object* (input/output)**

The initialized mutex attributes or read-write lock attributes object.

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in deleting the mutex attributes or read-write lock attributes object.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5I7	Severity	3
	Msg_No	5703
	Message	Address exception accessing pthread_mutexattr_t object specified by <i>attr_object</i> .
CEE5I8	Severity	4
	Msg_No	5704
	Message	Address exception while referencing storage allocated by mutex initialization for pthread_mutex_t object specified by <i>attr_object</i> .

Condition		
CEE5ID	Severity	3
	Msg_No	5709
	Message	The pthread_mutexattr_t object specified by <i>attr_object</i> was not initialized.
CEE5IF	Severity	3
	Msg_No	5711
	Message	The pthread_mutexattr_t object specified by <i>attr_object</i> has been changed since it was initialized.
CEE5IL	Severity	4
	Msg_No	5717
	Message	Unable to free storage allocated by mutex attribute initialization for pthread_mutexattr_t object specified by <i>attr_object</i> .
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock attribute object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	An addressing exception occurred referencing system storage allocated by lock attributes object initialization.
CEE5KC	Severity	3
	Msg_No	5772
	Message	The lock attribute object specified by <i>attr_object</i> was not initialized.
CEE5KE	Severity	3
	Msg_No	5774
	Message	A pthread_rwlockattr_t object specified by <i>attr_object</i> has been changed since it was initialized
CEE5KK	Severity	4
	Msg_No	5780
	Message	Unable to free storage allocated by lock attribute initialization for attributes object specified by <i>attr_object</i> .

CEEOPXG

C library interface: *pthread_mutexattr_getkind_np()*

CEEOPXG returns an integer value indicating mutex or read-write lock attributes specified by a mutex or read-write lock attributes object. For attribute values, see "CEEOPXS" on page 580.

Syntax

void CEEOPXG (*attr_object*, *attr_value*, *call_type*, [*fc*])

```
CEE_LOCKATTR *attr_object;
INT4         *attr_value;
INT4         *call_type;
FEED_BACK    *fc;
```

CEEOPXG

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)   CAA address is in R12
L   R15,0332(,R15)
BALR R14,R15
```

***attr_object* (input/output)**

The initialized attributes object.

***attr_value* (output)**

The location that will contain values specifying mutex or read-write lock attributes. The values returned are described under “CEEOPXS” on page 580, the description of *new_attr_value*.

***call_type* (input)**

A full word integer with the following defined values:

- X'00000001' = settype (setkind_np) attributes for private
- X'00000002' = setpshared attributes for shared.

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in interrogating the mutex or read-write lock attributes object.

Note: *call_type* values can be combined, so that X'00000003' would indicate that both settype and setpshared values may be changed.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	This service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5I8	Severity	4
	Msg_No	5704
	Message	Address exception while referencing storage allocated by mutex attribute initialization for pthread_mutexattr_t object specified by <i>attr_object</i> .
CEE5ID	Severity	3
	Msg_No	5709
	Message	The pthread_mutexattr_t object specified by <i>attr_object</i> is not valid (not initialized).
CEE5IJ	Severity	3
	Msg_No	5715
	Message	Address exception attempting to store attribute value at return address specified by <i>attr_value</i> .

CEEOPXG CWI

Condition		
CEE5IM	Severity	4
	Msg_No	5718
	Message	Valid attribute value for pthread_mutexattr_t object specified by <i>attr_object</i> not found in attribute object storage.
CEE5K7	Severity	4
	Msg_No	5767
	Message	An addressing exception occurred referencing system storage allocated by lock attributes object initialization.
CEE5KC	Severity	3
	Msg_No	5772
	Message	The lock attribute object specified by <i>attr_object</i> was not initialized.
CEE5KI	Severity	3
	Msg_No	5778
	Message	Address exception attempting to store attribute value at return address specified by <i>attr_value</i> .
CEE5KL	Severity	4
	Msg_No	5781
	Message	Valid lock attribute value for lock attribute object specified by <i>attr_object</i> not found in lock attribute storage.

CEEOPXI

C library interface: *pthread_mutexattr_init()*, *pthread_rwlockattr_init()*

CEEOPXI initializes the mutex attributes or read-write lock attributes object specified by *attr_object*. Upon successful completion of this function, the attributes object is set to this implementation's default. For attribute values, see "CEEOPXS" on page 580.

Syntax

void CEEOPXI (*attr_object*, *lock_type*, [*fc*])

```
CEE_LOCKATTR *attr_object;  
INT4 *lock_type;  
FEED_BACK *fc;
```

CEEOPXI

Call this CWI interface as follows:

```
L R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12  
L R15,0324(,R15)  
BALR R14,R15
```

***attr_object* (input)**

The caller-provided attributes object.

***lock_type* (input)**

A full word integer with the following defined values:

- X'00000000' = mutex
- X'00000001' = read-write lock

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in initializing the new attributes object.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5I7	Severity	3
	Msg_No	5703
	Message	Address exception accessing pthread_mutexattr_t object specified by <i>attr_object</i> .
CEE5I8	Severity	4
	Msg_No	5704
	Message	Address exception while referencing storage allocated by mutex attribute initialization for pthread_mutexattr_t object specified by <i>attr_object</i> .
CEE5IG	Severity	3
	Msg_No	5712
	Message	The pthread_mutexattr_t object specified by <i>attr_object</i> was already initialized.
CEE5IH	Severity	3
	Msg_No	5713
	Message	Insufficient storage to initialize the pthread_mutexattr_t object specified by <i>attr_object</i> .
CEE5IL	Severity	4
	Msg_No	5717
	Message	Unable to free storage allocated by mutex attribute initialization for pthread_mutexattr_t object specified by <i>attr_object</i> .
CEE5K6	Severity	3
	Msg_No	5766
	Message	An addressing exception occurred referencing a lock attribute object.
CEE5K7	Severity	4
	Msg_No	5767
	Message	An addressing exception occurred referencing system storage allocated by lock attributes object initialization.
CEE5KF	Severity	3
	Msg_No	5775
	Message	The pthread_rwlockattr_t object specified by <i>attr_object</i> was already initialized.

Condition		
CEE5KG	Severity	3
	Msg_No	5776
	Message	Insufficient storage to initialize the pthread_rwlockattr_t object specified by <i>attr_object</i> .
CEE5KK	Severity	4
	Msg_No	5780
	Message	Unable to free storage allocated by lock attribute initialization for attributes object specified by <i>attr_object</i> .

CEEOPXS

C library interface: *pthread_mutexattr_setkind_np()*

CEEOPXS changes the mutex or read-write lock attributes specified by a mutex attributes object or read-write lock attributes object. For each attribute pair described below a default is listed. This is the default resulting from calling CEEOPXI to initialize the attribute object prior to using CEEOPXS to change any attribute value. Valid attributes are:

RECURSIVE

If a mutex object is initialized with an attribute object which has been assigned the attribute RECURSIVE, the mutex is given the attribute RECURSIVE. A thread can lock and relock a recursive mutex any number of times. However, only the thread owning the recursive mutex can relock it, and only the thread owning the recursive mutex can unlock it. A recursive mutex must be unlocked as many times as it is locked, and relocked to relinquish ownership. The default attribute for a read-write lock is recursive. Unlike a mutex, more than one thread may lock and relock a read-write lock so long as it is only locked for read. For read-write locks, only the RECURSIVE attribute is supported.

NONRECURSIVE

The default attribute for a mutex is nonrecursive. If a thread attempts to relock a nonrecursive mutex that it owns (has already locked), the lock request fails. A read-write lock is by definition recursive and never should be changed to nonrecursive.

DEBUG

State changes to this mutex or read-write lock should be reported to the debug interface. For mutexes, the default is DEBUG.

NODEBUG

State changes to this mutex or read-write lock should *not* be reported to the debug interface even though it is present. For read-write locks the default is NODEBUG. For read-write locks only the NODEBUG attribute is supported.

ERRORCHECK

Prior to increasing the lock object count, determine if increasing the count would result in an error condition. For both mutexes and read-write locks, the default is ERRORCHECK.

NOERRORCHECK

No checking for error conditions is done. NOERRORCHECK can only be set for mutexes.

PRIVATE

The mutex or read-write lock will reside in local memory. The default for mutexes and read-write locks is PRIVATE.

SHARED

The mutex or read-write lock will reside in shared memory.

Syntax

void CEEOPXS (*attr_object*, *new_attr_value*, *call_type*, [*fc*])

```
CEE_LOCKATTR  *attr_object;
INT4          *new_attr_value;
INT4          *call_type;
FEED_BACK    *fc;
```

CEEOPXS

Call this CWI interface as follows:

```
L    R15,CEECAALE0V-CEECAA(,R12)    CAA address is in R12
L    R15,0336(,R15)
BALR R14,R15
```

***attr_object* (input/output)**

The location that contains the token of an initialized attributes object.

***new_attr_value* (input/output)**

The location of a value specifying a mutex or read-write lock attribute. Valid values for the low order 16 bits of *new_attr_value* are:

Settype: For *pthread_mutexattr_setkind_np* (*pthread_mutexattr_settype*):

- 0 NONRECURSIVE+DEBUG+ERRORCHECK
- 1 RECURSIVE+DEBUG+ERRORCHECK
- 2 NONRECURSIVE+NODEBUG+ERRORCHECK
- 3 RECURSIVE+NODEBUG+ERRORCHECK
- 4 NONRECURSIVE+DEBUG+NOERRORCHECK
- 5 RECURSIVE+DEBUG+NOERRORCHECK
- 6 NONRECURSIVE+NODEBUG+NOERRORCHECK
- 7 RECURSIVE+NODEBUG+NOERRORCHECK

Setpshared: For *pthread_mutexattr_setpshared* (and *pthread_rwlockattr_setpshared*):

- 0 PRIVATE
- 8 SHARED

The value for the high order 16 bits of *new_attr_value* must be zero.

***call_type* (input)**

A full word integer with the following defined values:

- X'00000001' = settype (setkind_np) attributes for private
- X'00000002' = setpshared attributes for shared.

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in setting the mutex or read-write lock attributes object. The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5I8	Severity	4
	Msg_No	5704
	Message	Address exception while referencing storage allocated by mutex attribute initialization for pthread_mutexattr_t object specified by <i>attr_object</i> .
CEE5ID	Severity	3
	Msg_No	5709
	Message	The pthread_mutexattr_t object specified by <i>attr_object</i> is not valid (not initialized).
CEE5IN	Severity	3
	Msg_No	5719
	Message	The attribute value specified by <i>new_attr_value</i> is not valid.
CEE5K7	Severity	4
	Msg_No	5767
	Message	An addressing exception occurred referencing system storage allocated by lock attributes object initialization.
CEE5KC	Severity	3
	Msg_No	5772
	Message	The lock attribute object specified by <i>attr_object</i> was not initialized.
CEE5KM	Severity	3
	Msg_No	5782
	Message	The lock attribute value specified by <i>new_attr_value</i> was not valid.

Note: call_type values can be combined, so that X'00000003' would indicate that both settype and setpshared values may be changed.

Thread synchronization — condition variables

The CWIs in this section support POSIX condition variables thread synchronization functions.

CEEOPCB

C library interface: *pthread_cond_broadcast()*

CEEOPCB unblocks all threads (if any) that are waiting (blocked) on the condition object referred to by *cond*. This call has no effect if there are no threads waiting on the condition object.

Syntax

void CEEOPCB (*cond*, [*fc*])

CEE_TOKEN **cond*;
FEED_BACK **fc*;

CEEOPCB

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)   CAA address is in R12
L   R15,0224(,R15)
BALR R14,R15
```

cond (**input**)

The condition object on which other threads can be blocked.

fc (**output/optional**)

The feedback code returned by the service. It indicates the degree of success in signaling the condition.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5IU	Severity	3
	Msg_No	5726
	Message	The condition object specified by <i>cond</i> is not initialized.
CEE5IV	Severity	3
	Msg_No	5727
	Message	Unexpected return code from z/OS UNIX condition post, BPX1CPO, callable service.
CEE5J0	Severity	4
	Msg_No	5728
	Message	Address exception while referencing storage allocated by condition initialization for cond_t object specified by <i>cond</i> .
CEE5J3	Severity	3
	Msg_No	5731
	Message	Unexpected return code from z/OS UNIX condition setup, BPX1CSE, callable service.
CEE5L3	Severity	3
	Msg_No	5795
	Message	The callable service, BPX1SMC, failed during shared condition variable processing. The system return code was <i>return_code</i> . The reason code was <i>return_code</i> X'00'.

CEEOPCD

C library interface: *pthread_cond_destroy()*

CEEOPCD destroys the condition object specified by *cond*. Attempting to destroy a condition object associated with threads in condition wait or timed wait results in an error condition.

Syntax

void CEEOPCD (*cond*, [*fc*])

CEE_COND **cond*;
FEED_BACK **fc*;

CEEOPCD

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)   CAA address is in R12
L   R15,0216(,R15)
BALR R14,R15
```

cond (input)

The condition object to be destroyed.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in destroying the condition object.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5IU	Severity	3
	Msg_No	5726
	Message	The cond_t object specified by <i>cond</i> is not valid (not initialized).
CEE5J0	Severity	4
	Msg_No	5728
	Message	Address exception while referencing storage allocated by condition initialization for cond_t object specified by <i>cond</i> .
CEE5JC	Severity	3
	Msg_No	5740
	Message	Address exception accessing cond_t object specified by <i>cond</i> .
CEE5JH	Severity	4
	Msg_No	5745
	Message	Unable to free storage allocated by condition initialization for cond_t object specified by <i>cond</i> .

Condition		
CEE5JJ	Severity	3
	Msg_No	5747
	Message	The cond_t object specified by <i>cond</i> is busy.
CEE5JK	Severity	3
	Msg_No	5748
	Message	Cond_t object specified by <i>cond</i> is damaged.
CEE5L3	Severity	3
	Msg_No	5795
	Message	The callable service, BPX1SMC, failed during shared condition variable processing. The system return code was <i>return_code</i> . The reason code was <i>return_code X'00'</i> .

CEEOPCI

C library interface: *pthread_cond_init()*

CEEOPCI initializes the condition object referred to by *cond* with the attributes identified by *condattr*. If this function fails, the condition object is not initialized and the contents of *cond* are undefined. If a condition attribute object is not specified, default condition attributes are used.

Syntax

void CEEOPCI (*cond*, *condattr*, [*fc*])

```
CEE_COND    *cond;
CEE_TOKEN   *condattr;
FEED_BACK   *fc;
```

CEEOPCI

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0212(,R15)
BALR R14,R15
```

cond (input/output)

The condition object to be initialized.

condattr (input/optional)

The condition attributes object used to initialize the condition object.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in initializing the condition object.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.

CEEOPCI CWI

Condition		
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5J0	Severity	4
	Msg_No	5728
	Message	Address exception while referencing storage allocated by initialization for object specified by <i>cond</i> or <i>condattr</i> .
CEE5JC	Severity	3
	Msg_No	5740
	Message	Address exception accessing object specified by <i>cond</i> or <i>condattr</i> .
CEE5JE	Severity	3
	Msg_No	5742
	Message	The <i>cond_t</i> object specified by <i>cond</i> is already initialized.
CEE5JF	Severity	3
	Msg_No	5743
	Message	The <i>condattr_t</i> object specified by <i>condattr</i> is not valid (not initialized).
CEE5JG	Severity	3
	Msg_No	5744
	Message	Insufficient storage to initialize the <i>cond_t</i> object specified by <i>cond</i> .
CEE5JH	Severity	4
	Msg_No	5745
	Message	Unable to free storage allocated by condition initialization for <i>cond_t</i> object specified by <i>cond</i> .

CEEOPCS

C library interface: *pthread_cond_signal()*

CEEOPCS unblocks at least one of the threads (if any) that are waiting (blocked) on the condition object referred to by *cond*. This call has no effect if there are no threads waiting on the condition object.

Syntax

void CEEOPCS (*cond*, [*fc*])

CEE_TOKEN **cond*;
FEED_BACK **fc*;

CEEOPCS

Call this CWI interface as follows:

```
L R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L R15,0220(,R15)
BALR R14,R15
```

cond (input)

The condition object on which other threads can be blocked.

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in signaling the condition.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5IU	Severity	3
	Msg_No	5726
	Message	The condition object specified by <i>cond</i> is not initialized.
CEE5IV	Severity	3
	Msg_No	5727
	Message	Unexpected return code from z/OS UNIX condition post, BPX1CPO, callable service.
CEE5J0	Severity	4
	Msg_No	5728
	Message	Address exception while referencing storage allocated by condition initialization for <i>cond_t</i> object specified by <i>cond</i> .
CEE5J3	Severity	3
	Msg_No	5731
	Message	Unexpected return code from z/OS UNIX condition setup, BPX1CSE, callable service.
CEE5L3	Severity	3
	Msg_No	5795
	Message	The callable service, BPX1SMC, failed during shared condition variable processing. The system return code was <i>return_code</i> . The reason code was <i>return_code X'00'</i> .

CEEOPCT

C library interface: *pthread_cond_timedwait()*

This function is same as the CEEOPCW function, except that an error is returned if the absolute time specified by *tv_sec* and *tv_nsec* passes before the condition specified by *cond* is signaled. The associated mutex is reacquired before return.

Syntax

void CEEOPCT (*cond*, *mutex*, *tv_sec*, *tv_nsec*, [*fc*])

CEEOPCT CWI

```
CEE_TOKEN    *cond;
CEE_MUTEX    *mutex;
INT4         *tv_sec;
INT4         *tv_nsec;
FEED_BACK    *fc;
```

CEEOPCT

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECOA(,R12)    CAA address is in R12
L    R15,0232(,R15)
BALR R14,R15
```

cond (input)

The condition variable to wait on.

mutex (input)

The associated locked mutex.

tv_sec (input)

Specifies time in seconds from midnight, January 1, 1970 UTC when CEEOPCT should cease wait for a condition signal. The value specified must be greater than zero (0) and less than 2,147,483,648 seconds. If a positive value is specified and is less than or equal to current calendar time expressed seconds from midnight, January 1, 1970 UTC, CEEOPCT returns immediately, thus indicating that the time to wait has elapsed.

tv_nsec (input)

Specifies time in nanoseconds to be added to *tv_sec* to determine when CEEOPCT should cease wait for a condition signal. The value specified must be greater than or equal to zero (0) and less than 1,000,000,000 (1,000 million).

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in waiting for the condition variable.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5I5	Severity	3
	Msg_No	5701
	Message	The mutex specified by <i>mutex</i> is not initialized.
CEE5I8	Severity	4
	Msg_No	5704
	Message	Address exception while referencing storage allocated by mutex initialization for pthread_mutex_t object specified by <i>mutex</i> .

Condition		
CEE5IA	Severity	3
	Msg_No	5706
	Message	The mutex specified by <i>mutex</i> is not owned by the calling thread.
CEE5IO	Severity	4
	Msg_No	5720
	Message	Mutex specified by <i>mutex</i> not reacquired after condition wait terminated because thread was forced to terminate.
CEE5IU	Severity	3
	Msg_No	5726
	Message	The condition variable specified by <i>cond</i> is not initialized.
CEE5J0	Severity	4
	Msg_No	5728
	Message	Address exception while referencing storage allocated by condition initialization for <i>cond_t</i> object specified by <i>cond</i> .
CEE5J1	Severity	3
	Msg_No	5729
	Message	The mutex specified by <i>mutex</i> is a recursive mutex.
CEE5J2	Severity	3
	Msg_No	5730
	Message	A mutex other than the mutex specified by <i>mutex</i> is already associated with the condition specified by <i>cond</i> .
CEE5J3	Severity	3
	Msg_No	5731
	Message	Unexpected return code from z/OS UNIX condition setup, BPX1CSE, callable service.
CEE5J5	Severity	3
	Msg_No	5731
	Message	The value specified by <i>tv_sec</i> is not valid.
CEE5J6	Severity	3
	Msg_No	5734
	Message	The value specified by <i>tv_nsec</i> is not valid.
CEE5J7	Severity	3
	Msg_No	5735
	Message	System time of day clock is not valid.
CEE5J8	Severity	1
	Msg_No	5736
	Message	The time specified by <i>tv_sec</i> and <i>tv_nsec</i> to wait for condition signal has passed.
CEE5J9	Severity	3
	Msg_No	5737
	Message	Unexpected return code from condition timedwait, BPX1CTW or BPX1STE, callable service.

Condition		
CEE5L3	Severity	3
	Msg_No	5795
	Message	The callable service, BPX1SMC, failed during shared condition variable processing. The system return code was <i>return_code</i> . The reason code was <i>return_code X'00</i> .

Note: The Language Environment date/time services can be used to obtain the needed number of seconds for the *abstime*. The user needs to obtain the number of seconds from 00:00:00 14 Oct 1585 until 00:00:00 1 Jan 1970. When calculated, that time can be saved and used to determine the absolute time.

CEEOPCW

C library interface: *pthread_cond_wait()*

CEEOPCW blocks the calling thread until another thread calls the condition signal or broadcast service specifying the same condition object *cond*. CEEOPCW releases the associated mutex object *mutex* before blocking the thread. Before returning to the caller, CEEOPCW reacquires (locks) the mutex again.

Syntax

void CEEOPCW (*cond*, *mutex*, [*fc*])

```
CEE_TOKEN    *cond;
CEE_MUTEX    *mutex;
FEED_BACK    *fc;
```

CEEOPCW

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0228(,R15)
BALR R14,R15
```

cond (input)

The condition object to wait on.

mutex (input)

The associated locked mutex.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in waiting for the condition variable.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.

Condition		
CEE5I5	Severity	3
	Msg_No	5701
	Message	The mutex specified by <i>mutex</i> is not initialized.
CEE5I8	Severity	4
	Msg_No	5704
	Message	Address exception while referencing storage allocated by condition initialization for pthread_mutex_t object specified by <i>mutex</i> .
CEE5IA	Severity	3
	Msg_No	5706
	Message	The mutex specified by <i>mutex</i> is not owned by the calling thread.
CEE5IO	Severity	4
	Msg_No	5720
	Message	Mutex specified by <i>mutex</i> not reacquired after condition wait terminated because thread was forced to terminate.
CEE5IU	Severity	3
	Msg_No	5726
	Message	The condition variable specified by <i>cond</i> is not initialized.
CEE5J0	Severity	4
	Msg_No	5728
	Message	Address exception while referencing storage allocated by condition initialization for cond_t object specified by <i>cond</i> .
CEE5J1	Severity	3
	Msg_No	5729
	Message	The mutex specified by <i>mutex</i> is a recursive mutex.
CEE5J2	Severity	3
	Msg_No	5730
	Message	A mutex other than the mutex specified by <i>mutex</i> is already associated with the condition specified by <i>cond</i> .
CEE5J3	Severity	3
	Msg_No	5731
	Message	Unexpected return code from z/OS UNIX condition setup, BPX1CSE, callable service.
CEE5J4	Severity	3
	Msg_No	5732
	Message	Unexpected return code from z/OS UNIX condition wait, BPX1CWA, callable service.
CEE5L3	Severity	3
	Msg_No	5795
	Message	The callable service, BPX1SMC, failed during shared condition variable processing. The system return code was <i>return_code</i> . The reason code was <i>return_code</i> X'00'.

CEEOPDD

C library interface: *pthread_condattr_destroy()*

CEEOPDD destroys a condition attributes object.

Syntax

void CEEOPDD (*condattr*, [*fc*])

```
CEE_TOKEN    *condattr;
FEED_BACK    *fc;
```

CEEOPDD

Call this CWI interface as follows:

```
L    R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L    R15,0248(,R15)
BALR R14,R15
```

condattr (input/output)

The location that contains the token of an initialized condition attributes object.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in deleting the condition attributes object.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5J0	Severity	4
	Msg_No	5728
	Message	Address exception while referencing storage allocated by condition attribute initialization for condattr_t object specified by <i>condattr</i> .
CEE5JC	Severity	3
	Msg_No	5740
	Message	Address exception accessing condattr_t object specified by <i>condattr</i> .
CEE5JD	Severity	4
	Msg_No	5741
	Message	Unable to free storage allocated by condition attribute initialization for condattr_t object specified by <i>condattr</i> .
CEE5JF	Severity	3
	Msg_No	5743
	Message	The condattr_t object specified by <i>condattr</i> is not valid (not initialized).

Condition		
CEE5JI	Severity	3
	Msg_No	5746
	Message	Condattr_t object specified by <i>condattr</i> is damaged.

CEEOPDG

C library interface: *pthread_condattr_getkind_np()*

CEEOPDG returns an integer value indicating attributes specified by a condition variable attributes object. Valid attributes are:

DEFAULT

N/A

NODEBUG

Indicates that condition variable services should not report state changes for a condition variable which was initialized with this attribute.

PRIVATE

Indicates that condition variable resides in local memory. This is the default.

SHARED

Indicates that condition variable resides in shared memory.

Syntax

void CEEOPDG (*condattr*, *kind*, [*fc*])

```
CEE_TOKEN *condattr;
INT4      *kind;
FEED_BACK *fc;
```

CEEOPDG

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L   R15,0252(,R15)
BALR R14,R15
```

condattr (input/output)

The location that contains the token of an initialized condition variable attributes object.

kind (output)

The location of a value specifying a condition variable attribute. Values returned by CEEOPDG are:

```
0      DEFAULT
2      NODEBUG+PRIVATE
8      DEBUG+SHARED
10     NODEBUG+SHARED
```

In addition to the other attributes, if the `_OPEN_SYS_MUTEX_EXT` feature switch is set, the `PRIVATE` and `SHARED` attributes set by *pthread_condattr_setpshared* is returned.

***fc* (output/optional)**

The feedback code returned by the service. It indicates the degree of success in interrogating the condition variable attributes object.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	This service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5J0	Severity	4
	Msg_No	5728
	Message	Address exception while referencing storage allocated by condition variable attribute initialization for <i>condattr_t</i> object specified by <i>condattr</i> .
CEE5JF	Severity	3
	Msg_No	5743
	Message	The <i>condattr_t</i> object specified by <i>condattr</i> is not valid (not initialized).
CEE5JM	Severity	3
	Msg_No	5750
	Message	Address exception attempting to store attribute value at address specified for <i>kind</i> .
CEE5JN	Severity	4
	Msg_No	5751
	Message	Valid attribute value for <i>condattr_t</i> object specified by <i>condattr</i> not found in attribute object storage.

CEEOPDI

C library interface: *pthread_condattr_init()*

CEEOPDI initializes a condition attributes object. There are no implementation defined attributes for condition variables.

Syntax

void CEEOPDI (*condattr*, [*fc*])

```
CEE_TOKEN    *condattr;
FEED_BACK    *fc;
```

CEEOPDI

Call this CWI interface as follows:

```
L    R15,CEECAALE0V-CEECAA(,R12)    CAA address is in R12
L    R15,0244(,R15)
BALR R14,R15
```

condattr (input)

The location of the caller-provided condition attributes object.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in initializing the new condition attributes object.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5J0	Severity	4
	Msg_No	5728
	Message	Address exception while referencing storage allocated by condition attribute initialization for condattr_t object specified by <i>condattr</i> .
CEE5JA	Severity	3
	Msg_No	5738
	Message	The condattr_t object specified by <i>condattr</i> is already initialized.
CEE5JB	Severity	3
	Msg_No	5739
	Message	Insufficient storage to initialize the condattr_t object specified by <i>condattr</i> .
CEE5JC	Severity	3
	Msg_No	5740
	Message	Address exception accessing condattr_t object specified by <i>condattr</i> .
CEE5JD	Severity	4
	Msg_No	5741
	Message	Unable to free storage allocated by condition attribute initialization for condattr_t object specified by <i>condattr</i> .

CEEOPDS

C library interface: *pthread_condattr_setkind_np()*

CEEOPDS changes attributes specified by a condition variable attributes object. Valid attributes are:

DEFAULT

N/A

NODEBUG

Indicates that condition variable services should not report state changes for a condition variable which was initialized with this attribute.

Syntax

void CEEOPDS (*condattr*, *kind*, [*fc*])

```
CEE_TOKEN  *condattr;
INT4      *kind;
FEED_BACK *fc;
```

CEEOPDS

Call this CWI interface as follows:

```
L   R15,CEECAALEOV-CEECAA(,R12)    CAA address is in R12
L   R15,0256(,R15)
BALR R14,R15
```

condattr (input/output)

The location that contains the token of an initialized condition variable attributes object.

kind (input/output)

The location of a value specifying a condition variable attribute. Valid values are:

```
0      DEFAULT
2      NODEBUG
X'8000' PRIVATE
X'8008' SHARED
```

CEEOPDS can be used to set the normal condition variable attributes, or to set the PRIVATE / SHARE attributes for `pthread_condattr_setpshared`. These two types of request cannot be combined in a single call to CEEOPDS. If the `_OPEN_SYS_MUTEX_EXT` feature switch is set, you can specify the PRIVATE / SHARED attributes for `pthread_condattr_setpshared`.

fc (output/optional)

The feedback code returned by the service. It indicates the degree of success in setting the condition variable attributes object.

The following message identifiers and associated severities can be returned by the service in the feedback code *fc*.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4S9	Severity	3
	Msg_No	5001
	Message	The service is unavailable unless POSIX(ON) runtime option specified and z/OS UNIX System Services are started.
CEE5J0	Severity	4
	Msg_No	5728
	Message	Address exception while referencing storage allocated by condition variable attribute initialization for <code>condattr_t</code> object specified by <i>condattr</i> .

Condition		
CEE5JF	Severity	3
	Msg_No	5743
	Message	The condattr_t object specified by <i>condattr</i> is not valid (not initialized).
CEE5JL	Severity	3
	Msg_No	5749
	Message	The attribute value specified by <i>kind</i> is not valid.

Process control functions support

In addition to those functions that are explicitly provided for threading applications, there are a number of POSIX 1003.1 functions that have either expanded definitions or new definitions. This section contains the CWIs that support those functions.

CEEEOEXEC

CEEEOEXEC replaces the prior POSIX process image with a new process image for the executable file being run, supporting the POSIX 1003.1 `exec()` function.

Syntax

void CEEEOEXEC (*path_name_length*, *path_name*, *argument_count*, *argument_length_list*, *argument_list*, *environment_count*, *environment_data_length*, *environment_data_list*, [*fc*])

```
INT4    *path_name_length;
VSTRING *path_name;
INT4    *argument_count;
POINTER *argument_length_list;
POINTER *argument_list;
INT4    *environment_count;
POINTER *environment_data_length;
POINTER *environment_data_list;
FEEDBACK *fc;
```

CEEEOEXEC

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L    R15,3288(,R15)
BALR R14,R15
```

path_name_length (input)

Specifies the name of a full word containing the length of the path name of the file (program) to be run. The length can be up to 1023 bytes long.

path_name (input)

Specifies the name of a field of length *file_name_length* containing the fully qualified path name of the file (program) to be run. Each component of the path name can be up to 255 characters long. The complete path name can be up to 1023 characters long and does not require a terminating character.

argument_count (input)

Specifies the name of a full word containing a count of the number of pointers in the *argument_length_list* and the *argument_list* lists. If the program needs no arguments, specify zero.

argument_length_list (input)

Specifies the address of the first in a list of pointers. Each pointer in the list is the address of a full word giving the length of one of the arguments to be passed to the specified program. If the program needs no arguments, specify zero.

argument_list (input)

Specifies the address of a list of pointers. Each pointer in the list is the address of a character string which is an argument to be passed to the specified program. Each argument is of the length specified by the corresponding element in the *argument_length_list*. If the program needs no arguments, specify zero.

environment_count (input)

Specifies the name of a full word containing a count of the number of pointers in the *environment_data_length* and the *environment_data_list* lists. If the program needs no arguments, specify zero.

environment_data_length (input)

Specifies the address of the first in a list of pointers. Each pointer in the list is the address of a full word giving the length of one of the environment variables to be passed to the specified program. If the program does not use environment variables, specify zero.

environment_data_list (input)

Specifies the address of a list of pointers. Each pointer in the list is the address of a character string consisting of one of the environment variables to be passed to the specified program. Each environment list argument is of the length specified by the corresponding element in the *environment_length_list*. If the program does not use environment variables, specify zero.

fc (output/optional)

The parameter in which the CWI service feedback code is placed. The following conditions can result from this CWI service.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4SA	Severity	3
	Msg_No	5002
	Message	POSIX function not available. z/OS UNIX System Services were not started.
CEE512	Severity	3
	Msg_No	5154
	Message	The requested exec() failed because it was invoked from a multithread environment.
CEE519	Severity	3
	Msg_No	5161
	Message	The z/OS UNIX callable service BPX1EXC for the exec() function was unsuccessful. The system return code was [return_code]; the reason code was [reason_code].

Qualifying Data:

No.	Name	Input/ Output	Type	Value
1	<i>parm_count</i>	Input	INT4	3
2	<i>return_code</i>	Input	INT4	Return code from kernel, BPX1EXC function <i>nn</i> Codes defined by ANSI C, POSIX, and z/OS UNIX
3	<i>reason_code</i>	Input	INT4	Reason code from kernel, BPX1EXC function <i>nn</i> Codes defined by ANSI C, POSIX, and z/OS UNIX

Usage notes:

1. Replaces the prior process image with a new process image for the executable file being run.
2. Target of `exec()` must be a C program and live in the POSIX file system and POSIX(ON) runtime option must be present.
3. Establishes an exit routine with the kernel, to gain control after the kernel has validated the `exec()` and prior to replacing the process image. In this exit routine, Language Environment drives member languages for enclave termination.
4. The values of the *return_code* and *reason_code* are as defined in the *z/OS UNIX System Services Programming: Assembler Callable Services Reference* and the *OpenEdition for VM/ESA: Callable Services Reference*.
5. This function is accessible independent of the POSIX runtime option.

CEEEOFORK

CEEEOFORK creates a new POSIX process, called a child process, supporting the POSIX 1003.1 `fork()` function. CEEEOFORK supports the XPG4 standard `vfork()` function.

Note: The CEEEOFORK CWI does not support the use of fork handlers. Support for the registration and execution of fork handlers only exists in the C/C++ Runtime Library.

Syntax

void CEEEOFORK (*function_code*, *pid*, [*fc*])

```
INT4      *function_code;
INT4      *pid;
FEED_BACK *fc;
```

CEEEOFORK

Call this CWI interface as follows:

```
L   R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L   R15,3284(,R15)
BALR R14,R15
```

***function_code* (input)**

A full word binary integer that specifies whether the function is `fork()` or `vfork()`. It must contain one of the following values:

- 0** Language Environment has been requested to perform `fork()`, if the member language tolerate the `fork()`. If so, perform `fork()`.

- 1 Language Environment has been requested to perform `vfork()`, if the member language tolerate the `vfork()`. If so, perform `vfork()`.

pid (output)

A full word binary integer to contain the process identifier of the newly-created child process. When `fork()` or `vfork()` returns to the calling (parent) process, it returns the process identifier of the newly create child. Because the child is a duplicate, it contains the same call to `fork()` or `vfork()` that was in the parent. Execution of the child begins with this `fork()` or `vfork()` call returning a value of zero, the child then proceeds with normal execution. If the `pid` is returned as minus 1, then no child process was created, for the reason specified in feedback token CEE510.

fc (output/optional)

The parameter in which the CWI service feedback code is placed. The following conditions can result from this CWI service.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE4SA	Severity	3
	Msg_No	5002
	Message	POSIX function not available. z/OS UNIX system services were not started.
CEE50V	Severity	3
	Msg_No	5151
	Message	Language Environment member identifier number [<i>member_id</i>] cannot tolerate the POSIX <code>fork()</code> or <code>vfork()</code> function.
CEE510	Severity	3
	Msg_No	5152
	Message	The z/OS UNIX callable service for the <code>fork()</code> function was unsuccessful. The system return code was [<i>return_code</i>]; the reason code was [<i>reason_code</i>]. Note: The system return code and reason code are documented in the z/OS UNIX manual.
CEE512	Severity	3
	Msg_No	5154
	Message	The requested <code>fork()</code> or <code>vfork()</code> service failed because it was invoked from a multithread environment.

Qualifying data (when CEE510):

No.	Name	Input/ Output	Type	Value
1	<i>parm_count</i>	Input	INT4	3
2	<i>return_code</i>	Input	INT4	Return code from kernel, BPX1FRK function <i>nn</i> Codes defined by ANSI C, POSIX, and z/OS UNIX

No.	Name	Input/ Output	Type	Value
3	<i>reason_code</i>	Input	INT4	Reason code from kernel, BPX1FRK function <i>nn</i> Codes defined by ANSI C, POSIX, and z/OS UNIX

Usage notes:

1. The new process (called the *child process*) is a duplicate of the process that calls `fork()` (called the *parent process*).
2. Member languages are notified (event code 24 and function code 1) that a `fork()` has been requested and can the member tolerate a `fork()`.
The event handler CEEEVnnn sets the return code in R15 to the following values:
 - 0 Member language can tolerate `fork()`
 - 4 Member language cannot tolerate `fork()`
 - 16 Event handler encountered an unrecoverable error.
3. Member languages are notified (event code 24 and function code 2) to perform any required cleanup in the child process.
4. The values of the `return_code` and `reason_code` (*nn*) are as defined in the *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
5. This function is accessible independent of the POSIX runtime option.

CEEOSPWN

CEEOSPWN creates a new POSIX process and immediately loads the process image from an executable file in the z/OS UNIX file system. The kernel callable service, BPX1SPN, is invoked to perform most of this function. The child process is created in a new address space unless the environment variable `_BPX_SHAREAS` has a value of 'YES', in which case the child process shares the address space with its parent. If the `_BPX_SHAREAS` value is 'NO' the process runs in a separate address space from the caller's address space.

Syntax

void CEEOSPWN (*path_name_length*, *path_name*, *argument_count*, *argument_length_list*, *argument_list*, *environment_count*, *environment_data_length*, *environment_data_list*, *filedesc_count*, *filedesc_list*, *inherit_area_len*, *inherit_area*, *process_id*, [*fc*])

```

INT4      *path_name_length;
VSTRING   *path_name;
INT4      *argument_count;
POINTER   *argument_length_list;
POINTER   *argument_list;
INT4      *environment_count;
POINTER   *environment_data_length;
POINTER   *environment_data_list;
INT4      *filedesc_count;
VSTRING   *filedesc_list;
INT4      *inherit_area_len;
VSTRING   *inherit_area;
INT4      *process_id;
FEED_BACK *fc;

```

CEEOSPWN

Call this CWI interface as follows:

```
L      R15,CEECAACELV-CEECAA(,R12)      CAA address is in R12
L      R15,0100(,R15)
BALR   R14,R15
```

***path_name_length* (input)**

Specifies the name of a full word containing the length of the pathname of the file (program) to be run. The length can be up to 1023 bytes long.

***path_name* (input)**

Specifies the name of a field of length *path_name_length* containing the fully qualified pathname of the file (program) to be run. Each component of the pathname can be up to 255 characters long. The complete pathname can be up to 1023 characters long, and does not require a terminating character.

***argument_count* (input)**

Specifies the name of a full word containing a count of the number of pointers in the *argument_length_list* and the *argument_list* lists. If the program need no arguments, specify zero.

***argument_length_list* (input)**

Specifies the address of the first in a list of pointers. Each pointer in the list is the address of a full word giving the length of one of the arguments to be passed to the specified program. If the program needs no arguments, specify zero.

***argument_list* (input)**

Specifies the address of a list of pointers. Each pointer in the list is the address of a character string which is an argument to be passed to the specified program. Each argument is of the length specified by the corresponding element in the *argument_length_list*. If the program needs no arguments, specify zero.

***environment_count* (input)**

Specifies the name of a full word containing a count of the number of pointers in the *environment_data_length* and the *environment_data_list* lists. If the program need no arguments, specify zero.

***environment_data_length* (input)**

Specifies the address of the first in a list of pointers. Each pointer in the list is the address of a full word giving the length of one of the environment variables to be passed to the specified program. If the program does not use environment variables, specify zero.

***environment_data_list* (input)**

Specifies the address of a list of pointers. Each pointer in the list is the address of a character string consisting of one of the environment variables to be passed to the specified program. Each environment list argument is of the length specified by the corresponding element in the *environment_length_list*. If the program does not use environment variables, specify zero.

***filedesc_count* (input)**

Specifies the name of a full word containing a count of the number of file descriptors the child process shall inherit. It may take a value from -1 to OPEN_MAX. If the value is -1, all file descriptors from the parent are inherited without remapping by the child and the *filedesc_list* is ignored. If the value is 0, no file descriptors are inherited by the child and the *filedesc_list* is ignored.

***filedesc_list* (input)**

Specifies the name of a list of full word file descriptor remap values. Except for

those file descriptors designated by **SPAWN_FDCLOSED**, each of the child's file descriptors in the range zero to *filedesc_count*-1 shall inherit file descriptor remap values *filedesc_list*(1) to *filedesc_list*(*filedesc_count*). (The constant **SPAWN_FDCLOSED** is defined in the z/OS UNIX macro, BPXYCONS.)

***inherit_area_len* (input)**

Specifies the name of a full word that contains the length of the inheritance structure specified in *inherit_area*. If this argument contains a value of zero, the *inherit_area* argument is ignored.

***inherit_area* (input)**

Specifies the name of a data area that contains the inheritance structure for the child process. (The inheritance structure is defined in the z/OS UNIX macro, BPXYINHE.)

***process_id* (output)**

Specifies the process id(PID) of the child process.

***fc* (output/optional)**

Specifies the optional feedback token where the CWI feedback code will be placed. If this argument is omitted and the CWI will return a feedback code other than **CEE000**, the CWI will 'raise' this feedback code as an error condition.

The following conditions can result from this CWI service:

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE513	Severity	3
	Msg_No	5155
	Message	The z/OS UNIX callable service, BPX1SPN, for the spawn() and spawnp() functions was unsuccessful. The system return code was [<i>return_code</i>], the reason code was [<i>reason_code</i>].

Qualifying data (when CEE513):

No.	Name	Input/ Output	Type	Value
1	<i>parm_count</i>	input	INT4	3
2	<i>return_code</i>	input	INT4	Return code from kernel, BPX1SPN function <i>nn</i> codes defined by ANSI C, POSIX, and z/OS UNIX
3	<i>reason_code</i>	input	INT4	Reason code from kernel, BPX1SPN function <i>nn</i> codes defined by ANSI C, POSIX, and z/OS UNIX

Usage notes:

1. The new process (called the *child process*) inherits the following attribute from the process that calls **spawn/spawnp**. For further details, see POSIX .4b draft 8.
2. Member languages are notified that a **spawn/spawnp** has been requested and given the opportunity to indicate whether they can tolerate a **spawn/spawnp**.

CEEOSPWN CWI

The event handler CEEVnnn sets the return code in register 15 to 0 if member language can tolerate being **spawn/spawnp**, -4 if member language cannot tolerate being **spawn/spawnp**, and 16 if the event handler encountered an unrecoverable error.

3. The values for the return code and reason code are defined in the *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
4. This function is accessible independent of the POSIX runtime option.

Miscellaneous utilities

The CWIs in this section provide varied functions.

CEEEXIT

CEEEXIT CWI service terminates the calling process.

Syntax

void CEEEXIT (*status*)

INT4 *status;

CEEEXIT

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12
L    R15,3292(,R15)
BALR R14,R15
```

status (output)

A full word binary integer containing the status field conforming to the allowable exit status values of:

X'000000'xx

Normal Termination. The child process ended due to a normal termination of the process with status code indicated as *xx*, where *xx* can be any value.

Usage notes:

1. If the application was invoked due to an `exec()` or `fork()`, the kernel does not return to the caller. If it cannot complete its processing successfully, an EC6 abend is issued.
2. If an incorrect exit status is specified, the kernel issues an EC6 abend and a reason code X'0B19C00F'.
3. This function is accessible independent of the POSIX runtime option.

CEEEXEXE

When a POSIX process is to exec a new file (terminate a current process), Language Environment must be able to run clean up functions to terminate its environment properly. CEEEXEXE is the exec exit routine given control to perform normal enclave termination.

Syntax

void CEEEXEXE (*plist*)

POINTER *plist;

***plist* (input)**

Specifies the name of a full word containing the address of the user exit parameter list. This value is in R1 when the user exit is invoked. If the user exit does not require parameters, specify 0.

Support for POSIX functions `getenv()`, `setenv()`, and `clearenv()`

Environment variables are contained in an array of null-terminated strings of the form *name=value* in the POSIX process. These environment variables are manipulated by the functions listed below, or by the external variable **extern char **environ**. The names cannot contain the equal sign character (=).

Access to environment variables using ****environ** cannot be guaranteed in a POSIX application that uses multiple threads. Access by these functions serializes access to the environment variables and guarantees consistency of the environment variable array in a threaded environment.

The functions that manipulate environment variables are listed as follows:

Syntax

```
#include <sys/types.h>
char *getenv(const char *name);
int  setenv(const char *name, char *newvalue, int overwrite);
int  clearenv(void);
```

getenv()

Searches the environment variable list for a string of the form *name=value* and returns a pointer to *value* if such a string is present, NULL otherwise. For details, see the POSIX 1003.1 definition.

setenv()

Searches the environment variable list for a string of the form *name=value*. If found and the *overwrite* argument is nonzero, the *newvalue* is substituted for the current value. If the string is not found, add it to the environment variable list. For details, see the POSIX 1003.1 definition.

clearenv()

Clears all of the environment variables in the POSIX process. For details, see the POSIX 1003.1 definition.

Errors

The errors that can occur, beyond those defined by POSIX, are:

EMVSBADCHAR

Bad input character

ENOMEM

Not enough memory available

CEEENV

Syntax

```
void CEEENV (function_code, name_length, name, value_length, value, overwrite, [fc])
INT4      *function_code;
INT4      *name_length;
VSTRING   *name;
```

CEEENV CWI

```
INT4    *value_length;  
POINTER *value;  
INT4    *overwrite;  
FEEDBACK *fc;
```

CEEENV

Call this CWI interface as follows:

```
L    R15,CEECAACELV-CEECAA(,R12)    CAA address is in R12  
L    R15,3416(,R15)  
BALR R14,R15
```

function_code (input)

A full word binary integer containing the function code of the one of the following values:

- 1 Perform `getenv()`. Searches the environment list for environment variable specified by *name* and if found returns a pointer to *value*.
- 2 Perform `setenv()`. Adds, changes, or deletes an environment variable in the environment list.
- 3 Perform `clearenv()`. Clears all environment variables in the environment list.
- 4 Perform internal `getenv()`. Functionally the same as (1) except that it is used internally within the library. The environment variable is returned in a buffer that is independent of the buffer used by external callers of `getenv()`.

name_length (input/output)

A full word binary integer containing the length of the name for the environment variable. If request is `clearenv()`, this argument is ignored.

name (input/output)

Specifies the address of the name of an environment variable. If request is `clearenv()`, this argument is ignored.

value_length (input/output)

A full word binary integer containing the length of the value for the environment variable. This argument is output from `getenv()`, and input to `setenv()`. If request is `clearenv()`, this argument is ignored. A length of zero indicates a delete request.

value (input/output)

Specifies the address of a field which contains the address of a null terminated string containing the value of the environment variable, or zero if this is a delete request. This argument is output from `getenv()`, and input to `setenv()`. If request is `clearenv()`, this argument is ignored.

overwrite (input)

A full word binary integer. If nonzero, `setenv()` changes the existing value of existing *name* to *value* or deletes the existing environment variable and adds a new environment variable. If request is `getenv()` or `clearenv()`, this argument is ignored.

fc (output/optional)

The parameter in which the CWI service feedback code is placed. The following conditions can result from this CWI service.

Condition		
CEE000	Severity	0
	Msg_No	0000
	Message	The service completed successfully.
CEE51O	Severity	3
	Msg_No	5176
	Message	Not enough memory available.
CEE51P	Severity	3
	Msg_No	5177
	Message	Bad input character detected for name or value.
CEE51Q	Severity	3
	Msg_No	5178
	Message	Bad address detected for the envvar anchor or environment variable array.
CEE51R	Severity	3
	Msg_No	5179
	Message	A parameter to the environment variable processing routine contained an invalid value.
CEE51S	Severity	0
	Msg_No	5180
	Message	The specified environment variable name already exists.

Usage notes:

1. The environment variables are always available, independent of the POSIX(ON|OFF) setting.
2. This function is also available from the CEEENV callable service. For more information, see *z/OS Language Environment Programming Reference*.
3. The environment array is searched sequentially, and the first occurrence of *name* is used.
4. Access to the environment variable array is as follows:
ceedbenvi points to a field (either *ceedenvvar* or C's ***environ* in writable static) which points to a null terminated array of null terminated character strings of the format *name=value*.
5. Because an application can manipulate the environment using the *environ* pointer, Language Environment cannot guarantee a single instance of any particular environment variable.
6. This function can manipulate the value of the pointer *environ*, copies of that pointer need not be valid after call to this function.
7. For a *getenv()* request, the storage returned for the value character string is supplied by Language Environment. There is one buffer per thread. Thus, it is the user's responsibility to use or save the value prior to the next call to *getenv()* on that thread.
8. Environment variable names that begin with “_BPXK_” are passed to the kernel through the callable service, BPX1ENV. Language Environment members and their users should not define environment variables that begin with the characters “_BPXK_”; otherwise, there may be conflicts with z/OS UNIX-defined variable names that begin with those characters.

Chapter 17. COBOL-specific vendor interfaces

This section describes the COBOL-specific interfaces, ILBOLLDX, IGZCXCC, IGZXAPI, and IGZCSF.

ILBOLLDX — OS/VS COBOL library load/delete exit

Purpose

When you link-edit an OS/VS COBOL NORES program with SCEELKED and include certain CSECTs, you can make the OS/VS COBOL NORES program act like a RES program. For more information about link-editing OS/VS COBOL applications with SCEELKED and having them act like RES programs, see *Enterprise COBOL for z/OS, V4R2, Compiler and Runtime Migration Guide, GC23-8527* or *Enterprise COBOL for z/OS, V3R4, Compiler and Runtime Migration Guide, GC27-1409*.

Syntax

```
void ILBOLLDX
```

R1 (input)

R1 contains a function code in byte 2 and a library routine identifier in byte 3 as shown below.

```
byte 0 1 2 3
```

```
R1= |xx|xx|ff|ll|
```

Where:

Reserved (xx)	Function (ff)	Library Routine Identifier (ll)
----- Always hex 00	----- 01 - Load 02 - Delete	----- 01 - ILBOCOM0 02 - ILBOSR

R0 (output)

R0 must be set to the address of the OS/VS COBOL library load module when a load function is done.

Usage notes

- The OS/VS COBOL library load/delete exit will get control when normally a load or delete of ILBOCOM0 or ILBOSR would occur. The OS/VS COBOL library load/delete exit can then provide a unique copy of the ILBOCOM and ILBOSRV modules per task (TCB).
- To enable the OS/VS COBOL library load/delete exit, a customer written CSECT called ILBOLLDX must be link edited with the following ILBO load modules:
 - ILBONTR (which is in the SCEERUN data set)
 - ILBOSRV (which is in the SCEERUN data set and the SCEELKED data set)
 - ILBOSTT (which is in the SCEERUN data set and the SCEELKED data set)

ILBOLLDX

Language Environment does not provide any usermod jobs to perform the link editing of the OS/VS COBOL library load/delete exit into Language Environment. It is the responsibility of the customer who is using the OS/VS COBOL library load/delete exit to do this. Additionally, all OS/VS NORES programs must be relink-edited using the modified SCEELKED data set.

When ILBOLLDX is link edited with the ILBO routines, the link-edit attributes must not be altered and all of the ALIASes associated with each load module must be preserved. For link-edit information to link-edit ILBOLLDX with the ILBO routines, see Figure 112, Figure 113, and Figure 114.

```
Required link edit parameters: NCAL,RENT,REFR

Link edit control cards:

INCLUDE SCEERUN(ILBONTR)
INCLUDE YOURLIB(ILBOLLDX)
ORDER ILBONTR
ENTRY ILBONTR
ALIAS ILBONTR0
NAME ILBONTR(R)
```

Figure 112. Link edit information to enable ILBOLLDX in ILBONTR

```
Required link edit parameters: LET,NCAL,REUS

Link edit control cards:

INCLUDE SCEERUN(ILBOSRV)
INCLUDE YOURLIB(ILBOLLDX)
ORDER ILBOSRV
ORDER IGZEOB2
ALIAS ILBOSR,ILBOSRV0,ILBOSRV1,ILBOSR3,ILBOSR5,ILBOST
ALIAS ILBOSTP0,ILBOSTP1
ENTRY ILBOSRV
NAME ILBOSRV(R)
```

Figure 113. Link edit information to enable ILBOLLDX in ILBOSRV

```
Required link edit parameters: LET,NCAL,RENT,REFR

Link edit control cards:

INCLUDE AIGZMOD1(ILBOSTT)
INCLUDE YOURLIB(ILBOLLDX)
ORDER ILBOSTT
ORDER IGZEOB2
ALIAS ILBOSTT0
ALIAS ILBOSTT2
ENTRY ILBOSTT
NAME ILBOSTT(R)
```

Figure 114. Link edit information to enable ILBOLLDX in ILBOSTT

- On entry, R14 contains the return address and R15 the entry point address.
- No save area will be passed to ILBOLLDX to save the caller's registers.
- Registers 2-13 must be preserved.
- ILBOLLDX must be REENTRANT (because it will be link-edited with reentrant ILBO routines).
- ILBOLLDX must be AMODE 24, RMODE 24.
- ILBOLLDX will be entered in AMODE 24 and must return in AMODE 24.

- If ILBOLLDX detects an error condition, it must ABEND, as the caller does not expect return except when the operation is successful.
- ILBOLLDX must support the concept of “use counts”. An instance of a routine (the copy associated with a given task, for example) should not be deleted unless the count of delete requests for that instance equals the count of load requests. For example, if the following sequence of events is received for an instance of given library routine, the instance must not be deleted until the last delete in the sequence: load, load, delete, load, delete, delete.
- OS/VS COBOL RES programs cannot be run with the ILBOLLDX support. Unpredictable results will occur if done.
- ILBOLLDX is not called when the OS/VS COBOL NORES program is running as NORES.
- Once ILBOLLDX is link-edited with the ILBO routines, ILBOLLDX must support all environments in which it is used.

IGZCXCC — COBOL call/cancel routine

Purpose

The COBOL CALL/CANCEL interface, IGZCXCC, can be called from an assembler program to get the equivalent function of doing a COBOL dynamic call and a COBOL CANCEL. You can use the interface to call and cancel any programs that can be dynamically called or cancelled from a COBOL program. IGZCXCC provides two functions:

1. CALL with program name provided.
2. CANCEL with program name provided.

Note: Starting with OS/390 Version 2 Release 6, the IGZCXCC functions of CALL with entry point provided and CANCEL with entry point provided are no longer available because they can cause problems when used with other high level languages. If you have a need to perform the load and delete activity, use the Language Environment preinitialization support (CEEPIPI), and provide your own load and delete service routines. For information about CEEPIPI and its support of user-supplied service routines, see *z/OS Language Environment Programming Guide*.

Syntax

```
void IGZCXCC
```

R1 (input)

R1 contains the address of a structure that provides a function code and the necessary information for each function code. The structure can be in storage above the 16M line.

- The structure for CALL a program with program name is shown in Figure 115 on page 612
- The structure for CANCEL a program with program name is shown in Figure 116 on page 612

R15 (output)

Content varies, depending on whether this is a CALL or CANCEL request. For CALL, R15 is set to the R15 value returned from the program that is the target of the CALL. Figure 115 on page 612 shows the structure for CALL with name;

note that all fields shown are input only.

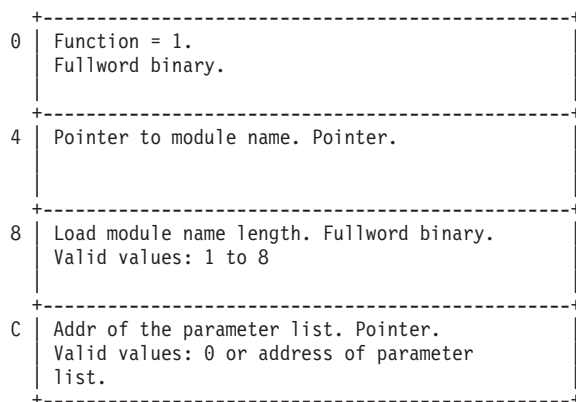


Figure 115. Structure for CALL with name

For CANCEL, R15 is set to 0. Figure 116 shows the structure; all fields are input only.

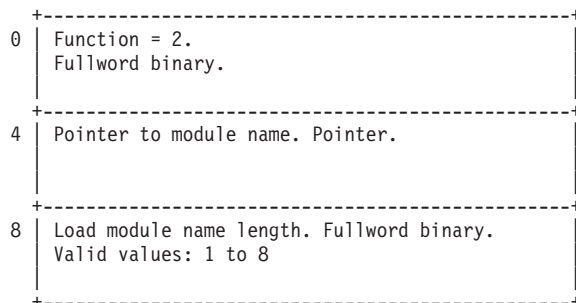


Figure 116. Structure for CANCEL with name

Usage notes

- IGZCXCC can only be used on z/OS non-CICS. IGZCXCC is supported in a multithread environment.
- IGZCXCC is AMODE(31) and RMODE(ANY). It expects to be entered in AMODE(31) via BASSM. It returns using a BSM. Assembler routines that call IGZCXCC must access it by issuing a LOAD SVC instruction. IGZCXCC must not be link edited with any other load modules, otherwise it will cause upward compatibility problems when moving from one release of Language Environment to another.
- IGZCXCC can not be invoked with a LINK. If it is invoked with a LINK, it will signal condition IGZ0099C with error code 4.
- IGZCXCC expects to be called from a Language Environment-enabled assembler program.
- R12 must point to the CAA on entry.
- R13 must point to a DSA with a valid NAB.
- The COBOL portion of Language Environment must be initialized prior to the call to IGZCXCC.
- If a parameter list is passed, the high order bit of the address of the last parameter must be on to indicate the end of the parameter list.

- There are no restrictions when mixing COBOL dynamic CALLs, COBOL CANCELs, IGZCXCC CALL with name provided (function code 1), and IGZCXCC CANCEL with name provided (function code 2).
- If a Language Environment-enabled assembler program that is going to call IGZCXCC is called by OS/VS COBOL programs or VS COBOL II programs, specify NAB=NO and MAIN=NO in the CEEENTRY macro.
- When a IGZCXCC CANCEL call is done for a program that either has not been dynamically called or has already been canceled, no action is taken, and no condition is signaled. This is the same behavior as a COBOL CANCEL.

IGZXAPI — COBOL file and runtime information query routine

Purpose

IGZXAPI is a loadable module, which can be called to query information about a running COBOL program. Below is sample code for calling the routine:

```
LOAD EP=IGZXAPI
..
LA R1,[parm area] see note below
BALR R14,R15
..
```

Note: [parm area] is an array of addresses. Only the first address entry is used, which points to XINFO, XINFO2, and everything below DSECT.

Syntax

```
void IGZXAPI
```

R1 (input)

Points to the parameter area.

(output)

The data structure (XINFO or XINFO2) is populated. Registers R2-R13 remain unchanged.

File information query

To query file information of a running COBOL program (language member identifier 4).

The DSECT XINFO is the parameter structure to communicate with the file information routine. The caller should take note of the following data fields:

XNREST

This is the number of files remaining to be returned from the query.

On the first call, the caller has to set this field to zero. This indicates the first call. The routine returns information of the first file; XNREST is set by the routine to the number of files remaining.

On subsequent calls, the caller should not modify any of the data fields returned from the previous call. The routine will return information of the next file.

When the last file has been returned, this field is set to zero by the routine.

XNFILES

This is an output field, and indicating the total number of files. This is returned by the routine in the first call, and remains unchanged in subsequent calls.

XDSA

This is the DSA address of the COBOL program being queried. This is set by the caller in the first call, and must not be modified in subsequent calls.

Only COBOL files in this program are returned.

XLEN

Length of the XINFO data structure. Caller should set this field to the length of XINFO, 124.

Caller should provide storage for XINFO.

XFNCODE

This is the function code. Set to 1 by caller for file information query.

All other data fields are output from the routine. The caller should not modify any of these fields, including the ones marked as reserved for future use.

Runtime information query

To query general runtime information. The DSECT XINFO2 is the parameter area to communicate with the runtime information routine. On input:

XDSA2

The DSA address of the COBOL program to be queried.

XLEN2

Length of XINFO2.

Caller should set this to 48.

Caller should provide memory for this data structure.

XFNCODE

This is the function code. Set to 2 for general runtime information query.

All other fields are output from the routine. The calling convention is the same as the file information routine.

Note: SD files are not physical data files. They do not actually exist until the sort is active. The current sort record, if present, is provided by way of the runtime information query (in XINFO2).

Layout of the parameter area

The layout of the parameter area is described below.

```

-----
*      1      2      3      4      5      6      7
*
* 34567890123456789012345678901234567890123456789012345678901
*
XINFO  DSECT
XFNCODE DS  F      Input: Function code, =1 for file information
XFILLER0 DS  A      reserved
XSIG   DS  F      X'COB00501'
XVER   DS  H      Version of this parameter block
XLEN   DS  H      Input: Length of this data structure, 120
XDSA   DS  A      Input: DSA of COBOL program to be queried
*
XNFILES DS  H      Total number of files
    
```



```

*
* BNREST is input field during the first call, set it to zero;
*       on return, contains the remaining number of files to go
*
*
XNREST  DS    H           Input in first call, set to zero
*
XDCB    DS    A           Address of DCB or ACB
XDDNAME DS    A           Address of DDNAME (8 characters)
XFNAME  DS    A           Address of file name (30 characters)
*
XCFLAG1 DS    X           Compile time information flags
XSOPTNL EQU  X'80'       SELECT OPTIONAL
XRCDSPN EQU  X'40'       Record format spanned
XBLKED  EQU  X'20'       Record format blocked
XLINAGE EQU  X'10'       Linage is specified
XLINFOOT EQU X'08'       Linage FOOTING is specified
XLINTOP EQU  X'04'       Linage TOP is specified
XLINBOT EQU  X'02'       Linage BOTTOM is specified
XBUFUSE EQU  X'01'       Buffer usage indicator
*
XCFLAG2 DS    X           Compile time information flags
XEXTFILE DS   X'80'       External file

FILLER2 DS    X           Reserved
FILLER3 DS    X           Reserved
*
XORG1   DS    X           File Type
XVSAM   EQU  X'01'       VSAM
XLSEQ   EQU  X'02'       Line Sequential
XQSAM   EQU  X'03'       QSAM
*
XORG2   DS    X           File Organization
XORGSEQ EQU  X'01'       Sequential
XORGIND EQU  X'02'       Indexed
XORGREL EQU  X'03'       Relative
*
XACCESS DS    X           File Access Mode
XACCSEQ EQU  X'01'       Sequential
XACCRAN EQU  X'02'       Random
XACCDYN EQU  X'03'       Dynamic
*
XRECFM  DS    X           Record Format
XRECFIX EQU  X'01'       Fixed
XRECVAR EQU  X'02'       Variable
XRECUND EQU  X'03'       Undefined
*
XRFLAG1 DS    X           Run time information flags 1
XOPOPT  EQU  X'80'       OPEN, missing optional file
XOPREV  EQU  X'40'       OPEN REVERSED (valid when XOPENED)
XOPNOREW EQU X'20'       OPEN, NO REWIND (valid when XOPENED)
XCLNOREW EQU X'10'       CLOSE, NO REWIND (valid when XCLOSED)
XCLLOCK EQU  X'08'       CLOSE, LOCK (valid when XCLOSED)
XCLREMOV EQU X'04'       CLOSE FOR REMOVAL (valid when XCLOSED)
XSOKACT EQU  X'02'       A successful action since OPEN
*
XRFLAG2 DS    X           Run time information flags 2
XPEND   EQU  X'20'       OPEN or CLOSE pending
XSEOF   EQU  X'10'       Previous READ hit end of file
XEOP    EQU  X'08'       End of page
XMOPTNL EQU  X'04'       OPTIONAL FILE MISSING
XADVAFT EQU  X'02'       WRITE AFTER ADVANCING x LINES
XADVBEF EQU  X'01'       WRITE BEFORE ADVANCING x LINES

FILLER4 DS    H           Reserved
*
XFMODE  DS    X           Current file mode
XOPENED EQU  X'01'       Opened

```

IGZXAPI

```

XCLOSED EQU X'02'      Closed
XNEVERO EQU X'03'      Never opened
*
XOMODE DS X           Information about OPEN (valid when XOPENED)
XOPIN EQU X'01'       OPEN INPUT
XOPOUT EQU X'02'      OPEN OUTPUT
XOPIO EQU X'03'       OPEN IO
XOPEXT EQU X'04'      OPEN EXTENDED
*
XCMODE DS X           Information about CLOSE (valid when XCLOSED)
XCLFILE EQU X'01'     CLOSE
XCLUNIT EQU X'02'     CLOSE REEL/UNIT
*
XLASTREQ DS X         Last operation on file
XLASTRD EQU X'01'     READ
XLASTWRT EQU X'02'    WRITE
XLASTRWT EQU X'03'    REWRITE
XLASTSTR EQU X'04'    START
XLASTDLT EQU X'05'    DELETE
XLASTOPN EQU X'06'    OPEN
XLASTCLO EQU X'07'    CLOSE
*
* Various LINAGE values
*
XLNLING DS F          Linage
XLNFOOT DS F          Linage footing
XLNTOP DS F           Linage top
XLNBOT DS F           Linage bottom
XLNCTR DS F           Linage counter
*
* File Status
*
XFSTAT DS X           File status, in 2 hex bytes
XVSMCOD DS X          VSAM feedback code
XVSMRET DS X          VSAM return code
XVSMFUNC DS X         VSAM function code
*
XADVVAL DS F          Write after/before advancing value
*
XRECLEN DS F          Record length; max length for variable rec
XBLKLEN DS F          Block size
XRECLAD DS A          Address of address of record
XBUFAD DS A           Address of buffer provided by DFSMS
*
XPNAME DS A           Address of program name
XPNAMELEN DS H        Program name length
FILLER5 DS H          Reserved
*
FILLER6 DS 4F         Reserved
*
*           1           2           3           4           5           6           7
*
* 345678901234567890123456789012345678901234567890123456789012345678901
*
-----
*
*           1           2           3           4           5           6           7
*
* 345678901234567890123456789012345678901234567890123456789012345678901
*
XINF02 DSECT
XFNCODE2 DS F         Input: Function code, =2 for RT information
FILLER10 DS A         reserved
XSIG2 DS F            X'C0B00501'
XVER2 DS H            Version of this parameter block
XLEN2 DS H            Input: Length of this data structure, 48
XDSA2 DS A            Input: DSA of COBOL program to be queried

```

```

*
FILLER11 DS   X      Reserved
FILLER12 DS   X      Reserved
FILLER13 DS   X      Reserved
*
XRFLAG4  DS   X      Run Time Flags
XINSORT  EQU  X'02'   Sort is active
XISMAIN  EQU  X'01'   Program is main
*
XSDREC   DS   A      Address of active sort record
XSDLEN   DS   F      Sort record len
FILLER14 DS   4F     Reserved
*
*
*           1           2           3           4           5           6           7
*
* 345678901234567890123456789012345678901234567890123456789012345678901
*
-----

```

Usage notes

- R13 must point to the caller's DSA.
- R14 is the return address.
- R15 is the entry point address of IGZXAPI.
- On return from the routine, R15 is set to zero if the call is successful. R15 is set to non-zero if the call is not successful and the requested information is not available.

IGZCXSF — COBOL extract side file routine

Purpose

The COBOL Extract Side File interface, IGZCXSF, can be called from a Language Environment-conforming program to extract the following information from a COBOL SYSDEBUG side file:

- Compilation information
- Procedure table
- Expanded program source

Syntax

```
call IGZCXSF(side-file-name, program-name, output-structure)
```

R1 (input)

The address of the parm list which contains the addresses of the following 3 parms:

side-file-name (input) – Character(1024) Varying

The SYSDEBUG side file name preceded by a half-word length. It contains the data set name, data set name with member name, or fully qualified HFS name.

program-name (input) – Character(160) Varying

The program name preceded by a half-word length. It contains the program name associated with the desired compile unit within the SYSDEBUG side file.

output-structure (output) – 7 fullwords

The structure to contain the output data. The format of the structure is as follows:

Table 65. Output-structure format

	Description
00	Address of compilation information. Fullword address.
04	Length of compilation information. Fullword binary.
08	Address of procedure table. Fullword address.
0C	Length of procedure table. Fullword binary.
10	Address of expanded program source. Fullword address.
14	Length of expanded program source. Fullword binary.
18	LRECL of expanded program source. Fullword binary.

R15 (output)

The return code. The possible values are listed below.

- 1 Unsupported environment
- 0 Successful
- 4 Data set/File not found
- 8 Allocate error
- 12 Deallocate error
- 16 Open error (for example, member not found)
- 20 Close error
- 24 Read error
- 28 Decompress error
- 32 Storage not available
- 36 Invalid function code
- 40 Invalid file attribute (for example, PDS but no member, member but not PDS)
- 44 Verification failed (for example, not a valid side file, program name not found)
- 48 Unexpected EOF
- 52 No TIOT ENQ

Usage notes

- IGZCXSF can only be used on z/OS non-CICS.
- IGZCXSF is AMODE(31) and RMODE(ANY). It expects to be entered in AMODE(31). It returns using a BSM. Routines that call IGZCXSF must access it by a dynamic load (for example, issuing a LOAD SVC, using CEELoad, COBOL dynamic call, PL/I fetch, C fetch). IGZCXSF must not be linked with any other load modules, otherwise it will cause upward compatibility problems when moving from one release of Language Environment to another.
- IGZCXSF expects to be called from a Language Environment-conforming program.
- R12 must point to the CAA on entry.

- R13 must point to a DSA with a valid NAB.
- The user is responsible for freeing each of the three storage areas (address of compilation information, address of procedure table, and address of expanded program source that are returned in the output structure) using Language Environment free heap storage services (for example, CALL CEEFRST(address,feedback-code) or CALL CEEVFRST(address,feedback-code)). If the storage is not freed by the user, it will be implicitly freed by Language Environment at enclave termination.

Compilation information

1. The address and length of the compilation information are returned in the output structure.
2. This maps to TIMEVRS through USER LEVEL INFO plus 1 word that contains the length of the procedure division code

```
+00 CL14  YYYYMMDDHHMMSS (compile date and time)
+0E CL6   VVRRMM (compiler version/release/modification level)
+14 H     CCSID value
+16 XL2   unused
+18 XL2   Info bytes 28-29
+1A XL2   Year Window value
+1C XL23  Info bytes 1-23
+33 XL1   COBOL signature level
+34 F #   DATA DIVISION statements
+38 F #   PROCEDURE DIVISION statements
+3C XL4   Info bytes 24-27
+40 XL4   User compiler level
+44 F     Length of the procedure division code
```

3. TIMEVRS through USER LEVEL INFO is documented in the Enterprise COBOL Programming Guide 2.6.4.4.1 Example: program initialization code and following sections. Here is an example of the TIMEVRS through USER LEVEL INFO:

000068	F2F0F0F9	DC	CL4'2009'	@TIMEVRS: YEAR OF COMPILATION
00006C	F0F9F3F0	DC	CL4'0930'	MONTH/DAY OF COMPILATION
000070	F1F0F4F8	DC	CL4'1048'	HOURS/MINUTES OF COMPILATION
000074	F1F6	DC	CL2'16'	SECONDS FOR COMPILATION DATE
000076	F0F4F0F2F0F0	DC	CL6'040200'	VERSION/RELEASE/MOD LEVEL OF PROD
00007C	0474	DC	X'0474'	UNSIGNED BINARY CODE PAGE CCSID VALUE
00007E	0000	DC	AL2'0'	AVAILABLE HALF-WORD
000080	0000	DC	X'0000'	INFO. BYTES 28-29
000082	076C	DC	X'076C'	SIGNED BINARY YEARWINDOW OPTION VALUE
000084	A0487C4C2000	DC	X'A0487C4C2000'	INFO. BYTES 1-6
00008A	000000080000	DC	X'000000080000'	INFO. BYTES 7-12
000090	000000000800	DC	X'000000000800'	INFO. BYTES 13-18
000096	0000000000	DC	X'0000000000'	INFO. BYTES 19-23
00009B	00	DC	X'00'	COBOL SIGNATURE LEVEL
00009C	00000001	DC	X'00000001'	# DATA DIVISION STATEMENTS
0000A0	00000003	DC	X'00000003'	# PROCEDURE DIVISION STATEMENTS
0000A4	000080	DC	X'000080'	INFO. BYTES 24-26
0000A7	00	DC	X'00'	INFO. BYTE 27
0000A8	40404040	DC	C' '	USER LEVEL INFO (LVLINFO)

Procedure table

1. The address and length of the procedure table are returned in the output structure.
2. Each entry is 6 bytes long.
3. The format of each entry is as follows:
 - a. The line number of the statement is in the first 20 bits.
 - b. The verb number (v) on the line is in the next 3 bits.

- c. The existence of a Paragraph Name or Section Name on the line (p) is in the next 1 bit.
- d. The displacement from the start of the CSECT for this statement is in the next 3 bytes.
- e. The following table displays the entry format:

Table 66. Entry format for the procedure table

Byte 1	Byte 2	Byte 3			Byte 4	Byte 5	Byte 6
Bits: 8	Bits: 8	Bits: 4	3	1	Bits: 8	Bits: 8	Bits: 8
line number of statement			v	p	displacement from CSECT start		

Expanded program source

1. The address and length of the expanded program source are returned in the output structure.
2. The record length (LRECL) is returned in the output structure.
3. The data contained in each record depends on the LRECL
 - a. For LRECL 78:
 - 1) Columns 1-6 contains the compiler generated source code line number.
 - 2) Columns 7-78 contains the COBOL sequence number and program source (from columns 1-72 of the source file).
 - b. For LRECL 86:
 - 1) Columns 1-6 contains the compiler generated source code line number.
 - 2) Columns 7-78 contains the COBOL sequence number and program source (from columns 1-72 of the source file).
 - 3) Columns 79-86 contains the COBOL suffix area (from columns 73-80 of the source file).

Chapter 18. PL/I-specific vendor interfaces

This section describes the PL/I-specific interface, IBMPXSF.

IBMPXSF — PL/I extract side file routine

Purpose

The PL/I extract side file interface, IBMPXSF, can be called from a Language Environment-conforming program to extract the following information from a PL/I SYSDEBUG side file:

- Compilation information
- Statement table
- Expanded program source

Syntax

Call IBMPXSF(side-file-name, program-name, output-structure)

R1 (input)

The address of the parm list which contains the addresses of the following 3 parms:

side-file-name (input) – character(1024) varying

The SYSDEBUG side file name preceded by a half-word length. It contains the data set name, data set name with member name, or fully qualified HFS name.

program-name (input) – character(160) varying

The program name preceded by a half-word length. It contains the program name associated with the desired compile unit within the SYSDEBUG side file.

output-structure (output) – 7 fullwords

The structure to contain the output data. The format of the structure is as follows:

Table 67. Output-structure format

	Description
00	Address of compilation information. Fullword address.
04	Length of compilation information. Fullword binary.
08	Address of statement table. Fullword address.
0C	Length of statement table. Fullword binary.
10	Address of expanded program source. Fullword address.
14	Length of expanded program source. Fullword binary.
18	Max LRECL of expanded program source. Fullword binary.

R15 (output)

The return code. The possible values are listed below.

IBMPXSF

-1	Unsupported environment
-2	Unsupported version/release of the compiler
0	Successful
4	Data set/File not found
8	Allocate error
12	Deallocate error
16	Open error (for example, member not found)
20	Close error
24	Read error
28	Decompress error
32	Storage not available
36	Invalid function code
40	Invalid file attribute (for example, PDS but no member, member but not PDS)
44	Verification failed (for example, not a valid side file, program name not found)
48	Unexpected EOF
52	No TIOT ENQ

Usage notes

- IBMPXSF can only be used on z/OS non-CICS.
- IBMPXSF is AMODE(31) and RMODE(ANY). It expects to be entered in AMODE(31). It returns using a BSM. Routines that call IBMPXSF must access it by a dynamic load (for example, issuing a LOAD SVC, using CEELoad, COBOL dynamic call, PL/I fetch, C fetch). IBMPXSF must not be linked with any other load modules, otherwise it will cause upward compatibility problems when moving from one release of Language Environment to another.
- IBMPXSF expects to be called from a Language Environment-conforming program.
- R12 must point to the CAA on entry.
- R13 must point to a DSA with a valid NAB.
- The user is responsible for freeing each of the three storage areas (address of compilation information, address of statement table, and address of expanded program source that are returned in the output structure) using Language Environment free heap storage services (for example, CALL CEEFRST(address,feedback-code) or CALL CEEVFRST(address,feedback-code)). If the storage is not freed by the user, it will be implicitly freed by Language Environment at enclave termination.
- The minimum Enterprise PL/I compiler level supported by IBMPXSF is V4R1M0.
- The following compiler options are required to ensure that the SYSDEBUG side file contains the complete expanded program source and statement table:
 - TEST(SEPARATE) – the ALL and NOHOOK sub-options are also recommended but not required.
 - GONUMBER(SEPARATE) – required to produce the statement table in the SYSDEBUG side file.

- MACRO or PP(MACRO) is required if there are %INCLUDE statements in the source (using the MACRO suboption CASE(ASIS) will leave the case of the source unchanged).
- LISTVIEW(AFTERALL) – required if include files, EXEC CICS commands, or SQL code are in the source.

Compilation information

1. The address and length of the compilation information are returned in the output structure.
2. This maps to the following side file header

```
+00 H    Length of side file header info in halfwords
+02 H    Version of side file header
+04 CL3  Eyecatcher ('SID')
+07 CL17 YYYYYMDDHHMMSSTTT (compile date and time)
+18 CL6  VVRRMM (compiler version/release/modification level)
+1E H    # of include files
+20 CL8  YYYYYMDD (compiler build date; version >= 2 only)
```

Statement table

1. The address and length of the statement table are returned in the output structure.
2. The statement table consists of one or more block statement structures which represent the blocks in the compile unit.
3. Each block statement structure may contain one or more statement sections depending on the length of the code for the block.
4. The format of each block statement structure entry is as follows (may be repeated as needed):
 - a. Block statement header


```
+00 F    Offset of block entry within compile unit
+04 F    Length of generated code for block
```
 - b. Statement section (one for every x'8000' bytes of code in the block)


```
+08 F    Section end offset (from the start of statement table)
+0C XL6  Struct statement table entry (repeated until section end offset is reached)
+00 XL2  Statement offset (including reserved flags)
+02 XL4  Statement number and index
          BL10  file index (should be 0 if the correct compiler options were used)
          BL17  statement number
          BL5   reserved
```
 - c. End block delimiter


```
+nn XL6  X'0E0E0E0E0E0E'
```
5. The end of the statement table may contain up to 8 bytes of padding.
6. When there are multiple statement sections for a block, the statement offset requires a base increment of x'8000' for each statement section after the first one to determine the actual offset into the block.
7. To determine the statement offset value, AND the 2 bytes with x'7FFE' to remove the high order and low order bits.
8. To determine the offset into the module, the block offset value from the statement block header must also be added to the statement offset value.
9. The following is the layout for the block statement structure:

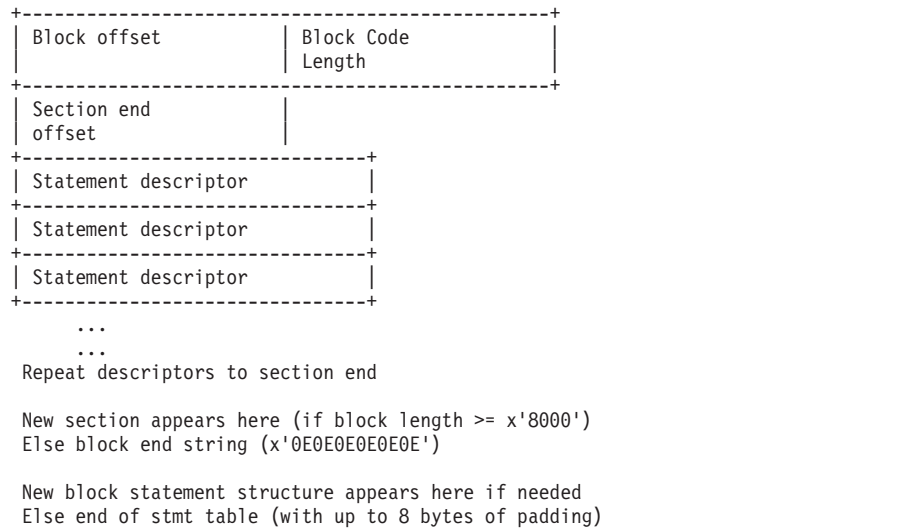


Figure 117. Structure for the block statement

10. The following displays the entry layout for the statement table:

Table 68. Entry layout for statement table

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
1	Bits: 14	1	10	17	5
x	statement offset value	x	file index	statement number	x

Expanded program source

- The address and length of the expanded program source are returned in the output structure.
- The data is mapped as follows:
 - +00 F number of records
 - +04 F max LRECL
 - Expanded source code (repeated as needed)
 - +08 CL2+n struct variable length strings
 - +00 H LRECL
 - +02 CLn statement
- The expanded source code does not contain the statement numbers; the source lines should be numbered sequentially starting with 1.

Chapter 19. C/C++ special purpose interfaces for IEEE floating-point

This section describes the C/C++ special purpose interfaces for IEEE floating-point.

IEEE binary floating-point introduction

Starting with the IBM S/390 Generation 5 Server, support for IEEE binary floating-point (IEEE floating-point) as defined by the ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, is included. Starting with Version 2 Release 6, OS/390 (including the Language Environment and C/C++ components) has added support for IEEE floating-point.

Note:

1. You must have OS/390 Release 6 or later to use the IEEE floating-point instructions. In OS/390 Version 2 Release 6, the base control program (BCP) was enhanced to support the new IEEE floating-point hardware in the IBM S/390 Generation 5 Server. This enabled programs running on OS/390 Release 6 or later to use the IEEE floating-point instructions and 16 floating-point registers. In addition, the BCP provided simulation support for all the new floating-point hardware instructions. This enabled applications that make light use of IEEE floating-point, and which could tolerate the overhead of software simulation, to execute on OS/390 Release 6 or later without requiring an IBM S/390 Generation 5 Server.
2. The terms “binary floating-point” and “IEEE floating-point” are used interchangeably. The abbreviations BFP and HFP, which are used in some function names, refer to binary floating-point and S/390 hexadecimal floating-point (hexadecimal floating-point), respectively.
3. IEEE binary floating-point is fully supported in a CICS environment only if CICS TS Version 4 or later is in use.

The z/OS XL C/C++ compiler provides a FLOAT option to select the format of floating-point numbers produced in a compile unit. The FLOAT option allows you to select either IEEE floating-point or hexadecimal floating-point format. For information on the z/OS XL C/C++ compiler options, see *z/OS XL C/C++ User's Guide*.

The C/C++ runtime library interfaces support both IEEE floating-point and hexadecimal floating-point formats. These interfaces are documented in *z/OS XL C/C++ Runtime Library Reference*.

The primary documentation for the IEEE floating-point support is contained in the *z/Architecture Principles of Operation, SA22-7832*, and the *z/OS XL C/C++ User's Guide*.

IEEE floating-point is provided on S/390 primarily to enhance interoperability and portability between S/390 and other platforms. It is anticipated that IEEE floating-point will be most commonly used for new and ported applications, and in emerging environments, such as Java. Customers should not migrate existing applications that use hexadecimal floating-point to IEEE floating-point, unless there is a specific reason (such as a need to interoperate with a non-S/390 platform).

IEEE binary floating-point

IBM does not recommend mixing floating-point formats in an application. However, for applications which must handle both formats, the C/C++ runtime library does provide some support which is described below.

IEEE decimal floating-point introduction

Starting with z/OS V1R8, including the Language Environment and C/C++ components, support has been added for IEEE decimal floating-point as defined by the ANSI/IEEE Standard P754/D0.15.3, IEEE Standard for Floating-Point Arithmetic.

Note:

1. You must have z/OS V1R8 or higher to use IEEE decimal floating-point, the hardware must have the Decimal Floating Point Facility installed, and the `__STDC_WANT_DEC_FP__` feature test macro must be defined.
2. The abbreviation DFP refers to IEEE decimal floating-point.
3. IEEE decimal floating-point is fully supported in a CICS environment only if CICS TS Version 4 or later is in use.

The z/OS XL C/C++ compiler provides a DFP option to include support for IEEE decimal floating-point numbers. For details on the z/OS XL C/C++ support, see the description of the DFP option in *z/OS XL C/C++ User's Guide*.

New C/C++ runtime library interfaces, which support IEEE decimal floating-point numbers have been added for z/OS V1R8, and other existing interfaces have been updated to support DFP. These interfaces are documented in *z/OS XL C/C++ Runtime Library Reference*.

The primary documentation for the IEEE decimal floating-point support is contained in *z/Architecture Principles of Operation, SA22-7832* and *z/OS XL C/C++ User's Guide*.

Reference information for IEEE floating-point can also be found in *z/OS XL C/C++ Language Reference*.

Selection of fdlibm or fdlibm replacement functions

In 1999, the C/C++ Runtime Library provided IEEE754 floating-point arithmetic support in support of IBM's Java group. The Java language had a bit-wise requirement for its math library, meaning that all platforms needed to produce the same results as Sun Microsystems' fdlibm (Freely Distributed LIBM) library. Therefore, Sun Microsystems' fdlibm code was ported to the C/C++ Runtime Library to provide IEEE754 floating-point arithmetic support. Subsequent to the C/C++ Runtime Library's 1999 release of IEEE754 floating-point math support, IBM's Java group provided their own support of IEEE754 floating point arithmetic and no longer use the C/C++ Runtime Library for this support.

Beginning in z/OS V1R9, a subset of the original fdlibm functions are being replaced by new versions that are designed to provide improved performance and accuracy. The new versions of these functions are replaced at the existing entry points. However, as a migration aid, IBM has provided new entry points for the original fdlibm versions. Applications that take no action will automatically use the updated functions. There are two methods for accessing the original functions. The details about the two methods are as follows:

To access the original fdlibm functions, you can use the following methods:

1. If the application has not included `math.h` or uses feature test macro `_FP_MODE_VARIABLE`, environment variable `_EDC_IEEEV1_COMPATIBILITY_ENV` can be set to `ON` in order to access the original versions of the functions. If the environment variable is not set or set to any value other than `ON`, the new versions of the functions will be used. This method does not require the application to be recompiled. Note that if the application is running in variable mode and was either compiled `FLOAT(HEX)` or has used `__fp_setmode()` to switch over to hexadecimal floating-point mode, the hexadecimal versions of the functions will be called no matter the setting of the environment variable.
2. If the application includes `math.h`, does not use feature test macro `_FP_MODE_VARIABLE`, and uses `FLOAT(IEEE)` compiler option, the application will need to be recompiled with feature test macro `_IEEEV1_COMPATIBILITY` defined so that the affected math functions can be mapped to the new entry points that provide the old behavior. This method requires the application to be recompiled. See *z/OS XL C/C++ Runtime Library Reference* for more information on the `_IEEEV1_COMPATIBILITY` feature test macro.

Note: IBM suggests always including `math.h`, so it is likely that the application will need to use the previous second method if it is desired to use the old versions of the functions.

IEEE floating-point functions

The following sections describe the IEEE floating-point functions.

`__chkbfp()` — check IEEE facilities usage Standards

Standards / Extensions	C or C++	Dependencies
		z/OS V1.8 (for DFP)

Syntax

```
#include <_Ieee754.h>
int __chkbfp(void);
```

General description

The system sets a flag in the secondary task control block (STCB) when IEEE floating-point hardware facilities or simulated facilities (including additional floating-point (AFP) registers in hexadecimal floating-point) are first accessed by a task. The `__chkbfp()` function returns the state of this flag.

Return values

- 0 IEEE floating-point facilities (including AFP registers in hexadecimal floating-point mode) have *not* been used by the task.
- 1 IEEE floating-point facilities have been used by the task. (This includes both IEEE binary and decimal floating-points.)

Usage information

To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Related information

- “__fp_level() — determine type of IEEE facilities available” on page 631

__fp_btoh() — convert from IEEE floating-point to hexadecimal floating-point

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
int __fp_btoh(void *src_ptr, int src_type,
              void *trg_ptr, int trg_type,
              int rmode);
```

General description

The __fp_btoh() function converts data in IEEE floating-point format, pointed to by *src_ptr*, to hexadecimal floating-point format, and stores the hexadecimal floating-point value at the location pointed to by *trg_ptr*. *src_ptr* and *trg_ptr* point to C floating-point variables of type float, double, or long double as indicated by *src_type* and *trg_type*. Valid values for *src_type* and *trg_type* are `_FP_FLOAT`, `_FP_DOUBLE`, and `_FP_LONG_DOUBLE`. *rmode* specifies rounding mode for inexact mappings. Valid values are:

Value Description

- `_FP_BH_NR` No rounding
- `_FP_BH_RZ` Rounding toward zero
- `_FP_BH_BRN` Biased round to nearest
- `_FP_BH_RN` Round to nearest
- `_FP_BH_RP` Round toward +infinity
- `_FP_BH_RM` Round toward -infinity

Return values

If invalid *src_type*, *trg_type*, or *rmode* is specified, __fp_btoh() returns -1. Otherwise, it returns the following values:

- 0 Zero (IEEE floating-point +zero or -zero value mapped to hexadecimal floating-point +zero or -zero value, respectively).
- 1 Underflow (IEEE floating-point value is too small to map to hexadecimal

floating-point). In this case **trg_ptr* is set to the hexadecimal floating-point value corresponding to the smallest convertible IEEE floating-point value.

- 2 Success (with rounding performed as indicated by *rmode*).
- 3 Overflow (IEEE floating-point value is too large to map to hexadecimal floating-point). In this case **trg_ptr* is set to the hexadecimal floating-point value corresponding to the largest convertible IEEE floating-point value.

Related information

- “__fp_htob() — convert from hexadecimal floating-point to IEEE floating-point” on page 630

__fp_cast() — cast between floating-point data types

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
int __fp_cast(int mode, void *src_ptr, int src_type,
              void *trg_ptr, int trg_type);
```

General description

The `__fp_cast()` function casts between C floating-point data types, when the data format does not match the format specified by the `FLOAT` compiler option. The *mode* parameter indicates the format of source and target floating-point values pointed to by *src_ptr* and *trg_ptr*. Valid values for the *mode* parameter are `_FP_HFP_MODE` for hexadecimal floating-point format and `_FP_BFP_MODE` for IEEE floating-point format.

src_type and *trg_type* indicate the C data type (float, double, or long double) of the source and target floating-point values, respectively. Valid values for *src_type* and *trg_type* are `_FP_FLOAT`, `_FP_DOUBLE` or `_FP_LONG_DOUBLE`.

Return values

If invalid values for *mode*, *src_type* or *trg_type* are specified, `__fp_cast()` returns -1. Otherwise, it performs the requested cast and returns 0.

Related information

- “__fp_setmode() — set IEEE or hexadecimal mode” on page 632
- “__fp_swapmode() — set IEEE or hexadecimal mode” on page 633
- “__isBFP() — determine application floating-point mode” on page 639

__fp_htob() — convert from hexadecimal floating-point to IEEE floating-point

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
int __fp_htob(void *src_ptr, int src_type,
              void *trg_ptr, int trg_type,
              int rmode);
```

General description

The `__fp_htob()` function converts data in hexadecimal floating-point format, pointed to by `src_ptr`, to IEEE floating-point format, and stores the IEEE floating-point value at the location pointed to by `trg_ptr`. `src_ptr` and `trg_ptr` point to C floating-point variables of type float, double, or long double as indicated by `src_type` and `trg_type`. Valid values for `src_type` and `trg_type` are `_FP_FLOAT`, `_FP_DOUBLE`, and `_FP_LONG_DOUBLE`. `rmode` specifies rounding mode for inexact mappings. Valid values are:

Value Description

<code>_FP_HB_NR</code>	No rounding
<code>_FP_HB_RZ</code>	Rounding toward zero
<code>_FP_HB_BRN</code>	Biased round to nearest
<code>_FP_HB_RN</code>	Round to nearest
<code>_FP_HB_RP</code>	Round toward +infinity
<code>_FP_HB_RM</code>	Round toward -infinity

Return values

If invalid `src_type`, `trg_type`, or `rmode` is specified, `__fp_htob()` returns -1. Otherwise, it returns the following values:

- 0 Zero (hexadecimal floating-point +zero or -zero value mapped to IEEE floating-point +zero or -zero value, respectively).
- 1 Underflow (hexadecimal floating-point value is too small to map to IEEE floating-point). In this case, `*trg_ptr` is set to the IEEE floating-point value corresponding to the smallest convertible hexadecimal floating-point value.
- 2 Success (with rounding performed as indicated by `rmode`).
- 3 Overflow (hexadecimal floating-point value is too large to map to IEEE floating-point). In this case, `*trg_ptr` is set to the IEEE floating-point value corresponding to the largest convertible hexadecimal floating-point value.

Related information

- “__fp_btoh() — convert from IEEE floating-point to hexadecimal floating-point” on page 628

__fp_level() — determine type of IEEE facilities available**Standards**

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
int __fp_level(void);
```

General description

The system provides simulation of IEEE floating-point hardware (including additional floating-point registers in hexadecimal mode). The __fp_level() function determines the level of IEEE floating-point support available.

Return values

- 0 No IEEE floating-point support available.
- 1 IEEE floating-point simulation is available.
- 2 IEEE floating-point hardware is available.

Related information

- “__chkbf() — check IEEE facilities usage” on page 627

__fp_read_rnd() — determine rounding mode**Standards**

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <float.h>
__fprnd_t __fp_read_rnd(void);
```

General description

For an application running in IEEE floating-point mode, the __fp_read_rnd() function returns the current rounding mode indicated by the rounding mode field of the floating-point control (FPC) register. For an application running in hexadecimal floating-point mode, __fp_read_rnd() returns 0.

Note: This function does not return or update decimal floating-point rounding mode bits.

Return values

For an application running in IEEE floating-point mode, __fp_read_rnd() returns the following:

Value Rounding Mode

- _FP_RND_RZ**
Round toward 0
- _FP_RND_RN**
Round to nearest
- _FP_RND_RP**
Round toward +infinity
- _FP_RND_RM**
Round toward -infinity

For an application running in hexadecimal floating-point mode, __fp_read_rnd() returns 0.

Related information

- “__fp_setmode() — set IEEE or hexadecimal mode”
- “__fp_swap_rnd() — swap rounding mode” on page 634
- “__isBFP() — determine application floating-point mode” on page 639

__fp_setmode() — set IEEE or hexadecimal mode

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
void __fp_setmode(int mode);
```

General description

The __fp_setmode() function sets a flag to tell C/C++ runtime library functions whether to interpret parameters as IEEE floating-point or hexadecimal floating-point values based on the value of *mode* as follows:

Value Description

- _FP_MODE_RESET**
Use the FLOAT compile option to determine the format of floating-point parameters.
- _FP_HFP_MODE**
Interpret parameters as hexadecimal floating-point values.
- _FP_BFP_MODE**
Interpret parameters as IEEE floating-point values.

Note: The compiler defines the __BFP__ macro if the FLOAT(IEEE) compile option is chosen. Otherwise, it undefines the __BFP__ macro. Headers related to floating-point, <float.h>, <limits.h>, and <math.h>, use the __BFP__ macro to select floating-point-type-specific bindings for functions and constants at compile-time. Applications that use __fp_setmode() must use the _FP_MODE_VARIABLE macro

to prevent type-specific compile-time binding of functions and constants as illustrated by the following example:

```
#define _FP_MODE_VARIABLE
#include <float.h>
#include <limits.h>
#include <math.h>
...
```

Return values

None

Related information

- “__fp_cast() — cast between floating-point data types” on page 629
- “__fp_swapmode() — set IEEE or hexadecimal mode”
- “__fp_swap_rnd() — swap rounding mode” on page 634
- “__isBFP() — determine application floating-point mode” on page 639

__fp_swapmode() — set IEEE or hexadecimal mode

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
int __fp_swapmode(int mode);
```

General description

The __fp_swapmode() function sets a flag to tell C/C++ runtime library functions whether to interpret parameters as IEEE floating-point or hexadecimal floating-point values based on the value of *mode* as follows:

Value Description

__FP_MODE_RESET

Use the FLOAT compile option to determine the format of floating-point parameters.

__FP_HFP_MODE

Interpret parameters as hexadecimal floating-point values.

__FP_BFP_MODE

Interpret parameters as IEEE floating-point values.

Usage Notes:

1. Language Environment Library code and non-Language Environment Library code which have the Language Environment library bit set must use __fp_swapmode() to explicitly set floating point behavior. Failure to do so could result in incorrect floating point values.

__fp_swapmode

2. Users of the Language Environment library bit which call c-rtl functions which could potentially use floating point values, need to use the `__fp_swapmode()` function as in the following example.
 - a. Customer application compiled for IEEE floating point math, calls the C++ class library.
 - b. the C++ class library calls `__fp_swapmode()`, as follows:

```
int custmode;

/*save customer fp mode and set fp mode to HFP */
custmode = __fp_swapmode (_FP_HFP_MODE);

/* perform call to c-rtl */
sprintf();

/* restore customer fp mode */
__fp_setmode(custmode);
```

Return values

`__fp_swapmode()` returns a flag (same values as above) with the changed from floating point mode of operation. If `__fp_swapmode()` is passed a value for `fpmode` other than the values shown above, the changed-to floating point mode will remain unchanged. The return value will continue to be the changed-from floating point mode.

Related information

- “`__fp_cast()` — cast between floating-point data types” on page 629
- “`__fp_setmode()` — set IEEE or hexadecimal mode” on page 632
- “`__fp_swap_rnd()` — swap rounding mode”
- “`__isBFP()` — determine application floating-point mode” on page 639

__fp_swap_rnd() — swap rounding mode

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <float.h>
__fprnd_t __fp_swap_rnd(__fprnd_t rmode);
```

General description

For an application running in IEEE floating-point mode, the `__fp_swap_rnd()` function returns the current rounding mode specified by the rounding mode field of the floating-point control (FPC) register and sets the rounding mode field in the FPC based on the value of *rmode* as follows:

Value Rounding Mode

`_FP_RND_RZ`

Round toward 0

- _FP_RND_RN**
Round to nearest
- _FP_RND_RP**
Round toward +infinity
- _FP_RND_RM**
Round toward -infinity

Note:

1. When processing IEEE floating-point values, the C/C++ runtime library math functions require IEEE rounding mode of round to nearest. The C/C++ runtime library takes care of setting round to nearest rounding mode while executing math functions and restoring application rounding mode before returning to the caller.
2. This function does not return or update decimal floating-point rounding mode bits.
3. For an application running in hexadecimal floating-point mode, __fp_swap_rnd() returns 0.

Return values

For an application running in IEEE floating-point mode, __fp_swap_rnd() function returns the following values; for an application running in hexadecimal floating-point mode, __fp_swap_rnd() returns 0.

Value	Description
_FP_RND_RZ	Round toward 0
_FP_RND_RN	Round to nearest
_FP_RND_RP	Round toward +infinity
_FP_RND_RM	Round toward -infinity

Related information

- “__fp_read_rnd() — determine rounding mode” on page 631
- “__fp_setmode() — set IEEE or hexadecimal mode” on page 632
- “__fp_swapmode() — set IEEE or hexadecimal mode” on page 633
- “__isBFP() — determine application floating-point mode” on page 639

__fpc_rd() — read floating-point control register

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
void __fpc_rd(_FP_fpcreg_t *fpc_ptr);
```

General description

The __fpc_rd() function stores the contents of the floating-point control (FPC) register at the location pointed to by *fpc_ptr*.

Note: This function does not return or update decimal floating-point rounding mode bits.

Return values

None

Related information

- “__fpc_rs() — read floating-point control register and change rounding mode field”
- “__fpc_rw() — read and write the floating-point control register” on page 637
- “__fpc_sm() — set floating-point control register rounding mode field” on page 638
- “__fpc_wr() — write the floating-point control register” on page 639
- “__fp_read_rnd() — determine rounding mode” on page 631

__fpc_rs() — read floating-point control register and change rounding mode field

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
void __fpc_rs(_FP_fpcreg_t *cur_ptr, _FP_rmode_t rmode);
```

General description

The __fpc_rs() function stores the current contents of the floating-point control (FPC) register at the location pointed to by *cur_ptr* and then sets the rounding mode field of the FPC based on the value specified by *rmode* as follows:

Value Rounding Mode

- _RMODE_RN**
Round to nearest
- _RMODE_RZ**
Round toward zero
- _RMODE_RP**
Round toward +Infinity
- _RMODE_RM**
Round toward -Infinity

Note:

1. When processing IEEE floating-point values, the C/C++ runtime library math functions require IEEE rounding mode of round to nearest. The C/C++ runtime

library takes care of setting round to nearest rounding mode while executing math functions and restoring application rounding mode before returning to the caller.

2. This function does not return or update decimal floating-point rounding mode bits.

Return values

None

Related information

- “__fpc_rd() — read floating-point control register” on page 635
- “__fpc_rw() — read and write the floating-point control register”
- “__fpc_sm() — set floating-point control register rounding mode field” on page 638
- “__fpc_wr() — write the floating-point control register” on page 639
- “__fp_swap_rnd() — swap rounding mode” on page 634

__fpc_rw() — read and write the floating-point control register

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
void __fpc_rw(_FP_fpcreg_t *cur_ptr, _FP_fpcreg_t *new_ptr);
```

General description

The __fpc_rw() function stores the current contents of the floating-point control (FPC) register at the location pointed to by *cur_ptr* and then replaces the contents of the floating-point control (FPC) register with the value pointed to by *new_ptr*.

Note:

1. When processing IEEE floating-point values, the C/C++ runtime library math functions require IEEE rounding mode of round to nearest. The C/C++ runtime library takes care of setting round to nearest rounding mode while executing math functions and restoring application rounding mode before returning to the caller.
2. This function does not return or update decimal floating-point rounding mode bits.

Return values

None

Related information

- “__fpc_rd() — read floating-point control register” on page 635

- “__fpc_rs() — read floating-point control register and change rounding mode field” on page 636
- “__fpc_sm() — set floating-point control register rounding mode field”
- “__fpc_wr() — write the floating-point control register” on page 639
- “__fp_swap_rnd() — swap rounding mode” on page 634

__fpc_sm() — set floating-point control register rounding mode field

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
void __fpc_sm(_FP_rmode_t rmode);
```

General description

The __fpc_sm() function changes the rounding mode field of the floating-point control (FPC) register based on the value of *rmode* as follows:

Value Description

- _RMODE_RN**
Round to nearest
- _RMODE_RZ**
Round toward zero
- _RMODE_RP**
Round toward +infinity
- _RMODE_RM**
Round toward -infinity

Note:

1. When processing IEEE floating-point values, the C/C++ runtime library math functions require IEEE rounding mode of round to nearest. The C/C++ runtime library takes care of setting round to nearest rounding mode while executing math functions and restoring application rounding mode before returning to the caller.
2. This function does not return or update decimal floating-point rounding mode bits.

Return values

None

Related information

- “__fpc_rd() — read floating-point control register” on page 635
- “__fpc_rs() — read floating-point control register and change rounding mode field” on page 636
- “__fpc_wr() — write the floating-point control register” on page 639

- “__fpc_rw() — read and write the floating-point control register” on page 637
- “__fp_swap_rnd() — swap rounding mode” on page 634

__fpc_wr() — write the floating-point control register

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>
void __fpc_wr(_FP_fpcreg_t *fpc_ptr);
```

General description

The __fpc_wr() function replaces the contents of the floating-point control (FPC) register with the value pointed to by *fpc_ptr*.

Note:

1. When processing IEEE floating-point values, the C/C++ runtime library math functions require IEEE rounding mode of round to nearest. The C/C++ runtime library takes care of setting round to nearest rounding mode while executing math functions and restoring application rounding mode before returning to the caller.
2. This function does not return or update decimal floating-point rounding mode bits.

Return values

None

Related information

- “__fpc_rd() — read floating-point control register” on page 635
- “__fpc_rs() — read floating-point control register and change rounding mode field” on page 636
- “__fpc_rw() — read and write the floating-point control register” on page 637
- “__fpc_sm() — set floating-point control register rounding mode field” on page 638
- “__fp_swap_rnd() — swap rounding mode” on page 634

__isBFP() — determine application floating-point mode

Standards

Standards / Extensions	C or C++	Dependencies
	Both	OS/390 V2R6

Syntax

```
#include <_Ieee754.h>  
int __isBFP(void)
```

General description

The `__isBFP()` function determines the application floating-point mode.

Return values

`__isBFP()` returns 1, if the floating-point mode of the caller is IEEE; it returns 0, if the floating-point mode of the caller is hexadecimal.

Related information

- “`__fp_read_rnd()` — determine rounding mode” on page 631
- “`__fp_setmode()` — set IEEE or hexadecimal mode” on page 632
- “`__fp_swapmode()` — set IEEE or hexadecimal mode” on page 633
- “`__fp_swap_rnd()` — swap rounding mode” on page 634

`__to_xx()` – C/C++ compiler casting support

Standards

Standards / Extensions	C or C++	Dependencies
		z/OS V1R8

Syntax

The following prototypes are not supplied in any header file, so they must be defined before these functions can be used. See Table 69 on page 642 for a description of the `conv` and `value_p` arguments.

```

#ifdef __cplusplus
extern "C" {
#endif

float    __to_b1(unsigned int conv, void *value_p);
double   __to_b2(unsigned int conv, void *value_p);
long double __to_b4(unsigned int conv, void *value_p);
__Decimal32 __to_d1(unsigned int conv, void *value_p);
__Decimal64 __to_d2(unsigned int conv, void *value_p);
__Decimal128 __to_d4(unsigned int conv, void *value_p);
float    __to_h1(unsigned int conv, void *value_p);
double   __to_h2(unsigned int conv, void *value_p);
long double __to_h4(unsigned int conv, void *value_p);

#ifdef __cplusplus
}
#endif

#pragma map(__to_b1, "\174\174T0\174B1")
#pragma map(__to_b2, "\174\174T0\174B2")
#pragma map(__to_b4, "\174\174T0\174B4")
#pragma map(__to_d1, "\174\174T0\174D1")
#pragma map(__to_d2, "\174\174T0\174D2")
#pragma map(__to_d4, "\174\174T0\174D4")
#pragma map(__to_h1, "\174\174T0\174H1")
#pragma map(__to_h2, "\174\174T0\174H2")
#pragma map(__to_h4, "\174\174T0\174H4")

```

Note: The only names that can be called are the #pragma mapped names beginning with @@. These names are also the default compiler short names.

General description

These functions convert an input floating-point number pointed to by *value_p* to an output floating-point number of the return type shown in the prototypes in the previous format section. The *conv* parameter specifies the type of the input floating-point number, as well as the rounding mode to use. The return values of `__to_b1()`, `__to_b2()`, and `__to_b4()` are always binary floating-point numbers of the indicated length. The return values of `__to_h1()`, `__to_h2()`, and `__to_h4()` are always hexadecimal floating-point numbers of the indicated length.

Table 69. Arguments for __to_xx()

Argument	Description
conv	<p>Conversion descriptor:</p> <p>Bits 0-18 Must be 0</p> <p>Bit 19 Allow or suppress exceptions</p> <p>0 Do not suppress any hardware exceptions during the conversion</p> <p>1 Suppress any hardware exceptions using the FPC register or program check shunting in Language Environment Program check shunting does not suppress HFP exponent overflow exceptions when all of the following conditions are met:</p> <ul style="list-style-type: none"> • TRAP(ON,NOSPIE) is in effect. • Using __to_h1() to convert input hexadecimal floating-point double or long double values, or using __to_h2() to convert input hexadecimal long double values. • The input value is too large to convert to the shorter format, causing the hardware to report an HFP exponent overflow. <p>Bits 20-23 Type of input value; other values are not valid:</p> <p>0 Hexadecimal float</p> <p>1 Hexadecimal double</p> <p>2 Hexadecimal long double</p> <p>5 Binary float</p> <p>6 Binary double</p> <p>7 Binary long double</p> <p>8 _Decimal32</p> <p>9 _Decimal64</p> <p>A _Decimal128</p> <p>Bits 24-27 Exception control flags. These bits must all be zero when converting from DFP input to DFP output, BFP input to BFP output, or HFP input to HFP output. They are honored only when converting between any two of the following: DFP, BFP, HFP.)</p> <p>Bit 24 Inexact suppression control</p> <p>0 IEEE-inexact exceptions are recognized and reported in the normal manner</p> <p>1 IEEE-inexact exceptions are not recognized</p> <p>Bit 25 Must be zero</p> <p>Bit 26 HFP-overflow control; this bit must be zero unless the output value is HFP.</p> <p>0 HFP-overflow exceptions are reported as IEEE-invalid-operation exceptions and are subject to the IEEE-invalid-operation mask</p> <p>1 HFP-overflow exceptions are reported as IEEE-overflow exceptions and are subject to the IEEE-overflow mask</p>

Table 69. Arguments for __to_xx() (continued)

Argument	Description
<i>conv</i> (continued)	<p>Bit 27 HFP-underflow control/DFP quantum control</p> <ul style="list-style-type: none"> • If the output number is HFP (underflow control) : <ul style="list-style-type: none"> 0 HFP underflow causes the result to be set to a true zero with the same sign as the input, and the underflow is not reported. The result in this case is inexact and is subject to the inexact-suppression control (Bit 24) 1 HFP underflow is reported as an IEEE-underflow exception, and is subject to the IEEE-underflow mask. • If the output number is DFP (quantum control) : <ul style="list-style-type: none"> 0 The preferred quantum for exact DFP results is the maximum possible, which means that trailing zeroes are removed and the exponent is adjusted upward, if possible. For example, <code>_Decimal32 100.0</code> becomes <code>+0000001.E+02</code>. 1 The preferred quantum for exact DFP results is 1, which means that the exponent is 0, when possible, and the output number is an integer that may have trailing zeroes. For example: <code>_Decimal32 100.0</code> becomes <code>+0000100.E+00</code>, and <code>_Decimal32 1000000000.0</code> becomes <code>+1000000.E+03</code>. <p>Bits 28-31</p> <p>A rounding mode; other values are not valid:</p> <ul style="list-style-type: none"> 0 according to DFP rounding mode in FPC 1 according to BFP rounding mode in FPC 8 round to nearest, ties to even 9 round towards 0 A round toward +infinity B round toward -infinity C round to nearest, ties away from 0 D round to nearest, ties toward 0 E round away from 0 F round to prepare for shorter precision <p>Note:</p> <ol style="list-style-type: none"> 1. When converting a binary floating-point value to a shorter binary floating-point value, rounding mode must be 1. 2. When converting a decimal floating-point value to a shorter decimal floating-point value, the rounding mode must be 0, 8, 9, A, B, C, D, E, F. 3. When converting a hexadecimal floating-point value to another hexadecimal floating-point value, the rounding mode must be valid, but does not affect the result, which is rounded by the hardware. 4. When converting a decimal floating-point value to a longer decimal floating-point value, or a binary floating-point value to a longer binary floating-point value, the rounding mode must be valid, but is otherwise ignored (there is no rounding). 5. When the input and output type is the same (no conversion), the rounding mode must be valid, but is otherwise ignored.
<i>value_p</i>	Pointer to the input floating-point value to be converted to the return type for this function. The type of the floating-point value depends on the <i>conv</i> parameter.

Return values

These functions return floating-point values as shown in the prototypes. When the *conv* parameter is not valid, the floating-point return value is 0.0. The return value is undefined when the input floating-point number cannot be converted to a return value of the requested type.

These functions do not set `errno`.

Note: When either the input value or output value are not hexadecimal floating-point, the raising of IEEE exceptions is allowed or suppressed by the combination of the five exception control flags and the current value of the exception mask bits in the floating-point control register (FPC). If both the input and output values are HFP, the raising of exceptions is controlled by the program mask in the PSW and Bit 19 in the exception control flags.

Related information

There are no prototypes provided for these functions. These functions are called by the compiler to support casting operations.

When executing on hardware that does not have the PFPO facility installed, the IEEE Interruption-Simulation (IIS) facility reports some of the exceptions that can occur when the conversion between numbers is in any two of the following formats:

- BFP
- DFP
- HFP

IIS might cause the contents of the floating-point control (FPC) register to be different from regular IEEE exceptions. In particular, the FPC flags and DXC bytes are different. See for more information about the FPC register contents after IIS events.

Part 2. Language Environment vendor interfaces for AMODE 64 applications

This part of the book applies to AMODE 64.

Chapter 20. Common interfaces and conventions for AMODE 64 applications

This section describes the common runtime library components of Language Environment for AMODE 64 applications.

Common runtime environment

A thread is represented by a Library Anchor Area (LAA). All thread- and enclave-related resources can be located either directly within the LAA or through the LAA. An enclave is one or more executable programs each containing one or more compilation units. The executable program that contains the main routine is known as the root executable. An enclave can consist of multiple executable programs. Fetch mechanisms, such as the C `fetch()` function or DLL load, introduce a new executable program into the enclave.

Library not all linkable

Most Language Environment routines cannot be statically linked. In general, it is not possible to make a complete, self-contained AMODE 64 executable.

Reentrancy

All Language Environment library code is reentrant. All read/write areas are dynamically acquired from stack or heap. Language Environment provides a reentrant environment for compiled code.

Recursion

All Language Environment-supplied library code can be called recursively. For example, if an interrupt occurs in a Language Environment routine and the exception is signaled to some other code (user, Language Environment, or language-specific), that code could, in turn, during its exception processing, use the function that originally caused the exception. This does not mean that the application itself is recursive.

Special handling of certain situations, such as short-on-storage conditions, cause recursive entry to be detected and handled appropriately.

AMODE/RMODE

All Language Environment library routines run AMODE 64.

Member code AMODE restrictions

Language Environment can allocate any of its control blocks above the 2 GB Bar. Any member code that accesses a Language Environment control block must run in AMODE 64 to have addressability to the control blocks.

External names

Language Environment supports external names such as files, programs, and data structures in the same manner as the host system. External names are limited to eight SBCS characters. No supported host system permits DBCS names.

Language Environment Conventions

Some languages permit longer names to be used when referring to externally named objects. In order to conform to the host system requirements, each language can use an algorithm to convert a long internal name to a shorter name that is acceptable to the host system.

Language Environment does not define a common naming convention or name conversion algorithm. Users are responsible for ensuring that names are not ambiguous when long names are converted. External and internal forms of names must match after conversion to a shorter form of the name.

Routine layout

The following table shows the two types of AMODE 64 entry points that Language Environment recognizes as Language Environment-conforming routines.

Table 70. AMODE 64 entry points

Entry point type is...	If...
Language Environment-conforming XPLINK	The entry point minus 16 is X'00C300C500C500F1'. XPLINK linkage conventions are used. For layout detail see Figure 118.
CELQSTRT CSECT	The entry point + 32 is CL8'CEESTART'.

The layout entry for XPLINK routines is shown in Figure 118. The layout entry for XPLINK routines is defined by the Version field at offset X'00' in the PPA1, see Figure 122 on page 651.

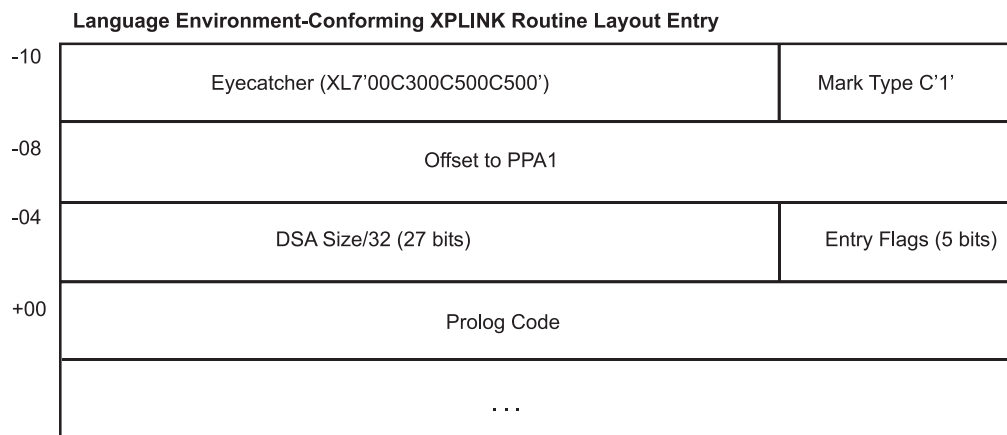


Figure 118. Layout entry of Language Environment-conforming routines – XPLINK

Eyecatcher

A 7-byte field containing the XPLINK eyecatcher, XL7'00C300C500C500'.

Mark Type

Field marking the type of code. Entry code is C'1'.

Offset to PPA1

A signed fullword representing the offset from the start of the entry marker to the start of the PPA1.

DSA Size/32

A 27-bit field representing the size of the routine's DSA in 32-byte increments.

Entry Flags

A 5-bit field containing flag bits to identify the type of routine. If bit 1 is on, the routine is an XPLEAF routine. XPLEAF routines save caller's registers in their own stack frame, but do not update the stack pointer. Bit 2 indicates whether the routine uses the `alloca()` service.

The compiler emits an XPLINK stack extension marker in front of the call to Language Environment for the overflow prolog sequence for the +4K DSA scenario. Figure 119 depicts this marker.

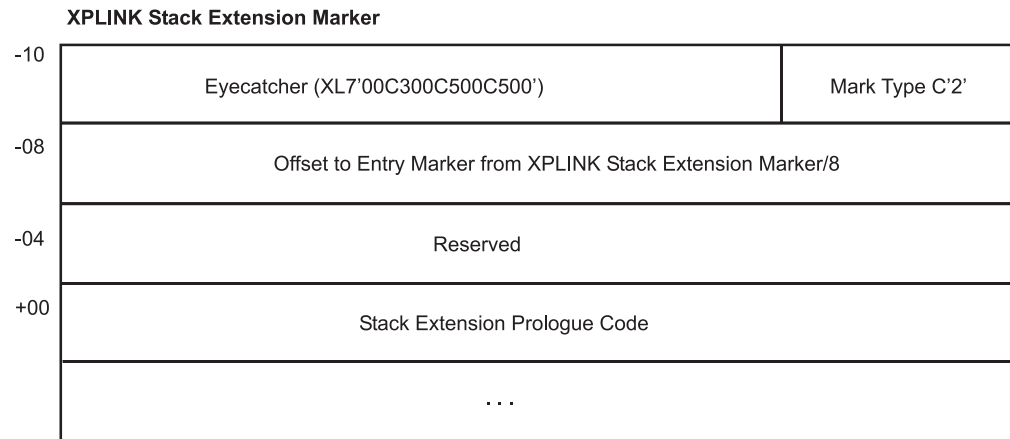


Figure 119. XPLINK stack extension marker

Eyecatcher

A 7-byte field containing the XPLINK eyecatcher, XL7'00C300C500C500'.

Mark Type

Field marking the type of code. XPLINK stack extension is C'2'.

Offset to entry marker from XPLINK stack extension marker/8

The signed offset from the start of the XPLINK stack extension marker to the start of the entry point marker in doublewords.

The XPLINK end of data marker is placed after, or at the end of a section of code, where the compiler may have placed constants. Language Environment's asynchronous signal deliverer uses this in its scan backwards to identify that a signal did not arrive inside a function's prolog. Figure 120 depicts this marker.

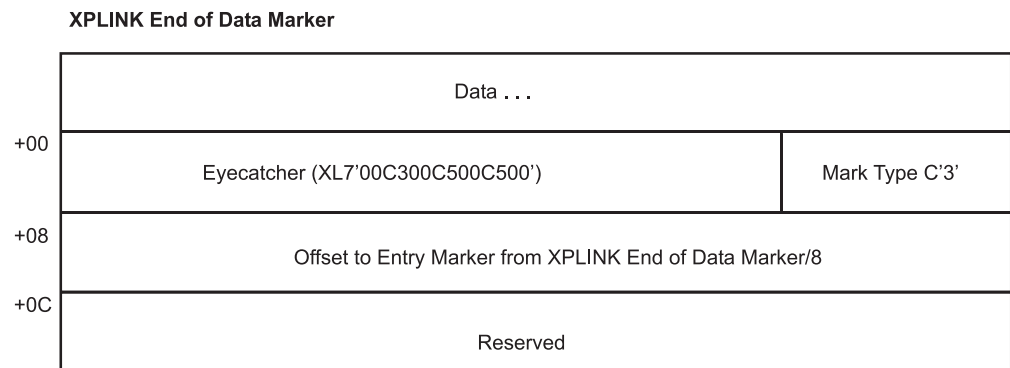


Figure 120. XPLINK end of data marker

Language Environment Conventions

Eyecatcher

A 7-byte field containing the XPLINK eyecatcher, XL7'00C300C500C500'.

Mark Type

Field marking the type of code. XPLINK end of data is C'3'.

Offset to entry marker from XPLINK end of data marker/8

The signed offset from the start of XPLINK end of data marker to the start of the entry point marker in doublewords.

Language Environment implements an 8-byte XPLINK stub entry marker for Language Environment and C runtime stubs. Figure 121 depicts this marker.

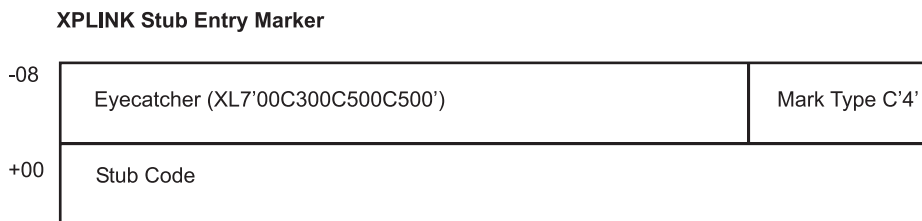


Figure 121. XPLINK stub entry marker

Eyecatcher

A 7-byte field containing the XPLINK eyecatcher, XL7'00C300C500C500'.

Mark Type

Field marking the type of code. XPLINK Stub is Entry C'4'.

Prolog information blocks

The prolog information blocks for the XPLINK layout are defined in Figure 122 on page 651, Figure 123 on page 652, and Figure 129 on page 659. The prolog information exists for every block or internal procedure.

Program Prolog Area-1 (PPA1) appears for every Language Environment entry point. There is a one-to-one correlation between a PPA1 and a DSA. The content of the entry/label name field is defined by member languages. The name can be SBCS characters or DBCS characters bracketed by shift-codes. Member-defined information can be placed starting at offset X'20'. Fields described as fullword offsets are treated as signed offsets.

Program Prolog Area-2 (PPA2) appears once for each compile unit and can immediately follow the primary PPA1. The control level field indicates the change level of the prolog. The timestamp and version information normally appears at the end of PPA2 and is optional. The version and release data fields identify the level of the compiler that produced the object code. You can use the PPA2 field at offset X'10' to determine the primary entry point for the compilation unit. It is zero if the compilation unit primary entry point does not exist. Member-defined information can be placed at the end of PPA2.

Program Prolog Area-3 (PPA3), if available, appears once for every Language Environment entry point. It provides additional information about an entry point, and typically contains information relevant for problem determination tools. There is a one-to-one correlation between a PPA1 and a PPA3. The PPA3 layout may differ among different member languages.

Program Prolog Area-4 (PPA4), if available, appears once for each compilation unit. It provides additional information about a compilation unit, and typically contains information relevant for problem determination tools. There is a one-to-one correlation between a PPA2 and a PPA4. The PPA4 layout may differ among different member languages.

In the timestamp block, as shown in Figure 131 on page 660, the two characters that indicate the version are to be used at the discretion of the high level language that produces the block; they are not interrogated by Language Environment. In addition, the dump service uses the service level field to add the module service level information to the traceback.

PPA1 in support of XPLINK

To optimize the space used for control purposes, the structure and contents of the PPA1 for XPLINK have been redefined. The control block is made up of a fixed part followed by a contiguous optional part, with the presence of optional fields indicated by flag bits. Optional fields, if present, are stored immediately following the fixed part of the PPA1 aligned on fullword boundaries in the order specified below.

PPA1: XPLINK Entry Point Block Fixed Area (Version 3)

+00	Version	LE Signature 'X'CE' (Lan Env Signature)	Saved GPR Mask	
+04	Signed Offset to PPA2 from start of PPA1			
+08	PPA1 Flags 1	PPA1 Flags 2	PPA1 Flags 3	PPA1 Flags 4
+0C	Length/4 of Parm		Length/2 of Prolog	Alloca Reg Offs/2 R4 Chg
+10	Length of Code			

Figure 122. Prolog constants format – level 4 (XPLINK), PPA1: entry point block (Version 3)

The PPA1 is located through an offset field preceding the entry point which provides flexibility to group all PPA1s either by compilation unit or by module. The new PPA1 content is extensible in that a Version field identifies the particular table structure. Program prolog areas are mandatory for languages participating in XPLINK. Each entry point must have a corresponding PPA1 associated with it.

Language Environment Conventions

PPA1 fixed area fields:

+0	Version	CEL Signature X'CE' (Lang Env Signature)	Saved GPR Mask	
+4	Signed offset to PPA2 from start of PPA1			
+8	PPA1 Flag 1 0 DSA Format 0: 32 bit 1: 64 bit 1 0: Short form PPA1 1: Reserved 2 Exception Model 0: Own 1: Caller's 3 PPA3 type flags 0: tiny PPA3 1: full PPA3 4 Invoke member for DSA exit event 5 XPLink Exit DSA 6 Special Linkage 7 Vararg function	PPA1 Flag 2 0 Procedure 0: Internal 1: External 1 Reserved, 0 2 Reserved, 0 3 Reserved, 0 4 Reserved, 0 5 Reserved, 0 6 Reserved, 0 7 Reserved, 0	PPA1 Flag 3 0 State Variable Locator 1 Argument Area Length 2 FPR Mask 3 AR Mask 4 Member PPA1 Word 5 Block Debug Info 6 Interface Mapping Flags 7 Java Method Locator Table Indicating fields in optional area	PPA1 Flag 4 0 Reserved, 0 1 Reserved, 0 2 Reserved, 0 3 Reserved, 0 4 Reserved, 0 5 Reserved, 0 6 Reserved, 0 7 Name Length and Name Indicating fields in optional area
+12 0x0c	Length/4 of Parm		Length/2 of Prolog	Alloca Reg
+16 0x10	Length of Code			
				Offset/2 to StackPointer Update

Figure 123. PPA1: XPLINK entry point block fixed area (Version 3) details

Version: An 8-bit field that is set to X'02' to identify this PPA1 as having the Level 4, XPLINK (Version 3) layout.

Language Environment Signature: An 8-bit field that must be set to X'CE'.

Saved GPR mask: A 16-bit mask, indicating which registers are saved and restored by the associated routine. Bit 0 indicates register 0, followed by bits for registers 1 to 15 in order.

Signed offset to PPA2 from the start of PPA1: The offset of the PPA2 block belonging to the compilation unit containing the function described by this PPA1.

PPA1 flag 1: Program flags (PPA1 offset X'08') are shown in Figure 123 and are described below.

```

'0.....'B GPR Save area is 32-bit.
'1.....'B GPR Save area is 64-bit.
'.0.....'B Indicates that this is a short form of the PPA1.
'..0....'B Own exception model.
'...1....'B Inherited exception model.
'....0...'B Tiny PPA3.
'....1...'B Full PPA3.
'....0...'B Do Not call member for Exit DSA event.
'....1...'B Call member for Exit DSA event.
'.....0..'B Stack frame is Not an XPLINK Exit DSA.
'.....1..'B Stack frame is an XPLINK Exit DSA.
'.....0.'B This is not a Special linkage routine.
'.....1.'B This is a Special linkage routine.
'.....0'B Not a Vararg routine.
'.....1'B Vararg routine.

```

Figure 124. Language Environment PPA1 flag 1 offset X'08'

The PPA1 flag 1 field (PPA1 offset X'08') contains 8 bits, as shown above, and are described as follows:

Bit location	Description
Bit 0	Format of General Purpose Registers (GPR) save area 0 Indicates that GPRs are saved as 32-bit quantities. 1 Indicates that GPRs are saved as 64-bit quantities.
Bit 1	Format of PPA1 0 Indicates that this is a short form of the PPA1. 1 Reserved.
Bit 2	Exception Model Flag 0 Indicates that this routine uses it's own exception model. 1 Indicates that this routine inherited the exception model from its caller.
Bit 3	PPA3 Type Flag 0 Indicates that the PPA3 is a tiny PPA3. 1 Indicates that the PPA3 is a full PPA3.
Bit 4	Call Member for DSA Exit flag 0 Indicates that the owning member of the DSA should not be called for Exit DSA processing. 1 Indicates that the owning member of the DSA should be called for Exit DSA processing.
Bit 5	XPLINK Exit DSA Flag 0 Indicates that the associated stack frame is not an XPLINK Exit DSA. 1 Indicates that the associated stack frame is an XPLINK Exit DSA and its GPR7 (return addr) should be given control during stack collapse.
Bit 6	Special Linkage Flag 0 Indicates that this is not a special linkage routine. 1 Indicates that this is a special linkage routine used to handle calls between XPLINK and non-XPLINK routines or to handle calls that cause a stack segment extension.
Bit 7	Vararg Flag 0 Indicates that this is not a variable argument (Vararg) routine. 1 Indicates that this is a Vararg routine.

Language Environment Conventions

PPA1 flag 2: Program flags (PPA1 offset X'09') are shown in Figure 123 on page 652 and are described below.

```
'0.....'B Internal procedure
'1.....'B External procedure
'.0000000'B Reserved for future use (must all be zero).
```

Figure 125. Language Environment PPA1 flag 2 offset X'09'

Bit location	Description
Bit 0	Internal/External procedure
0	Indicates that this procedure is an internal procedure with a nesting level greater than zero.
1	Indicates that this procedure is an external procedure with a nesting level of zero.
Bit 1 - 7	Reserved for future use.

PPA1 flag 3: Program flags (PPA1 offset X'0A') are shown in Figure 123 on page 652 and are described below.

```
'0.....'B State Variable locator field is not in optional area.
'1.....'B State Variable locator field is in the optional area.
'.0.....'B Argument Area Length is not in the optional area.
'.1.....'B Argument Area Length is in the optional area.
'..0....'B FP Register Mask is not in the optional area.
'..1....'B FP Register Mask is in the optional area.
'...0...'B No ARs are saved. AR mask not in optional area.
'...1...'B ARs are saved. AR mask in optional area.
'....0...'B Member PPA1 word is not present in optional area.
'....1...'B Member PPA1 word is present in the optional area.
'.....0..'B Offset to PPA3 is not present in optional area.
'.....1..'B Offset to PPA3 is present in the optional area.
'.....0.'B Interface mapping flags not in the optional area.
'.....1.'B Interface mapping flags in the optional area.
'.....0'B Java Method Locator Table not in the optional area.
'.....1'B Java Method Locator Table in the optional area.
```

Figure 126. Language Environment PPA1 flag 3 offset X'0A'

Bit location	Description
Bit 0	State Variable Locator Flag
0	Indicates that this field is not present in the optional part of the PPA1.
1	Indicates that this field is present in the optional part of the PPA1.
Bit 1	Argument Area Length
0	Indicates that this field is not present in the optional part of the PPA1.
1	Indicates that this field is present in the optional part of the PPA1.

Bit location	Description
Bit 2	Floating-Point Registers Flag
	<p>0 Indicates that the Floating-Point registers are not saved in the DSA.</p> <p>1 Indicates that the Floating-Point registers are saved in the DSA and that the FPR mask and Offset to FPR savearea is present in the optional PPA1 area. If this field is present, the entire word containing FPR Mask and AR Mask is present in the optional area.</p>
Bit 3	Access Registers Flag
	<p>0 Indicates that the Access Registers are not saved in the DSA.</p> <p>1 Indicates that the Access Registers (as indicated by the Saved AR Bit Mask field) are saved in the DSA and the AR mask in the optional area. If this field is present, the entire word containing FPR Mask, Alloca Reg, and AR Mask is present in the optional area.</p>
Bit 4	Member PPA1 Word Flag
	<p>0 Indicates that this field is not present in the optional part of the PPA1.</p> <p>1 Indicates that this field is present in the optional part of the PPA1.</p>
Bit 5	Offset to PPA3 Flag
	<p>0 Indicates that this field is not present in the optional part of the PPA1.</p> <p>1 Indicates that this field is present in the optional part of the PPA1.</p>
Bit 6	Interface Mapping Flag
	<p>0 Indicates that this field is not present in the optional part of the PPA1.</p> <p>1 Indicates that this field is present in the optional part of the PPA1.</p>
Bit 7	Java Method Locator Table
	<p>0 Indicates that this field is not present in the optional part of the PPA1.</p> <p>1 Indicates that this field is present in the optional part of the PPA1.</p>

PPA1 flag 4: Program flags (PPA1 offset X'0B') are shown in Figure 123 on page 652 and are described below.

```
'0000000.'B Reserved for future optional fields (must all be zero).
'.....0.'B Name length and name are not in the optional area.
'.....1.'B Name length and name in the optional area.
```

Figure 127. Language Environment PPA1 flag 4 offset X'0B'

Bit location	Description
Bit 0 - 6	Reserved for future optional fields
Bit 7	Procedure/Label Name Flag
	<p>0 Indicates that the length of name field and the entry/label name field are not present in the optional part of the PPA1.</p> <p>1 Indicates that the length of name field and the entry/label name field are present in the optional part of the PPA1.</p>

Length/4 of parms: Length of expected parameter area for this function in fullwords (for vararg functions, the length of the fixed portion of the parameter list). This is used for copying parameters on stack extension. For vararg functions, the entire caller's argument area must be copied on stack extension.

Language Environment Conventions

Length/2 of prolog: Length of prolog instruction sequence in halfwords starting from the entry point. The prolog is complete when all conditions described in this architecture are satisfied. This includes: saving the non-volatile registers used by the function, including FPRs and ARs; updating the stack pointer; and loading the `alloca()` register. Other instructions from the function body, including setting up various base registers, may be moved into the prolog, so no component can assume anything about the state of registers within the prolog without scanning the prolog code.

alloca() register: The register used to point to automatic storage (and other parts of the originally-allocated stack frame) in functions that use `alloca()`. This must be zero if `alloca()` is not used.

Offset/2 to stack pointer update: The offset in halfwords from the Entry Point to the beginning of the instruction that updates the stack pointer (GPR4). For XPLleaf routines, this field will be set to zero.

Length of code: The length of the code for this function, starting from the entry point marker associated with this PPA1 to the last instruction in the function, in bytes. This does not necessarily include instructions which are the target of "execute," which may be in other parts of the code section, the stack frame, or writable static.

PPA1 optional area fields: There are several optional PPA1 Fields; each one's presence indicated by a flag bit in PPA1 flags3 or PPA1 s4. Where an optional field is less than 4 bytes in length, the entire word is present if any of the fields in that word are present. Unused parts of the word are filled with zeroes. The optional fields are fullword aligned and appear in the order listed here. The field name and length are given:

Field name	Field length
State Variable Locator (PPA1 Flag 3, Bit 0)	4

Field name	Field length
Argument Area Length (PPA1 Flag 3, Bit 1)	4

Field name	Field length	
FPR mask (PPA1 Flag 3, Bit 2)	AR mask (PPA1 Flag 3, Bit 3)	4

Note: If either Bit 2 or Bit 3 of 3 is on, the fullword variable representing FPR mask and AR mask is present.

Field name	Field length
Floating Point Register Save Area Locator (PPA1 Flag 3, Bit 2)	4

Field name	Field length
Access Register Save Area Locator (PPA1 Flag 3, Bit 3)	4

Field name	Field length
PPA1 Member Word (PPA1 Flag 3, Bit 4)	4

Field name	Field length
Offset to PPA3 (PPA1 Flag 3, Bit 5)	4

Field name	Field length
Interface Mapping Flags (PPA1 Flag 3, Bit 6)	4

Field name	Field length
Java Method Locator Table (MLT) (PPA1 Flag 3, Bit 7)	8

Field name		Field length
Length of Name (PPA1 Flag 4, Bit 7)	Name of Function	variable length
Name of Function (continued)		

State variable locator: Defines the location of the state variable. Bits 0-3 contain the number of a GPR whose contents are added to the unsigned offset in bits 4-31 to calculate the address of the state variable. The register used to address the State Variable, typically the stack register or the `alloca()` register, must be set in the prolog and retain its value throughout the function. This field is optional; its presence is indicated by PPA1 flag 3, bit 0.

Argument area length: Length of argument area allocated by this function on the stack. If present, this field contains the size of the largest argument list used by this function. This field is optional; its presence is indicated by PPA1 flag 3, bit 1.

However, this field is **required** for every function that contains a call with an argument list longer than 128 bytes.

FPR mask: A 16-bit mask indicating which of FPRs are saved and restored by this routine. Bit 0 indicates FPR0, followed by bits for FPR1 to FPR 15. Space is reserved in the function's local storage for those FPRs actually saved by the function. This field is optional; its presence is indicated by PPA1 flag 3, bit 2. The word containing this field, if present, has either PPA1 flag 3 bits 2 or 3 on.

Access register mask: Reserved for future use.

Floating Point Register Save Area locator: Defines the location of the Floating Point Register Save Area. Bits 0-3 contain the number of a GPR whose contents are added to the unsigned offset in bits 4-31 to calculate the address of this save area. The register used to address this save area, typically the stack register or the `alloca()` register, must be set in the prolog and retain its value throughout the function. This field is optional; its presence is indicated by PPA1 flag 3, bit 2.

Language Environment Conventions

Access Register Save Area locator: Defines the location of the Access Register Save Area. Bits 0-3 contain the number of a GPR whose contents are added to the unsigned offset in bits 4-31 to calculate the address of this save area. The register used to address this save area, typically the stack register or the `alloca()` register, must be set in the prolog and retain its value throughout the function. This field is optional; its presence is indicated by PPA1 flag 3, bit 3.

Member PPA1 word: This word contains the following information for C/C++ when present.

```
'00000000000000000000000000000000.....'B Reserved (must be zero)
'.....0.....'B Argparse
'.....1.....'B No argparse
'.....0.....'B Redirection
'.....1.....'B No redirection
'.....0.....'B Execops
'.....1.....'B No execops
'.....00000'B Reserved (must be zero)
```

Figure 128. Language Environment PPA1 flag word as defined by C/C++

For C/C++, this word is used for flags as shown in the preceding figure and are described as follows:

Bit location	Description
Bit 0 - 23	Reserved (must be zero)
Bit 24	Noargparse 0 Indicates argparse. 1 Indicates no argparse.
Bit 25	Noredirection 0 Indicates redirection. 1 Indicates no redirection.
Bit 26	Noexecops 0 Indicates execops. 1 Indicates no execops.
Bit 27 - 31	Reserved (must be zero)

Offset to PPA3: Signed offset to PPA3 from the start of PPA1. This field is optional; its presence is indicated by PPA1 flag 3, bit 5.

Interface mapping flags: This field is provided to allow interface mapping by a glue routine when an XPLINK routine is called from non-XPLINK. It describes the linkage type, the floating-point parameters expected by this routine, and the format of the function return value. This field is optional; its presence is indicated by PPA1 flag 3, bit 6.

Java method locator table: Used to locate meta-information for Java classes. This field is optional; its presence is indicated by PPA1 flag 3, bit 7.

PPA2 in support of XPLINK

PPA2: Compile Unit Block

+00	Member Identifier	Member Subid	Member Defined	Control Level (= 4)
+04	Signed offset from PPA2 to CELQSTRT for load module			
+08	Signed offset from PPA2 to PPA4. Zero if PPA4 is not available.			
+0C	Signed offset from PPA2 to timestamp/version information or zero			
+10	Signed offset from PPA2 to the compilation unit's Primary Entry Point			
+14	Compilation flags			
			

Figure 129. Prolog constants format – level 4 (64-bit XPLINK), PPA2: compile unit block

Level 4 (XPLINK), PPA2: compile unit block bits:

'0.....	'B Indicates that program was compiled for hexadecimal floating-point
'1.....	'B Indicates that program was compiled for binary floating-point
'0.....	'B Indicates that the code is compiler generated user code
'1.....	'B Indicates that the code is associated with library code
'..0.....	'B Program does not contain service information
'..1.....	'B Program contains service information
'...0....	'B Not compiled with XPLINK(STOREARGS)
'...1....	'B Compiled with XPLINK(STOREARGS)
'...0....	'B Reserved
'...0....	'B Compiled unit is EBCDIC
'...1....	'B Compiled unit is ASCII
'...0....	'B No additional compiler information after service information
'...1....	'B Additional compiler information after service information
'.....0	'B Not compiled with XPLINK
'.....1	'B Compiled with XPLINK
'.....0	'B Reserved
'.....0	'B MD5 signature is not located at 16 bytes before the timestamp
'.....1	'B MD5 signature is located at 16 bytes before the timestamp
'.....0	'B Not compiled with FLOAT(AFP(VOLATILE))
'.....1	'B Compiled with FLOAT(AFP(VOLATILE))
'.....	...00000 00000000 00000000	'B Reserved

Figure 130. Level 4 (XPLINK), PPA2: compile unit block bits

The XPLINK(STOREARGS) and XPLINK flags were added in PPA2 Level 4.

Timestamp and version: Figure 131 on page 660 shows the format of the information in the timestamp and version.

Language Environment Conventions

00	CL4'yyyy' Year of compilation	
04	CL4'mmdd' Date of compilation	
08	CL4'hmmm' Time of compilation	
0C	CL2'ss' Time of compilation	CL2 'vv' Version
10	CL4'rrmm' Release/Modification	
14	Service level string length	Untruncated service level string

Figure 131. Timestamp and version information

C/C++ DWARF 64-bit PPA4 layout

PPA4 conforms to this layout under these conditions:

- Member identifier (PPA2 offset X'00') is 3
- PPA4 version in PPA4 program flags is 2
- PPA4 program flags indicates 64-bit compile

Table 71. C/C++ DWARF 64-bit PPA4 layout

Offset	Length	Description
X'00'	4	PPA4 debug flags for PPA4 version 2
X'04'	4	PPA4 program flags
X'08'	8	Signed offset from CELQSTRT address to NORENT static
X'10'	8	Signed offset from WSA to RENT static
X'18'	8	Signed offset from PPA4 to symbol offset table
X'20'	8	Signed offset from PPA4 to code csect
X'28'	8	Length of code csect (in bytes)
X'30'	8	Signed offset from PPA4 to DWARF line number table embedded in C_CDA class [optional field, check PPA4 debug flags]

PPA4 debug flags

PPA4 debug flags for PPA4 version 2 - PPA4 offset X'00' are shown in the following code sample:

```
'0.....'B DWARF line number table is not in C_CDA class.
'1.....'B DWARF line number table is in C_CDA class.
'.0.....'B Primary source file name is not available.
'.1.....'B Primary source file name follows DWARF sidefile name.
          (prefixed with 2 bytes string length)
'..0.....'B DWARF is not embedded in NOLOAD D_* class
```

```
'...1.....'B DWARF is embedded in NOLOAD D_* class
'...0.....'B DWARF is not embedded in LOAD_D_* class
'...1.....'B DWARF is embedded in LOAD D_* class
'....0....'B Compilation unit is compiled with DEBUG
'....1....'B Compilation unit is not compiled with DEBUG
'....000 00000000 00000000 00000000'B Reserved
```

PPA4 program flags

PPA4 program flags - PPA4 offset X'04' are shown in the following code example:

```
'00000000 00000...'B Reserved
'.....0..'B 31-bit compile
'.....1..'B 64-bit compile
'.....00..'B Reserved
'.....xxxxxxx..'B PPA4 version
0: DWARF information not present
1: COBOL V5 PPA4
2: C/C++ DEBUG(FORMAT(DWARF)) PPA4
'.....xxxxxxx'B Offset to file name (zero if not applicable)
file name is prefixed with 4 bytes string length
PPA4 version is 0: unsigned offset from PPA4 to source file name
PPA4 version is 2: unsigned offset from PPA4 to DWARF sidefile name
```

Language Environment dynamic storage area

An AMODE 64 XPLINK DSA (Dynamic Storage Area) is described in Figure 132. In an XPLINK function, the currently active DSA is located by GPR4. However, GPR4 is "biased" by x'800' (2048) bytes. This bias needs to be added to the contents of GPR4 to get the actual start of the XPLINK register save area. XPLINK DSAs can be back-chained using the value of GPR4 in the register save area. However, GPR4 is only optionally saved. The correct way to find the caller's DSA is to add the size of the current DSA to its location.

000	CEEDSAHP_BIAS - Stack Bias, DO NOT USE	Note 1
800	CEEDSAHP4T015 - Save area for GPRs 4-15	Note 2
860	Reserved for use by run-time	
870	CEEDSAHPTRAN - Debug Area	
878	CEEDSAHP_ARG_PRE - Argument prefix area	
880	CEEDSAHP_ARGLIST - Start of variable length argument list	Note 3

Figure 132. Language Environment dynamic storage area – XPLINK format for AMODE 64 applications

Note:

- CEEDSAHP_BIAS: This is the size of the bias between the actual value in the XPLINK stack register (GPR4) and the start of the DSA. This area is not usable by the current function. It will contain the DSAs of any called XPLINK functions.
- CEEDSAHP4T015: A called XPLINK function will only save the registers that might be altered during its execution.
- CEEDSAHP_ARGLIST: Area where argument list for called functions is built. Only parameters that are not passed in registers will be stored into the argument area.

Language Environment control block mappings

This section shows the control block mappings for AMODE 64 applications.

Language Environment library anchor area

The library anchor area (LAA), shown in the code example below, is a control block that is allocated during TCB initialization. This is key 0, authorized storage. For mapping information on the LAA, see the SCEEMAC(CEELAA) data set.

```

1 CEELAA
   Language Environment LIBRARY ANCHOR AREA (LAA)
   -----
   OFFSET  OFFSET
   DECIMAL  HEX  TYPE      LENGTH  NAME (DIM)  DESCRIPTION
   -----  -----
       0      (0) STRUCTURE    0  CEELAA      LAA mapping
       0      (0) CHARACTER    4  CEELAAEYE  Eyecatcher 'LAA '
       4      (4) SIGNED      4  CEELAA_VER  version
       8      (8) ADDRESS     4  CEELAA_PREV ptr to previous LAA
      12     (C) ADDRESS     4  CEELAA_NEXT ptr to next LAA
      16     (10) ADDRESS    4  CEELAA_STCB addr of associated STCB
      20     (14) BITSTRING  2  CEELAA_ASID asid of this LAA
      22     (16) CHARACTER  2  CEELAA_RSVD1 reserved
      24     (18) CHARACTER 112 CEELAA_COMPILER C
           +18x compiler dependent flds
      24     (18) CHARACTER  40  CEELAA_COMP_31B
           31 bit
      24     (18) ADDRESS    4  CEELAA_STACKFLOOR31
           stack floor
      28     (1C) ADDRESS    4  CEELAA_STACKOVFLOW31
           stack ovrl rtn
      32     (20) ADDRESS    4  CEELAA_GTAB31 GTAB addr
      36     (24) ADDRESS    4  CEELAA_LCA31  LCA addr
      40     (28) ADDRESS    4  CEELAA_TRT31 addr of trt spc
      44     (2C) CHARACTER  20  CEELAA_RSVD31_1
           reserved
      64     (40) CHARACTER  72  CEELAA_COMP_64B
           64 bit
      64     (40) CHARACTER    8  CEELAA_STACKFLOOR64
      64     (40) ADDRESS     4  CEELAA_STKFLR64_HI
      68     (44) ADDRESS     4  CEELAA_STKFLR64_LO
      72     (48) CHARACTER    8  CEELAA_STACKOVFLOW64
      72     (48) ADDRESS     4  CEELAA_STKOVFL64_HI
      76     (4C) ADDRESS     4  CEELAA_STKOVFL64_LO
      80     (50) CHARACTER    8  CEELAA_GTAB64 GTAB addr
      88     (58) CHARACTER    8  CEELAA_LCA64  LCA addr
      96     (60) CHARACTER    8  CEELAA_TRT64 addr -trt space
     104     (68) CHARACTER   32  CEELAA_RSVD64_1
           Reserved
     136     (88) CHARACTER    8  CEELAA_JIT_RSVD1
           Reserved for JIT
     144     (90) CHARACTER    8  CEELAA_JIT_RSVD2
           Reserved for JIT
     152     (98) CHARACTER   48  *          Reserved future
     200     (C8) BITSTRING    1  CEELAA_FLAG1 Flags *** CANNOT MOVE! ***
           CEELAA_LEACTIVE
           "X'80'" LE env is active
           .1.. .... CEELAA_LEPENDING
           "X'40'" LE env is pending
           ..1. .... CEELAA_IPT "X'20'" this is the IPT
           ...1 .... CEELAA_MEMLIMIT
           "X'10'" Memlimit hit during a
           stk ovflw request
           .... 1... CEELAA_RSTK_ACTIVE
           "X'08'" Reserve Stk active

   OFFSET  OFFSET
   DECIMAL  HEX  TYPE      LENGTH  NAME (DIM)  DESCRIPTION
   -----  -----
           .... .1.. CEELAA_RESERVE_STACK_REQUEST
           "X'04'" This bit is set by the
           Stack
           Overflow SRB routine to
           indicate to Cond.
           Management that a switch to the
           Rsv Stack is necessary
           .... ..1. CEELAA_OVERFLOW_ABEND
  
```


Library Anchor Area (LAA)

				"X'02'" Set by Stack Over- flow SRB to tell Cond. Mgmt to abend
1		CEELAA_FIRST_IN_CHAIN	"X'01'" Reserved
201	(C9) BITSTRING	1	CEELAA_FLAG2	Flags - byte 2
	1...		CEELAA_OVERFLOW_INVALID	"X'80'" Non-USER stack o/f
202	(CA) BITSTRING	1	CEELAA_FLAG3	Flags - byte 3
203	(CB) BITSTRING	1	CEELAA_FLAG4	Flags - byte 4
204	(CC) CHARACTER	36	*	Reserved future
240	(F0) CHARACTER	8	CEELAA_64BIT_CB_STG	Addr of control blks above bar
				Addr of control blks below bar
248	(F8) CHARACTER	8	CEELAA_31BIT_CB_STG	31 bit stuff
256	(100) CHARACTER	24	CEELAA_31BIT	system svce vector
256	(100) ADDRESS	4	CEELAA_SVCVEC31	addr of first SANC
260	(104) ADDRESS	4	CEELAA_SANC31	key of current stk
264	(108) CHARACTER	1	CEELAA_CURKEY31	reserved
265	(109) CHARACTER	15	CEELAA_RSVD31	64 bit stuff
280	(118) CHARACTER	24	CEELAA_64BIT	system svce vector
280	(118) CHARACTER	8	CEELAA_SVCVEC64	addr of first SANC
288	(120) CHARACTER	8	CEELAA_SANC64	key - current stack
296	(128) CHARACTER	1	CEELAA_CURKEY64	reserved
297	(129) CHARACTER	7	CEELAA_RSVD64	heap related fields
304	(130) CHARACTER	24	CEELAA_HEAP	addr of 64bit ENSQ
304	(130) CHARACTER	8	CEELAA_ENSQ64	64bit Library thread heap id
312	(138) CHARACTER	8	CEELAA_THDLHEAP64ID	31bit Library thread heap id
				address of LAA for the IPT
320	(140) CHARACTER	8	CEELAA_THDLHEAP31ID	reserved
328	(148) ADDRESS	4	CEELAA_IPTLAA	reserved
332	(14C) ADDRESS	4	CEELAA_MASTERLAA	reserved
336	(150) CHARACTER	48	CEELAA_RSVD3	reserved
384	(180) CHARACTER	1	CEELAA_END(0)	end of block
1		CEELAA_CURRENT_VERSION	"1"
1		CEELAA_VERSION_1	"1"
384	(180)		CEELAA_LEN	"*-CEELAA"

Figure 133 on page 664 provides the cross reference to the LAA.

Library Control Area (LCA)

1	CROSS REFERENCE	HEX OFFSET	HEX VALUE	LEVEL
	NAME	=====	=====	=====
	CEELAA	0		1
	CEELAA_ASID	14		2
	CEELAA_COMP_31B	18		2
	CEELAA_COMP_64B	40		2
	CEELAA_COMPILER_C	18		2
	CEELAA_CURKEY31	108		2
	CEELAA_CURKEY64	128		2
	CEELAA_CURRENT_VERSION	180	1	2
	CEELAA_END	180		2
	CEELAA_ENSQ64	130		2
	CEELAA_FIRST_IN_CHAIN	C8	1	2
	CEELAA_FLAG1	C8		2
	CEELAA_FLAG2	C9		2
	CEELAA_FLAG3	CA		2
	CEELAA_FLAG4	CB		2
	CEELAA_GTAB31	20		2
	CEELAA_GTAB64	50		2
	CEELAA_HEAP	130		2
	CEELAA_IPT	C8	20	2
	CEELAA_IPTLAA	148		2
	CEELAA_JIT_RSVD1	88		2
	CEELAA_JIT_RSVD2	90		2
	CEELAA_LCA31	24		2
	CEELAA_LCA64	58		2
	CEELAA_LEACTIVE	C8	80	2
	CEELAA_LEN	180	180	2
	CEELAA_LEPENDING	C8	40	2
	CEELAA_MASTERLAA	14C		2
	CEELAA_MEMPLIMIT	C8	10	2
	CEELAA_NEXT	C		2
	CEELAA_OVERFLOW_ABEND	C8	2	2
	CEELAA_OVERFLOW_INVALID	C9	80	2
	CEELAA_PREV	8		2
	CEELAA_RESERVE_STACK_REQUEST	C8	4	2
	CEELAA_RSTK_ACTIVE	C8	8	2
	CEELAA_RSVD1	16		2
	CEELAA_RSVD3	150		2
	CEELAA_RSVD31	109		2
	CEELAA_RSVD31_1	2C		2
	CEELAA_RSVD64	129		2
	CEELAA_RSVD64_1	68		2
	CEELAA_SANC31	104		2
	CEELAA_SANC64	120		2
	CEELAA_STACKFLOOR31	18		2
	CEELAA_STACKFLOOR64	40		2
	CEELAA_STACKOVFLOW31	1C		2
	CEELAA_STACKOVFLOW64	48		2
	CEELAA_STCB	10		2
	CEELAA_STKFLR64_HI	40		2
	CEELAA_STKFLR64_LO	44		2
	CEELAA_STKOVFL64_HI	48		2
	CEELAA_STKOVFL64_LO	4C		2
	CEELAA_SVCVEC31	100		2
	CEELAA_SVCVEC64	118		2
	CEELAA_THDLHEAP31ID	140		2
	CEELAA_THDLHEAP64ID	138		2
	CEELAA_TRT31	28		2
	CEELAA_TRT64	60		2
	CEELAA_VER	4		2
	CEELAA_VERSION_1	180	1	2
	CEELAA_31BIT	100		2
	CEELAA_31BIT_CB_STG	F8		2
	CEELAA_64BIT	118		2
	CEELAA_64BIT_CB_STG	F0		2
	CEELAAEYE	0		2

Figure 133. Library anchor area (LAA) field descriptions

Language Environment library control area

The library control area (LCA), Figure 134 on page 665, is a control block that is allocated in the key of the caller when Language Environment is initialized. The LCA is pointed to by CEELAA_LCA64. For mapping information, see the

SCEEMAC(CEELCA) data set.

1 CEELCA

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	864	CEELCA	Flags - byte 2
0	(0)	STRUCTURE	864	CEELCA	LCA mapping
0	(0)	CHARACTER	856	CEELCALEN	Length used for dword filler
0	(0)	CHARACTER	4	CEELCAEYE	eyecatcher 'LCA '
4	(4)	SIGNED	4	CEELCA_VER	version
8	(8)	ADDRESS	8	CEELCA_CAA	+8 ptr to the CAA
16	(10)	ADDRESS	4	CEELCA_DIA	+16 ptr to the DIA
20	(14)	CHARACTER	4	CEELCA_RSVD1B	reserved
24	(18)	ADDRESS	4	CEELCA_LAA	addr of associated LAA
28	(1C)	ADDRESS	4	CEELCA_OSSPL0	Ptr to OS call parm list
32	(20)	ADDRESS	8	CEELCA_SAVSTACK	Saved Stack Pointer when OS_NOSTACK linkage routine is called. After the call returns, the CEELCA_SAVSTACK field must be set back to zero. When the value is not zero, condition management and signal processing use this value as the current stack pointer. Asynchronous signals are put back if the interrupt occurs outside the bounds of the routine that owns the stack frame.
40	(28)	ADDRESS	256	CEELCA_CELQINIT	ptr to CELQINIT
48	(30)	CHARACTER	256	CEELCA_TRT	Space for C TRT
304	(130)	ADDRESS	8	CEELCA_SHUNT	ptr to the shunt routine
312	(138)	CHARACTER	4	CEELCA_FDSETFD	
					Work area used by 64-bit UU version for 31-bit mode field in CAA (reserved)
316	(13C)	CHARACTER	4	*	
320	(140)	ADDRESS	8	CEELCA_RSVFLD01	Unavailable for use
328	(148)	ADDRESS	8	CEELCA_RSVFLD02	Unavailable for use
336	(150)	ADDRESS	8	CEELCA_SAVSTACK_ASYNC	When the value is not zero, CEELCA_SAVSTACK_ASYNC contains the the address of a 8-byte field provided by the application that holds the Saved Stack Pointer when the register for the stack pointer is being used for other purposes. Zero otherwise. When the field exists and is not zero, Condition Management and signal processing will use this value as the current stack pointer. Asynchronous signals will be processed even if the interrupt occurs outside the bounds of the routine that owns the stack frame.
816	(330)	ADDRESS	8	CEELCA_CELQ6TLC	ptr to CELQ6TLC
824	(338)	CHARACTER	32	CEELCA_RSVD2	(reserved)
856	(358)	CHARACTER	8	*	Dword boundary filler

Figure 134. Library control area (LCA) field descriptions

Figure 135 on page 666 provides the cross reference to the LCA.

Common Anchor Area (CAA)

1 CROSS REFERENCE		HEX	HEX	
NAME	OFFSET	VALUE	LEVEL	
=====	=====	=====	=====	=====
CEELCA	0		1	
CEELCA_CAA	8		2	
CEELCA_CELQINIT	28		2	
CEELCA_CELQ6TLC	330		2	
CEELCA_DIA	10		2	
CEELCA_FDSETFD	138		2	
CEELCA_LAA	18		2	
CEELCA_LEN	338	358	2	
CEELCA_OSSPL@	1C		2	
CEELCA_RSVD1B	14		2	
CEELCA_RSVD2	338		2	
CEELCA_RSVFLD01	140		2	
CEELCA_RSVFLD02	148		2	
CEELCA_SAVSTACK	20		2	
CEELCA_SAVSTACK_ASYNC	150		2	
CEELCA_SHUNT	130		2	
CEELCA_TRT	30		2	
CEELCA_VER	4		2	
CEELCAEYE	0		2	
CEELCALEN	0		2	

Figure 135. Library control area (LCA) field descriptions (cross reference)

Language Environment common anchor area

Each thread is represented by a common anchor area (CAA), as the code example below shows. The CAA is generated during thread initialization and deleted during thread termination. It is pointed to by CEELCA_CAA. For mapping information on the CAA, see the SCEEMAC(CEECAA) data set.

1 CEECAA		OFFSET	OFFSET					
DECIMAL	HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION			
=====	=====	=====	=====	=====	=====	=====	=====	=====
0	(0)	STRUCTURE	0	CEECAA	, CAA mapping			
0	(0)	BITSTRING	1280	CEECAA_EXTERNAL(0)	Fields external in 31-bit mode			
0	(0)	BITSTRING	688	*	Reserved for "external" fields			
688	(2B0)	BITSTRING	2	*	Padding			
690	(2B2)	BITSTRING	2	CEECAA_INVAR(0)	Field that is at same fixed offset in both 31-bit and 64-bit CAAs			
690	(2B2)	BITSTRING	1	CEECAA_INVAR_0	Byte 0			
		1... ..		CEECAA_64	"X'80'" ON/OFF = 64/31-bit CAA			
					EQU X'40' Reserved			
					EQU X'20' Reserved			
					EQU X'10' Reserved			
					EQU X'08' Reserved			
					EQU X'04' Reserved			
					EQU X'02' Reserved			
					EQU X'01' Reserved			
DECIMAL	HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION			
=====	=====	=====	=====	=====	=====	=====	=====	=====
691	(2B3)	BITSTRING	1	CEECAA_INVAR_1	Byte 1			
					EQU X'80' Reserved			
					EQU X'40' Reserved			
					EQU X'20' Reserved			
					EQU X'10' Reserved			
					EQU X'08' Reserved			
					EQU X'04' Reserved			

Common Anchor Area (CAA)

					EQU X'02' Reserved
					EQU X'01' Reserved
692	(2B4)	BITSTRING	4	*	Padding
696	(2B8)	BITSTRING	72	*	Reserved for "external fields"
768	(300)	BITSTRING	16	*	Reserved
784	(310)	BITSTRING	80		CEECAAMEMBER_AREA(0)
					CGEN, for C Member
784	(310)	ADDRESS	8	CEECAACGENE	Reserved
792	(318)	BITSTRING	24	*	Reserved
816	(330)	ADDRESS	8	CEECAACTHD	Address of CTHD
824	(338)	ADDRESS	8	*	Reserved
832	(340)	ADDRESS	8	CEECAACPCB	Address of C PCB
840	(348)	ADDRESS	8	CEECAACEDB	Address of C CEDB
848	(350)	BITSTRING	16	*	
864	(360)	BITSTRING	3	*	Reserved
867	(363)	BITSTRING	1	CEECAAF2	2nd flags byte
					EQU X'80' Reserved
					EQU X'40' Reserved
	..1.			CEECAATIP	"X'20'" Thread termination in progress
	...1			CEECA_THREAD_INITIAL	"X'10'" If on, indicates this is the IPT
 1...			CEECA_TRACE_ACTIVE	"X'08'" If on, library trace is active (TRACE runtime option was set)
1..			CEECA_ALTSTK_ACTIVE	"X'04'" If on, alt stack active
1			CEECA_USRSTK_ACTIVE	EQU X'02' Reserved "X'01'" If on, context switching user stack is active
868	(364)	BITSTRING	1	CEECAALEVEL	LE/370 level identifier
869	(365)	BITSTRING	3	*	Reserved
872	(368)	ADDRESS	8	CEECAADMC	Addr of ESPIE Devil-May-Care rtn
880	(370)	BITSTRING	8	*	Reserved
888	(378)	ADDRESS	8	CEECAAERR	Addr of the current CIB
896	(380)	DBL WORD	8	CEECA_FIRSTDSA(0)	LE64 First DSA
896	(380)	ADDRESS	8	CEECAADDSA	Addr of the dummy DSA
904	(388)	ADDRESS	8	CEECAAEDB	Address of the EDB
912	(390)	ADDRESS	8	CEECAAPCB	Address of the PCB

The following two fields are used for the validation of the CAA

```
=====
```

920	(398)	ADDRESS	8	CEECAEYEPTR	Addr of CAA eyecatcher
928	(3A0)	ADDRESS	8	CEECAAPTR	Addr of this CAA
936	(3A8)	BITSTRING	40	*	Reserved
976	(3D0)	CHARACTER	8	CEECAATHDID	Posix thread id
984	(3D8)	ADDRESS	8	CEECAARCB	A(RCB)
992	(3E0)	BITSTRING	104	*	End

Figure 136 on page 668 shows the cross reference to the CAA.

Debugger Interfaces Area (DIA)

1	CROSS REFERENCE	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====	=====
	CEECAA	0		1
	CEECAA_ALTSTK_ACTIVE	363	4	2
	CEECAA_EXTERNAL	0		2
	CEECAA_FIRSTDSA	380		2
	CEECAA_INVAR	2B2		2
	CEECAA_INVAR_0	2B2		2
	CEECAA_INVAR_1	2B3		2
	CEECAA_THREAD_INITIAL	363	10	2
	CEECAA_TRACE_ACTIVE	363	8	2
	CEECAA_USRSTK_ACTIVE	363	1	2
	CEECAA_64	2B2	80	2
	CEECAACEDB	348		2
	CEECAACGENE	310		2
	CEECAACPCB	340		2
	CEECAACTHD	330		2
	CEECAADDSA	380		2
	CEECAADMC	368		2
	CEECAAEDB	388		2
	CEECAAERR	378		2
	CEECAAIEPTR	398		2
	CEECAAFIAG2	363		2
	CEECAALEVEL	364		2
	CEECAAMEMBER_AREA	310		2
	CEECAAPCB	390		2
	CEECAAPTR	3A0		2
	CEECAARCB	3D8		2
	CEECAATHDID	3D0		2
	CEECAATIP	363	20	2

Figure 136. Common anchor area (CAA) field descriptions (cross references) AMODE 64

Language Environment debugger interfaces area

The debugger interfaces area (DIA), as shown in the code example below, is a control block that is allocated in the key of the caller when Language Environment is initialized. The DIA is pointed to by CEELCA_DIA. For mapping information on the DIA, see the SCEEMAC(CEEDIA) data set.

1	CEEDIA	OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
=====	=====	=====	=====	=====	=====	=====	=====
		0	(0)	STRUCTURE	0	CEEDIA	DIA Mapping
		0	(0)	CHARACTER	680	CEEDIALEN(0)	Length used for dword filler
		0	(0)	CHARACTER	4	CEEDIAEYE	eyecatcher 'DIA '
		4	(4)	SIGNED	4	CEEDIAVER	version
		8	(8)	CHARACTER	68	CEEDIAHOOKS(0)	
		8	(8)	CHARACTER	4	CEEDIAALLOC	Hook control words ALLOCATE descr. Built
		12	(C)	CHARACTER	4	CEEDIASTATE	New statement begins
		16	(10)	CHARACTER	4	CEEDIAENTRY	Block entry
		20	(14)	CHARACTER	4	CEEDIAEXIT	Block exit
		24	(18)	CHARACTER	4	CEEDIAMEXIT	Multiple block exit
		28	(1C)	CHARACTER	32	CEEDIAPATHS(0)	
							PATH hooks
		28	(1C)	CHARACTER	4	CEEDIALABEL	At a label constant
		32	(20)	CHARACTER	4	CEEDIACALL	Before CALL
		36	(24)	CHARACTER	4	CEEDIAACALL	After CALL
		40	(28)	CHARACTER	4	CEEDIADO	DO block starting
		44	(2C)	CHARACTER	4	CEEDIAIFTRUE	True part of IF
		48	(30)	CHARACTER	4	CEEDIAIFFALSE	False part of IF
		52	(34)	CHARACTER	4	CEEDIAWHEN	WHEN group starting
		56	(38)	CHARACTER	4	CEEDIAOTHER	OTHERWISE group
		60	(3C)	CHARACTER	4	CEEDIACGOTO	GOTO hook for C
		64	(40)	CHARACTER	4	CEEDIARSVDH1	Reserved hook

Debugger Interfaces Area (DIA)

68	(44)	CHARACTER	4	CEEDIARSDVH2	Reserved hook
72	(48)	CHARACTER	4	CEEDIAMULTEVT	Multiple Event Hook
76	(4C)	BITSTRING	4	CEEDIAMEVMASK	Multiple Event Hook Mask
80	(50)	ADDRESS	8	CEEDIAHLLEXIT	HLL Exit
88	(58)	CHARACTER	80	CEEDIADBG(0)	CodeDT CAA Debug Fields
88	(58)	ADDRESS	8	CEEDIADBGCTLA	PL/I-CodeDT Interface
96	(60)	ADDRESS	8	CEEDIADBGVIEW	Bas-View CodeDT CB
104	(68)	ADDRESS	8	CEEDIADBGGOTO	Gto_Goto_Rec CodeDT CB
112	(70)	ADDRESS	8	CEEDIADBGFMET	DT Module Fetch Struct
120	(78)	ADDRESS	8	CEEDIABOSADDR	Bas_BOSS_Control DT CB
128	(80)	CHARACTER	16	CEEDIAOHPSW	PSW for Overlay Hooks
144	(90)	ADDRESS	8	CEEDIAOHRESUME	Overlay Hooks Resume
152	(98)	CHARACTER	8	CEEDIADBGFLAG(0)	CodeDT Flags Area
152	(98)	BITSTRING	1	CEEDIADBGFLG0	CodeDT Flag Byte 0
153	(99)	BITSTRING	1	CEEDIADBGFLG1	CodeDT Flag Byte 1
154	(9A)	BITSTRING	1	CEEDIADBGFLG2	CodeDT Flag Byte 2
155	(9B)	BITSTRING	1	CEEDIADBGFLG3	CodeDT Flag Byte 3
156	(9C)	BITSTRING	1	CEEDIADBGFLG4	CodeDT Flag Byte 4
157	(9D)	CHARACTER	3	*	Reserved
160	(A0)	SIGNED	4	CEEDIADBGINVS	Recursive CodeDT Invoc.
164	(A4)	CHARACTER	4	*	Reserved

```
CEEDIAHOOK Code to Pass Control to Hook Handler: STG
R8,2096(4) LLGT R8,PSALAA-PSA(,0) USING CEELAA,R8 LG
R8,CEELAA_LCA64 DROP R8 USING CEELCA,R8 LLGT R8,CEELCA_DIA
DROP R8 USING CEEDIA,R8 STMG 0,15,CEEDIA_R0 LG R6,DIA_DIMA
LMG R5,R6,0(R6) BASR R7,R6 NOPR 0
```

```
=====
```

168	(A8)	BITSTRING	46	CEEDIAHOOK	Code to pass control
216	(D8)	ADDRESS	8	CEEDIADIMA	A(debugger entry)

```
=====
```

OFFSET	OFFSET				
DECIMAL	HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION

```
=====
```

Register Save Area - When an event hook hits, an EX statement transfers control to the CEEDIAHOOK code via a BRAS and the users registers are saved in locations CEEDIAR0 through CEEDIAR15. Because a BRAS stores the return address in R0, we can use that value as the IP portion of the PSW which is constructed for use by an RP instruction that returns control back to the program.

```
=====
```

224	(E0)	CHARACTER	136	CEEDIAHOOKSA(0)	Hooks Save Area
224	(E0)	CHARACTER	16	CEEDIAPSW(0)	PSW
224	(E0)	SIGNED	8	CEEDIATOP	PSW Modes, Mask, etc
232	(E8)	ADDRESS	8	CEEDIAR0	Register 0
240	(F0)	ADDRESS	8	CEEDIAR1	Register 1
248	(F8)	ADDRESS	8	CEEDIAR2	Register 2
256	(100)	ADDRESS	8	CEEDIAR3	Register 3
264	(108)	ADDRESS	8	CEEDIAR4	Register 4
272	(110)	ADDRESS	8	CEEDIAR5	Register 5
280	(118)	ADDRESS	8	CEEDIAR6	Register 6
288	(120)	ADDRESS	8	CEEDIAR7	Register 7
296	(128)	ADDRESS	8	CEEDIAR8	Register 8
304	(130)	ADDRESS	8	CEEDIAR9	Register 9
312	(138)	ADDRESS	8	CEEDIAR10	Register 10
320	(140)	ADDRESS	8	CEEDIAR11	Register 11
328	(148)	ADDRESS	8	CEEDIAR12	Register 12
336	(150)	ADDRESS	8	CEEDIAR13	Register 13
344	(158)	ADDRESS	8	CEEDIAR14	Register 14
352	(160)	ADDRESS	8	CEEDIAR15	Register 15

CEEHCHK5 Save Area - When the hook handler is BASRed into, it has to determine if the calling routine was a leaf before adjusting R4 to obtain stack storage. CEEHCHK5 will be called to make this determination. The following fields and associated storage will be pre allocated and made available to construct a parameter list and to provide a save area for CEEHCHK5.

Debugger Interfaces Area (DIA)

360	(168)	CHARACTER	256	CEEDIACHK5(0)	Storage for CEEHCHK5 call
360	(168)	ADDRESS	8	CEEDIACHK5CAL	Address of CEEHCHK5
368	(170)	CHARACTER	48	CEEDIACHK5PRMS(0)	Storage for CHK5 Parms
368	(170)	ADDRESS	8	CEEDIAPRM1	CHK5 Parm 1
376	(178)	ADDRESS	8	CEEDIAPRM2	CHK5 Parm 2
384	(180)	ADDRESS	8	CEEDIAPRM3	CHK5 Parm 3
392	(188)	ADDRESS	8	CEEDIAPRM4	CHK5 Parm 4
400	(190)	ADDRESS	8	CEEDIAPRM5	CHK5 Parm 5
408	(198)	ADDRESS	8	CEEDIAPRM6	CHK5 Parm 6
416	(1A0)	CHARACTER	144	CEEDIACHK5SA	Save Area for HCHK5 call
560	(230)	SIGNED	4	CEEDIACHK5SRC	HCHK5 Return Code
564	(234)	SIGNED	4	*	Padding
568	(238)	ADDRESS	8	CEEDIACHK5DSA	HCHK5 Good DSA Pointer
576	(240)	ADDRESS	8	CEEDIACHK5EP	HCHK5 EP Pointer
584	(248)	CHARACTER	32	*	Reserved for CHK5 exp
616	(268)	CHARACTER	64	*	Reserved for DIA exp.
680	(2A8)	CHARACTER	8	*	Dword boundary filler
680	(2A8)			CEEDIA_LEN	"*-CEEDIA"

The following code sample provides the cross reference to the DIA.

1	CROSS REFERENCE	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====	=====
	CEEDIA	0		1
	CEEDIA_LEN	2A8	2B0	2
	CEEDIAACALL	24		2
	CEEDIAALLOC	8		2
	CEEDIABCALL	20		2
	CEEDIABOSADDR	78		2
	CEEDIACGOTO	3C		2
	CEEDIACHK5	168		2
	CEEDIACHK5CAL	168		2
	CEEDIACHK5DSA	238		2
	CEEDIACHK5EP	240		2
	CEEDIACHK5PRMS	170		2
	CEEDIACHK5SRC	230		2
	CEEDIACHK5SA	1A0		2
	CEEDIADBG	58		2
	CEEDIADBGCTLA	58		2
	CEEDIADBGFLAG	98		2
	CEEDIADBGFLG0	98		2
	CEEDIADBGFLG1	99		2
	CEEDIADBGFLG2	9A		2
	CEEDIADBGFLG3	9B		2
	CEEDIADBGFLG4	9C		2
	CEEDIADBGGOTO	68		2
	CEEDIADBGINVS	A0		2
	CEEDIADBGMFET	70		2
	CEEDIADBGVIEW	60		2
	CEEDIADIMA	D8		2
	CEEDIADO	28		2
	CEEDIAENTRY	10		2
	CEEDIAEXIT	14		2
	CEEDIAEYE	0		2
	CEEDIAHLLEXIT	50		2
	CEEDIAHOOK	A8		2
	CEEDIAHOOKS	8		2
	CEEDIAHOOKSA	E0		2
	CEEDIAIFFALSE	30		2
	CEEDIAIFTRUE	2C		2
	CEEDIALABEL	1C		2
	CEEDIALEN	0		2
	CEEDIAMEVMASK	4C		2
	CEEDIAMEXIT	18		2
	CEEDIAMULTEVT	48		2
	CEEDIAOHPSW	80		2
	CEEDIAOHRESUME	90		2
	CEEDIAOTHER	38		2

Debugger Interfaces Area (DIA)

CEEDIAPATHS	1C	2
CEEDIAPRM1	170	2
CEEDIAPRM2	178	2
CEEDIAPRM3	180	2
CEEDIAPRM4	188	2
CEEDIAPRM5	190	2
CEEDIAPRM6	198	2
CEEDIAPSW	E0	2
CEEDIARSVDH1	40	2
CEEDIARSVDH2	44	2
CEEDIAR0	E8	2
CEEDIAR1	F0	2
CEEDIAR10	138	2
CEEDIAR11	140	2
CEEDIAR12	148	2
CEEDIAR13	150	2
CEEDIAR14	158	2
CEEDIAR15	160	2
CEEDIAR2	F8	2
CEEDIAR3	100	2
CEEDIAR4	108	2
CEEDIAR5	110	2
CEEDIAR6	118	2
CEEDIAR7	120	2
CEEDIAR8	128	2
CEEDIAR9	130	2
CEEDIASTATE	C	2
CEEDIATOP	E0	2
CEEDIAVER	4	2
CEEDIAWHEN	34	2

Language Environment enclave data block

Each enclave is represented by an enclave data block (EDB), Figure 137 on page 672, which supports the program model. All enclave-related resources are provided in the EDB. It is generated during enclave initialization and deleted during enclave termination. For mapping information on the EDB, see the SCEEMAC(CEEEDB) data set.

Enclave Data Block (EDB)

1 CEEEDB						
OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION	
0	(0)	STRUCTURE	0	CEEEDB	0D EDB mapping	
0	(0)	STRUCTURE	0	CEEEDB	0D EDB mapping	
0	(0)	BITSTRING	512	CEEEDB_EXTERNAL(0)	External part in 31-bit mode	
0	(0)	BITSTRING	8	CEEEDBEYE	Eyecatcher 'CEEEDB '	
8	(8)	CHARACTER	248	*	Reserved area	
256	(100)	SIGNED	4	CEEEDBFLAGS(0)		
256	(100)	BITSTRING	1	CEEEDBFLAG1	EDB Flags	
		1... ..		CEEEDBMAINI	"X'80'" Main program initialized	
		..1.		CEEEDBACTIV	EQU X'40' Initial amode	
		...1		CEEEDBTIP	"X'20'" Environment is active	
	1..		CEEEDB_POSIX	"X'10'" Termination In Progress	
	1.		CEEEDBMULTITHREAD	EQU X'08' Pre-Init Compat. is active	
	1		CEEEDB_OMVS_DUBBED	"X'04'" POSIX environment active	
				CEEEDB_MULTITHREAD	"X'02'" Multi-threading environment	
				CEEEDB_OMVS_DUBBED	"X'01'" OpenMVS is dubbed	
257	(101)	BITSTRING	15	*	Reserved	
272	(110)	ADDRESS	8	CEEEDBOPTCB	A(options control block)	
280	(118)	BITSTRING	232	*	Reserved End	

Figure 137. Enclave data block (EDB) field descriptions (AMODE 64)

Figure 138 provides the cross reference to the EDB.

1 CROSS REFERENCE			
NAME	HEX OFFSET	HEX VALUE	LEVEL
CEEEDB	0		1
CEEEDB	0		1
CEEEDB_EXTERNAL	0		2
CEEEDB_OMVS_DUBBED	100	1	2
CEEEDB_POSIX	100	4	2
CEEEDBACTIV	100	20	2
CEEEDBEYE	0		2
CEEEDBFLAGS	100		2
CEEEDBFLAG1	100		2
CEEEDBMAINI	100	80	2
CEEEDBMULTITHREAD	100	2	2
CEEEDBOPTCB	110		2
CEEEDBTIP	100	10	2

Figure 138. Enclave data block (EDB) field descriptions (cross reference)

Language Environment process control block

Each process is represented by a process control block (PCB); Figure 139 on page 673 shows the format. All process resources are anchored, provided for, or can be obtained through the PCB. The PCB is generated during process initialization and deleted during process termination. For mapping information on the PCB, see the SCEEMAC(CEEPCB) data set.

```

1  CEEPCB

      OFFSET  OFFSET
      DECIMAL  HEX   TYPE      LENGTH  NAME (DIM)  DESCRIPTION
      =====  =====
      0        (0)  STRUCTURE  0       CEEPCB      , PCB mapping
      0        (0)  BITSTRING  448     CEEPCB_EXTERNAL(0)
                                     External part in 31-bit mode
      0        (0)  BITSTRING  8       CEEPCBEYE   Eyecatcher 'CEEPCB '
      8        (8)  CHARACTER  248     *           Reserved for "external" fields
      256      (100) BITSTRING  3       *           Reserved
      259      (103) BITSTRING  1       CEEPCBFLAG2
                                     EQU X'80' Reserved
                                     EQU X'40' Reserved
                                     EQU X'20' Reserved
                                     EQU X'10' Reserved
      =====  =====
                                     .... 1...
                                     CEEPCB_OMVS  "X'08'" OpenMVS is up and
                                               available
                                               EQU X'04' Reserved
                                               EQU X'02' Reserved
                                               EQU X'01' Reserved
      260      (104) CHARACTER  4       *           Padding
      264      (108) ADDRESS    8       CEEPCBDBGEH A(debug event handler)
      272      (110) BITSTRING  40      *           Reserved
      312      (138) ADDRESS    8       CEEPCBRCB   Address of the RCB
      320      (140) BITSTRING  128     *           Reserved
                                               End
  
```

Figure 139. Process control block (PCB) field descriptions (AMODE 64)

Figure 140 provides the cross reference to the PCB.

```

1  CROSS REFERENCE

      NAME                                     HEX  HEX  LEVEL
      =====  =====  =====  =====
      CEEPCB                                     0                                     1
      CEEPCB_EXTERNAL                           0                                     2
      CEEPCB_OMVS                               103                                8    2
      CEEPCBDBGEH                              108                                2
      CEEPCBEYE                                 0                                     2
      CEEPCBFLAG2                              103                                2
      CEEPCBRCB                                138                                2
  
```

Figure 140. Process control block (PCB) field descriptions (cross reference)

Language Environment region control block

Regions are defined to effectively manage the resources for multiple processes, allowing, for instance, for the reuse of resources. There is one RCB per instance of a Language Environment environment and there is no link between RCB in separate Language Environment environments. For mapping information on the RCB, see the SCEEMAC(CEERCB) data set. Figure 141 on page 674 shows the format of the RCB.

Region control block (RCB)

1	CEERCB	OFFSET	OFFSET			
		DECIMAL	HEX	TYPE	LENGTH	NAME (DIM)
		0	(0)	STRUCTURE	0	CEERCB
		0	(0)	BITSTRING	128	CEERCB_EXTERNAL(0)
		0	(0)	CHARACTER	8	CEERCBEYE
		8	(8)	CHARACTER	56	*
		64	(40)	BITSTRING	16	*
		80	(50)	SIGNED	4	CEERCB_VERSION_ID(0)
		80	(50)	BITSTRING	1	CEERCBPRODID
		81	(51)	BITSTRING	1	CEERCBVERID
		82	(52)	BITSTRING	1	CEERCBRELID
		83	(53)	BITSTRING	1	CEERCBMODID
		84	(54)	CHARACTER	4	*
		88	(58)	ADDRESS	8	CEERCB_PCBCHAIN
		96	(60)	CHARACTER	32	*

Figure 141. Region control block (RCB) field descriptions

Figure 142 provides the cross reference to the RCB.

1	CROSS REFERENCE	HEX	HEX	
	NAME	OFFSET	VALUE	LEVEL
	CEERCB	0		1
	CEERCB_EXTERNAL	0		2
	CEERCB_PCBCHAIN	58		2
	CEERCB_VERSION_ID	50		2
	CEERCBEYE	0		2
	CEERCBMODID	53		2
	CEERCBPRODID	50		2
	CEERCBRELID	52		2
	CEERCBVERID	51		2

Figure 142. Region control block (RCB) field descriptions (cross reference)

Chapter 21. Compiler-writer interfaces (CWIs) supported for AMODE 64 applications

The following table lists the CWIs that Language Environment supports for AMODE 64 applications.

Table 72. CWIs for AMODE 64 applications

CWI	Function	Page
<code>__dsa_prev()</code>	Returns the address of the DSA prior to <i>dsa_p</i> on the Language Environment stack	" <code>__dsa_prev()</code> — Chain back to previous DSA" on page 725
<code>__ep_find()</code>	Returns the address of the entry point of the function owning the <i>dsa_p</i> DSA	" <code>__ep_find()</code> — returns the address of the entry point of the function owning the <i>dsa_p</i> DSA" on page 728
<code>__vhm_event()</code>	Drives an event into any vendor heap manager	" <code>__vhm_event()</code> " on page 713

Chapter 22. CALL linkage convention for AMODE 64 applications

This chapter describes the program call linkage convention supported by Language Environment for AMODE 64 applications.

Terminology

The terminology around the call or function invocation is not exactly the same in all HLLs. Figure 143 summarizes the terminology in this section.

TERMINOLOGY REFRESHER

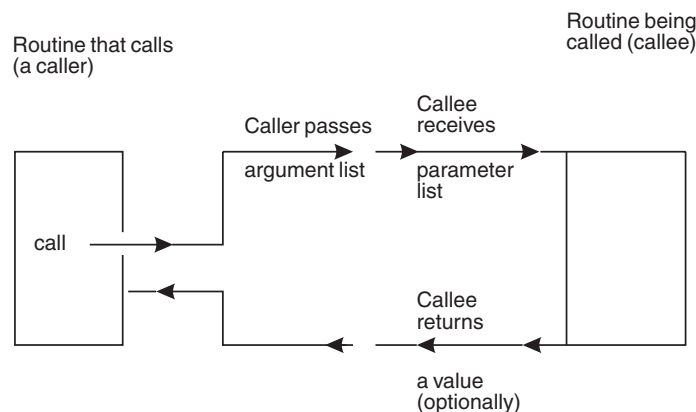


Figure 143. CALL terminology refresher

The formats of a Language Environment argument list and parameter list are identical. Rather than refer to both formats in this section, the term argument list only is used. However, everything that applies to an argument list format also applies to a parameter list format.

There are two access modes for arguments:

Direct The value of the argument is placed directly in the argument list body.

Indirect

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

Programming languages have two basic argument passing semantics:

By value

The value of the object is passed. No change made by the callee to the argument value is reflected in the calling routine.

By reference

Changes made by the callee to the argument value are reflected in the calling routine.

XPLINK CALL linkage conventions for AMODE 64 applications

The following sections describe the Language Environment XPLINK protocols for passing arguments to external routines.

The primary goal of XPLINK is to make subroutine calls as fast and efficient as possible by removing all nonessential instructions from the main path. This is achieved by introducing the following:

- growing the stack from higher to lower addresses ("negative-" or "downward-growing")
 - to eliminate overhead in stack frame allocation
 - to eliminate need for inline stack overflow check
 - to allow for an improved epilog
 - to allow addressability to information (such as parameters) in the caller's stack frame
- biasing the stack pointer (by 2048 bytes), so that small functions can save registers in their own stack frame before updating the stack pointer, avoiding address generation interlocks
- reassignment of registers to support more efficient saving and restoring of registers in function prologs and epilogs
- parameter passing in registers, accepting return values in registers
- elimination of Inter-language Call (ILC) overhead (marking of stack frame) for non-ILC calls
- faster call sequences for inter-module calls
- passing the address of the data area associated with a function, its "environment", to the function on entry
- no branching around CEL words
- use of relative branching for function calls where possible
- unification of the various (RENT and NORENT, DLL and NODLL) function pointer implementations, reducing the costs of all operations involving function pointers

An important additional goal is the reduction in size of the function in memory. This is accomplished by eliminating unused information in function control blocks.

Register usage and linkage

The following list shows the register use and linkage.

- GPR1-3 => arguments (depending upon type)
- GPR4 => the caller's stack frame in the downward-growing stack. This is biased and actually points to 2048 bytes before the real start of the stack frame.
- GPR5 => the called routine's environment pointer
- GPR6 => the entry point in the called routine if the call was made by a BASR instruction
- GPR7 => the return point in the caller's routine. The return point also contains information to determine if the call was made via BASR or branch relative.
- GPR8-15 => preserved
- FPRs => arguments (depending upon type)

Stack format

Figure 144 shows the Language Environment AMODE 64 XPLINK stack storage model.

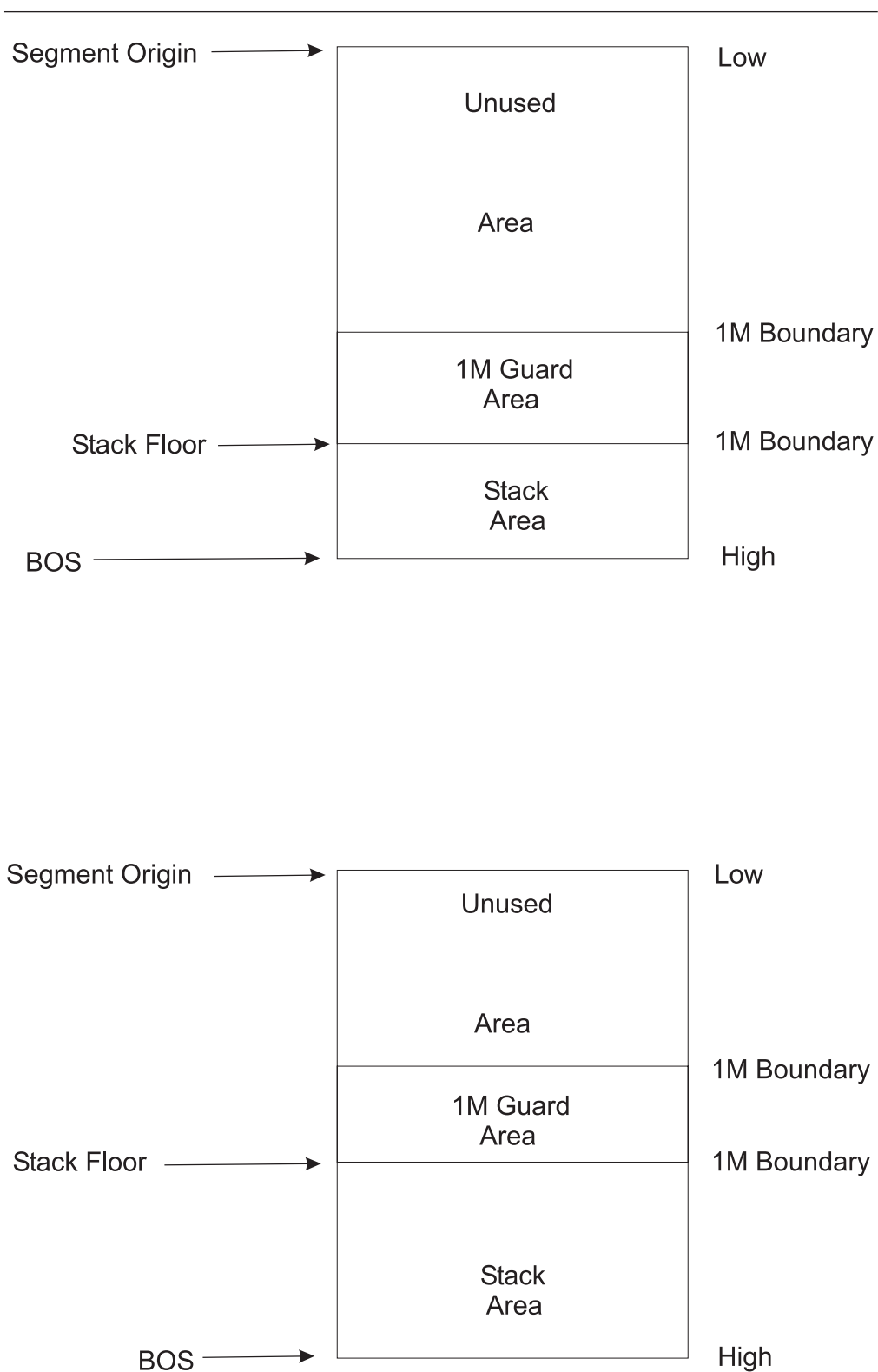


Figure 144. Language Environment AMODE 64 XPLINK stack storage model

Stack Frame Mapping

Stack frame mapping

The prolog of a function usually allocates space (referred to as a “frame”, “Stack Frame”, or “DSA”- dynamic storage area) in the Language Environment-provided stack segment for its own purposes and to support calls to other routines.

Figure 145 shows the stack frame layout. The stack register points to a location 2048 bytes before the stack frame for the currently active routine. It grows from numerically higher storage addresses to numerically lower ones, that is the stack frame for a called function is always at a lower address than the calling function. The stack frame is 32-byte-aligned.

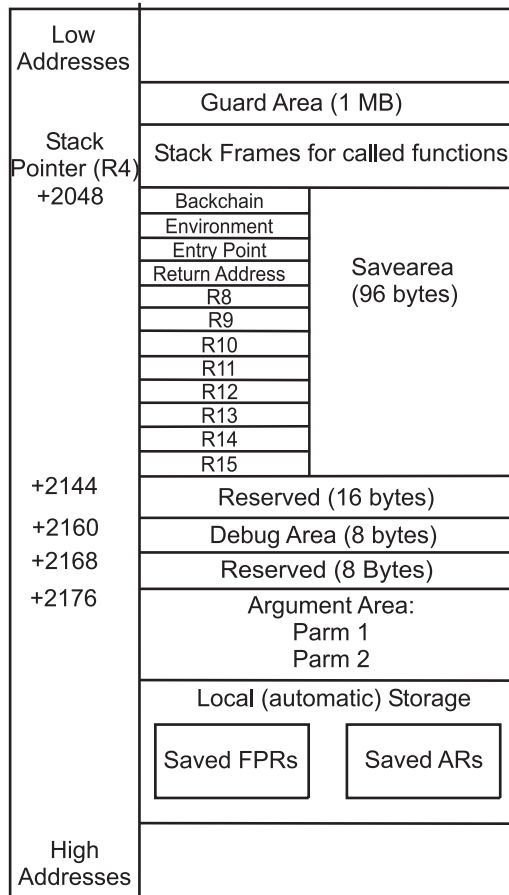


Figure 145. Language Environment XPLINK stack frame layout for AMODE 64 applications

Table 73 on page 681 describes the contents of each area within the stack frame shown in Figure 145.

Table 73. Content of XPLINK stack frame for AMODE 64 applications

AMODE 64 stack frame area	Content
Save area	<p>This area is always present when a stack frame is required. It holds up to 12 registers. The first two doublewords hold, optionally, GPRs 4 and 5, the registers containing the address of the previous stack frame and the environment address passed into the function. This is followed by the two doublewords containing GPR6, which may or may not hold the actual entry point address depending on the type of call, and GPR7, the return address. As many of the 8 non-volatile registers as are used by the called function are saved in the following 64 bytes.</p> <p>Except when registers are saved in the prolog, this area may not be altered by compiled code. The PPA1 GPR Save Mask indicates which GPRs are saved in this area by the prolog.</p> <p>Stack overflow is detected by the STMG instruction used to save registers in this save area.</p> <p>Storage of the Backchain field in the save area is triggered by the optional XPLINK(BACKCHAIN) compiler option, or at the convenience of the compiler.</p> <p>The environment address is stored when the TEST compiler option or the optional XPLINK(STOREARGS) compiler option is specified, or at the convenience of the compiler.</p> <p>The third doubleword in the save area contains the value in GPR6 on entry to the routine. If the routine was called with a BASR instruction, the address is that of the function entry point.</p> <p>The fourth doubleword contains the return address. The return point can be examined to determine how the function was called:</p> <ul style="list-style-type: none"> • If the function was called with a BASR instruction, the entry point address can be found in the third doubleword of the save area • If the function was called with a relative branch, the entry point can be computed from the return address and the branch offset contained in the relative branch instruction
Reserved	These areas are always present and are for the exclusive use of the runtime. It is uninitialized by compiled code.
Debugger area	This area is always present and is for the exclusive use of the debugger. It is uninitialized by compiled code.
Argument area	<p>This area is at the fixed DSA offset of 128 bytes into the caller's stack frame. It contains the argument lists passed on function calls made by the function associated with this stack frame. The called function finds its parameters in the caller's stack frame.</p> <p>Requirement: A minimum of four doublewords (32 bytes) must be always be allocated.</p>
Local storage	This is the space owned by the executing procedure and may be used for its local variables and temporaries.

Stack overflow

To maximize function call performance, XPLINK replaces the explicit inline check for overflow with a storage protect mechanism that detects stores past the end of the stack area.

The stack floor is the lowest usable address of the current stack area. In the lower storage addresses, it is preceded by a store-protected guard area used to detect stack overflows.

Availability of space for a stack frame is ensured in the function prolog usually by storing into the start of the called function's frame. In case of overflow, this triggers an exception which in turn causes a contiguous extension of the stack by Language

Stack Frame Mapping

Environment. Functions with a DSA larger than the guard area use the stack floor address in the LAA to verify space availability. Extensions to the stack area are transparent to the application.

Stores into the guard area done outside the prolog and done outside "alloca" built-in processing are treated as not valid and cause the application to be terminated.

Prolog/epilog examples

This section contains typical prolog and epilog code sequences for AMODE 64 XPLINK. These are examples, not definitive code sequences that must be generated by conforming compilers.

Table 74 shows the layout of a small size stack frame, where the dsasize is less than, or equal to, 2048 bytes ($dsasize \leq 2048$ bytes).

Table 74. Prolog/epilog example: small size (AMODE 64) stack frame

	DC	0D'0',XL8'00C300C500C500F1'.C.E.E.1	
	DC	A(*-8-PPA1),AL.27(dsasize/32),AL.5(flags)	
EP	STMG	6,lastused,2048-dsasize+16(4)	
	AGHI	4,-dsasize	update stack pointer
	...		
		<i>function body</i>	
	...		
	LMG	7,lastused,2048+24(4)	restore registers
	LA	4,dsasize(,4)	restore stack pointer
	B	2(,7)	return to caller

Table 75 shows the layout of an intermediate size stack frame. In this case, the dsasize is greater than 2048 bytes but less than 1 M ($2048 < dsasize \leq 1M$).

Table 75. Prolog/epilog example: intermediate size (AMODE 64) stack frame

	DC	0D'0',XL8'00C300C500C500F1'.C.E.E.1	
	DC	A(*-8-PPA1),AL.27(dsasize/32),AL.5(flags)	
EP	AGHI	4,-dsasize	update stack pointer
	STMG	6,lastused,2048+16(4)	
	...		
		<i>function body</i>	
	...		
	LMG	7,lastused,2048+24(4)	restore registers
	AGHI	4,dsasize	restore stack pointer
	B	2(,7)	return to caller

Table 76 shows a large size stack frame, where the dsasize is greater than 1 M ($dsasize > 1M$).

Table 76. Prolog/epilog example: large size (AMODE 64) stack frame

	DC	0D'0',XL8'00C300C500C500F1'.C.E.E.1	
--	----	-------------------------------------	--

Table 76. Prolog/epilog example: large size (AMODE 64) stack frame (continued)

	DC	A(*-8-PPA1),AL.27(dsasize/32),AL.5(flags)	
EP	DS	0D	
	STMG	2,3,2048+136(4)	
	LGR	0,4	Save caller's stack register
*	For this particular example, the instructions below assume dsasize = x'100140'		
	LGHI	2,H'-16'	
	SLLG	2,2,16	
	AGHI	2,H'-320'	
	AGR	4,2	update stack pointer
	LLGT	3,1208	get LAA address from PSALAA
*	Check bottom of stack		
	CG	4,CEELAA_STACKFLOOR64-CEELAA(,3)	
	JM	EXT	
STK	DS	0H	
	LGR	3,0	
	LG	3,2048+144(,3)	
	STMG	5,9,2048+8(4)	
	STG	0,2048(,4)	
	LGR	2,0	
	LG	2,2048+136(,2)	
	...		
	<i>function body</i>		
	...		
	LMG	4,lastused,2048(4)	restore registers
	B	2(,7)	return to caller
	...		
	DC	0D'0',XL8'00C300C500C500F2'.C.E.E.2	
	DC	A(this marker - entry point marker)/8	
EXT	DS	0D	
	LG	3,CEELAA_STACKOVFLOW64-CEELAA(,3)	
	BASR	3,3	call Language Environment stack extender
	NOPR	7	
	J	STK	

Table 77 shows an example with the following characteristics: XPLINK, no alloca, no storeargs, saves regs 5-9, dsasize=360256 (AMODE 64).

Table 77. Prolog/epilog example: XPLINK, no alloca, no storeargs, saves regs 5-9, DSA size=360256 (AMODE 64)

@1L0	DS	0D	
		=F'12779717'	
		=F'12910833'	
		=F'168'	
		=F'360256'	
main	DS	0D	
	STMG	r2,r3,2184(r4)	save 2nd parm + R3
	LGR	r0,r4	save original R4
	STMG	r5,r9,-358200(r4)	save caller's registers

Prolog/Epilog Examples

Table 77. Prolog/epilog example: XPLINK, no alloca, no storeargs, saves regs 5-9, DSA size=360256 (AMODE 64) (continued)

LGHI	r2,H'-5'	high 16 bits of DSA length
SLLG	r2,r2,16	move into proper spot
AGHI	r2,H'-32576'	low 16 bits of DSA length
AGR	r4,r2	R4 = new DSA address
STG	r0,2048(,r4)	save backchain in DSA
LGR	r2,r0	point to caller's DSA
LG	r2,2184(,r2)	restore 2nd parm
<i>function body</i>		
LMG	r4,r9,2048(r4)	restore caller's registers
SRG	r3,r3	R/C = 0
B	2(,r7)	return to caller

Stack extension

When the stack frame size is greater than the guard area size, the new stack pointer value must be compared to the CEELAA_STACKFLOOR64 field. When the stack pointer is less, then a stack expansion routine must be called explicitly to create the new stack increment.

DSA extension -- alloca: Sometimes a program's automatic (stack) storage requirements are not known until runtime. DSA extension allows a program to dynamically allocate additional automatic (stack) storage. (The C/C++ compiler built-in function `alloca` is the C/C++ implementation of DSA extension.) For XPLINK, allocating additional stack storage also requires moving the register save area at the beginning of the stack frame (the GPR4 value will change). This storage is automatically freed when the function in which it was acquired returns.

The following rules must be observed when handling `alloca` in XPLINK:

- The stack pointer (GPR4) must always point to a location 2048 bytes before the current function's stack frame. This may or may not be within the guard area.
- Functions that use “`alloca`” must use a different register (called the “`alloca` register”) to address their automatic storage and their parameters. This register must be set to point to automatic storage (computed from GPR4) in the prolog; it must keep this value throughout the function (until register contents are restored in the epilog).
- A function that uses “`alloca`” must acquire a stack frame and its prolog must store GPRs 4, 6 and 7 in its stack frame. Such a function cannot be considered an XPLleaf routine and may not be marked as such in the PPA.
- The argument area used to construct argument lists for called function must be addressed using the top of the stack pointer (GPR4).
- All live values from the beginning of the stack frame up to and including the entire argument area must be copied to the new start of the stack frame. This includes all saved registers. It does not include the Debug Area or the Reserved fields. If an argument list is under construction when `alloca` is called, it includes those arguments already constructed. When an external call is made to the runtime for `alloca` the generated code must ensure that any live values in the argument area are copied; the runtime is responsible for copying the entire 96-byte savearea.
- `alloca` rounds all requested storage amounts to a multiple of 32 bytes to maintain stack frame alignment.

Functions that use “alloca” require changes to their prologs and epilogs to maintain addressability to their automatic variables and parameter list. Also, as Table 78 shows, fields in the entry mask and PPA1 must correctly indicate that the routine uses a DSA extension.

Table 78. Prolog/epilog example: changes needed to maintain addressability

	DC	0D'0',XL8'00C300C500C500F1'	.C.E.E.1
	DC	A(*-8-PPA1),AL.27	(dsasize/32),AL.5(flags)
EP	STM	4,lastused,2048-dsasize(4)	
	STMG	1,Rx,2112(4)	if XPLINK(STOREARGS), TEST, or varargs
	AGHI	4,-dsasize	update stack pointer
	LA	Ry,128+argsize(,4)	set alloca register
	...		
		function body (addresses auto storage using the alloca register)	
	...		
	L	7,2048+24(,4)	restore return address
	LMG	8,lastused,2048+32	restore remaining registers
	L	4,dsasize(,4)	restore stack pointer
	BR	7	return to caller

Exceptions

The following sections describe rules and exceptions applicable to prologs and epilogs. Note that, in these rules, “pointing to stack frame” means “pointing to 2048 bytes before the stack frame”.

Rules applicable to prologs:

- The prolog must be contiguous (except for the out-of-line call to the stack extender) and less than or equal to 128 bytes in length.
- When a function requires a stack frame, it must check the stack segment for space availability in the prolog and it must save GPRs 6 and 7 in the Save Area. GPR6 must be saved by the instruction that checks for stack space availability.
- Saved GPRs must always be saved in their canonical location, which is as if an STMG 4,15,2048(4) had been executed.
- When a routine does not require a stack frame, it must maintain the contents of GPR7 (return address) and GPR6 received at entry at all times (not just during prolog execution) for exception handling purposes.
- GPRs 6 and 7 may not be changed in the prolog.
- Any instruction that is part of the window ranging from the entry point up to and including the instruction updating GPR4, may not introduce any potential exceptions other than as might be caused by an invalid GPR4.
- Except for a NOP, a prolog may not start with a Branch on Condition instruction (opcode 0x47).
- If the stack pointer (GPR4) is updated before the registers are saved GPR0 must be set to the value in GPR4 at function entry before GPR4 is updated. GPR0 is updated by Language Environment during stack extension; the updated value should be stored in the backchain field of the stack frame.
- GPR4 points to the caller's stack frame, the new stack frame, or the proposed new stack frame location (possibly in the guard area) throughout the prolog. No other value is allowed.

Exceptions

- GPRs 5-15 may not be modified in the prolog until after GPR4 is updated to point to the new stack frame.
- If an explicit check for stack overflow is not done in the prolog using the "End of Stack" field maintained by Language Environment, the first instruction that touches the new stack frame must be one of the following:
 - STMG 4,x,2048(4)
 - STMG 5,x,2056(4)
 - STMG 6,x,2064(4)
 - STMG 4,x,2048–dsasize(4)
 - STMG 5,x,2056–dsasize(4)
 - STMG 6,x,2064–dsasize(4)

Rules applicable to epilogs:

- The epilog must be contiguous and less than or equal to 128 bytes in length.
- Except for XPLleaf routines, epilog code must extract the return address from the savearea, and it must do this before updating GPR4 to point to the caller's stack frame. In XPLleaf routines, the return address must be taken from GPR7, which remains unaltered by compiled code throughout the life of the function. This allows the runtime to steal the return address for its own purposes.
- GPR4 must point to the current function's stack frame on entry to the epilog; when it's updated it must point to the caller's stack frame. No other value is allowed.
- The epilog contains no call, including alloca.
- Compiled code may not refer to its own stack frame after updating GPR4 .

XPLleaf routines: XPLleaf routines are functions that make no function calls (including alloca); do not contain try, catch, or throw statements; and do not acquire their own stack frame. **Restriction:** GPRs 4, 6 and 7 must not be altered by the routine.

Stack overflow exception: In XPLINK stack frame allocation is designed to trigger a protection exception when insufficient storage remains in the current stack area. This exception requires proper handling in the Language Environment interrupt exit.

A valid request for stack extension can be recognized by Language Environment as follows:

- The exception is caused by STMG 4,x,nnnn(4), STMG 5,x,nnnn(4), or STMG 6,x,nnnn(4).
- The target address in nnnn(4) is within the guard area.
- The exception address is within the prolog defined by the PPA1 of the function experiencing the exception.

Exception processing may need to distinguish between a request made in the function prolog and through "alloca". For example, set up and initialization of an extension may be different in the two cases (for example, copying of parameters). The prolog length field in the PPA1 is provided for this purpose.

For requests in the prolog, the required stack frame size is available in the entry point marker; for requests in alloca, it must be taken from GPR1.

When a stack overflow occurs, the caller's arguments must be made available in the newly-created stack segment.

Stack unwinding: Because XPLINK does not always provide a backchain, a new method for unwinding the stack must be followed:

1. Determine if the current instruction address is in a function prolog by following the directions in "Determining if an execution point is in a prolog."
2. If the current point of execution is in a prolog, determine whether GPR4 has been updated (the offset of the beginning of the instruction updating GPR4 is in the PPA1).
3. If GPR4 has been updated, reverse this by adding the DSA size (found in the entry point marker for the function) to GPR4. This is the address of the previous stack frame.
4. At this point, GPR4 points to a 2048 bytes before a valid stack frame (the caller's in the case on an incomplete prolog).
5. Using the current GPR4 value, locate the entry point of the function associated with the stack frame.

Locate the return address of the function in the fourth doubleword of the current stack frame at 2072(4). At the return address, find the call type to determine the instruction making the call.

- If it's a relative branch compute the target offset from the branch instruction contents and its address to determine the entry point.
 - If it's a BASR instruction, the entry point to the function is the value passed into the function in GPR6 and stored in the third doubleword of the current stack frame at 2064(4).
6. The current entry point can be used to locate the PPA1 for this function, but this is not required for stack unwinding:
 - a. Subtract 16 from the entry point address to get the address of the entry point marker.
 - b. Add the fullword at 8 bytes past this address (the PPA1 offset) to this value.
 7. "Special linkage" stack frames contain identifying markers; Language Environment architecture specifies how to use information in this stack frame to get to the previous stack frame.
 8. The entry point marker contains a flag to indicate whether `alloca` is used in the function. If it is not, the entry point marker contains the `dsasize` of the function associated with the current stack frame; add this value to the current stack frame address to get the address of the previous stack frame.
 9. If `alloca` is used in the function the previous value of GPR4 (2048 bytes before the previous stack frame) is stored at 2048(4).
 10. Continue, as required.

Determining if an execution point is in a prolog: From a point of execution, scan backwards for up to 128 bytes, looking for a doubleword-aligned marker as described in "Code markers" on page 688.

- If a marker is not found, the current point of execution is not in a prolog.
- If a marker is found but it is not an entry point marker, the current point of execution is not in a prolog.
- If a marker is found and it is an entry point marker:
 - In the entry point marker, the fullword at offset +8 contains the offset, from the marker, of the associated PPA1.

Exceptions

- The PPA1 contains the length of the prolog. If the current point of execution is not within this range (from the entry point, the doubleword following the entry point marker), the current point of execution is not in a prolog.

Finding the entry point of the current function:

1. Determine if the current point of execution is in a prolog. If it is, the entry point is at the beginning of the prolog.
2. Locate the return address of the function in the fourth doubleword of the current stack frame at 2072(4). At the return address find the call type, to determine the instruction making the call. If it's a relative branch compute the target offset from the branch instruction contents and its address to determine the entry point. If it is a BASR instruction, the entry point to the function is the value passed into the function in GPR6 and stored in the third doubleword of the current stack frame at 2064(4).

Code markers

This section describes the following sequences that identify points in code that are significant to Language Environment. Each of these is doubleword-aligned and has the same initial 7-byte sequence. Markers that could be found in the body of compiled code (types 2 and 3) contain *offset/8* of the associated entry point marker at *offset+8*.

- Entry point marker (type 1)
- Stack extension marker (type 2)
- Data marker (type 3)
- Stub marker (type 4)

Table 79 shows the format of entry point marker type 1.

Table 79. Entry point marker (type 1) AMODE64

+0	0x00	'C'	0x00	'E'	0x00	'E'	0x00	'1'
+8	offset of PPA1 from entry point marker				dsasize/32			EP flags

In an entry point marker, the fullword at offset +8 is an offset from the beginning of the Entry Point marker to the PPA1 associated with the entry point. "EP flags" is:

.	1	...		Function is an XPLleaf routine, saving registers in its own stack frame but not updating the stack pointer
.	.	1	..	Function uses alloca
0		0	0	Must be zero

The stack extension marker (type 2), shown in Table 80 on page 689, identifies stack extension code that is logically part of the function's prolog but not within the range of instructions defined to be part of the prolog by the PPA1 "(Length of prolog)/2" field.

Table 80. Stack extension marker (type 2) AMODE64

+0	0x00	'C'	0x00	'E'	0x00	'E'	0x00	'2'
+8	offset to entry point marker from this Marker/8				Reserved			

The data marker (type 3), shown in Table 81, follows any data in the code section that might be confused for a "real" marker because it contains the values in the first seven bytes of any marker style.

Table 81. Data marker (type 3) AMODE64

+0	0x00	'C'	0x00	'E'	0x00	'E'	0x00	'3'
+8	offset to entry point marker from this Marker/8				Reserved			

The stub marker (type 4), shown in Table 82, marks the beginning of runtime stubs.

Table 82. Stub marker (type 4) AMODE64

+0	0x00	'C'	0x00	'E'	0x00	'E'	0x00	'4'
----	------	-----	------	-----	------	-----	------	-----

Function calls

In XPLINK, each function has a data area associated with it, its environment, whose address is passed by a caller in GPR5. For externally visible functions, the environment must be representable by an ESD record. For internal functions, the value is the address of the stack pointer for the immediately containing lexical scope.

Callers need two pieces of information for each function they call. This information, organized in two consecutive doublewords on a doubleword boundary, is referred to as a function descriptor:

- Address of the called function's environment area
- Address of the called function's entry point

Resolution of function linkage is done at the stage in the compile–bind–execute process where enough information is available to make the proper choice with respect to performance and flexibility. In some cases, calls can be resolved at compile time. For calls outside a compilation unit the resolution is postponed to the binder for best results, and when DLLs are used, to the runtime environment.

Calling scheme: Excluding parameter handling, calls are made up of a sequence of instructions (CALL) that load the called function's Environment area address, load the called function's entry point address, and invoke the called function. Details of the generated sequences for different types of calls are described in separate sections below. Calls to routines in Dynamic Link Libraries (DLLs) are supported naturally without special compiler options.

Exceptions

With XPLINK, the function entry point address is not always passed to the called function. To allow Language Environment and other tools to find the entry point of the currently executing routine, every call site, which is located by the "return address" field of the current stack frame, contains information necessary to locate the entry points of both the calling and called functions and, if required, information about floating-point parameters passed. This is done by encoding information in a no-op instruction at the return point. The following diagrams show how this is encoded using a 2-byte NOPR instruction.

Call information field:

	CALL			
* *	NOPR	0(call type)		Shown as <i>NOPR type</i> in subsequent sequences

"Call type" is a 4-bit field describing the type of call. The call is not required to pass the function entry point address; the NOP following the call, which can be found through the return address (in GPR7), provides the information required to compute the entry point address in cases where it is not passed in register.

Call Type	
0000	Function is called with a BASR 7,6 instruction. GPR 6 contains its entry point address.
0001	Function is called with a BRAS 7,EP instruction. The called function does not have a base register on entry.
0010	Reserved
0011	Function is called with a BRASL 7,EP instruction. The called function does not have a base register on entry.
0100	Reserved
0101	Reserved
0110	Reserved
0111	Special linkage
1...	Reserved

Calls by name: The following sections describe the instructions that the compiler will generate when calls to functions are made by name.

Calling name: For all calls by name (inter-module calls and calls to imported functions), the compiler generates sequence of instructions that accomplish the following:

Calls without long relative branch:

	LMG	5,6,...	load environment and function addresses
	...		
	BASR	7,6	call the function
	NOPR	calltype	

--

Function Descriptor (space reserved by compiler):

DC	AD(environment)	address of function's environment
DC	AD(func)	address of function

Calls with long relative branch:

LMG	5,...	load environment and function addresses
...		
BRASL	7, called function	call the function, offset received by binder
NOPR	calltype	

Intra-module calls: When functions are bound within the same program object as the caller, the address constants to the function's environment and entry point are resolved directly by the binder and loader.

Calling imported functions: For calls to imported functions, the compiler will generate the same instruction sequence as for intra-module calls. The function descriptors for all calls to imported functions should be initialized by the binder as required for delayed DLL loading.

Function descriptor, unresolved:

DC	AD(function ID)	function ID
DC	AD(DLL loader)	address of function

Function Descriptor, resolved:

DC	AD(environment)	address of function's environment
DC	AD(func)	address of function

Calls by pointer: The following sections describe the instructions that the compiler will generate when calls to functions are made by a pointer.

Function pointers: A function pointer is a data type whose values range over procedure names. Variables of this type are usually used in procedure call contexts where the particular procedure to be called cannot be determined at compile time. They can also be passed as arguments of a call or used in comparison expressions.

Exceptions

Function pointers are a doubleword quantity that is the address of a function descriptor. With some exceptions, there is only one "call-by-pointer" function descriptor per entry point for calls made by function pointer. The exceptions are:

- pointers to internal (nested) functions, discussed below
- pointers to fetched functions and function pointers created by fetched function because the same function can be fetched more than once

This is different from previous linkage where more than one function descriptor - and different function pointer values - could exist for one function, each created in the WSA of the routine that takes the address of or calls the function. With a unique function pointer value, long to pointer casting works as expected when used with DLLs, providing the same result as with S/390 non-DLL and on most other platforms. Also, function pointer comparisons are significantly faster.

Language Environment creates function descriptors for functions whose address is taken in a separate dynamically acquired storage area based on information added to a module by the binder.

Calling sequence:

	LG	Rx,fp	load address of descriptor from function pointer
	...		
1	LMG	5,6,0(Rx)	load environment and function addresses
	...		
	BASR	7,6	call the function
	NOPR	calltype	

Function Descriptor, unresolved:

DC	AD(function ID)	address of function's environment
DC	AD(DLL loader)	address of function

Function Descriptor, resolved:

DC	AD(environment)	address of function's environment
DC	AD(func)	address of function

Reentrancy: Reentrant programs are structured to allow more than one user to share a single copy of a program object. Users create reentrant programs by writing code that does not modify data in the executable. This is referred to as a naturally-reentrant program. In many languages, users can also request that the compiler create reentrant programs on their behalf by allocating external data in

the writable static area; this is referred to as constructed-reentrancy. If a function refers to data in the writable static, its environment must also reside in writable static.

When a program is naturally-reentrant it may be desirable to bypass constructed reentrancy to avoid allocation and initialization of a writable static area.

Argument passing register conventions: The following tables describe the XPLINK register conventions used for passing arguments.

Register	Conventions on function entry	Volatility
	exit	
GPR 0	undefined	not preserved
GPR 1	1st doubleword of argument list or undefined	n/a
	part of return value or undefined	
GPR 2	2nd doubleword of argument list or undefined	n/a
	part of return value or undefined	
GPR 3	3rd doubleword of argument list or undefined	n/a
	part of return value or undefined	
GPR 4	Pointer to caller's stack frame - 2048	preserved
GPR 5	Address of environment	not preserved
GPR 6	undefined	not preserved
GPR 7	Return address	not preserved
GPR 8-11	Undefined	preserved
GPR 13-15	Undefined	preserved

Register	Conventions on function entry	Volatility
	exit	
FPR 0	FP parameter 1 or undefined	not preserved
	part of return value or undefined	
FPR 2	FP parameter 2 or part of FP parameter 1 in register pair 0,2 (for long double) or undefined	not preserved
	part of return value or undefined	
FPR 4	FP parameter or undefined	not preserved
	part of return value or undefined	
FPR 6	FP parameter or part of an FP parameter in register pair 4,6 (for long double) or undefined	not preserved
	part of return value or undefined	
FPR 1, 3, 5 and 7	undefined	not preserved
FPR 8-15	undefined	preserved

Argument passing: XPLINK uses a logical argument list consisting of contiguous doublewords, where some arguments are passed in registers and some in storage. The argument list is located in the caller's stack frame at a fixed offset (+2176) from the beginning of the stack frame. It provides space for all arguments, including those passed in registers. Its size is sufficient to contain all the arguments passed

Argument Passing

on any call statement from a procedure associated with the stack frame. The argument list must not include arguments that are pointers to locations in the argument list.

The rules for argument passing in registers are as follows:

- The first three doublewords of the argument area, regardless of their composition or source, are passed in GPRs 1, 2, and 3, and not in the argument area, except for:
 - Floating point values, including the real or imaginary constituents of complex types
- Except for arguments in the variable part of a vararg parameter list, up to four floating-point value arguments (the first four) are loaded into floating-point registers FPR0, FPR2, FPR4, FPR6 and not passed in the argument area, although space is set aside for these arguments in the argument area. In this fashion, up to four floating-point arguments can be passed depending on their precision (single, double, extended), provided each of these can be fully (considering the constituent parts of complex arguments separately) contained in the remaining available FPRs.

An extended precision floating point parameter (long double) is always passed in FPR0/2 or FPR4/6. If, for example, the first floating point parameter is double (passed in FPR0) and the second floating point parameter is long double FPR2 will be unused in the parameter list.

If a floating point argument occupies one of the first three doublewords in the argument area, a prototype for the function is visible, and the argument is not part of the vararg portion of a parameter list, the corresponding GPR's value is undefined on entry to the called function.

- Normally, arguments passed in registers are not stored in the argument list although a doubleword in the argument list is reserved for them.

Exception: If it is required that part of a floating point value be stored in the argument area, then the entire floating value is stored in the argument area. This situation arises in calls to unprototyped functions or in the vararg portion of a parameter list when part of the floating point parameter is in the first three doublewords of the argument area.

For calls to unprototyped functions, where the caller cannot know whether the called function contains a variable vararg portion, the argument list must be constructed to allow a call to either a vararg or non-vararg function. In this situation, floating-point arguments in the first three doublewords of the parameter list are passed in both GPRs and FPRs. Other floating point arguments passed in FPRs are also passed in the argument list.

To support vararg functions and calls to unprototyped functions, the minimum argument area length must be 32 bytes.

The compiler passes the maximum number of parameters that fit this encoding scheme so the parameters in registers match between caller and called functions. When calling a vararg routine, no argument in the variable portion of the argument is passed in a Floating Point Register. When calling unprototyped functions, floating point parameters are passed in FPRs matching this encoding scheme and are also shadowed by the caller, in GPRs or memory.

Function return values: Functions return their values according to type:

1. Integral and pointer data types ≤ 64 bits in length are widened to 64 bits and returned in GPR3.

2. Floating point types, including complex types, are returned FPR0, FPR2, FPR4 and FPR6, using as many registers as required.

Restriction: Not every language supports complex types. For the purposes of argument passing and function return values, in every language every aggregate that is (a) not a union, and (b) contains exactly two floating-point types of the same size (4, 8, or 16 bytes) is treated as a complex type.

3. Aggregates or packed decimal types 1-8 bytes in length are returned left adjusted in GPR1.
4. Aggregates or packed decimal types 9-16 bytes in length are returned left adjusted in GPRs 1 and 2.
5. Aggregates or packed decimal types 17-24 bytes in length are returned left adjusted in GPRs 1, 2, and 3.
6. Any other type is always completely returned in a buffer allocated by the caller. The address of this buffer is passed as a hidden first argument. For example, `struct {double, long double, double}` is returned entirely in a buffer, with no part of the aggregate returned in registers.
7. Functions returning a return value and a reason code will pass the return value in GPR3 and the reason code in GPR2. In this case, both the return value and the reason code must be integral types that are less than or equal to 64 bits in length; or, they may be aggregates consisting of a single integral type that are less than or equal to 64 bits in length.

Function Return Values

Chapter 23. Program initialization and termination for AMODE 64 applications

Initialization and termination establishes the state of the components of the Language Environment program model for AMODE 64 supporting C/C++ and Language Environment-conforming Assembler applications. Specifically, this section discusses the initialization and termination of a process, an enclave, and a thread.

Initialization overview

The program model describes three major constructs of a program structure. The constructs are:

Process

A collection of resources (code and data)

Enclave

A collection of program units consisting of exactly one main and zero or more subroutines

Thread

The basic unit of execution and owner of an exception handler, a stack, and the machine state

Initialization provides for the construction of the entities described in this model. Brief descriptions of process, enclave, and thread initialization follow.

Process Initialization

Process initialization sets up the framework to manage the enclave. It is during process initialization that the library anchor area (LAA) is obtained and initialized. For more information, see Chapter 20, "Common interfaces and conventions for AMODE 64 applications," on page 647.

Enclave Initialization

Enclave initialization creates the framework to manage enclave-related resources and the threads that run within the enclave.

Thread Initialization

Thread initialization consists of the acquisition of a stack and the enablement of the condition manager for the thread.

The first user routine to gain control within the enclave is the main routine. If user parameters are passed from the host system/subsystem, the user parameters are made available to the main routine. By the time the main routine receives control, the following resources are available:

- Stack storage
- Heap storage
- Condition handling
- Message services
- Math library

Termination overview

The following section covers enclave and process termination.

Enclave termination

An enclave terminates when one of the following events occurs:

- The last thread in the enclave terminates.
- The main routine in the enclave returns to its caller through an implicit or explicit return.
- An HLL construct issues a request for the termination of an enclave (for example, using C's `abort()`, `raise(SIGTERM)`, `_exit()`, or `exit()` functions).
- When a severity 2 or greater condition remains unhandled, the thread terminates. When a thread terminates due to an unhandled condition, the enclave also terminates.

When an enclave terminates, Language Environment releases resources allocated on behalf of the enclave and performs various other activities such as the following

- Language Environment exception handlers are canceled.
- All modules loaded by Language Environment are deleted.
- All storage obtained by way of Language Environment services is freed.
- All Language Environment control blocks for the enclave are freed.
- Return code and reason code are set in R15 and R0, respectively.
- The program mask and registers are restored to their values at the call to enclave initialization.
- Control is returned to the enclave creator.

Process termination

Process termination occurs after enclave termination. Process termination returns to the creator of the process. The resources allocated on behalf of the process are released. Language Environment explicitly relinquishes all resources that were obtained by Language Environment. Routines that obtain resources directly from the host system (such as opening a DCB) need to explicitly relinquish the resource, because Language Environment does not have any knowledge of its acquisition.

Putting initialization and termination together

This section contains an overview of running an application. Many details are omitted, but the overview summarizes how all of the pieces fit together.

- The operating system passes control to the application providing a save area, which is known as the O/S Save Area.
- The application does a STMG into the O/S Save Area to preserve the operating system's registers.
- The application calls CELQBST with R13 pointing to the O/S Save Area (and some other parameters as well).
- While running CELQBST, Language Environment initializes the process and the enclave.
- The main routine is called.
- If the user code completes through an HLL construct such as `exit()`, or if the main routine returns to its caller, the enclave is terminated.
- The return code and reason code are set into R15 and R0 and returned.

- Control is returned through the save area that was passed to CELQBST during Language Environment initialization. First the registers are restored from the O/S Save Area, including R14. Then control is returned using R14. In this example, control is returned to the operating system.

Member interfaces for initialization

The following section covers enclave initialization. CELQBST is the Language Environment initialization routine that establishes a Language Environment environment (the process and the enclave within the process) in which an application can run. CELQBST relies on a number of components to be bound with the application. Language Environment uses these components to describe the contents of the application, and to locate other elements contained in the application. A description of these components follows.

CELQSTRT

The CELQSTRT CSECT is a required part of each application; it identifies an application. Language Environment must be able to access CELQSTRT throughout the duration of the Language Environment environment. It must not be bound with a program object that is deleted during application execution. Language Environment provides a default version of CELQSTRT, but it can also be generated by the compiler.

CELQSTRT is the entry point for any language that provides a CELQMAIN main. Entry into CELQSTRT causes the Language Environment environment to be initialized and execution to be passed to the main routine as specified in CELQMAIN.

CELQSTRT is also used for any language that provides a CELQFMAN fetchable subroutine. However, entry into CELQSTRT for a fetchable subroutine is not allowed. Subroutines must be invoked using other methods, such as C fetch().

As the code sample below shows, CELQSTRT is logically divided into five sections; Table 83 on page 700 describes their content. It is intended that the section structure and fields currently defined in CELQSTRT will remain constant over time. It is also intended that necessary changes to CELQSTRT will be made in an upwardly compatible manner, to preserve the structure and fields as currently defined.

SECTION 1

```

CELQSTRT CSECT      ,
CELQSTRT AMODE     64
CELQSTRT RMODE     ANY
           WXTRN    CELQMAIN
           WXTRN    CELQFMAN
           WXTRN    CELQBST
           EXTRN    CELQETBL
           EXTRN    CELQLLST

```

SECTION 2

```

           NOP      0
           NOP      2
           STMG     14,12,8(13)  SAVE CALLER REGS
           BRU      AROUND        BRANCH AROUND SIGNATURE
SIGNATUR EQU      *              START OF SIGNATURE
           DC       AL2(AROUND-SIGNTUR) LEN OF SIGNATURE
           DC       X'CE'         CEL Signature
           DC       X'03'         CEL Member ID
           DC       X'03'         CEL Version number

```

Initialization

```

        DC      X'0F'      CEL Coded release number
        DC      AD(PLIST)  Point to parameter list
        DC      CL8'CEESTART' EYE-CATCHER
        DC      X'01'      1 = XPLINK Main
        DC      X'00'      Reserved
AROUND  EQU      *

SECTION 3

        BALR   3,0        ADDRESSABILITY
        USING  *,3
        LG     15,BSTRAP  Get address of CELQBST
        LTGR   15,15     See if no CELQBST
        BNZ    BALR       OK -- go call CELQBST
        ABEND  CEEABND_INIT,REASON=CEERSN_64_NOMAIN,DUMP 4093-536
        BALR   BALR       0,15      Branch to bootstrap

SECTION 4

        PLIST  DS      0D
        DC      AD(CELQMAIN) CELQMAIN CSECT ADDRESS or 0
        DC      H'-3'      VERSION MARKER
        DC      AL2(CEESTLEN) CELQSTRT parameter length
        DC      XL4'00'    Pad
        DC      AD(0)      Reserved
        DC      AD(0)      Reserved
        DC      AD(0)      Reserved
        DC      AD(SIGNATUR) A(SIGNATURE ABOVE)
        DC      AD(0)      Reserved
        DC      AD(CELQFMAN) CELQFMAN CSECT ADDRESS or 0
        DC      AD(CELQLLST) Language list address
        DC      AD(0)      Reserved
        DC      AD(CELQETBL) Table of CEL External Entries
        CEESTLEN EQU    *-PLIST    Length of parameter list

SECTION 5

        BSTRAP DC      AD(CELQBST)  Address of bootstrap routine
        DC      CL8'CELQSTRT' 64-bit-only Eyecatcher
        CEEABND_INIT EQU 4093      Abend completion code
        CEERSN_64_NOMAIN EQU 536    Abend reason code per CEEEXABCD
        C_DATA64_CATTR RMODE(ANY),ALIGN(3)
        END      CELQSTRT

```

Table 83. Contents of the CELQSTRT CSECT

Section Number	Content
Section 1	Declarations for the entry points and external routines.
Section 2	<p>Additional entry points and signature. The signature is used for identification and provides access to the parameter list found in section 4.</p> <p>mm Member identifier of the creator. The HLL compilers should set this value to their corresponding member identifier.</p> <p>vv Member-defined version level; Language Environment has no dependencies on it. This contains a version level corresponding to the CELQSTRT defined by Language Environment or the compiler.</p> <p>rr Member-defined release level; Language Environment has no dependencies on it. This contains a release level corresponding to the CELQSTRT defined by Language Environment or the compiler.</p>
Section 3	Executable code that invokes the bootstrap routine CELQBST. Control is not returned to CELQSTRT once the bootstrap routine is invoked. Minimal logic is contained within this section of CELQSTRT.

Table 83. Contents of the CELQSTRT CSECT (continued)

Section Number	Content
Section 4	<p>Parameter list that is passed to the bootstrap routine. This parameter list is also intended to remain unchanged in future releases.</p> <p>AD(CELQMAIN) Points to the CELQMAIN CSECT for the main, or 0.</p> <p>VERSION MARKER An identifying characteristic for the CELQSTRT PLIST.</p> <p>CEESTLEN Indicates the number of bytes contained within this PLIST.</p> <p>AD(SIGNATUR) Points to the CELQSTRT signature contained in section 2.</p> <p>AD(CELQFMAN) Points to the CELQFMAN CSECT that is used during fetch, or 0.</p> <p>AD(CELQETBL) Points to the Language Environment externals table. It is through the externals table that Language Environment passes program object information into initialization.</p>
Section 5	<p>Bootstrap routine addresses. This provides the routine address to the initialization bootstrap routine.</p> <p>BSTRAP Address of the bootstrap routine. The Language Environment provided version of CELQSTRT has a WXTRN to CELQBST and requires that CELQBST be INCLUDED during bind of the application.</p>

CELQMAIN

The CELQMAIN CSECT contains the address of the main routine.

RENT CELQMAIN				
+0	0x04	0x00	0x00	0x01
+4	0x00	0x00	0x00	0x00
+8	AD(main entry point)			
+10	AD(CELQINPL)			
+18	A(0), or -1 if no environment			
+1C	Q(environment), or -1 if no environment			

NORENT CELQMAIN				
+0	0x05	0x00	0x00	0x01
+4	0x00	0x00	0x00	0x00
+8	AD(main entry point)			
+10	AD(CELQINPL)			

Initialization

NORENT CELQMAIN	
+18	AD(environment), or 0 if no environment

CELQFMAN

The CELQFMAN CSECT contains the address of a fetchable subroutine.

- RENT CELQFMAN

Offset				
+0	0x04	0x00	0x00	0x01
+4	0x00	0x00	0x00	0x00
+8	AD(fetchable entry point)			
+10	A(0), or -1 if no environment			
+14	Q(environment), or -1 if no environment			

- NORENT CELQFMAN

Offset				
+0	0x05	0x00	0x00	0x01
+4	0x00	0x00	0x00	0x00
+8	AD(fetchable entry point)			
+10	AD(environment), or 0 if no environment			

CELQBST operation

The Language Environment bootstrap routine CELQBST takes the actions that are described in Table 84.

Table 84. Bootstrap behavior

Enclave Initialized?	MAIN?	FMAIN?	Comments
No	Yes	No	Initialize the enclave and execute MAIN
No	Yes	Yes	Initialize the enclave and execute MAIN
No	No	No	Abend 4093-536
No	No	Yes	Abend 4093-536
Yes	Yes	No	Abend 4093-516
Yes	Yes	Yes	Abend 4093-516
Yes	No	Yes	Abend 4093-536
Yes	No	No	Abend 4093-536

Note:

1. **Enclave Initialized** is No if CELQBST has not yet been called.
2. **MAIN** refers to the address of the main routine contained in the CELQMAIN CSECT.
3. **FMAIN** refers to the address of the main routine contained in the CELQFMAN CSECT.

CELQETBL — Language Environment externals table

The CELQETBL CSECT, shown in Figure 146, is bound with any Language Environment-conforming AMODE 64 application program. The externals table contains various external references to entities in the executable program, which allows Language Environment to locate entities if they exist in the executable program.

```

CELQETBL CSECT ,
CELQETBL AMODE 64
CELQETBL RMODE 31
        WXTRN CELQUOPT
        WXTRN IEWBLIT
ETBL_ENTRIES DC F'10'      Number of doublewords in this table
              DC F'0'      Padding
ETBL_A_1     DC AD(0)      Reserved
ETBL_A_2     DC AD(0)      Reserved
ETBL_A_CELQLLST DC F'0'
              DC V(CELQLLST) Language List
ETBL_A_CELQUOPT DC F'0'
              DC V(CELQUOPT) User declared runtime option table
ETBL_A_CELQTRM DC F'0'
              DC V(CELQTRM) Termination stub routine address
ETBL_A_6     DC AD(0)      Reserved
ETBL_A_7     DC AD(0)      Reserved
ETBL_A_IEWBLIT DC F'0'
              DC V(IEWBLIT) Loader information table
ETBL_A_9     DC AD(0)      Reserved
        END

```

Figure 146. CELQETBL CSECT format

The CELQETBL contains the following information:

- A fullword containing the number of doublewords in CELQETBL, where the first doubleword is this fullword and the fullword of 0s that follows.
- A doubleword of 0s.
- A doubleword of 0s.
- A fullword of 0s followed by the fullword address of the language list (CELQLLST). This is a vector of weak external references for the signature CSECTs. When an entry in the vector is nonzero, the corresponding HLL is present in the executable program and its language-specific initialization is performed. (This is provided by Language Environment.)
- A fullword of 0s followed by the fullword address of the user declared option table (CELQUOPT) or zero. If a zero is discovered, then user-defined runtime options are not available (for example, bound with the application).
- A fullword of 0s followed by the fullword address of the termination stub (CELQTRM) that releases the resources obtained in CELQBST. Essentially, the termination stub deletes the routine loaded by CELQBST and returns using R14 found in the save area provided on entry to CELQBST.
- A doubleword of 0s.
- A doubleword of 0s.
- A fullword of 0s followed by the fullword address of the loader information table (IEWBLIT), which is created by the Binder.
- A doubleword of 0s.

CELQLLST — Language Environment language list

Note: AMODE 64 Language Environment supports C/C++ and Language Environment-conforming assembler. No other members are available. There are currently no member event handlers. The definition of the Language List is provided for completeness.

The language list is a vector of WXTRNs of the signature CSECTs and is generated by Language Environment. Language Environment checks for the presence of a member in the application in the language list. If the member represented by a specific offset in this list is not present or requires no special initialization, its WXTRN is unresolved. If the WXTRN is resolved, then Language Environment dynamically loads the event handler routine for that member, and stores the address in the member list. Language Environment then calls the event handler, passing an event code to the event handler routine. The language list has zero through seventeen entries statically allocated in Language Environment. Language Environment uses the number of entries in the language list as a loop counter when it is necessary to loop through the language list entries. The format of the language list is shown in the following code sample.

```

CELQLLST CSECT ,      LANGUAGE ENVIRONMENT LANGUAGE LIST HEADER
CELQLLST AMODE 64
CELQLLST RMODE 31
      DC      CL4'LLHD'
      DC      AL2(CEELLIST-CELQLLST)   Length of list header
      DC      AL2(1)                   Lang Env list version number
      DC      A((LLISTEND-CEELLIST)/8) Number of list entries
      DC      F'0'                      Padding
      DC      AD(CEELLIST)              Pointer to the language list
CEELLIST DS      0D                    Lang Env language list
      WXTRN  CELQSG00
      DC      AD(CELQSG00) 00 RSVD
      WXTRN  CELQSG01
      DC      AD(CELQSG01) 01 Language Environment
      WXTRN  CELQSG02
      DC      AD(CELQSG02) 02 RSVD
      WXTRN  CELQSG03
      DC      AD(CELQSG03) 03 C/C++
      WXTRN  CELQSG04
      DC      AD(CELQSG04) 04 RSVD
      WXTRN  CELQSG05
      DC      AD(CELQSG05) 05 RSVD for COBOL
      WXTRN  CELQSG06
      DC      AD(CELQSG06) 06 RSVD for Debug Tool
      WXTRN  CELQSG07
      DC      AD(CELQSG07) 07 RSVD for Fortran
      WXTRN  CELQSG08
      DC      AD(CELQSG08) 08 RSVD (do not use)
      WXTRN  CELQSG09
      DC      AD(CELQSG09) 09 RSVD
      WXTRN  CELQSG10
      DC      AD(CELQSG10) 10 RSVD for PL/I
      WXTRN  CELQSG11
      DC      AD(CELQSG11) 11 RSVD for Enterprise PL/I
      WXTRN  CELQSG12
      DC      AD(CELQSG12) 12 RSVD (do not use)
      WXTRN  CELQSG13
      DC      AD(CELQSG13) 13 RSVD
      WXTRN  CELQSG14
      DC      AD(CELQSG14) 14 RSVD
      WXTRN  CELQSG15
      DC      AD(CELQSG15) 15 Assembler
      WXTRN  CELQSG16
      DC      AD(CELQSG16) 16 RSVD

```

```

          DC   AD(0)      Dummy entry must contain X'00'
          DS   0D
LLISTEND DC   AD(0)      Mark the end of list
          END

```

Signature CSECT

Note: AMODE 64 Language Environment supports C/C++ and Language Environment-conforming Assembler. No other members are available. There are currently no member event handlers. The definition of the Signature CSECT is provided for completeness.

Each language called by Language Environment for member-specific initialization and termination must generate a `CELQSGnn` signature CSECT. The signature CSECT denotes the presence of a member in the application. In addition, the signature CSECT provides a mechanism for the member to convey user load module information to the dynamically loaded member event handler. The *nn* value is the decimal member number for each language.

In addition, the signature CSECT can contain a list of member identifiers upon which this current member is dependent. Language Environment orders these dependencies and calls the member-specific initializations in the dependent order. Termination is performed in the reverse order. Language Environment assumes that circular dependencies do not occur.

The format of the signature CSECT is shown in Figure 147. During enclave initialization, the signature CSECT can be accessed indirectly through the initialization parameter list.

```

CELQSGnn CSECT
CELQSGnn AMODE 64
CELQSGnn RMODE 31
          DC   CL4'S0nn'    Eye catcher
          DC   AL2(CELQSGND-CELQSGnn) Length of csect
          DC   H'257'      Version id (0101)
          DC   H'0'        Number of dependent member IDs
          DC   H'0'        Offset from the start of the CSECT...
*                                     ...to the one-byte member IDs
          DC   F'0'        reserved
          DC   53AD(0)     reserved
          CELQSGND DS   0X      End of CELQSGnn
          END

```

Figure 147. Signature CSECT format

Initialization parameter list

As Figure 148 on page 707 shows, the initialization parameter list is presented in two parts. The first part only contains the following items:

- Doubleword address of a doubleword which contains the address of the entry point. For HLLs that do not have multiple entry points, the entry point is the address of the main routine.
- A fullword offset, from offset 0 of the first part of the initialization parameter list, to the second part of the initialization parameter list. The offset is treated as a signed offset.

The second part of the initialization parameter list consists of the following information:

Signature CSECT

- A byte indicating the control level. The value is X'01'.
- Two reserved bytes of 0s
- A byte of flags (currently unused).
- A reserved fullword of 0s.
- The doubleword address of a doubleword containing the address of the main entry point of the application.
- The doubleword address of the CELQSTRT CSECT.
- The doubleword address of the CELQETBL CSECT.
- A fullword containing the member identifier that created this initialization parameter list.
- A reserved fullword of 0s.
- A doubleword that is used by the member identified by the above member ID.
- A fullword containing options related to the main (currently unused).
- A reserved fullword of 0s.

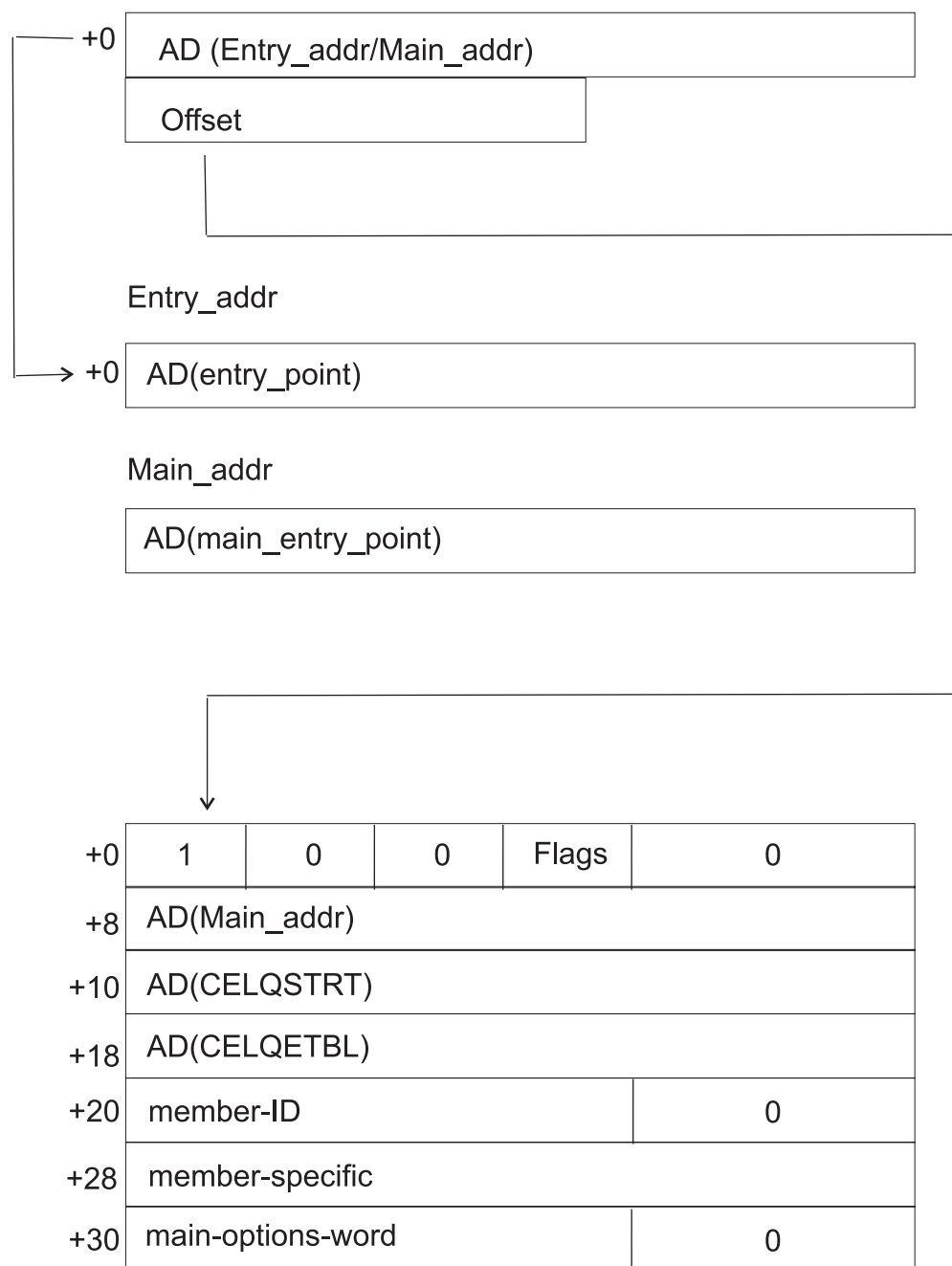


Figure 148. Format of the initialization parameter list for AMODE 64 applications

Member interfaces for termination

For normal enclave termination, the following sequence occurs:

1. Termination is requested.
2. Normal handling of the condition occurs without regard to the condition itself.
3. Call "at-termination" routines, where applicable.
4. If a debug tool is active, it is called for enclave termination.
5. Terminate the enclave.

Signature CSECT

When Language Environment cannot successfully terminate the enclave, it abends with user completion code 4094. For example, this can occur when the program has overwritten Language Environment storage, causing Language Environment control blocks to no longer be valid. The reason code associated with the ABEND U4094 indicates the cause of the failure. The reason codes are described in *z/OS Language Environment Runtime Messages*.

Language Environment transforms abends into signaled conditions, which, if unhandled, result in nonzero return and feedback codes. In the case that Language Environment finds that its operation is severely compromised, it terminates the process with a U4xxx abend. Abends treated this way have return codes in the range 4000 to 4095. Termination is immediate (using SVC 13).

CEECOPP — Runtime Option Compiler Service

Purpose

CEECOPP allows compilers to convert runtime options strings specified in a source program to an options control block (OCB). This interface also supports the runtime options that are not part of the OCB, specifically REDIR, EXECOPS, and ARGPARSE. These options are returned in the Supplementary Options Control Block (SOCB). The compiler would then create the OCB in the same format as the CELQUOPT CSECT file. This service is loadable and requires multiple calls, one to obtain the size of the working storage block (which includes the size of the OCB), and subsequent calls for the HLL to pass the runtime options string and the working storage and receive the parsed output.

CEECOPP is called by loading the executable named CEECOPP (using the LOAD SVC service), which resides in the SCEERUN data set. Then, call the entry point returned from the load using the syntax shown.

Syntax

```
void CEECOPP (function_code, storage_size, storage_addr, options, ocb_addr, socb_addr,
roet_addr, ocb_status, socb_status, rc)
INT4      *function_code;
INT4      *storage_size;
POINTER   *storage_addr;
PREFIXSTR *options;
POINTER   *ocb_addr;
POINTER   *socb_addr;
POINTER   *roet_addr;
POINTER   *ocb_status;
POINTER   *socb_status;
INT4      *rc;
```

function_code (input)

Indicates the type of request. The valid function codes and meanings are:

- 4 Obtain the size of working storage. The first call is required to communicate to the caller how much storage is required by Language Environment to parse the options, the size of the resulting OCB, and the size of the error table. It is the caller's responsibility to acquire the storage and return the address to Language Environment in the second call.

- 5 Initialize OCB and parse the supplied options. The second call is used to initialize the OCB and to parse the options and save them in the OCB.
- 6 Parse the supplied options. Subsequent calls are used to parse the options save them in the OCB created by function code 5.

storage_size (output)

The amount of storage required by Language Environment to do the parse. This size includes the amount of working storage needed to parse the string, the resulting OCB, and an error table. This is used in conjunction with *function_code* equal to 4.

storage_addr (input)

The address of storage of the length returned by Language Environment in the first call. This is used in conjunction with *function_code* 5 and 6.

options (input)

A character string containing the runtime options. This is a halfword-prefixed length string. The string is not altered and can reside in read-only storage. This is used with *function_code* 5 and 6.

ocb_addr (output)

The address of the options control block that was created with the parsed options. The compiler should convert this block into a CELQUOPT CSECT. The storage used for the OCB is obtained from the storage provided by the caller. The length of the OCB is found directly within the OCB itself. The OCB is constructed so that there are no relocatable address constants and is essentially a stream of hex information. This is used with *function_code* 5 and 6. For an example of an options control block, see Appendix A, "Options control block and supplementary options control block," on page 811.

socb_addr (output)

The address of a supplementary options control block (SOCB) that was created with the parsed options. The compiler should convert this block into a format that is suited to the caller. Language Environment does not retain this information. The storage used for the SOCB is obtained from the storage provided by the caller. The length of the SOCB is found directly within the SOCB itself. The SOCB is constructed so that there are no relocatable address constants and is essentially a stream of hex information. This is used with *function_code* 5 and 6. For an example of a supplementary options control block, see Appendix A, "Options control block and supplementary options control block," on page 811.

roet_addr (output)

The address of the runtime options error table created. The caller could convert this error table into error messages as part of the compiler output in its normal way of outputting errors. This is used with *function_code* 5 and 6. The format of the runtime options error table is shown in Figure 149 on page 711.

ocb_status (output)

A fullword integer that contains the status of output OCB. If zero, no OCB entries were made. If nonzero, OCB entries have been made.

socb_status (output)

A fullword integer that contains the status of output SOCB. If zero, no SOCB entries were made. If nonzero, SOCB entries have been made.

rc (output)

A fullword integer that contains the return code. This is used in conjunction with both *function_code* 5 and 6. The possible values are:

Signature CSECT

- 0 Options parsed with no errors, OCB entries made.
- 4 Invalid function code detected. No action performed.
- 8 Invalid function code sequence. Function code 6 (parse only) was received before function code 5 (initialize and parse).

Usage notes

- In the OCB, there are no address constants; therefore, no RLDs need to be created.
- Options string length limitation is 64K bytes.
- CEECOPP is reentrant and is marked AMODE(31)/RMODE(ANY). It is the caller's responsibility to insure the proper AMODE upon entry. CEECOPP does not switch AMODEs.
- Invocation of CEECOPP is through BALR 14,15.
- If the *OCB_status* parameter is zero, the compiler should not generate the CELQUOPT CSECT.
- If the *roet_error_count* field in the ROET is not zero, errors occurred in the parse of the options string. The errors are contained in the table.
- The *roet_error_code* field is in the format of a Language Environment condition token, which is described in Figure 150 on page 718. The message numbers associated with the feedback codes that could be found in the runtime options error table are between CEE3601I and CEE3629I. For a description of these messages, see *z/OS Language Environment Runtime Messages*.
- Figure 149 on page 711 shows the format of the runtime options error table.

Run-Time Options ErrorTable

0	Reserved
1	Error count
2	Error array entries · · ·
401	

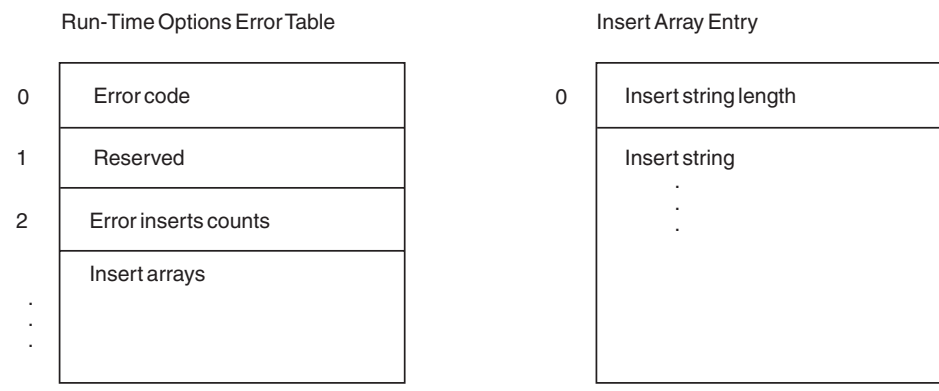


Figure 149. Runtime options error table (64-bit)

Chapter 24. Storage management for AMODE 64 applications

The Language Environment storage manager provides services that control the stack and heap storage used at run time. The initial allocation of stack and above the bar storage for heap is done during Language Environment initialization. The storage manager:

- Manages heap storage.
- Manages stack storage.
- Interfaces with host operating system to allocate/free storage
- Detects the out-of-storage condition and signals the exception handler
- Releases (or keeps track) of free heap storage segments
- Cleans-up resources at termination

Stack storage is allocated as a large memory object, guarded and based on the STACK64 runtime option (see *z/OS Language Environment Programming Reference*).

In addition to the storage manager, Language Environment provides an interface to a vendor heap manager for use with C/C++ applications.

Vendor heap manager interface for AMODE 64 applications

The vendor heap manager interface allows an external heap manager product to support C/C++ applications by an event driven interface. The following routines are supported:

- `malloc()` (C++ default operator `new` and default operator `new[]` are included)
- `calloc()`
- `realloc()`
- `free()` (C++ default operator `delete` and default operator `delete[]` are included)

The following routines are not supported:

- `__malloc31()`
- `__malloc24()`

Requirements from the vendor

A vendor, wishing to provide a replacement for functions that obtain or release storage from the user heap, needs to provide a DLL that:

- resides in either the z/OS UNIX file system or a PDSE
- runs AMODE 64
- contains the following exported function:

```
void __cee_heap_manager(int, void *);
```

The purpose of this routine is to be the communication vehicle between Language Environment and the vendor heap manager (VHM). The communication will be in the form of event codes and data areas. The prototype for the function is in the header file, `<edcwccwi.h>`.

The replacement should provide a "memory manager" that is fast (when not running in debug mode), thread-safe, and storage efficient.

Support provided for the vendor heap manager interface

The communication between Language Environment and the vendor heap manager (VHM) is through events and data structures. The C header, `<edcwccwi.h>`, contains the interfaces required to create a vendor heap manager. It is located in member EDCWCCWI of the SCEESAMP data set. To include `<edcwccwi.h>` in an application, the header file must be copied into a PDS or z/OS UNIX file system in which the C/C++ compiler will find it. This includes the C structures required as input to the VHM event calls.

The following events are supported and are defined in: `<edcwccwi.h>`

- `_VHM_INIT` - Initialization event
- `_VHM_TERM` - Termination event
- `_VHM_REPORT` (optional)

Initialization event (`_VHM_INIT`)

Initialization event (`_VHM_INIT`): This event is driven during initialization of the Language Environment enclave prior to any user code being given control. The purpose of this event is for the VHM to give Language Environment the addresses of the replacement services. Language Environment will use these routines, instead of its own, to manage the user heap. The VHM can, at this time, use `getenv()` to query any environment variables it has defined that will customize its operation.

The VHM should initialize its environment at this time, possibly allocating its own control blocks and the initial user heap segment.

The data area passed is defined as follows:

```

struct __event1_s {
    void * __ev1_free;
    void * __ev1_malloc;
    void * __ev1_realloc;
    void * __ev1_calloc;
    void * __ev1_xp_free;
    void * __ev1_xp_malloc;
    void * __ev1_xp_realloc;
    void * __ev1_xp_calloc;
    unsigned int __ev1_le_xplink : 1,
                __ev1_le_reserved : 31;
    unsigned int __ev1_vhm_xplink : 1,
                __ev1_vhm_reserved : 31;
};

```

Termination event (`_VHM_TERM`)

This optional event is driven during termination of the Language Environment enclave, after all application code has completed, but before the C library resources are terminated. There is no data area passed with this event. The purpose of this event is for the VHM to write, to `stderr`, any reports, as necessary, and then cleanup the user heap storage it has managed for the enclave.

Usage notes

- Regarding serialization, the VHM must be thread-safe. One way to detect a multi-threaded environment is to test the `CEEEDBMULTITHREAD` bit; see page Table 15 on page 66 for more information.
- The VHM should not use `malloc()`, `free()`, `calloc()` or `realloc()` from within the replacement services, to avoid potential recursive calls.

Activating the vendor heap manager

Users choose the option to use the vendor heap manager at run time. They do this by setting the `_CEE_HEAP_MANAGER` environment variable. This environment variable is set by the end-user or the application to indicate that the vendor heap manager (VHM) will be used to manage the user heap. This environment variable must be set using one of the following mechanisms:

- `ENVAR` runtime option
- inside the file specified by the `_CEE_ENVFILE` environment variable
- `export _CEE_HEAP_MANAGER`

Each of these locations is before any user code gets control, meaning prior to the static constructors, and/or `main()` getting control. Setting of this environment variable, once the user code has begun execution, will not activate the VHM, but the value of the environment variable will be updated.

`__vhm_event()`

This function drives an event into the vendor heap manager. Note that a vendor heap manager **must** be active.

Vendor heap manager interface

Syntax

```
#include <edcwccwi.h>
```

```
void __vhm_event (int event,...)
```

event

identifies the VHM event to execute. The function calls the `__cee_heap_manager()` inside the vendor heap manager function with the event as the argument. It supports the `_VHM_REPORT` event.

...

an optional argument that can be used to set special options in the event to be driven.

`__alcaxp()` — AMODE 64 DSA extension (`alloca`)

This function is invoked by C/C++ compiler generated code to extend an XPLINK downward-growing stack frame. The linkage will be normal XPLINK conventions for call-by-name. It will appear like a function that takes an integer for input and returns void. It is used by the compiler to implement the compiler built-in function `alloca()`.

Syntax

```
#include <edcwccwi.h>
```

```
void __alcaxp (long storage_size)
```

storage_size

the amount of additional stack storage being requested in bytes. This value will be rounded up to a multiple of 16 to ensure that the stack frame remains on a quadword boundary.

Usage Notes:

1. This function changes the value of the stack pointer (R4) and moves the register save area.
2. The argument area is never copied. The compiler must never assume that something placed in the argument area is still there across a call to this function.
3. The address of this function is resolved like other C-RTL functions for XPLINK (through a side deck). There is no stub for non-XPLINK.
4. If there is not sufficient room in the current stack segment, this routine handles stack expansion.
5. It is the responsibility of the caller to calculate the address of the allocated storage. The allocated storage is located immediately following the argument area. The reason for this is that the compiler, which will know the size of the argument area, can generate more efficient code to perform the calculation.
6. The Vendor Interfaces header file, `<edcwccwi.h>`, is located in member `EDCWCCWI` of the `SCEESAMP` data set. In order to include `<edcwccwi.h>` in an application, the header file must be copied into a PDS or into a directory in the z/OS UNIX file system where the C/C++ compiler will find it.

Memory object dump priority

When obtaining memory objects, Language Environment uses the IARV64 DUMPPRIORITY keyword to identify the relative priorities in which the objects are to be included in a dump. All Language Environment stack memory objects are given a priority of 5; all heap memory objects are given a priority of 15. Other AMODE 64 programs, such as Java, can allocate memory objects and assign their own dump priorities.

Memory object user tokens

Language Environment uses the IARV64 USERTKN keyword to identify all memory objects that it allocates on behalf of an AMODE 64 application. This token is used to refer to the memory objects as a set; for example, when fork() is called to create a process, or when cleaning up above the bar resources at termination. The token is a double word (8 bytes). In the high half of the double word, Language Environment places the address of the Library Anchor Area (LAA) of the Initial Process Thread (IPT). The low half of the token varies depending on the environment:

- Non-Preinit applications: the low half of the token is set to zero.
- Preinit applications: For memory objects related to base Language Environment structures and work areas, the low half of the token is set to one; for memory objects related to the current enclave, the low half of the token is set to zero.

Applications that obtain their own above the bar storage can use this user token to associate their memory objects with those of Language Environment. Depending on the actual token value used, such an association allows:

- These memory objects to be dumped along with those of Language Environment.
- These memory objects to be propagated on a fork().
- These memory objects to be cleaned up during environment termination.

Note: To use this format of user token, IARV64 requires that the caller be authorized.

When building a user token, applications can locate the address of the LAA of the IPT by first locating the address of the LAA for the current pthread, pointed to by field PSALAA in the system prefix save area (IHAPSA). Within this LAA that is mapped by macro CEELAA, field CEELAA_IPTLAA contains the address of the LAA of the IPT for the current process. When building the user token, if the code might not always be executed when a valid AMODE 64 Language Environment exists, the code must first check whether the flag CEELAA_LeActive in the LAA is on. This ensures that field CEELAA_IPTLAA is valid.

Saving the stack pointer

Language Environment provides two fields where the stack pointer can be saved:

CEELCA_SAVSTACK

The CEELCA_SAVSTACK field can be used by an application or a compiler to save the stack pointer before calling a routine using OS_NOSTACK linkage. After the call returns, the CEELCA_SAVSTACK field must be set back to zero. The value in CEELCA_SAVSTACK is used as the current stack frame when:

`__alcexp()`

1. The Language Environment ESPIE exit routine, ESTAE exit routine or signal interface routine (SIR) gets control.
2. The value in `CEELCA_SAVSTACK` is not zero.

For asynchronous signal processing, typically the interrupt PSW is outside the routine that owns the stack frame and the signal is put back.

The c macro `__LE_SAVSTACK_ADDR` defined in sample header file `edcwccwi.h` is the address of the `CEELCA_SAVSTACK` field.

`CEELCA_SAVSTACK_ASYNC`

The `CEELCA_SAVSTACK_ASYNC` field can be used by applications that have large sections of code that does not require access to the Language Environment stack but can benefit from having an additional register available. The `CEELCA_SAVSTACK_ASYNC` field is a pointer to the field where the stack pointer will be saved. Language Environment initializes `CEELCA_SAVSTACK_ASYNC` to zero. The application needs to set up the field where the stack pointer will be saved and store the address of that field in `CEELCA_SAVSTACK_ASYNC`. The storage for the field must be in the application key and persist for the life of the thread.

When initializing `CEELCA_SAVSTACK_ASYNC`, appropriate action needs to be taken if `CEELCA_SAVSTACK_ASYNC` is not zero. Because it is possible to directly access the field where the stack pointer will be stored, consider the consequences if some part of the application is doing so.

Whenever the Language Environment stack is being used, either `CEELCA_SAVSTACK_ASYNC` must be zero or the field pointed to by `CEELCA_SAVSTACK_ASYNC` must be zero.

The value in the field pointed to by `CEELCA_SAVSTACK_ASYNC` is used as the current stack frame when:

1. The Language Environment ESPIE exit routine, ESTAE exit routine, or signal interface routine (SIR) gets control.
2. `CEELCA_SAVSTACK_ASYNC` is not zero.
3. The value in the field pointed to by `CEELCA_SAVSTACK_ASYNC` is not zero

For asynchronous signal processing, the signal is always handled as if the interrupt PSW was inside the routine that owns the stack frame.

The C macro `__LE_SAVSTACK_ASYNC_ADDR`, defined in the sample header file `edcwccwi.h`, is the address of the `CEELCA_SAVSTACK_ASYNC` field.

Chapter 25. Condition representation for AMODE 64 applications

This chapter describes the format and use of condition representation within Language Environment for AMODE 64 applications.

Conditions can be defined in a number of ways. Some examples are hardware- or software-detected events (which might or might not be critical for the application to run properly), asynchronous events, or the completion of a unit of work (successfully or unsuccessfully).

Systems communicate information about conditions in a variety of ways. Return and condition codes are examples of condition information. Also, common usage is almost nonexistent in representing or communicating these conditions across IBM products or platforms. Therefore, Language Environment defines a consistent data type to represent conditions and communicate information about them to enable ILC and cross-system source code portability of applications.

The methodology presented here is required for the representation and communication of condition-related information:

- As a feedback code (return information) from some Language Environment callable services
- As input to the Language Environment condition manager
- As input to the Language Environment message services

Condition representation model

A condition in Language Environment is communicated with a 16-byte (128-bit) condition token data type. The return information (feedback code) from a Language Environment callable service is an instance of this data type.

The advantages of the condition token data type include:

- The shared data type ties together the Language Environment callable services, condition management, and message services components of Language Environment.
- A message that can be displayed or logged in a file is associated with each instance of a condition.
- As a feedback code, the data type can be stored or logged for later processing (if the message associated with the feedback code has inserts, the message must be obtained before it is saved).
- Symbolic names can be equated to defined feedback codes and hardware conditions.

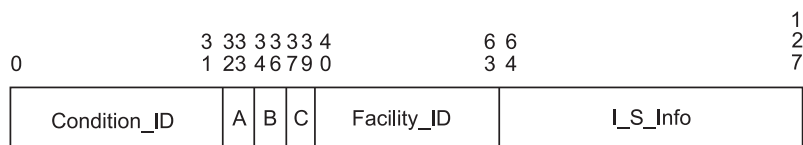
The format of the condition token data type allows four different cases, or types, of conditions to be represented. Two of the four types are **cross-system consistent**. The other two are reserved for future expansion or describe platform-specific conditions. Some Language Environment callable services use this condition token data type to return information as a feedback code.

Data objects

Language Environment condition representation data objects are defined in this section.

Condition token data type

The condition token data type communicates with message services, condition management, Language Environment callable services, and user applications. For the detailed layout of the condition token data type, see Figure 150.



A = Case
 B = Severity
 C = Control

Cases of Condition_ID are:

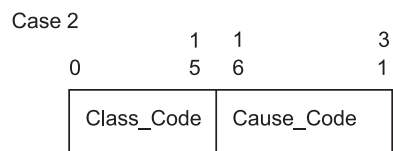
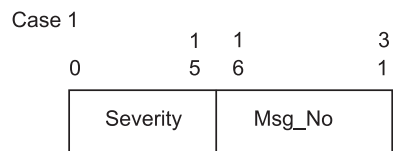


Figure 150. Language EnvironmentCondition token for AMODE 64 applications

An instance of a condition token is 16 bytes (128 bits) long, as shown in Figure 151 on page 719.


```

CEECTOK      DSECT DS 0D
CONDITION_ID DS 0F
*
* Case 1 definitions for CONDITION_ID
*
SEVERITY     DS H      Condition severity (0-4)
MSG_NUMBER   DS H      Related message number
*
* Case 2 definitions for CONDITION_ID
*
          ORG  CONDITION_ID
CLASS_CODE   DS H      Message associated with the class
CAUSE_CODE   DS H      Message associated with the cause
*
* Common part of the feedback code
*
FLAGS        DS X      Bits for Case/Severity/Control
*
* Case definitions
*
          B'xx.....'
CASE1        EQU B'01000000'
CASE2        EQU B'10000000'
*
* Severity definitions
*
          B'..xxx...'
SEV0         EQU B'00000000'  Severity 0 condition
SEV1         EQU B'00001000'  Severity 1 condition
SEV2         EQU B'00010000'  Severity 2 condition
SEV3         EQU B'00011000'  Severity 3 condition
SEV4         EQU B'00100000'  Severity 4 condition
*
* Control definitions
*
          B'.....xxx'
IBM_ASSIGN   EQU B'00000001'  IBM assigned the facility id
CTL_RSVD1    EQU B'00000010'  Reserved - must be 0
CTL_RSVD2    EQU B'00000100'  Reserved - must be 0
*
* Facility ID
*
FACILITY_ID  DS CL3      3 char string that ids the product
*
* Instance Specific Information Token
*
I_S_Info     DS D        Token to the ISI

```

Figure 151. Condition token for AMODE 64 applications

CONDITION_ID

A 4-byte identifier that describes the condition with the FACILITY_ID. The case field determines the type of identifier. Two identifiers are defined to be cross-system consistent:

1. **Case 1 - Service Condition**, which is used by all Language Environment callable services and most application programs.

SEVERITY

A 2-byte binary integer with the following possible values:

- | | |
|---|--|
| 0 | Information only (or, if the entire token is zero, no information). |
| 1 | Warning — service completed, probably correctly. |
| 2 | Error detected — correction attempted; service completed, perhaps incorrectly. |
| 3 | Severe error — service not completed. |
| 4 | Critical error — service not completed; condition signaled. |

Although the field is obviously capable of containing other values, these are not architected. If a critical error (severity = 4)

Condition Representation

occurs during a Language Environment callable service, it is always signaled to the condition manager, rather than returned synchronously to the caller.

MSG_NUMBER

A 2-byte binary number that identifies the message associated with the condition. The combination of Facility_ID and Msg_No uniquely identifies a condition.

2. **Case 2 - Class/Cause Code Condition**, which is used by some operating systems and compiler runtime libraries.

CLASS_CODE

A 2-byte, binary number that identifies the message subid associated with the **class** of the condition.

CAUSE_CODE

A 2-byte, binary number that identifies the message ID associated with the **cause** of the condition.

Note: The message subid and the message identifier are tags found in the message source file.

FACILITY_ID

A 3-character, alphanumeric string that identifies a product or component within a product. Note that special characters, including space, cannot be used.

The Facility_ID is associated with the repository (for example, a file) of the runtime messages. The conventions for naming the message repository, however, are platform-specific. The Facility_ID need not be unique within the system and can be determined by the application writer. If a unique ID is required (for IBM and non-IBM products), an ID can be obtained by contacting an IBM project office.

A Facility_ID assigned by IBM to an IBM product must begin with one of the letters A through I, inclusive. A Facility_ID assigned by IBM to a product other than an IBM's must not begin with a letter A through I. For information on how to indicate if the Facility_ID has been assigned by IBM, see Control below. There are no constraints (other than the alphanumeric requirement) on a Facility_ID not assigned by IBM.

Language Environment constructs a load name consisting of the form **T | | Facility_ID | | MSGT:**

T The character 'T' if the Facility_ID was assigned by IBM, or the character 'U' if the Facility_ID was **not** assigned by IBM.

Facility_ID

The three character facility ID as described above.

MSGT

The four characters MSGT.

For example, given an IBM assigned facility ID of CEE, the constructed load name would be ICEEMSGT.

Note: The Msg_No/Facility_ID identifies a condition for a Language Environment-enabled product. This identification is required to be persistent beyond the scope of a single session. This allows the meaning of the condition and its associated message to be determined after the session

that produced the condition has ended. The message inserts and the I_S_Info need to be explicitly saved to allow persistence after the session has concluded.

Case A 2-bit field that defines the format of the Condition_ID portion of the token. The value 1 identifies a case 1 condition, the value 2 identifies a case 2 condition. The values 0 and 3 are reserved.

Severity

A 3-bit field indicating a condition's severity. Severity values are the same as defined under a case 1 Condition_ID. When evaluating the severity, the same rules apply for signaling case 2 conditions as for case 1 conditions. For a case 1 condition, this field contains the same value as the Severity field in the Condition_ID.

Note: This field is valid for both case 1 and 2 conditions. It can be used with either condition token to evaluate the condition's severity.

Control

A 3-bit field containing flags describing or controlling various aspects of condition handling, as follows:

- ..1 Indicates Facility_ID has been assigned by IBM.
- .1 Reserved.
- 1.. Reserved.

I_S_INFO

A doubleword containing a token that identifies the Instance Specific Information (ISI) associated with the given condition. If an ISI is not associated with a given condition token, the ISI field contains binary zero. The ISI token provides access to various instance specific information such as message inserts and qualifying data.

Feedback code

A feedback code is an instance of a condition token. A feedback code is returned from a Language Environment callable service if the caller has passed a reference to an area to hold it. To test a feedback code for equivalence, the first eight bytes should be compared because they are static. The last eight bytes can change from instance to instance.

Chapter 26. National language support and message services for AMODE 64 applications

This chapter describes Language Environment National Language Support (NLS) and message handling services for AMODE 64 applications.

National language support

Language Environment provides services to support many NLS machine readable information (MRI) requirements, such as: message formatting, message delivery, casing, folding, and normalization. Language Environment formats messages for any national language known to it. Language Environment provides runtime messages for the following national languages:

- **ENU** (Mixed-case English USA)
 - Message text is made up of SBCS characters and consists of both uppercase and lowercase letters.
 - Message inserts can contain DBCS characters.
 - Long messages are split at an SBCS blank if possible or split by the output line length if a blank separator does not exist.
- **UEN** (Uppercase English USA)

This is identical to the mixed-case USA English language except the message text consists of uppercase letters. Message inserts can be in lowercase or might use lowercase codepoints to make use of SBCS Katakana capabilities.
- **JPN** (Japanese)

This language supports devices that have both DBCS and SBCS capabilities; its characteristics are:

 - Message text can be made interchangeably of SBCS and DBCS characters.
 - If a long message extends beyond the print line and the text is SBCS, it is split at a blank when possible. If a blank separator does not exist, text is split by the output line length. If the text is DBCS, the message is split at a DBCS blank if possible. If a blank separator does not exist, it is split at the last DBCS character that allows a shift-in to be inserted. The next line begins with a shift-out character.

The national language can be set using the NATLANG runtime option. One current language is maintained at the enclave level and remains in effect until it is changed. If the message text is not available for the current national language setting, the IBM-supplied default is used instead.

Language Environment message services

Language Environment provides message services to format and deliver runtime messages. The following C functions are extensions to the C runtime library:

`__le_msg_write()`
writes a message string to stderr.

`__le_msg_get_and_write()`
takes a message associated with a condition and writes it to stderr.

National language support

`__le_msg_get()`

retrieves, formats, and stores message data for a condition.

`__le_msg_add_insert()`

creates a message insert.

`__le_condition_token_build()`

builds a 16-byte condition token for use in retrieving messages from a Language Environment message repository.

For more information about the functions, see *z/OS XL C/C++ Runtime Library Reference*.

C/C++-specific vendor interfaces

For information on the C/C++-specific vendor interfaces, see “C/C++-specific vendor interfaces” on page 244.

Chapter 27. Condition management for AMODE 64 applications

This section describes what constitutes a condition in Language Environment, how Language Environment supplements existing HLL condition handling methods, and how the Language Environment condition handling model works. It describes in detail the steps involved in condition handling under Language Environment, HLL-specific condition handling considerations, Language Environment — POSIX signal handling interactions, and how you can communicate events that happen in a routine to another routine.

For a discussion of Language Environment condition handling models in the POSIX(ON) and POSIX(OFF) environments, see *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*.

Application programming interfaces (APIs)

The APIs provided by Language Environment for condition management for AMODE 64 applications are `__dsa_prev()`, `__ep_find()` and `__far_jump()`.

`__dsa_prev()` — Chain back to previous DSA Purpose

The `__dsa_prev()` function returns the address of the DSA prior to `dsa_p` on the Language Environment stack. Two types of backchaining request are supported -- logical and physical. The `req_type` parameter is used to select either logical or physical backchaining. For physical backchaining, the address of the DSA immediately prior to `dsa_p` is always returned. That DSA can be a transition or overflow DSA, or the DSA of a normal routine. For logical backchaining, `__dsa_prev()` keeps looking backward on the Language Environment stack until a normal DSA is found, skipping over any transition or overflow DSAs.

If the dummy Language Environment DSA is reached while backchaining, a NULL pointer is returned, and `errno` is set to **ESRCH**.

`__dsa_prev()` can be used when the Language Environment stack of interest is not in the current address space. To access storage outside the current address space, the user must provide the `callback_p` parameter. `callback_p` is a pointer to a user-written function that fetches all required data for `__dsa_prev()`. Generally, the `(*callback_p)()` function would obtain the data using some application-dependent method (like BPX1PTR) and move it into the current address space, where `__dsa_prev()` can access it directly. If the Language Environment stack of interest is in the same address space and is directly accessible to `__dsa_prev()`, `callback_p` can be NULL.

Syntax

```
#include <edcwccwi.h>

void __dsa_prev(const void * dsa_p, int req_type, int dsa_fmt, void * (*callback_p)(void
*data_p, size_t data_l), const void *caa_p, int *prev_fmt, void **ph_callee_dsa_p, int
*ph_callee_dsa_fmt);
```

*const void *dsa_p*

Pointer to the current DSA. `__dsa_prev()` returns a pointer to the DSA logically or physically previous to *dsa_p*, depending on the value of the *req_type* parameter. *dsa_p* may point to a DSA in another address space or in some other place not directly accessible by `__dsa_prev()`. If this address is not directly accessible, the *callback_p* parameter must be non-NULL. The callback function will be used to access *dsa_p* indirectly.

int req_type

Controls if transition DSAs are returned. The allowed values for *req_type* are:

`__EDCWCCWI_PHYSICAL`

Physical backchaining causes `__dsa_prev()` to return the address of the DSA immediately prior to *dsa_p*. The returned DSA can be either a transition or normal DSA.

`__EDCWCCWI_LOGICAL`

Logical backchaining causes `__dsa_prev()` to skip over any transition DSAs that it finds while backchaining, and not pass them back. The address of the most recent normal DSA previous to *dsa_p* is returned. Doing logical backchaining is the same as doing physical backchaining one or more times, stopping when a normal DSA is found.

int dsa_fmt

The format of the DSA pointed to by *dsa_p*. The allowed value for *dsa_fmt* is:

`__EDCWCCWI_DOWN`

This value indicates that *dsa_p* points to a 64-bit DSA.

*void * (*callback_p)()*

Pointer to a user-provided function that fetches data not normally accessible by `__dsa_prev()`. If *callback_p* is NULL, `__dsa_prev()` accesses *dsa_p* and any other required Language Environment data areas directly in the current address space. The Language Environment stack and all other data needed for backchaining must be directly accessible to `__dsa_prev()` in this case.

The user-provided *(*callback_p)()* function is passed the address and length of data to access. It must fetch the data in some application-dependent manner, and make the data available in the current address space in a place accessible to `__dsa_prev()`. *(*callback_p)()* must return a pointer to the copied data. This data must remain available to `__dsa_prev()` until the next call to *(*callback_p)()*, or until `__dsa_prev()` returns to its caller, whichever happens first. On subsequent calls, *(*callback_p)()* is allowed to reuse the same data passback area.

There is no provision for *(*callback_p)()* to pass back an error return code, indicating that the requested data could not be obtained. If *(*callback_p)()* cannot return the requested data, it must not return to `__dsa_prev()`. When an error occurs, *(*callback_p)()* may:

- `longjmp()` back to some error return point in the user code that called `__dsa_prev()`.

- ABEND or otherwise terminate abnormally.
- `exit()`, `pthread_exit()`, etc.
- Raise a caught signal where the catcher does `longjmp()` so as not to return to `__dsa_prev()`.
- Use Language Environment condition management to bypass `__dsa_prev()` after the error and resume in user code.
- Recover in some other way that does not involve returning to `__dsa_prev()`.

`__dsa_prev()` calls `(*callback_p)()` with two parameters:

*void *data_p*

Pointer to the start of the required data. This address might not be in the current address space.

size_t data_l

The number of bytes of data required. *data_l* will never exceed 16 bytes. If `(*callback_p)()` cannot pass back the complete data requested, it must not return to `__dsa_prev()`.

*const void *caa_p*

Pointer to the Language Environment CAA for the thread owning the *dsa_p* DSA. This parameter must be non-NULL whenever *callback_p* is non-NULL, and it may point to a CAA in some other address space. If *callback_p* is NULL, *caa_p* may also be NULL. If *caa_p* is NULL, the current CAA (of the thread where `__dsa_prev()` is running) is used. In this case, it is assumed that *dsa_p* points to a DSA on the Language Environment stack for the caller's thread.

*int *prev_fmt*

Pointer to an optional passback area where `__dsa_prev()` will return the DSA format of the prior DSA. The possible values passed back in this field are the same as the values for *dsa_fmt*.

If *prev_fmt* is NULL, the DSA format for the previous DSA is not passed back. If `__dsa_prev()` cannot find the previous DSA and returns a NULL value, the field pointed to by *prev_fmt* is not altered.

*void **ph_callee_dsa_p*

Pointer to an optional passback area where `__dsa_prev()` will return the address of the DSA of the physical callee. The physical callee is the function called by the function owning the returned DSA. The physical callee can be a Language Environment overflow or stack expansion routine, or it can be a normal user or Language Environment function. If physical backchaining is requested, *ph_callee_dsa_p* will be the same as *dsa_p* after `__dsa_fmt()` returns.

If *ph_callee_dsa_p* is NULL, the address of the physical callee DSA is not passed back.

If `__dsa_prev()` cannot find the previous DSA and returns a NULL value, the field pointed to by *ph_callee_dsa_p* is not altered.

*int *ph_callee_dsa_fmt*

ph_callee_dsa_fmt is a pointer to an optional passback area where `__dsa_prev()` will return the DSA format of the physical callee's DSA. The possible values passed back in this field are the same as the values for *dsa_fmt*.

If *ph_callee_dsa_fmt* is NULL, the format of the physical callee DSA is not passed back. If `__dsa_prev()` cannot find the previous DSA and returns a NULL value, the field pointed to by *ph_callee_dsa_fmt* is not altered.

Return values

- If successful, `__dsa_prev()` returns the address of the previous DSA. In addition, if `errno` is zero when `__dsa_prev()` is called, one of the following `errno` values may be set to pass back additional information:

EACCES

Indicates that the returned DSA pointer is for the Language Environment dummy DSA (pointed to by the CAA `ceecaaddsa` field). This is not an error, and all returned or passed-back information is valid.

EALREADY

Indicates that the input DSA pointer (*dsa_p*) is for the Language Environment dummy DSA (pointed to by the CAA `CEECAADDSA` field). This is not an error, and all returned or passed-back information is valid.

- If unsuccessful, `__dsa_prev()` returns a NULL pointer, and sets `errno` to one of the following values:

ESRCH

Indicates that there was no DSA previous to *dsa_p* that could satisfy the physical or logical backchaining request. This error also occurs if *dsa_p* is NULL when `__dsa_prev()` is called.

EINVAL

This error can occur if:

- *caa_p* was NULL and *callback_p* was not NULL.
- *req_type* was not `__EDCWCCWI_PHYSICAL` or `__EDCWCCWI_LOGICAL`.
- *dsa_fmt* was not `__EDCWCCWI_DOWN`.

Usage notes

- If the return code from `__dsa_prev()` is NULL, the listed `errno` values are set even if `errno` was non-zero when `__dsa_prev()` was called. When the return code from `__dsa_prev()` is not NULL, `errno` is not changed if it was not zero when `__dsa_prev()` was called.
- `__dsa_prev()` may cause program checks if it accesses invalid addresses. This is especially likely to happen if *callback_p* is NULL and the Language Environment stack being looked at is corrupted. For this reason, the caller should consider having a signal catcher set up to handle SIGSEGV with appropriate error recovery.
- The Vendor Interfaces header file, `<edcwccwi.h>`, is located in member `EDCWCCWI` of the `SCEESAMP` data set. In order to include `<edcwccwi.h>` in an application, the header file must be copied into a PDS or into a directory in the z/OS UNIX file system where the C/C++ compiler will find it.

`__ep_find()` — returns the address of the entry point of the function owning the *dsa_p* DSA

Purpose

The `__ep_find()` function returns the address of the entry point of the function owning the *dsa_p* DSA. `__ep_find()` can be used when the passed-in DSA is not in the current address space. To access storage outside the current address space, the user must provide the *callback_p* parameter, which is a pointer to a user-written function that fetches all data required by `__ep_find()`. Generally, the `(*callback_p)()` function would obtain the data using some application-dependent method (like

BPX1PTR) and move it into the current address space, where `__ep_find()` can access it directly. If the passed-in DSA is in the same address space and is directly accessible to `__ep_find()`, `callback_p` can be NULL.

Syntax

```
#include <edcwcwi.h>
```

```
void *__ep_find (const void *dsa_p, int dsa_fmt, void (*callback_p)(void *data_p, size_t data_l))
```

*const void * dsa_p*

Pointer to the DSA. *dsa_p* may point to a DSA in another address space or in some other place not directly accessible by `__ep_find()`. If this address is not directly accessible, the *callback_p* parameter must be non-NULL. The callback function will be used to access *dsa_p* indirectly.

int dsa_fmt

The format of the DSA pointed to by *dsa_p*. The allowed values for *dsa_fmt* are:

`__EDCWCWI_UP`

This value indicates that *dsa_p* points to a non-XPLINK DSA.

`__EDCWCWI_DOWN`

This value indicates that *dsa_p* points to an XPLINK DSA.

*void * (*callback_p) ()*

Pointer to a user-provided function that fetches data not normally accessible by `__ep_find()`. If *callback_p* is NULL, `__ep_find()` accesses *dsa_p* and any other required Language Environment data areas directly in the current address space. All required data must be directly accessible to `__ep_find()` in this case. The user-provided *(*callback_p)()* function is passed the address and length of data to access. It must fetch the data in some application-dependent manner, and make the data available in the current address space in a place accessible to `__ep_find()`. *(*callback_p)()* must return a pointer to the copied data. This data must remain available to `__ep_find()` until the next call to *(*callback_p)()*, or until `__ep_find()` returns to its caller, whichever happens first. On subsequent calls, *(*callback_p)()* is allowed to reuse the same data passback area. There is no provision for *(*callback_p)()* to pass back an error return code, indicating that the requested data could not be obtained. If *(*callback_p)()* cannot return the requested data, it must not return to `__ep_find()`. When an error occurs, *(*callback_p)()* may:

- `longjmp()` back to some error return point in the user code that called `__ep_find()`
- `abend` or otherwise terminate abnormally
- `exit()`, `pthread_exit()`
- Raise a caught signal where the catcher does `longjmp()` so as not to return to `__ep_find()`
- Use Language Environment condition management to bypass `__ep_find()` after the error and resume in user code
- Recover in some other way that does not involve returning to `__ep_find()`.

`__ep_find()` calls *(*callback_p)()* with two parameters:

__ep_find()

*void * data_p*

Pointer to the start of the required data. This address might not be in the current address space.

size_t data_l

The number of bytes of data required. *data_l* will never exceed 16 bytes. If (**callback_p*()) cannot pass back the complete data requested, it must not return to `__ep_find()`.

Return values

- If successful, `__ep_find()` returns the entry point address of the function owning the *dsa_p* DSA.
- If unsuccessful, `__ep_find()` returns a NULL pointer, and sets `errno` to one of the following values:

ESRCH

This error indicates that the entry point could not be located for the passed-in DSA. This error also occurs if *dsa_p* is NULL when `__ep_find()` is called.

EINVAL

This error occurs if *dsa_fmt* is not `__EDCWCCWI_UP` or `__EDCWCCWI_DOWN`.

Usage notes

- `__ep_find()` may cause program checks if it accesses invalid addresses. This is especially likely to happen if *callback_p* is NULL and the DSA being looked at is not valid. For this reason, the caller should consider having a signal catcher set up to handle SIGSEGV with appropriate error recovery.
- The Vendor Interfaces header file, `<edcwccwi.h>`, is located in member EDCWCCWI of the SCEESAMP data set. To include `<edcwccwi.h>` in an application, the header file must be copied into a PDS or into a directory in the UNIX file system where the z/OS XL C/C++ compiler will find it.

__far_jump() — Perform far jump

Purpose

The `__far_jump()` interface performs a function similar to `longjmp()`. However, it does not require a `setjmp()` to be performed previously. The information required to perform this "nonlocal goto" is provided by the user in the `__jumpinfo` structure. This information includes registers and signal mask. The target address of the jump is not supplied separately. It is supplied as two of the register values in the GPR set in the `__jumpinfo` structure, register 4 for the target DSA address and register 7 for the target code address.

Syntax

```
#include <edcwccwi.h>

void __far_jump (struct __jumpinfo * JumpInfo);
```

*struct __jumpinfo * JumpInfo*

The `__jumpinfo` structure must be cleared before it is filled in to ensure that all reserved areas are zero. The `__jumpinfo` structure appears in the following format:

```

{
  char __ji_u1[68];
  char __ji_mask_saved;
  char __ji_u2[3];
  sigset_t __ji_sigmask;
  char __ji_u3[11];
  unsigned __ji_fl_fp4 :1;
  unsigned __ji_fl_fp16 :1;
  unsigned __ji_fl_fpc :1;
  unsigned __ji_fl_res1a :1;
  unsigned __ji_fl_res1b :1;
  unsigned __ji_fl_res2 :1;
  unsigned __ji_fl_exp :1;
  unsigned __ji_fl_res2a :1;
  char __ji_u4[36];
  long __ji_gr[16];
  int __ji_u5[16];
  double __ji_fpr[16];
  int __ji_fpc;
  char __ji_u6[60];
} __jumpinfo_t;

```

long __ji_gr[16]

Contains the values of the 16 general purpose registers. The value for Register 7 is used as the target address of the jump. The value for Register 4 is used as the target DSA address.

double __ji_fpr[16]

Contains the values of the floating-point registers as indicated by the `__ji_fl_fp4` and `__ji_fl_fp16` flags. When `__ji_fl_fp16` is one, it contains all 16 floating-point registers. When `__ji_fl_fp16` is zero and `__ji_fl_fp4` is one, it contains only floating-point registers 0, 2, 4, and 6 in fields `__ji_fpr[0]`, `__ji_fpr[2]`, `__ji_fpr[4]`, and `__ji_fpr[6]`. When `__ji_fl_fp16` is zero and `__ji_fl_fp4` is zero, it contains no floating-point registers.

char __ji_mask_saved

Set to non-zero value when the signal mask field (`__ji_sigmask`) is valid.

sigset_t __ji_sigmask

Contains the signal mask value when `__ji_mask_saved` is a nonzero value.

int __ji_fpc

Contains the floating point control register value when `__ji_fl_fpc` is set to one.

unsigned __ji_fl_fp4:1

Set to one when values for only floating point registers 0, 2, 4, and 6 are provided in `__ji_fpr`.

unsigned __ji_fl_fp16:1

Set to one when values for all 16 floating point registers are provided in `__ji_fpr`.

unsigned __ji_fl_fpc:1

Set to one when value for the floating point control register is provided in `__ji_fpc`.

unsigned __ji_fl_exp:1

Set to one when explicit backchaining is complete to the target DSA.

Return values

The `__far_jump()` function has no returned value. When `__far_jump()` completes, program execution continues at the target address.

Usage notes

- The library does not attempt to verify the contents of the `__jumpinfo` structure. Incorrect data can lead to unpredictable results.
- The caller of `__far_jump()` can optionally supply a signal mask suitable to the target of the jump.
- The caller of `__far_jump()` provides the GPR & FPR sets needed for the target of the `__far_jump()`. The GPR set is always complete. The FPR set is 0, 4, or 16 registers, as indicated by the `__ji_fl_fp4` and `__ji_fl_fp16` fields.
- The Vendor Interfaces header file, `<edcwccwi.h>`, is located in member EDCWCCWI of the SCEESAMP data set. In order to include `<edcwccwi.h>` in an application, the header file must be copied into a PDS or a directory in which the C/C++ compiler will find it.

Language Environment shunt routine for AMODE 64 applications

Along with application interfaces, Language Environment provides a shunt routine for condition management of AMODE 64 applications. A shunt is a low-level error handling routine intended for use by language library routines and debug tools. A shunt is typically used when a segment of code needs to protect itself from a likely error. An incorrect address while following a control block chain is an example of an error that activates a shunt routine.

A shunt is usually established for short periods of time while the library routines or debug tools are providing services to the application. Language Environment establishes an ESPIE error recovery routine for program interrupts and an ESTAE recovery routine for abends. These recovery routines check for and setup for retry to a shunt, as appropriate. Shunt routines do not return to the Language Environment condition manager. There is no return code from the shunt routine.

Establishing a program interrupt shunt service

A program interrupt shunt routine is established by setting its address in the CAA (CEECAADMC). When the shunt address gains control, the AMODE is the AMODE at the time of the program interrupt. Setting an address in the CEECAADMC effectively cancels the previously established shunt routine, if any. Only one shunt routine can be in effect at a time. Language Environment does not provide any facility for stacking the shunt addresses. A save is not needed prior to establishing your own shunt routine.

The shunt routine is removed by removing its address from the CEECAADMC. A value of zero should be assigned to CEECAADMC as soon as possible. A shunt routine should be removed as soon as it is not needed. Information about the error is provided to the shunt routine through the CEECAAPRGCK field in the CAA, which is set to the value of the program interrupt code.

Usage Notes:

1. R0 through R15 have the same value when the shunt routine gains control as they did when the program check occurred.
2. The shunt routine cannot assume that the range of the base registers used at the time that the program check occurred extends to the shunt routine. The shunt routine might need to re-establish addressability upon entry.
3. The CEECAADMC field should be cleared as soon as it is no longer needed.

4. A shunt routine should never span a call statement. A shunt routine that gains control with another program's registers will usually fail on the first branch attempt. The routine that is called does not have to save the address of your shunt routine.
5. The Language Environment condition manager clears the CEECAADMC field when the program interrupt shunt routine is called.

Other Language Environment condition manager topics

For information about Language Environment default condition handling, see *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*. For information about Language Environment runtime options, see *z/OS Language Environment Programming Reference*.

Language Environment condition information block

Each condition is represented by a Condition Information Block (CIB). The CIB is built by the condition manager and is used as an information repository for data required by the condition handling facilities. The CIB is not intended to be altered by the user. The complete CIB is listed in the *z/OS Language Environment Debugging Guide*.

Errors during condition handling

Every effort should be made to ensure that further exceptions do not occur during the condition handling process. However, errors may still occur. To identify the state (or point in time) of the Language Environment condition manager, a state setting is contained in the CIB. The valid states, constant values, and actions taken by the Language Environment condition manager are listed in Table 85.

Table 85. CEECIB state variable, constant values, and associated actions for AMODE 64 applications

State Value	Value	Variable Meaning	Condition Manager Actions with Nested Condition
cib_state_enable	1	The language-specific enablement handler is in control. This is set by the Language Environment condition manager.	Terminate the enclave via abend 4087-C.
cib_state_eh	2	A user condition handler, registered via <code>_set_exception_handler()</code> , is in control. This is set by the Language Environment condition manager.	Terminate the enclave via abend 4087-2.
cib_state_memb	3	A language-specific exception handler is in control. This is set by the Language Environment condition manager.	Terminate the enclave via abend 4087-3.
cib_state_SF0	4	A language-specific exception handler is in control for stack frame zero. This is set by the Language Environment condition manager.	Terminate the enclave via abend 4087-4.
cib_state_evnt	5	A language-specific exception handler is in control for incidental service. This is set by the Language Environment condition manager.	Terminate the enclave via abend 4087-5.
cib_state_ipat	6	The debug tool is in control. This is set by the Language Environment condition manager.	Call the debug tool event handler indicating this event, then terminate the enclave via abend 4087-6.

Condition Management

Table 85. CEECIB state variable, constant values, and associated actions for AMODE 64 applications (continued)

State Value	Value	Variable Meaning	Condition Manager Actions with Nested Condition
cib_state_msg	7	Language Environment message services are being called by the Language Environment condition manager; this is set by the Language Environment condition manager.	Terminate the enclave via abend 4087-7.
cib_state_dump	8	Used when traceback or dump services are being called.	Terminate the enclave via abend 4087-8.
cib_state_Memb_AR_MODE	9	Used for member processing when recursion is allowed.	While in this state, the Language Environment condition manager tolerates the occurrence of a nested condition.
cib_state_ab_term_exit	10	Used when an abnormal termination exit is called; the cib_state_ab_term_exit variable contains the name of the exit.	Terminate the enclave via abend 4087-A.
cib_state_recursion	100	A language-specific user handler is in control. This value is set by the language-specific exception handler. While in this state, the Language Environment condition manager tolerates the occurrence of a nested condition. This is set by subordinate condition handlers and debug tools when calling user code.	Tolerate nested conditions.

Language Environment-issued abends

Language Environment issues abends for some fatal errors. For these errors, the Language Environment condition manager terminates the process without the subordinate exception handlers being called.

Language Environment issues user abends with codes of 4000 and above. When Language Environment issues an abend, the normal condition processing does not occur. Language Environment percolates the abend if the abend drives Language Environment's ESTAE exit. User abends of 4000 and above that are not issued by Language Environment are **not** percolated.

The products running under Language Environment should be aware that abend codes that are 4000 through 4095 are reserved for Language Environment use. These abend codes are used by Language Environment and possibly the members to signify that the environment is no longer usable.

In general, other abend codes are intercepted by the Language Environment condition manager. These produce messages and possibly dumps. The philosophy of the Language Environment exception manager is to provide diagnostic messages and not abend.

Chapter 28. Debugging and performance analysis for AMODE 64 applications

Language Environment provides interfaces upon which a debug tool, such as Debug Tool, can be built. The interfaces defined by Language Environment to a debug tool fall into the following classes: callable service, event handlers, and data areas. These interfaces, and the actions Language Environment takes on the behalf of a debug tool, are described in the following sections.

Language Environment also provides interfaces upon which a performance analysis tool, which is often called a profiler, can be built. This support is described in “Performance analysis support” on page 361. Much of this support is similar to the support Language Environment provides for debugging tools. Therefore, a debugging tool and a profiler cannot be used at the same time.

Language Environment-provided functions for the debug tool

__le_debug_set_resume_mch() — set resume machine state

The `__le_debug_set_resume_mch()` function allows the debug event handler to modify the machine state that will be used to resume after the debug event handler returns a result code of resume (110). (This only applies to event codes for which result code is a parameter.)

A recommended approach for using this function is to start with the current resume machine state. This can be obtained from the CIB. Changes then can be made to the registers, PSW, or other components in your local copy of the machine state. Later, if the debug event handler returns a result code of resume, the information from the updated machine state is used to resume the application program.

Syntax

```
#include <__le_api.h>
```

```
void __le_debug_set_resume_mch (__mch_t *position, _FEEDBACK *fc)
```

position (input)

A pointer to a valid machine state to which the resume cursor is be moved.

fc (output/optional)

A pointer to a 16-byte Feedback Code where the results of this function will be stored. Feedback codes returned include:

CEE000	Severity	0
	Msg_No	N/A
	Message	The service completed successfully.
CEE07V	Severity	2
	Msg_No	0255
	Message	position parameter is not a machine state.

__le_debug_set_resume_mch()

Usage notes

- When an interrupt has occurred in a routine that has saved the stack pointer in the CEELCA_SAVSTACK field or in the field pointed to by the CEELCA_SAVSTACK_ASYNC field, the resume cursor is initially set up so that the stack pointer is restored to that field if the application is resumed. However, if the resume cursor is moved, the stack pointer is not restored to that field unless certain fields in the machine state are set.
- To restore the stack pointer to the CEELCA_SAVSTACK field, the flags INT_SF_VALID and SAVSTACK must be set to 1 and the field INT_SF must contain the stack pointer.
- To restore the stack pointer to the field pointed to by the CEELCA_SAVSTACK_ASYNC field, the flags INT_SF_VALID and SAVSTACK_ASYNC must be set to 1 and the field INT_SF must contain the stack pointer.
- Only the stack pointer that was saved at the time of the interrupt can be restored and only be restored to the field where it was saved.

__setHookEvents() — specify execute hook events for target process

The `__setHookEvents()` function sets the execute hook events state for all threads owned by the target enclave and referenced using `asfTargetThreadRef` as specified by the `eventsMask` parameter. Callback functions let you provide address space free access to storage in the target process.

Restriction: Because C and C++ linkage conventions are incompatible, `__setHookEvents()` cannot receive a C++ function pointer as one of the callback routine function pointers. If you attempt to pass a C++ function pointer to `__SetHookEvents()`, the compiler will flag it as an error. You can pass a C or C++ function to `__SetHookEvents()` by declaring it as `extern "C"`.

Syntax

```
#include <__ledebug.h>
```

```
int __setHookEvents (int eventsMask, const asfCallback Functions,  
*asfCallbacks,  
                    const asfTargetRef *asfTargetThreadRef,  
                    const threadSpec  
*reservedForFutureUse);
```

eventsMask

Used as a bit mask to specify which types of instruction hook events to enable and which events to disable. For each bit in `eventsMask` that is set to 1, the corresponding instruction hook event is enabled. For each bit that is set to 0, the corresponding instruction hook event is disabled. Bits that do not correspond to instruction hook events are reserved and must be set to 0. The following macros define the bit values corresponding to the instruction events:

```
THOOK_LABEL
THOOK_STATEMENT
THOOK_ACALL
THOOK_DO
THOOK_IFTRUE
THOOK_IFFALSE
THOOK_WHEN
THOOK_OTHER
THOOK_POST
THOOK_BCALL
THOOK_GOTO
THOOK_EXIT
THOOK_MEXIT
THOOK_MULTIEVT
THOOK_ALLOC
THOOK_ENTRY
```

const asfCallbackFunctions *asfCallbacks

Specifies the callback functions for copying data between the controlling process and the target process. If the controlling and target processes are the same or if they are running in the same address space, asfCallbacks can be a null pointer. The addresses of the callback functions are specified by the following structure type:

```
typedef struct {
    /******
    /* callback function copies data to controlling */
    /* process buffer from target process memory */
    /******
    asfCallbackResult (*asfGetStoreCallback)(
        void *localDest,
        const asfTargetRef *targetSrce,
        size_t *dataLength);

    /******
    /* callback function copies data to target process */
    /* memory from controlling process buffer */
    /******
    asfCallbackResult (*asfSetStoreCallback)(
        const asfTargetRef *targetDest,
        const void *localSrce,
        size_t *dataLength);
} asfCallbackFunctions;
```

- *asfGetStoreCallback* is a pointer to a function that copies the amount of data specified by **dataLength* bytes from the target process memory specified by *targetSrce* to *localDest*. *localDest* must point to a buffer with a capacity of at least **dataLength* bytes. On return, **dataLength* is set to the number of bytes actually copied into *localDest*. If any of the requested target process data cannot be copied, all bytes starting from the target process address specified by *targetSrce* up to the first non-copyable byte are copied to *localDest*. **dataLength* is set to the number of bytes copied, and (**asfGetStoreCallback*()) returns the appropriate error value. If all the requests are copied successfully, **dataLength* is unchanged and (**asfGetStoreCallback*()) returns *asfResultOK* .
- *asfSetStoreCallback* is a pointer to a function that copies **dataLength* bytes of data from *localSrce* to the target process memory specified by *targetDest*. On return, **dataLength* is set to the number of bytes that could have been copied into *targetDest*. If any of the requested target process data cannot be updated, none of the target process' memory is changed, **dataLength* is set to the difference between the target process address specified by *targetDest* and

__setHookEvents()

the next lowest non-updatable target process address, and *(*asfSetStoreCallback)()* returns the appropriate error value. If all of the target process memory was updated successfully, **dataLength* is unchanged and *(*asfSetStoreCallback)()* returns *asfResultOK*.

The two callback functions must return an appropriate value to the caller. They must not *exit()*, *longjmp()*, execute a PL/I ON clause or C++ throw statement, or transfer control to any routine that bypasses returning to the caller. The type of a target process memory reference is defined as follows:

```
typedef struct {
    int asid;           /* target address space identifier */
    void *addr;        /* memory address within target address
                       * space */
} asfTargetRef;
```

- *asid* contains the identifier of the address space that contains the referenced target process memory.
- *addr* is the virtual address of the target process memory within the specified address space.

The return type of the address space free callback functions is defined as follows:

```
typedef enum {
    asfResultOK,
    asfResultAddressSpaceNotAvailable,
    asfResultPageNotMapped,
    asfResultPageNotAvailable,
    asfResultPageNotAccessible
} asfCallbackResult;
```

- *asfResultOK* specifies that the callback function returned successfully. Memory in the controlling process or target process is updated as requested.

The remaining values indicate an error in locating or accessing the target process memory. If one of the following values is returned, no memory in the target process is updated. If data is being copied from the target process to the controlling process, the largest contiguous length of memory is copied, starting from the specified target process address:

- *asfResultAddressSpaceNotAvailable*: the *asid* member of the target process memory reference is not valid, or the address space to which it refers is not available to the controlling process.
- *asfResultPageNotMapped*: the target process address space is available to the controlling process, but the specified virtual address is not mapped within that address space.
- *asfResultPageNotAvailable*: the target process address space is available and the virtual address is mapped, but the data contained in that page is not available to the controlling process. For example, the target process memory is paged out and the target process is suspended, or the target process memory is contained in a dump that does not include the requested memory location.
- *asfResultPageNotAccessible*: the target process address space is available, the virtual address is mapped and available, but the controlling process does not have access to the storage because of key, page or segment protection.

const asfTargetRef *asfTargetThreadRef

Specifies the address space identifier and virtual address of the target Language Environment environment anchor associated with a particular target thread in the target enclave. For AMODE 31 applications, this is the address of the CAA, which is loaded into register R12 while the thread is running. For AMODE 64 applications, it is the address of the LAA, stored in the prefix page at PSALAA while the thread is running. If the calling thread is also the target thread, *asfTargetThreadRef* can be a null pointer. If *asfCallbacks* is a null pointer, the *asid* member of **asfTargetThreadRef* is ignored. If *asfCallbacks* is not a null pointer, *asfTargetThreadRef* and *asfTargetThreadRef->addr* must also not be a null pointers.

const threadSpec *reservedForFutureUse

Specifies a null pointer. It is included to simplify future specifications of particular threads, rather than all threads in the target enclave.

Returned value

If successful, *__setHookEvents()* returns 0.

If an error occurs, the execute hook event state of the target process is unchanged and a negative value is returned:

- If any parameter is not valid, -1 is returned.
- If the target process runtime environment does not support instruction hook events, -2 is returned.

Usage notes

- **Restriction:** Because C and C++ linkage conventions are incompatible, *__setHookEvents()* cannot receive a C++ function pointer as one of the callback routine function pointers. If you attempt to pass a C++ function pointer to *__setHookEvents()*, the compiler flags it as an error. You can pass a C or C++ function to *__setHookEvents()* by declaring it as `extern 'C'`.
- The bit value macros can be bit-wise ORed to calculate the *eventsMask* value.

Debug tool-provided event handlers

One of the most important things a debug tool must do to be called by Language Environment is provide an event handler to handle debug events. The address of this event handler is maintained by Language Environment in the PCB field, CEEPCBDBGEH. When Language Environment initializes, this field is initialized to zero; when Language Environment loads the debug event handler, it sets this field to the address of the debug event handler.

Debug tool event handler

The debug event handler is a DLL with an exported function called one CELQVDBG. The default name of the DLL is also CELQVDBG. The `__CEE_DEBUG_FILENAME64` environment variable can be used to specify a different DLL name. Language Environment checks for the environment variable. If the variable exists, Language Environment uses the value specified as the name of the debug event handler DLL and loads it.

You can specify the debug tool to be used at run time by exposing its name to the system for Language Environment to LOAD. A load failure indicates to Language Environment that a debug tool is not available while this program is running. The debug event handler is loaded and initialized when any one of the following occur:

Debug Interfaces

- An initial command string or PROMPT is discovered and the TEST runtime option is in effect.
- The error condition is raised for the first time and the TEST runtime option is in effect with the ERROR suboption specified.
- Any condition is raised for the first time and the TEST runtime option is in effect with the ALL suboption specified.
- A call to `__ctestc` is made, regardless of the TEST runtime option setting.

Language Environment notifies the debugger of events by calling the `CELQVDBG` function. The event handler interface is defined in Table 86 and the bit map descriptions are in Table 87 on page 742.

Table 86. Debugger Language Environment event handler interface for AMODE 64 applications

Debug Tool Event	Debug Tool Event Code	Parm 2	Parm 3	Parm 4
Condition raised	101	CIB	result code	
Unhandled condition	103	CIB	result code	
User handler next	105	CIB	1 2	function pointer for user handler function pointer for member handler
Goto	111	DSA	DSA format	
PIPI Sub Initialization	115			
PIPI Sub Termination	116			
Enclave init	118	creator's EDB		
Enclave term	119			
Thread init	120	creator's CAA		
Debug tool term	121			
Thread term	122			
External entry	123	<ul style="list-style-type: none"> • Parm 2 = DSA (see note) • Parm 3 = cmd string • Parm 4 = INPL • Parm 5 = DSA format 		
Module load	124	DSA	module descriptor	DSA format
Module delete	125	DSA	module name	DSA format
Storage free	126	storage	storage length	
Condition promote	127	CIB	result code	
Condition goto	128	DSA	DSA format	
Debug tool program check	130	result code		
Message redirect	131	msg_text	ddname	
CALL CEETEST	132	DSA (see note 1)	cmd string	DSA format

Table 86. Debugger Language Environment event handler interface for AMODE 64 applications (continued)

Debug Tool Event	Debug Tool Event Code	Parm 2	Parm 3	Parm 4
Execute Hook invocation	133	<ul style="list-style-type: none"> • Parm 2 = DSA • Parm 3 = hook offset • Parm 4 = DSA format • Parm 5 = A buffer containing general purpose registers • Parm 6 = Return address to the routine that was interrupted • Parm 7 = Entry point to the routine that was interrupted 		
mutex_init	140	initializing thread_id	mutex	(for bit mask descriptions, see Table 54 on page 355)
mutex_destroy	141	destroying thread_id	mutex	
mutex_lock	142	owner thread_id	mutex	
mutex_unlock	143	thread_id releasing mutex	mutex	
mutex_wait	144	waiting thread_id	mutex	
mutex_unwait	145	posted thread_id	mutex	
mutex_relock	146	owner thread_id	mutex	
mutex_unrelock	147	owner thread_id	mutex	
cond_init	150	initializing thread_id	condition var	cv attr object
cond_destroy	151	destroying thread_id	condition var	
cond_wait	152	waiting thread_id	condition var	mutex
cond_unwait	153	posted thread_id	condition var	mutex
Initial thread create	160	initial thread_id	nil	stack_size
Initial thread exit	161	initial thread_id		
Pthread create	162	creating thread_id	created thread_id	stack_size
Pthread created	163	created thread_id	nil	stack_size
Pthread exit	164	created thread_id		
Pthread wait	165	joining thread_id	joined thread_id	
Pthread unwait	166	joining thread_id	joined thread_id	
Imminent CAA Chain Addition	167			
CAA Chain Addition Complete	168			
Imminent CAA Chain Deletion	169			
CAA Chain Deletion Complete	170			
POSIX fork() imminent	171	thread_id		
In child process	172			
POSIX exec() imminent	173			
Process clean up imminent	174			

Debug Interfaces

Table 86. Debugger Language Environment event handler interface for AMODE 64 applications (continued)

Debug Tool Event	Debug Tool Event Code	Parm 2	Parm 3	Parm 4
Spawn is imminent	175			
UNIX file system load module	176	DSA	UNIX file system module descriptor	DSA format
Delete UNIX file system load module	177	DSA	UNIX file system module name	DSA format
In parent process	178			
After spawn	179			
rwlock lock for read	181	thread_id	rwlock	
rwlock lock for write	182	thread_id	rwlock	
rwlock wait for read	183	thread_id	rwlock	
rwlock wait for write	184	thread_id	rwlock	
Multiple event Execute Hook invocation	189	<ul style="list-style-type: none"> • Parm 2 = DSA • Parm 3 = hook offset • Parm 4 = DSA format • Parm 5 = A buffer containing general purpose registers • Parm 6 = Return address to the routine that was interrupted • Parm 7 = Entry point to the routine that was interrupted • Parm 8 = Event mask 		

Note:

1. This is the requestor's DSA, which means an HLL library routine DSA is likely the requestor of the Language Environment service or user DSA.
2. If DSA format is 1 in a 64-bit environment, i.e. XPLink DSA, 64-bit address of 64-bit'ized DSA

Table 87. Debugger Language Environment event handler bit mask descriptions for AMODE 64 applications

Bit mask	Description
'00000000'X	The object is a private mutex with the non-recursive characteristic.
'00000001'X	The object is a private mutex with the recursive characteristic.
'00800000'X	The object is a shared mutex with the non-recursive characteristic.
'00800001'X	The object is a shared mutex with the recursive characteristic.
'08000001'X	The object is a private rwlock with the recursive characteristic.
'08800001'X	The object is a shared rwlock with the recursive characteristic.

CAA

A doubleword binary integer that contains the address of the CAA.

CIB

A doubleword binary integer that contains the address of the CIB.

DSA

A doubleword binary integer that contains the address of the DSA.

DSA format

A fullword binary integer set to:

- 1 The format of the DSA is XPLINK style.

General purpose registers

A 128-byte buffer containing the general purpose registers stored in order 0 to 15 at the time the debug hook was executed. If the debugger changes these register values, the new values will be used when control is returned to the routine that executed the debug hook.

return_address

A doubleword pointer containing the address of the instruction where control will be returned to the routine that executed the debug hook. If the debugger changes this address, control will be returned to the new location.

entry_ptr

A fullword pointer containing the address of the entry point of the routine that contains the debug hook.

EDB

A doubleword binary integer that contains the address of the EDB.

module name

A halfword-prefixed string of the module name being deleted.

UNIX file system module name

A fullword-prefixed string of the module name being deleted.

module descriptor

A structure describing the module that was just loaded. The structure is as follows:

```

dcl 1 module descriptor,
    3 load point pointer(64),
    3 module size fixed,
        3 * char(4),
        3 entry point pointer(64),
    3 name length fixed(15),
    3 module name char(255);

```

UNIX file system module descriptor

A structure describing the module that was just loaded. The structure is as follows:

```

dcl 1 UNIX file system module descriptor,
    3 load point pointer(64),
    3 module size fixed,
        3 * char(4),
        3 entry point pointer(64),
    3 name length fixed(31),
    3 module name char(255);

```

result code

A fixed(31) binary value action for condition manager to take. The supported values are:

- 110 — Resume at the resume cursor
- 120 — Percolate to next condition handler

storage length

A fixed(31) binary value containing the number of bytes of storage.

cmd string

A halfword-prefixed string containing the debug command.

Debug Interfaces

msg_text

A halfword-prefixed string of the text that is transmitted by Language Environment message services.

ddname

An 8-byte character string, left-justified, padded right with blanks of the target ddname.

INPL

The initialization parameter list. For the format of the INPL, see Figure 54 on page 151.

start_rtn

A function pointer to the start routine for the pthread.

thread_id

An 8-byte thread identifier.

mutex

A pointer to a mutex object.

recursive

A recursive type mutex.

nonrecurs

A nonrecursive type mutex.

condition var

A pointer to a condition variable object.

cv attr object

A pointer to a condition variable attributes object.

stack_size

A fixed (63) stack size attribute (in bytes) of initial or created thread.

nil

Unused; null pointer.

event mask

a fullword binary value in which each bit represents a different hook event. When the bit is '1'b, the event occurred. The values of the bits are:

Bit	Event
0-11	Not used
12	Multiple Event Hook
13	Allocate Descriptor Built
14	Block Entry
15	Not used
16	User label
17	Begin of statement
18	Call return
19-20	Not used
21	Start of loop
22	If evaluated TRUE
23	If evaluated FALSE
24	Switch/case/select choice start
25	Switch/case/select default start
26	Multiple flows join
27	Not used
28	Call begin
29	Goto
30	Procedure exit
31	Multiple exit

Usage Notes:

1. A message is issued if the load fails because the Debug tool is not available.
2. All parameters are passed by reference.
3. Return codes (in decimal) are placed in R3

00	Success
16	Critical error in the debug tool; do not invoke again.
4. The debugger signals a CEE2F1 condition when it needs to quit from a nested enclave.

Language Environment actions for the interactive debug tool

This section discusses the actions Language Environment takes on behalf of a debug tool.

Language Environment parses the TEST runtime option on behalf of the debug tool and sets the appropriate flags within the Language Environment options control block. Language Environment sets the initial values for the test level and the debug tool event handler in the PCB. After its initial setting during the initialization of the first enclave within the process, this field is updated only by debug tool commands such as the SET TEST command. It is not influenced by nested enclave invocations. For every new enclave spawned and every thread being terminated, if the debug tool has been initialized, Language Environment thread initialization/termination calls the debug event with an enclave initialization or termination event code.

If the debug tool has been initialized, Language Environment messages and messages using Language Environment services are delivered to the debug tool by calling the debug event handler. In addition, the Language Environment error handler calls the debug event handler for all enabled conditions. The debug event handler is called after the enablement phase and prior to calling the exception handlers. It is also called when a condition is promoted.

Language Environment interactive debug data areas

Language Environment provides data areas for a debug tool's use. These areas are described in this section. The CAA fields are as follows:

- Initial command string address and length is contained within the Language Environment options control block.
- The TEST option's command file ddname is contained within the Language Environment options control block.
- Indication of ALL, ERROR, or NONE TEST suboption is contained within the Language Environment options control block.

Execute hook support

The compiled execute hook can be a single event hook or a multiple event hook. A multiple event hook represents the simultaneous occurrence of more than one execute hook event. The multiple event hook collapses multiple EX instructions into a single EX instruction, followed by a NOP instruction.

Invoking the event handler:

- Single event hook:

If the debugger has been initialized when a single event hook is enabled and executed, the debugger event handler is invoked with the following interface:

1. Event code 133

Debug Interfaces

2. A DSA that was in control when the hook was executed
 3. The offset of the hooks within the hook set that was executed (a multiple of 4 ranging from 0 to 15 inclusive)
 4. DSA format
 5. A buffer containing general purpose registers
 6. Return address to the routine that was interrupted
 7. Entry point to the routine that was interrupted
- Multiple event hook:

If the debugger has been initialized when a multiple event hook is enabled and executed and the hook for at least one of the events is active, the debugger event handler is invoked with the following interface:

 1. Event code 189
 2. A DSA that was in control when the hook was executed
 3. The offset of a multiple event hook is a specific number determined by the events
 4. DSA format.
 5. A buffer containing general purpose registers
 6. Return address to the routine that was interrupted
 7. Entry point to the routine that was interrupted
 8. Event mask

Use `__setHookEvents()` to enable or disable execution hooks.

Performance analysis support

This section describes Language Environment's support for performance analysis, or profiler tools. You can use a profiler tool to determine the performance level of an application; for example, trace data from a profiler tool can reveal the areas of an application that require the most processing time.

The C/C++ Performance Analyzer is available with the IBM C/C++ Productivity Tools for the z/OS product. Use the Performance Analyzer to help analyze, understand, and tune your C and C++ applications for improved performance.

Profile tool event handler

The profile event handler is A DLL named CELQVPRF with an exported function called CELQVPRF. The profiler event handler is loaded and initialized if the PROFILE runtime option is in effect and the TEST runtime option is not specified.

Reminder: If the TEST runtime option is specified, the PROFILE runtime option is ignored and a profiler tool is not loaded. A load failure occurs if Language Environment cannot find the CELQVPRF routine or if the routine is not available.

Language Environment calls the CELQVPRF function to notify the profiler tool of certain events. These events, which are described in Table 88 on page 747, are a subset of the notifications and parameters that Language Environment passes to the debug tool event handler.

Table 88. Profile tool — Language Environment event handler interface for AMODE 64 applications

Profile Tool Event	Profile Tool Event Code	Parm 2	Parm 3	Parm 4
Condition raised	101	CIB	result code	
Unhandled condition	103	CIB	result code	
Enclave init	118	creator's EDB		
Enclave term	119			
Thread init	120	creator's CAA		
Profile tool term	121			
Thread term	122			
External entry	123	DSA address (see note)	profiler invocation string	<ul style="list-style-type: none"> • Parm 4 = INPL • Parm 5 = DSA format
Condition promote	127	CIB	result code	
Execution Hook invocation	133	<ul style="list-style-type: none"> • Parm 2 = DSA • Parm 3 = hook offset • Parm 4 = DSA format • Parm 5 = A buffer containing general purpose registers • Parm 6 = Return address to the routine that was interrupted • Parm 7 = Entry point to the routine that was interrupted • Parm 8 = Eight-byte clock value returned by the STORE Clock (STCK) instruction • Parm 9 = Eight-byte elapsed CPU time in microseconds returned by the TIMEUSED assembler service 		
Initial thread create	160	initial_thread_id	nil	stack_size
Initial thread exit	161	initial_thread_id		
Pthread create	162	creating_thread_id	created_thread_id	stack_size
Pthread created	163	created_thread_id	nil	stack_size
Pthread exit	164	created_thread_id		
POSIX fork() imminent	171	thread_id		
In child process	172			
POSIX exec() imminent	173			
Process clean up imminent	174			
Spawn is imminent	175			
In parent process	178			
After spawn()	179			

Performance Analysis Support

Table 88. Profile tool — Language Environment event handler interface for AMODE 64 applications (continued)

Profile Tool Event	Profile Tool Event Code	Parm 2	Parm 3	Parm 4
Multiple event Execute Hook invocation	189	<ul style="list-style-type: none"> • Parm 2 = DSA • Parm 3 = hook offset • Parm 4 = DSA format • Parm 5 = A buffer containing general purpose registers • Parm 6 = Return address to the routine that was interrupted • Parm 7 = Entry point to the routine that was interrupted • Parm 8 = eight-byte clock value returned by the STORE Clock (STCK) instruction • Parm 9 = eight-byte elapsed CPU time in microseconds returned by the TIMEUSED assembler service • Parm 10 = Event mask 		

Note: This is the requestor's DSA, which means an HLL library routine DSA is likely the requestor of the Language Environment service or user DSA.

CAA

A doubleword binary integer that contains the address of the CAA.

CIB

A doubleword binary integer that contains the address of the CIB.

DSA

A doubleword binary integer that contains the address of the DSA.

EDB

A doubleword binary integer that contains the address of the EDB.

Hook offset

A fullword binary integer that contains the offset of the hook that was executed within the hook set. (This value is a multiple of 4 ranging from 0 to 52 inclusive.)

DSA format

A fullword binary integer set to:

- 1 The format of the DSA is XPLINK style.

General purpose registers

A 128-byte buffer containing the general purpose registers stored in order 0 to 15 at the time the debug hook was executed. If the debugger changes these register values, the new values will be used when control is returned to the routine that executed the debug hook.

return_address

A doubleword pointer containing the address of the instruction where control will be returned to the routine that executed the debug hook. If the debugger changes this address, control will be returned to the new location.

entry_ptr

A doubleword pointer containing the address of the entry point of the routine that contains the debug hook.

result code

A fixed(31) binary value action for condition manager to take. The supported values are:

- 110 — Resume at the resume cursor
- 120 — Percolate to next condition handler

storage length

A fixed (31) binary value containing the number of bytes of storage.

profiler invocation string

A halfword-prefixed string that contains the invocation string of the profiler tool. This value, which is specified as the *string* parameter of the PROFILE runtime option, it is translated to upper case characters. For more information about the runtime option, see *z/OS Language Environment Programming Reference*.

INPL

The Initialization Parameter List. For the format of the INPL, see Figure 54 on page 151.

thread_id

An 8-byte thread identifier.

stack_size

A fixed (63) stack size attribute (in bytes) of initial or created thread.

nil

Unused; null pointer.

event mask

a fullword binary value in which each bit represents a different hook event. When the bit is '1'b, the event occurred. The values of the bits are:

Bit	Event
0-11	Not used
12	Multiple Event Hook
13	Allocate Descriptor Built
14	Block Entry
15	Not used
16	User label
17	Begin of statement
18	Call return
19-20	Not used
21	Start of loop
22	If evaluated TRUE
23	If evaluated FALSE
24	Switch/case/select choice start
25	Switch/case/select default start
26	Multiple flows join
27	Not used
28	Call begin
29	Goto
30	Procedure exit
31	Multiple exit

Language Environment actions for profiler

Language Environment parses the PROFILE runtime option on behalf of the profile tool and sets the appropriate flags and profiler invocation string with the Options Control Block (OCB). If the TEST runtime option has also been specified, Language Environment issues a message to indicate that the TEST option will take precedence; that is, Language Environment will load the specified debug tool and will not load the specified profiler tool. If the NOTEST runtime option is specified, Language Environment loads module CELQVPRF.

Chapter 29. Anchor support for AMODE 64 applications

For AMODE 64 applications, register 12 can no longer be relied upon to contain the address of Language Environment common anchor area (CAA). Instead a new Language Environment anchor, library anchor area (LAA), is being defined for AMODE 64 applications that is anchored in the prefix save area (PSA) field PSALAA.

On every TCB ATTACH of an AMODE 64 application, the Language Environment LAA is allocated and initialized along with the STCB (Key 0 and subpool 253, ELSQA). It is anchored from both the PSALAA and the STCB field STCBLAA. Since this is Key 0 authorized storage, it can be read by Language Environment and other unauthorized programs. When a TCB is dispatched, the contents of STCBLAA are copied into PSALAA.

The LAA points to a new library common area (LCA). This control block is allocated when Language Environment is initialized. It is allocated along with the Language Environment control blocks in 31-bit storage in the key of the caller. It contains information and pointers that can be set or reset by the application in the key of the caller, including a pointer to the CAA.

To get the address of the Language Environment CAA in an AMODE 64 environment, the basing is:

```
PSALAA -> CEELAA_LCA64 -> CEELCA_CAA -> CAA
```

For more information on the library anchor area and library common area, see Chapter 22, "CALL linkage convention for AMODE 64 applications," on page 677.

Chapter 30. Preinitialized Environments for Authorized Programs for AMODE 64 applications

Preinitialized Environments for Authorized Programs is a feature of Language Environment for AMODE 64 applications. It allows authorized components to create pre-initialized Language Environment environments that are able to execute C/C++ and Language Environment-conforming assembler routines. To use Preinitialized Environments for Authorized Programs, the caller must be running supervisor state, with PSW key 1 to 7. The caller's PSW key must be the same for all requests. When running in cross-memory mode, all data used by the routine must be in the current primary address space. Access registers are not used to address this data.

Preinitialized Environments for Authorized Programs are created, initialized, and ended asynchronous to the execution of the C, C++, and Language Environment-conforming Assembler routines. Each environment is a self-contained process and has its own stack and heap. You have the option of managing the environments (user-managed) or allowing Preinitialized Environments for Authorized Programs to manage them (system-managed).

Creating Preinitialized Environments for Authorized Programs

You can initialize environments with a call using the CELAAUTH macro. On this call, you can specify the characteristics of the environments you want to create, including the management characteristics, runtime options, and the number of environments. Environment initialization can only be performed in the home address space in TCB or SRB mode.

Restriction: Cross-memory mode initialization is not allowed.

Preinitialized Environments for Authorized Programs are based on the AMODE 64 version of Language Environment and have the following characteristics:

- The linkage model is XPLINK.
- The storage for the stack, user heap, and most Language Environment control blocks is allocated above the bar.
- Only AMODE 64 runtime options are valid.

In order for authorized applications to use this support, it is required that they define SCEERUN2 and SCEERUN as authorized libraries. These libraries are part of the z/OS program search order for the address space's cross-memory resource owning (CMRO) task. You can do this in one of the following ways:

- Put SCEERUN2 and SCEERUN in the LNKLST.
- Define SCEERUN2 and SCEERUN to be APF-authorized, and placing them in the application's TASKLIB or STEPLIB/JOBLIB concatenation.

Preinitialized Environments for Authorized Programs supports a subset of the C/C++ library functions. For a list of these functions, see *z/OS XL C/C++ Runtime Library Reference*.

Creating a user-managed environment

When creating a user-managed environment, you can supply runtime options by passing a string of characters on the initialization call. These options are saved with the user-managed environment and are merged into the runtime options set when the runtime environment is created.

Each initialization call to CELAAUTH creates one environment. A token representing the environment is returned to the caller. This token is used to identify the newly-created environment on subsequent calls.

Creating a system-managed environment

When creating a set of system-managed environments, you need to create an authorized environment definition table (AEDT). The AEDT describes the attributes and management characteristics of the environments to be created. Each AEDT contains one or more environment definition entries (AEDE). Each AEDE is a set of characteristics that describe how and when an environment is to be created. On later routine calls, you need to select an AEDE to run the call by specifying the index in the AEDT corresponding to the appropriate AEDE.

Guidelines: When building the AEDT, for each AEDE specify:

- An optional runtime options string to be applied to any environment created for that entry. The runtime option string for each AEDE is applied to each environment on the first call using the environment, along with the options specified by the main(). This initial set of options remains in effect for the life of the environment, unless overridden by another runtime options string on a subsequent call that uses this environment.
- The initial number of environments to be created
- The amount of time, in microseconds, to wait if no environments are available. The minimum value is 0, which indicates that no wait is to be performed. CELAAUTH uses this value to wait once for an available environment before attempting to increment the number of environments. If the maximum number of environments has been reached, CELAAUTH waits once more for an available environment, using the specified time. Do not specify values greater than 20 microseconds because this could have an adverse affect on CELAAUTH processing.
- The number of environments to be incrementally created when all existing environments are in use
- The maximum number of environments to be created
- A deferred initialization attribute that indicates whether to defer the creation of the initial set of environments until the first call using that environment type

With system-managed environments, the caller needs to provide an 8-byte identifier that identifies the set of managed environments within the address space. This helps differentiate between sets of managed environments created by different components or applications. The actual content of the environment ID is up to the caller. For example, it could contain a pointer to an application's control block or an 8-character value.

Recommendation: When characters are used for the environment ID, the value should begin with the application's 3-4 character component ID to help avoid confusion with other identifiers. Managed environment IDs beginning with the characters "CEE", "CEL", and "EDC" are reserved for use by Language Environment.

Preinitialized Environments for Authorized Programs tasks

The following tasks are important to Preinitialized Environments for Authorized Programs when it performs specific functions:

- Resource-owning (ARO) TCB
- Preinitialized Environments for Authorized Programs worker task

Task-level resource managers are established within the address space the first time an environment is created. These resource managers detect when the ARO or worker task is terminating and performs any necessary cleanup.

Preinitialized Environments for Authorized Programs resource-owning TCB

The Preinitialized Environments for Authorized Programs resource-owning (ARO) TCB is the task to which CELAAUTH assigns ownership for system resources that it obtains on behalf of the user. This ensures that environments are independent of the dispatchable unit of work that created them and allows them to continue to function after the related CELAAUTH call has ended. The default ARO TCB is the cross memory resource owning (CMRO) TCB in the address space.

Preinitialized Environments for Authorized Programs worker task

CELAAUTH attaches a worker task to the ARO TCB. The worker task performs functions that must be done in task mode. These include loading routine load modules, DLLs, and fetchable modules requested by routines running in environments within the address space. Note the following restrictions:

- Preinitialized Environments for Authorized Programs does not control or override the search order used to locate a load module. The search order established for the ARO TCB is used. This includes any JOBLIBs, STEPLIBs, or TCBLIBs the application previously defined.
- Preinitialized Environments for Authorized Programs does not delete any of the load modules that were loaded using the worker task.
- Only reentrant routines are supported.
- All load modules must reside in a PDSE.

Executing a routine in Preinitialized Environments for Authorized Programs

An application can execute C, C++, and Language Environment-conforming Assembler code by calling CELAAUTH with information describing the routine to be run. The call to CELAAUTH must be made in primary ASC mode in the key in which the routine should run.

Recommendation: All routines should be reentrant to reduce the storage constraints.

In Preinitialized Environments for Authorized Programs, an application can call the following types of routines:

- main()

Preinitialized Environments for Authorized Programs supports C/C++ main() functions using one of the following:

- The name of the program object where main() resides if CELAAUTH needs to load the program object
- The address of the program object entry point if the calling program loaded the program object previously

- Fetchable function
Preinitialized Environments for Authorized Programs supports calls to functions that have been declared fetchable using the #pragma linkage compiler directive. The function is identified by:
 - The name of the program object where the function resides if CELAAUTH needs to load the program object
 - The address of the program object entry point if the calling program loaded the program object previously
- Exported function
Preinitialized Environments for Authorized Programs supports calls to exported functions from a DLL. The function is identified by supplying the name of the function and the name of the DLL in which the function resides.

When returning from the first call for a specific routine, CELAAUTH returns a token representing the routine. This *routine token* should be used on subsequent calls to the same routine to allow CELAAUTH to pass control to the routine more quickly.

Calling a main routine

When calling a main routine, additional runtime options are passed as a string of characters on the call. This string is merged with the initialization runtime options string, the application runtime options settings found in CELQUOPT, and the system-level defaults, before initializing the environment. For more information, see “Creating a user-managed environment” on page 754.

Each time a main is run within an environment, CELAAUTH reinitializes the environment. This includes:

- Refreshing the main's WSA
- Resetting Language Environment component states
- Cleaning up unneeded heap allocations from a previous use

Calling a subroutine

When calling a subroutine, the input runtime options string is ignored. The subroutine is run in an environment that is initialized using the initialization runtime options string merged with the system-level defaults before initializing the environment.

When a subroutine is run on an environment just after a main routine is run on the same environment, the subroutine receives a reinitialized environment. When a subroutine calls `exit()`, the `atexits` are invoked and environment terminates after the `atexits`. The next time this environment is used, it is reinitialized.

Using runtime options

Most runtime options are valid for Preinitialized Environments for Authorized Programs with the following exceptions:

- POSIX(ON) is not supported. If POSIX(ON) is specified, it is overridden by POSIX(OFF).
- The generation of CEEDUMPs is not supported. The only valid options for TERMTHDACT are QUIET, MSG, UAONLY, and UAIMM. If any other option is specified, it is overridden with TERMTHDACT(UAONLY).

During the first environment initialization within an address space, a copy is made of the currently active set of parmlib-level default runtime options. This set of

options is merged with runtime options from other sources when each environment is initialized. Subsequent changes to the parmlib-level options on the system do not affect this local copy of the options.

Selecting an environment

For user-managed environments, the user provides an environment token that identifies the environment in which the code should execute. If the environment is already in use, CELAAUTH returns a reason code indicating that the environment is unavailable.

For system-managed environments, CELAAUTH selects the environment with which to execute the code. A new environment might be created if all the existing environments are already being used. The user has no control over what specific environments are used with each routine. These routines should be "stateless", not relying on any data previously saved in the environment.

Recommendation: If this set of routines has complex state requirements or dependencies, the application must use user-managed environments.

Providing recovery

To provide recovery, CELAAUTH establishes one of the following each time it is called:

- an EUTFRR
- an ESTAE if RECOVERY=ESTAE is specified on the CELAAUTH macro invocation

In addition to capturing serviceability information, CELAAUTH uses this recovery routine to process Language Environment shunts, handle math overflows, and drive Language Environment condition management for the application routines.

Restriction: With an EUTFRR established, the program cannot issue SVCs or handle asynchronous interrupts.

Preinitialized Environments for Authorized Programs provides recovery for worker tasks established under the ARO within each address space where environments are created. This ensures that the task does not end prematurely. The worker task recovery attempts to isolate the problem and makes sure the task remains active. An ETXR routine is established in case worker task recovery cannot save the TCB.

Terminating Preinitialized Environments for Authorized Programs

CELAAUTH begins to clean up the environment when it receives a termination call. For user-managed environments, resources related to the specific environment that provided a token are freed. For system-managed environments, all existing environments are cleaned up. When the last environment is ended, CELAAUTH cleans any remaining tasks and resources that were obtained to manage the environments.

Examples of using Preinitialized Environments for Authorized Programs

Following are examples of using Preinitialized Environments for Authorized Programs.

Using Preinitialized Environments for Authorized Programs in service request block (SRB) mode

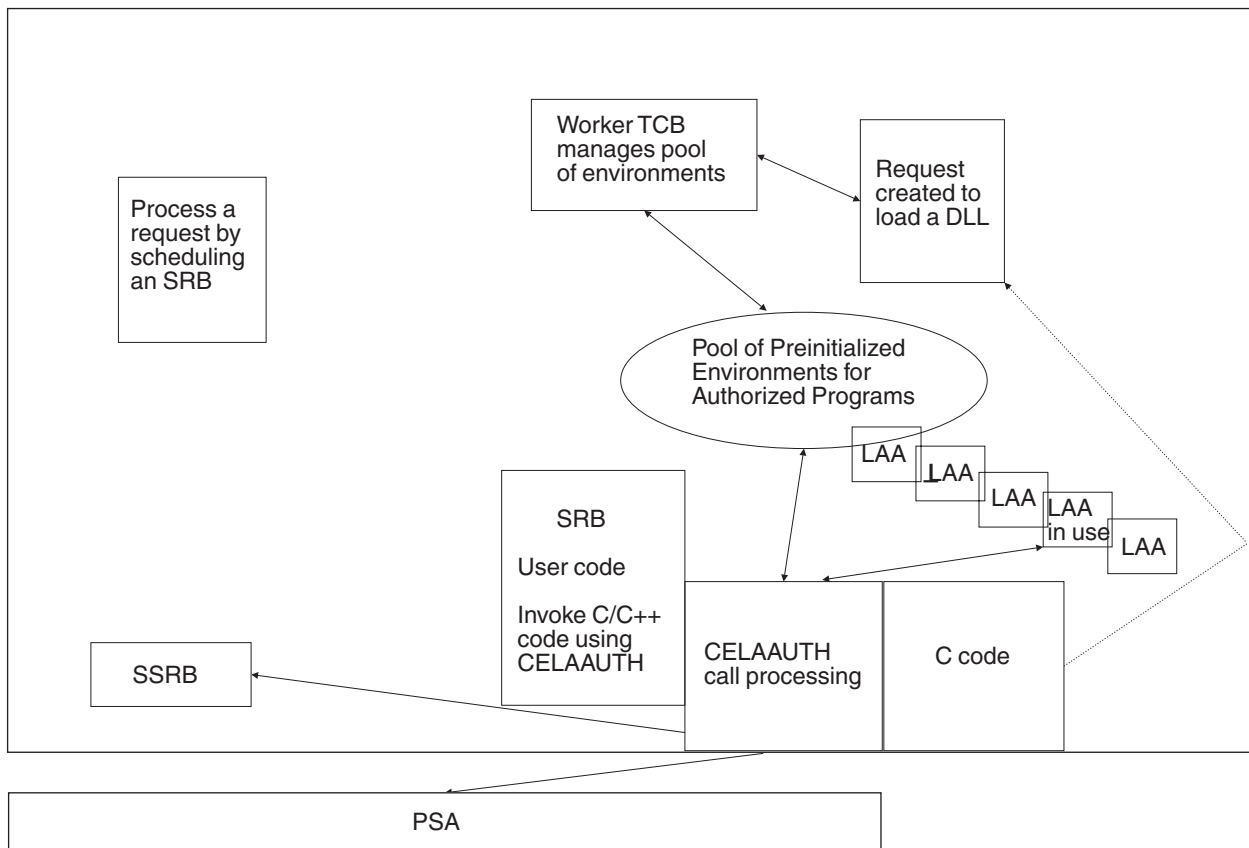


Figure 152. Using Preinitialized Environments for Authorized Programs in SRB mode

In this example, Preinitialized Environments for Authorized Programs is being used by an SRB mode exploiter. The user code running under the SRB uses the CELAAUTH macro to call a C/C++ routine within the address space. The CELAAUTH services locate an available environment, including an LAA, in which to run the routine. The address of the LAA is placed in PSALAA before calling the routine. If the routine requires an additional DLL to be loaded, CELAAUTH queues a request to the worker task, which performs the load and returns the information. If the SRB is preempted, the LAA address in PSALAA is saved in the SSRB. This value is restored to PSALAA when the SRB is re-dispatched.

Using Preinitialized Environments for Authorized Programs in cross-memory mode

In Figure 153 on page 759, Preinitialized Environments for Authorized Programs are being used in cross-memory mode. A TCB has performed a program call into the address space where environments have been initialized. The user code running as the target of the program call uses the CELAAUTH macro to call a C/C++ routine within the address space. The CELAAUTH services locate an available environment, including an LAA, in which to run the routine. The address of the LAA is placed in PSALAA and STCBLAA before calling the routine. If the routine requires an additional DLL to be loaded, CELAAUTH queues a request to the worker task, which performs the load and returns the load information. If the TCB is preempted, the LAA address can be restored to PSALAA from STCBLAA

when the TCB is re-dispatched.

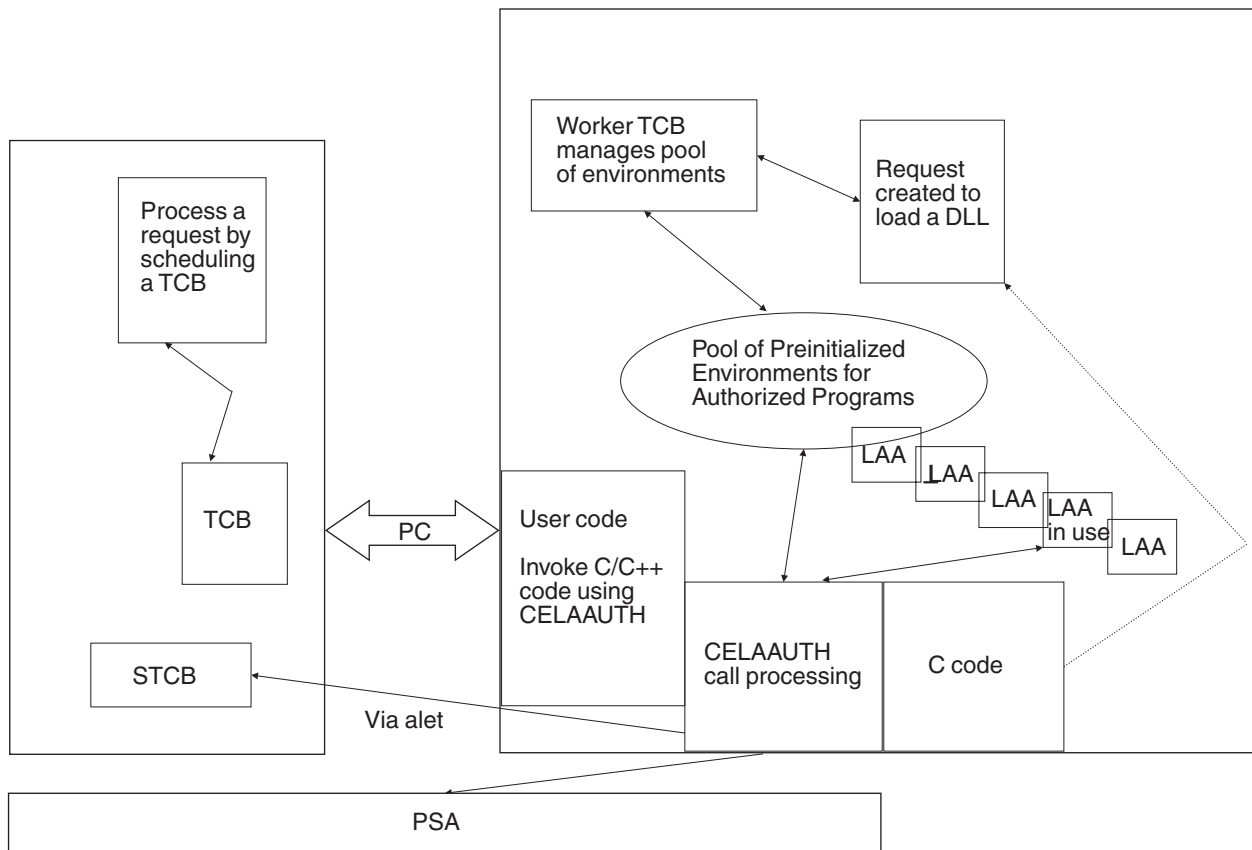


Figure 153. Using Preinitialized Environments for Authorized Programs in cross-memory mode

CELAAUTH macro

The CELAAUTH macro allows you to create Preinitialized Environments for Authorized Programs and to run C, C++, and Language Environment-conforming assembler routines within those environments for AMODE 64 applications. CELAAUTH is used to perform the following tasks:

- Environment initialization: Set up one or more environments for later use.
- Routine invocation: Call a C/C++ or Language Environment Assembler routine using a previously initialized environment.
- Environment termination: Clean up one or more environments.

CELAAUTH environments

There are two forms of Preinitialized Environments for Authorized Programs: user-managed and system-managed.

User-managed environment

In a user-managed environment, the invoker of CELAAUTH has complete control over the environments. This includes the number created, when they are created and destroyed, and the environment used to execute each called routine. For a user-managed environment, an application can use these CELAAUTH request types:

CELAAUTH

USERINIT

Initialize an environment that is managed by the user. See “Syntax for REQUEST=USERINIT” on page 761 for more information.

USERCALL

Call a routine using an environment that was initialized using USERINIT. See “Syntax for REQUEST=USERCALL” on page 765 for more information.

USERTERM

End an environment that was created using USERINIT. See “Syntax for REQUEST=USERTERM” on page 770 for more information.

System-managed environment

In a system-managed environment, CELAAUTH provides most of the management of the environments. The caller of CELAAUTH only needs to specify the number of environments to be created and the runtime options for these environments. For a system-managed environment, an application can use these CELAAUTH request types:

MNGDINIT

Define and initialize a set of environments that are to be managed by the system. See “Syntax for REQUEST=MNGDINIT” on page 772 for more information.

MNGDCALL

Call a routine using an environment that is part of the set of environments that was initialized using MNGDINIT. See “Syntax for REQUEST=MNGDCALL” on page 778 for more information.

MNGDUPDT

Call a routine using an environment that is part of the set of environments that was initialized using MNGDUPDT. See “Syntax for REQUEST=MNGDUPDT” on page 783 for more information.

MNGDTERM

End the set of environments that was created using MNGDINIT. See “Syntax for REQUEST=MNGDTERM” on page 786 for more information.

Environment overview

The requirements for the caller are:

	Requirement
Minimum authorization:	Supervisor state with PSW key 1–7
Dispatchable unit mode:	Task or SRB
Cross memory mode:	Any PASN, any HASN, any SASN
	Requests for USERCALL and USERTERM must be made in the same primary address space as a previous USERINIT request.
	Requests for MNGDCALL and MNGDTERM must be made in the same primary address space as a previous MNGDINIT request.
	Restriction: Requests for USERINIT and MNGDINIT cannot be made while in cross-memory mode.
AMODE:	64-bit
ASC mode:	Primary
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks may be held.

Control parameters: **Requirement**
 Control parameters must be in the primary address space.
 For REQUEST=USERCALL and REQUEST=MINGDCALL, all data to be accessed by any C or C++ routines must be in the primary address space as well.

Programming requirements

None.

Restrictions

None.

Input register information

When issuing the CELAAUTH macro, register 13 must contain the address of a 144-byte work area.

Attention: All 144 bytes of this work area might be changed during the CELAAUTH invocation. This is important if the caller is attempting to use its current register 13 save area as the work area. Important fields in the save area might be destroyed, such as the previous save area address at offset X'80' in an F4SA-style save area. If the caller is using a save area, it must ensure that important fields in the save area are preserved. Use care when the caller's register 13 points to a dynamic area containing local variables to ensure that any variables used by the CELAAUTH expansion are still addressable.

The caller does not have to place any information into any other registers unless using it in register notation for a particular parameter, or using it as a base register.

Output register information

When control returns to the caller, the GPRs contain:

Register	Contents
0	Reason code, if GPR 15 is non-zero
1	Used as a work register by the system
2-13	Unchanged
14	Used as a work register by the system
15	Return code

When control returns to the caller, the ARs contain:

Register	Contents
0-15	Unchanged

Some callers depend on register contents remaining the same before and after issuing a service. If the system changes the contents of registers on which the caller depends, the caller must save them before issuing the service, and restore them after the system returns control.

Syntax for REQUEST=USERINIT

CELAAUTH

<i>name</i>	<i>name</i> : symbol. Begin <i>name</i> in column 1.
b	One or more blanks must precede CELAAUTH.
CELAAUTH	
b	One or more blanks must follow CELAAUTH.

REQUEST=USERINIT

<u>,WRKJSTCB=SYSRULES</u> ,WRKJSTCB=NO ,WRKJSTCB=YES	Default: WRKJSTCB=SYSRULES
<u>,FULLINIT=YES</u> ,FULLINIT=NO	Default: FULLINIT=YES
,RTO= <i>rto</i>	<i>rto</i> : RS-type address or address in register (2) - (12)
,RTOLEN= <i>rtolen</i>	<i>rtolen</i> : RS-type address or address in register (2) - (12)
,ENVTOKEN= <i>envtoken</i>	<i>envtoken</i> : RS-type address or address in register (2) - (12)
<u>,RECOVERY=EUTFRR</u> ,RECOVERY=ESTAE	Default: RECOVERY=EUTFRR
,RETCODE= <i>retcode</i>	<i>retcode</i> : RS-type address or register (2) - (12).
,RSNCODE= <i>rsncode</i>	<i>rsncode</i> : RS-type address or register (2) - (12).
<u>,PLISTVER=IMPLIED_VERSION</u> ,PLISTVER=MAX ,PLISTVER=0	Default: PLISTVER=IMPLIED_VERSION
,MF= S ,MF=(L, <i>list addr</i>) ,MF=(L, <i>list addr,attr</i>) ,MF=(L, <i>list addr,OD</i>) ,MF=(E, <i>list addr</i>) ,MF=(E, <i>list addr,COMPLETE</i>)	Default: MF=S <i>list addr</i> : RS-type address or register (1) - (12)

Parameters for REQUEST=USERINIT

The parameters are explained as follows:

name

is an optional symbol, starting in column 1, that is the name on the CELAAUTH macro invocation. The name must conform to the rules for an ordinary assembler language symbol. The default is no name.

REQUEST=USERINIT

REQUEST=USERINIT creates Preinitialized Environments for Authorized

Programs in the current primary address space and returns a token that identifies the environment. Each USERINIT call establishes another environment. The application must manage the set of environments USERINIT creates.

,WRKJSTCB=SYSRULES

,WRKJSTCB=NO

,WRKJSTCB=YES

An optional parameter that indicates the job step attribute of any task that CELAAUTH may attach to the Cross Memory Resource Owning task. This keyword is only necessary if this task will be the first subtask attached under this task, so the job step attribute is not clear. The default is WRKJSTCB=SYSRULES.

,WRKJSTCB=SYSRULES

indicates that CELAAUTH will determine the proper job step attribute with which to attach the task. CELAAUTH will use the job step attribute of the current subtasks under the Cross Memory Resource Owning Task. If no subtasks are found, then CELAAUTH will attach the task as non-job step.

,WRKJSTCB=NO

indicates that the task must be a non-job step task.

,WRKJSTCB=YES

indicates that the attached task must be a job step task.

,FULLINIT=YES

,FULLINIT=NO

An optional parameter indicating whether the environment is to be fully initialized during this call. The default is FULLINIT=YES.

,FULLINIT=YES

indicates full initialization is requested.

,FULLINIT=NO

indicates that minimal initialization is requested. Complete environment initialization will occur upon first use.

,RTO=*rto*

An optional input parameter containing the runtime options to be associated with the environment that is to be initialized during this call. The length of the runtime options cannot exceed 4096 characters.

To code: Specify the RS-type address, or address in register (2)-(12), of a character field.

,RTOLEN=*rtolen*

When RTO=*rto* is specified, a required input parameter containing the length of the runtime option string pointed to by RTO.

To code: Specify the RS-type address, or address in register (2)-(12), of a doubleword field.

,ENVTOKEN=*envtoken*

A required output parameter that is to contain the token that will be used to identify the environment that was just created. The contents of this environment token will be provided by CELAAUTH, and must be used on subsequent user-managed type calls to identify the environment.

To code: Specify the RS-type address, or address in register (2)-(12), of a 16-character field.

CELAAUTH

,RECOVERY=EUTFRR

,RECOVERY=ESTAE

An optional parameter indicating the type of recovery routine to be set by CELAAUTH for this invocation.

,RECOVERY=EUTFRR

indicates that an EUTFRR will be set.

,RECOVERY=ESTAE

indicates that an ESTAE will be set. When invoked in SRB mode, an EUTFRR is always used.

,RETCODE=retcode

An optional output parameter into which the return code is to be copied from GPR 15.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,RSNCODE=rsncode

An optional output parameter into which the reason code is to be copied from GPR 0.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,PLISTVER=IMPLIED_VERSION

,PLISTVER=MAX

,PLISTVER=0

An optional input parameter that specifies the version of the macro. PLISTVER determines which parameter list the system generates. PLISTVER is an optional input parameter on all forms of the macro, including the list form. When using PLISTVER, specify it on all macro forms used for a request and with the same value on all of the macro forms. The values are:

- **IMPLIED_VERSION**, which is the lowest version that allows all parameters specified on the request to be processed. If you omit the PLISTVER parameter, IMPLIED_VERSION is the default.
- **MAX**, if you want the parameter list to be the largest size currently possible. This size might grow from release to release and affect the amount of storage that your program needs.

If you can tolerate the size change, IBM recommends that you always specify PLISTVER=MAX on the list form of the macro. Specifying MAX ensures that the list-form parameter list is always long enough to hold all the parameters you might specify on the execute form, when both are assembled with the same level of the system. In this way, MAX ensures that the parameter list does not overwrite nearby storage.

- **0**, if you use the currently available parameters.

To code: Specify one of the following:

- IMPLIED_VERSION
- MAX
- A decimal value of 0

,MF=S

,MF=(L,list addr)

,MF=(L,list addr,attr)

,MF=(L,list addr,0D)

,MF=(E,list addr)

,MF=(E,list addr,COMPLETE)

An optional input parameter that specifies the macro form.

Use MF=S to specify the standard form of the macro, which builds an inline parameter list and generates the macro invocation to transfer control to the service. MF=S is the default.

Use MF=L to specify the list form of the macro. Use the list form together with the execute form of the macro for applications that require reentrant code. The list form defines an area of storage that the execute form uses to store the parameters. Only the PLISTVER parameter may be coded with the list form of the macro.

Use MF=E to specify the execute form of the macro. Use the execute form together with the list form of the macro for applications that require reentrant code. The execute form of the macro stores the parameters into the storage area defined by the list form, and generates the macro invocation to transfer control to the service.

,list addr

The name of a storage area to contain the parameters. For MF=S and MF=E, this can be an RS-type address or an address in register (1)-(12).

,attr

An optional 1- to 60-character input string that you use to force boundary alignment of the parameter list. Use a value of 0F to force the parameter list to a word boundary, or 0D to force the parameter list to a doubleword boundary. If you do not code *attr*, the system provides a value of 0D.

,COMPLETE

Specifies that the system is to check for required parameters and supply defaults for omitted optional parameters.

Syntax for REQUEST=USERCALL

<i>name</i>	<i>name</i> : symbol. Begin <i>name</i> in column 1.
b	One or more blanks must precede CELAAUTH.
CELAAUTH	
b	One or more blanks must follow CELAAUTH.

REQUEST=USERCALL

,RTNNAME= <i>rtname</i>	<i>rtname</i> : RS-type address or address in register (2) - (12)
,RTNADDR= <i>rtnaddr</i>	<i>rtnaddr</i> : RS-type address or address in register (2) - (12)
,RNAMELEN= <i>rnamelen</i>	<i>rnamelen</i> : RS-type address or address in register (2) - (12)
,DLLNAME= <i>dllname</i>	<i>dllname</i> : RS-type address or address in register (2) - (12)
,RTO= <i>rto</i>	<i>rto</i> : RS-type address or address in register (2) - (12)
,RTOLEN= <i>rtolen</i>	<i>rtolen</i> : RS-type address or address in register (2) - (12)

CELAAUTH

<code>,PARMLIST=parmlist</code>	<i>parmlist</i> : RS-type address or address in register (2) - (12)
<code>,RTNTOKEN=rtntoken</code>	<i>rtntoken</i> : RS-type address or address in register (2) - (12)
<code>,ENVTOKEN=envtoken</code>	<i>envtoken</i> : RS-type address or address in register (2) - (12)
<code>,RTNRETCODE=rtnretcode</code>	<i>rtnretcode</i> : RS-type address or address in register (2) - (12)
<code>,RTNRSNCODE=rtnrsncode</code>	<i>rtnrsncode</i> : RS-type address or address in register (2) - (12)
<code>,RTNFDBKCODE=rtnfdbkcode</code>	<i>rtnfdbkcode</i> : RS-type address or address in register (2) - (12)
<code>,RECOVERY=<u>EUTFRR</u></code> <code>,RECOVERY=ESTAE</code>	Default: RECOVERY=EUTFRR
<code>,RETCODE=retcode</code>	<i>retcode</i> : RS-type address or register (2) - (12).
<code>,RSNCODE=rsncode</code>	<i>rsncode</i> : RS-type address or register (2) - (12).
<code>,PLISTVER=<u>IMPLIED_VERSION</u></code> <code>,PLISTVER=MAX</code> <code>,PLISTVER=0</code>	Default: PLISTVER=IMPLIED_VERSION
<code>,MF=<u>S</u></code> <code>,MF=(L,<i>list addr</i>)</code> <code>,MF=(L,<i>list addr</i>,<i>attr</i>)</code> <code>,MF=(L,<i>list addr</i>,<u>OD</u>)</code> <code>,MF=(E,<i>list addr</i>)</code> <code>,MF=(E,<i>list addr</i>,<u>COMPLETE</u>)</code>	Default: MF=S <i>list addr</i> : RS-type address or register (1) - (12)

Parameters for REQUEST=USERCALL

The parameters are explained as follows:

name

is an optional symbol, starting in column 1, that is the name on the CELAAUTH macro invocation. The name must conform to the rules for an ordinary assembler language symbol. The default is no name.

REQUEST=USERCALL

REQUEST=USERCALL indicates that CELAAUTH invoke the specified C, C++, or Language Environment-conforming Assembler routine, using the environment represented by the supplied environment token. The environment can be called serially multiple times. The values in the heap and WSA are reinitialized between invocations only if the function is a C/C++ main.

`,RTNNAME=rtnname`

`,RTNADDR=rtnaddr`

A required input parameter.

`,RTNNAME=rtnname`

A parameter containing the 1-1024 character name of the routine to be called. The length of this name is provided via the RNAMELEN keyword. When DLLNAME is specified, then RTNNAME must be the name of a

function which has been exported from the DLL. Otherwise, RTNNAME must be the name of a main() program or fetchable routine, and is limited to 8 characters in length.

To code: Specify the RS-type address, or address in register (2)-(12), of a character field.

,RTNADDR=*rtnaddr*

A parameter containing the address of the routine to be called.

To code: Specify the RS-type address, or address in register (2)-(12), of an eight-byte pointer field.

,RNAMELEN=*rnamelen*

When RTNNAME=*rtname* is specified, a required input parameter containing the length of the routine name specified on the RTNNAME keyword.

To code: Specify the RS-type address, or address in register (2)-(12), of a doubleword field. *rnamelen* must be in the range 1 through 1024.

,DLLNAME=*dllname*

When RTNNAME=*rtname* is specified, an optional input parameter containing the 1-8 character name, padded with blanks, of the DLL from which the function name specified on the RTNNAME keyword has been exported.

To code: Specify the RS-type address, or address in register (2)-(12), of an 8-character field.

,RTO=*rto*

An optional input parameter containing the runtime options to be associated with the environment when the specified main is called. The length of the runtime options cannot exceed 4096 characters. This value is ignored if the routine to be called is a subroutine.

To code: Specify the RS-type address, or address in register (2)-(12), of a character field.

,RTOLEN=*rtolen*

When RTO=*rto* is specified, a required input parameter containing the length of the runtime option string pointed to by RTO.

To code: Specify the RS-type address, or address in register (2)-(12), of a doubleword field.

,PARMLIST=*parmlist*

An optional input parameter containing the parameter list to be passed to the routine that is to be called.

- The parameter list is copied to the appropriate location in the stack frame.
- general purpose registers 1, 2 and 3 are loaded from the parameter list when the routine is executed.

The length of the parameter list is determined from the PPA1 of the routine. If the routine takes a variable length parameter list, the length of the parameter list is assumed to be 256 bytes. Floating point and complex values can only be passed by reference.

To code: Specify the RS-type address, or address in register (2)-(12), of a character field.

,RTNTOKEN=*rtntoken*

An optional input/output parameter containing the token that identifies the

CELAAUTH

routine to be called. This token is built upon first invocation of the routine within the environment, and returned to the caller for use on subsequent calls using the same environment.

To code: Specify the RS-type address, or address in register (2)-(12), of a 16-character field.

,ENVTOKEN=envtoken

A required input parameter containing the token that identifies the environment to be used for this call.

To code: Specify the RS-type address, or address in register (2)-(12), of a 16-character field.

,RTNRETCODE=rtnretcode

A required output parameter that is to contain the routine return code after the routine has completed. For a main(), this is the enclave return code. For a subroutine, this is the value that the subroutine provided on the return statement that caused the subroutine to end. If the subroutine caused the enclave to terminate due to an unhandled condition or a call to exit(), then this is the enclave return code. For more information on the enclave return code, see *z/OS Language Environment Programming Guide*.

To code: Specify the RS-type address, or address in register (2)-(12), of a fullword field.

,RTNRSNCODE=rtnrsncode

A required output parameter that is to contain the routine reason code after the routine has completed. This value is 0 if the routine ended normally. If the enclave is terminated due to an unhandled condition or a call to exit(), then this is the enclave reason code. For more information on the enclave return code, see *z/OS Language Environment Programming Guide*.

To code: Specify the RS-type address, or address in register (2)-(12), of a fullword field.

,RTNFDBKCODE=rtnfdbkcode

A required output parameter that is to contain the condition token indicating why the application terminated. For normal completion of the routine, CEE000 is returned. If the enclave is terminated due to an unhandled condition or a call to exit(), this field contains the enclave feedback code for termination.

To code: Specify the RS-type address, or address in register (2)-(12), of a 16-character field.

,RECOVERY=EUTFRR

,RECOVERY=ESTAE

An optional parameter indicating the type of recovery routine to be set by CELAAUTH for this invocation.

,RECOVERY=EUTFRR

indicates that an EUTFRR will be set.

,RECOVERY=ESTAE

indicates that an ESTAE will be set. When invoked in SRB mode, an EUTFRR is always used.

,RETCODE=retcode

An optional output parameter into which the return code is to be copied from GPR 15.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,RSNCODE=rsncode

An optional output parameter into which the reason code is to be copied from GPR 0.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,PLISTVER=IMPLIED_VERSION

,PLISTVER=MAX

,PLISTVER=0

An optional input parameter that specifies the version of the macro. PLISTVER determines which parameter list the system generates. PLISTVER is an optional input parameter on all forms of the macro, including the list form. When using PLISTVER, specify it on all macro forms used for a request and with the same value on all of the macro forms. The values are:

- **IMPLIED_VERSION**, which is the lowest version that allows all parameters specified on the request to be processed. If you omit the PLISTVER parameter, IMPLIED_VERSION is the default.
- **MAX**, if you want the parameter list to be the largest size currently possible. This size might grow from release to release and affect the amount of storage that your program needs.

If you can tolerate the size change, IBM recommends that you always specify PLISTVER=MAX on the list form of the macro. Specifying MAX ensures that the list-form parameter list is always long enough to hold all the parameters you might specify on the execute form, when both are assembled with the same level of the system. In this way, MAX ensures that the parameter list does not overwrite nearby storage.

- **0**, if you use the currently available parameters.

To code: Specify one of the following:

- IMPLIED_VERSION
- MAX
- A decimal value of 0

,MF=S

,MF=(L,list addr)

,MF=(L,list addr,attr)

,MF=(L,list addr,0D)

,MF=(E,list addr)

,MF=(E,list addr,COMPLETE)

An optional input parameter that specifies the macro form.

Use MF=S to specify the standard form of the macro, which builds an inline parameter list and generates the macro invocation to transfer control to the service. MF=S is the default.

Use MF=L to specify the list form of the macro. Use the list form together with the execute form of the macro for applications that require reentrant code. The list form defines an area of storage that the execute form uses to store the parameters. Only the PLISTVER parameter may be coded with the list form of the macro.

Use MF=E to specify the execute form of the macro. Use the execute form together with the list form of the macro for applications that require reentrant code. The execute form of the macro stores the parameters into the storage area defined by the list form, and generates the macro invocation to transfer control to the service.

CELAAUTH

,list addr

The name of a storage area to contain the parameters. For MF=S and MF=E, this can be an RS-type address or an address in register (1)-(12).

,attr

An optional 1- to 60-character input string that you use to force boundary alignment of the parameter list. Use a value of 0F to force the parameter list to a word boundary, or 0D to force the parameter list to a doubleword boundary. If you do not code *attr*, the system provides a value of 0D.

,COMPLETE

Specifies that the system is to check for required parameters and supply defaults for omitted optional parameters.

Syntax for REQUEST=USERTERM

name

name: symbol. Begin *name* in column 1.

b

One or more blanks must precede CELAAUTH.

CELAAUTH

b

One or more blanks must follow CELAAUTH.

REQUEST=USERTERM

,ENVTOKEN=envtoken

envtoken: RS-type address or address in register (2) - (12)

,RECOVERY=EUTFRR
,RECOVERY=ESTAE

Default: RECOVERY=EUTFRR

,RETCODE=retcode

retcode: RS-type address or register (2) - (12).

,RSNCODE=rsncode

rsncode: RS-type address or register (2) - (12).

,PLISTVER=IMPLIED_VERSION
,PLISTVER=MAX
,PLISTVER=0

Default: PLISTVER=IMPLIED_VERSION

,MF=S

Default: MF=S

,MF=(L,list addr)

list addr: RS-type address or register (1) - (12)

,MF=(L,list addr,attr)

,MF=(L,list addr,0D)

,MF=(E,list addr)

,MF=(E,list addr,COMPLETE)

Parameters for REQUEST=USERTERM

The parameters are explained as follows:

name

is an optional symbol, starting in column 1, that is the name on the CELAAUTH macro invocation. The name must conform to the rules for an ordinary assembler language symbol. The default is no name.

REQUEST=USERTERM

REQUEST=USERTERM ends a user-managed environment.

,ENVTOKEN=envtoken

A required input parameter that contains the environment token which identifies the environment to be terminated.

To code: Specify the RS-type address, or address in register (2)-(12), of a 16-character field.

,RECOVERY=EUTFRR

,RECOVERY=ESTAE

An optional parameter indicating the type of recovery routine to be set by CELAAUTH for this invocation.

,RECOVERY=EUTFRR

indicates that an EUTFRR will be set.

,RECOVERY=ESTAE

indicates that an ESTAE will be set. When invoked in SRB mode, an EUTFRR is always used.

,RETCODE=retcode

An optional output parameter into which the return code is to be copied from GPR 15.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,RSNCODE=rsncode

An optional output parameter into which the reason code is to be copied from GPR 0.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,PLISTVER=IMPLIED_VERSION

,PLISTVER=MAX

,PLISTVER=0

An optional input parameter that specifies the version of the macro. PLISTVER determines which parameter list the system generates. PLISTVER is an optional input parameter on all forms of the macro, including the list form. When using PLISTVER, specify it on all macro forms used for a request and with the same value on all of the macro forms. The values are:

- **IMPLIED_VERSION**, which is the lowest version that allows all parameters specified on the request to be processed. If you omit the PLISTVER parameter, IMPLIED_VERSION is the default.
- **MAX**, if you want the parameter list to be the largest size currently possible. This size might grow from release to release and affect the amount of storage that your program needs.

If you can tolerate the size change, IBM recommends that you always specify PLISTVER=MAX on the list form of the macro. Specifying MAX ensures that the list-form parameter list is always long enough to hold all the parameters you might specify on the execute form, when both are assembled with the same level of the system. In this way, MAX ensures that the parameter list does not overwrite nearby storage.

- **0**, if you use the currently available parameters.

CELAAUTH

To code: Specify one of the following:

- IMPLIED_VERSION
- MAX
- A decimal value of 0

,MF=S
,MF=(L, list addr)
,MF=(L, list addr, attr)
,MF=(L, list addr, 0D)
,MF=(E, list addr)
,MF=(E, list addr, COMPLETE)

An optional input parameter that specifies the macro form.

Use MF=S to specify the standard form of the macro, which builds an inline parameter list and generates the macro invocation to transfer control to the service. MF=S is the default.

Use MF=L to specify the list form of the macro. Use the list form together with the execute form of the macro for applications that require reentrant code. The list form defines an area of storage that the execute form uses to store the parameters. Only the PLISTVER parameter may be coded with the list form of the macro.

Use MF=E to specify the execute form of the macro. Use the execute form together with the list form of the macro for applications that require reentrant code. The execute form of the macro stores the parameters into the storage area defined by the list form, and generates the macro invocation to transfer control to the service.

,list addr

The name of a storage area to contain the parameters. For MF=S and MF=E, this can be an RS-type address or an address in register (1)-(12).

,attr

An optional 1- to 60-character input string that you use to force boundary alignment of the parameter list. Use a value of 0F to force the parameter list to a word boundary, or 0D to force the parameter list to a doubleword boundary. If you do not code *attr*, the system provides a value of 0D.

,COMPLETE

Specifies that the system is to check for required parameters and supply defaults for omitted optional parameters.

Syntax for REQUEST=MNGDINIT

<i>name</i>	<i>name</i> : symbol. Begin <i>name</i> in column 1.
b	One or more blanks must precede CELAAUTH.
CELAAUTH	
b	One or more blanks must follow CELAAUTH.

REQUEST=MNGDINIT

,MENVID=*menvid* *menvid*: RS-type address or address in register (2) - (12)

,WRKJSTCB=SYSRULES **Default:** WRKJSTCB=SYSRULES
 ,WRKJSTCB=NO
 ,WRKJSTCB=YES

,ENVDEFN=*envdefn* *envdefn*: RS-type address or address in register (2) - (12)

,RECOVERY=EUTFRR **Default:** RECOVERY=EUTFRR
 ,RECOVERY=ESTAE

,RETCODE=*retcode* *retcode*: RS-type address or register (2) - (12).

,RSNCODE=*rsncode* *rsncode*: RS-type address or register (2) - (12).

,PLISTVER=IMPLIED_VERSION **Default:** PLISTVER=IMPLIED_VERSION
 ,PLISTVER=MAX
 ,PLISTVER=0

,MF=**S** **Default:** MF=S
 ,MF=(L,*list addr*) *list addr*: RS-type address or register (1) - (12)
 ,MF=(L,*list addr,attr*)
 ,MF=(L,*list addr,OD*)
 ,MF=(E,*list addr*)
 ,MF=(E,*list addr,COMPLETE*)

Parameters for REQUEST=MNGDINIT

The parameters are explained as follows:

name

is an optional symbol, starting in column 1, that is the name on the CELAAUTH macro invocation. The name must conform to the rules for an ordinary assembler language symbol. The default is no name.

REQUEST=MNGDINIT

REQUEST=MNGDINIT creates Preinitialized Environments for Authorized Programs in the current primary address space. This set of environments is used to run routines during subsequent CELAAUTH MNGDCALL requests. Only one set of managed environments using a specific managed environment token can be established for an address space. All subsequent CELAAUTH MNGDINIT calls using the same token will be unsuccessful.

,MENVID=*menvid*

A required input parameter containing the 8-byte ID that the caller wishes to use to uniquely identify the new set of managed environments. This ID is used on subsequent calls to CELAAUTH MNGDCALL to identify the set of environments to be used with the call. The contents of this ID is completely up to the caller. It can be a pointer to a control block, or a sequence of characters uniquely identifying the set. When using characters, IBM recommends that the caller begin the character sequence with its component ID, to help ensure uniqueness. IBM reserves managed environment IDs beginning with the characters "CEE", "CEL", and "EDC", for Language Environment's own use.

CELAAUTH

To code: Specify the RS-type address, or address in register (2)-(12), of an 8-character field.

,WRKJSTCB=SYSRULES

,WRKJSTCB=NO

,WRKJSTCB=YES

An optional parameter that indicates the job step attribute of any task that CELAAUTH may attach to the Cross Memory Resource Owning task. This keyword is only necessary if this task will be the first subtask attached under this task, so the job step attribute is not clear. The default is WRKJSTCB=SYSRULES.

,WRKJSTCB=SYSRULES

indicates that CELAAUTH will determine the proper job step attribute with which to attach the task. CELAAUTH will use the job step attribute of the current subtasks under the Cross Memory Resource Owning Task. If no subtasks are found, then CELAAUTH will attach the task as non-job step.

,WRKJSTCB=NO

indicates that the task must be a non-job step task.

,WRKJSTCB=YES

indicates that the attached task must be a job step task.

,ENVDEFN=*envdefn*

A required input parameter containing the address of the authorized environment definition table (AEDT). This table is built by the caller. It defines the characteristics for the environments that are to be created, as well as how they should be managed. Each table must contain a header, as well as one or more environment definition entries (AEDE). An environment definition entry describes how and when an environment will be created. On later CELAAUTH REQUEST(MNGDCALL) calls, the caller must specify the index of the AEDE that is to be used to create or locate an environment to be used when calling the routine. All fields within the table must be set. There are no default values.

The AEDT and AEDE are mapped by the CEEAEDT macro. Refer to this macro for the complete details of the AEDT structure.

The header for the AEDT contains the following information:

AEDT_ID

CHAR(4) Table eyecatcher 'AEDT'

AEDT_VERSION

FIXED(16) Version number of the table

AEDT_FLAGS

BIT(16)

AEDT_NUMEDE

FIXED(64) Number of environment definition entries in this table

AEDT_DIAGRTN

PTR(64) Address of an optional diagnostic routine that can be provided by the authorized application. This routine is called by the Preinitialized Environments for Authorized Programs recovery routine after determining that a dump is to be taken after an abend or program check occurs while an application routine is in control within an authorized environment. This gives the application an opportunity to capture diagnostic information about the error.

The routine gains control in AMODE 64, supervisor state, key 0, in the dispatchable unit and cross memory mode at time of failure, using standard Format 4 save area (F4SA) linkage.

When the diagnostic routine is called, the relevant register contents are:

Register

Contents

- 0 The address of an 8-byte buffer that contains the diagnostic token the user specified in the AEDT during system-managed initialization
- 1 The address of the SDWA that was provided to the Preinitialized Environments for Authorized Programs recovery routine
- 13 The address of a Format 4 save area
- 14 The return address
- 15 The address of the diagnostic routine

When the diagnostic routine returns control to its caller, the relevant register contents are:

Register

Contents

- 15 A value set by the application diagnostic routine that indicates the actions that it wants the Preinitialized Environments for Authorized Programs recovery routine to take as a result of its processing of the error condition. The following values can be returned in register 15:
 - 0 The application requests that Preinitialized Environments for Authorized Programs take a dump for this problem, which then occurs.
 - 4 The application captured appropriate diagnostic information. Preinitialized Environments for Authorized Programs continues with its own error recovery processing, but does not take any additional dumps for this problem.

Note:

1. All other values returned by the diagnostic routine are treated as if 0 had been returned.
2. This field is ignored if AEDT_VERSION is less than #AEDTVersion2.

AEDT_DIAGTKN

CHAR(8) An optional 8-byte token that is associated with the application's diagnostic routine. The token is provided to the diagnostic routine when it is called. The contents of this token is completely up to the caller. It is typically used to anchor an application control block the diagnostic routine can use to locate application data. This field is ignored if AEDT_DIAGRTN is zero.

Note: This is field ignored if AEDT_VERSION is less than #AEDTVersion2.

CELAAUTH

Each Environment Definition entry contains the following information:

AEDE_FLAGS

BIT(32)

AEDE_FULLINIT

BIT(1), '1... ..' within AEDE_FLAGS. Indicates whether each environment is to be fully initialized during this call. '1'b indicates full initialization is requested. '0'b indicates that no initialization is requested; environment initialization will occur upon first use of each environment.

AEDE_INIT

FIXED(64) Number of environments to create initially. Minimum value is 1.

AEDE_WTIME

FIXED(64) Amount of time, in microseconds, to wait if no environments are available. Minimum value is 0. 0 indicates that no wait is to be performed. CELAAUTH uses this value to wait once for an available environment prior to attempting to increment the number of environments. If the maximum number of environments has been reached, CELAAUTH waits once more for an available environment, using the specified time.

AEDE_INCR

FIXED(64) Number of environments to create incrementally, when more are needed. Minimum value is 0.

AEDE_MAX

FIXED(64) Maximum number of environments for this Environment Definition Entry.

AEDE_RTO

PTR(64) Pointer to a field containing the runtime options to be used with the environments for this Environment Definition Entry. The length of the runtime options cannot exceed 4096 characters. When no runtime options are provided, this field should be set to zero. Other fields in the AEDE must be set to either the value used to initialize the environment or set to zero.

AEDE_RTOLN

FIXED(64) Length of the runtime option string pointed to by AEDE_RTO. Other fields in the AEDE must be set to either the value used to initialize the environment set or zero.

To code: Specify the RS-type address, or address in register (2)-(12), of an eight-byte pointer field.

,RECOVERY=EUTFRR

,RECOVERY=ESTAE

An optional parameter indicating the type of recovery routine to be set by CELAAUTH for this invocation.

,RECOVERY=EUTFRR

indicates that an EUTFRR will be set.

,RECOVERY=ESTAE

indicates that an ESTAE will be set. When invoked in SRB mode, an EUTFRR is always used.

,RETCODE=retcode

An optional output parameter into which the return code is to be copied from GPR 15.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,RSNCODE=rsncode

An optional output parameter into which the reason code is to be copied from GPR 0.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,PLISTVER=IMPLIED_VERSION

,PLISTVER=MAX

,PLISTVER=0

An optional input parameter that specifies the version of the macro. PLISTVER determines which parameter list the system generates. PLISTVER is an optional input parameter on all forms of the macro, including the list form. When using PLISTVER, specify it on all macro forms used for a request and with the same value on all of the macro forms. The values are:

- **IMPLIED_VERSION**, which is the lowest version that allows all parameters specified on the request to be processed. If you omit the PLISTVER parameter, IMPLIED_VERSION is the default.
- **MAX**, if you want the parameter list to be the largest size currently possible. This size might grow from release to release and affect the amount of storage that your program needs.

If you can tolerate the size change, IBM recommends that you always specify PLISTVER=MAX on the list form of the macro. Specifying MAX ensures that the list-form parameter list is always long enough to hold all the parameters you might specify on the execute form, when both are assembled with the same level of the system. In this way, MAX ensures that the parameter list does not overwrite nearby storage.

- **0**, if you use the currently available parameters.

To code: Specify one of the following:

- IMPLIED_VERSION
- MAX
- A decimal value of 0

,MF=S

,MF=(L,list addr)

,MF=(L,list addr,attr)

,MF=(L,list addr,0D)

,MF=(E,list addr)

,MF=(E,list addr,COMPLETE)

An optional input parameter that specifies the macro form.

Use MF=S to specify the standard form of the macro, which builds an inline parameter list and generates the macro invocation to transfer control to the service. MF=S is the default.

Use MF=L to specify the list form of the macro. Use the list form together with the execute form of the macro for applications that require reentrant code. The list form defines an area of storage that the execute form uses to store the parameters. Only the PLISTVER parameter may be coded with the list form of the macro.

Use MF=E to specify the execute form of the macro. Use the execute form together with the list form of the macro for applications that require reentrant

CELAAUTH

code. The execute form of the macro stores the parameters into the storage area defined by the list form, and generates the macro invocation to transfer control to the service.

,list addr

The name of a storage area to contain the parameters. For MF=S and MF=E, this can be an RS-type address or an address in register (1)-(12).

,attr

An optional 1- to 60-character input string that you use to force boundary alignment of the parameter list. Use a value of 0F to force the parameter list to a word boundary, or 0D to force the parameter list to a doubleword boundary. If you do not code *attr*, the system provides a value of 0D.

,COMPLETE

Specifies that the system is to check for required parameters and supply defaults for omitted optional parameters.

Syntax for REQUEST=MNGDCALL

name

name: symbol. Begin *name* in column 1.

b

One or more blanks must precede CELAAUTH.

CELAAUTH

b

One or more blanks must follow CELAAUTH.

REQUEST=MNGDCALL

,RTNNAME=rtnname

rtnname: RS-type address or address in register (2) - (12)

,RTNADDR=rtnaddr

rtnaddr: RS-type address or address in register (2) - (12)

,RNAMELEN=rnamelen

rnamelen: RS-type address or address in register (2) - (12)

,DLLNAME=dllname

dllname: RS-type address or address in register (2) - (12)

,RTO=rto

rto: RS-type address or address in register (2) - (12)

,RTOLEN=rtolen

rtolen: RS-type address or address in register (2) - (12)

,PARMLIST=parmlist

parmlist: RS-type address or address in register (2) - (12)

,RTNTOKEN=rtntoken

rtntoken: RS-type address or address in register (2) - (12)

,MENVID=menvid

menvid: RS-type address or address in register (2) - (12)

,ETINDEX=etindex

etindex: RS-type address or address in register (2) - (12)

,RTNRETCODE=rtnretcode

rtnretcode: RS-type address or address in register (2) - (12)

,RTNRSNCODE=rtnrsncode

rtnrsncode: RS-type address or address in register (2) - (12)

<code>,RTNFDBKCODE=<i>rtnfdbkcode</i></code>	<i>rtnfdbkcode</i> : RS-type address or address in register (2) - (12)
<code>,RECOVERY=<u>EUTFRR</u></code> <code>,RECOVERY=ESTAE</code>	Default: RECOVERY=EUTFRR
<code>,RETCODE=<i>retcode</i></code>	<i>retcode</i> : RS-type address or register (2) - (12).
<code>,RSNCODE=<i>rsncode</i></code>	<i>rsncode</i> : RS-type address or register (2) - (12).
<code>,PLISTVER=<u>IMPLIED_VERSION</u></code> <code>,PLISTVER=MAX</code> <code>,PLISTVER=0</code>	Default: PLISTVER=IMPLIED_VERSION
<code>,MF=<u>S</u></code> <code>,MF=(L,<i>list addr</i>)</code> <code>,MF=(L,<i>list addr</i>,<i>attr</i>)</code> <code>,MF=(L,<i>list addr</i>,<u>OD</u>)</code> <code>,MF=(E,<i>list addr</i>)</code> <code>,MF=(E,<i>list addr</i>,<u>COMPLETE</u>)</code>	Default: MF=S <i>list addr</i> : RS-type address or register (1) - (12)

Parameters for REQUEST=MNGDCALL

The parameters are explained as follows:

name

is an optional symbol, starting in column 1, that is the name on the CELAAUTH macro invocation. The name must conform to the rules for an ordinary assembler language symbol. The default is no name.

REQUEST=MNGDCALL

REQUEST=MNGDCALL indicates that CELAAUTH invoke the specified C, C++, or Language Environment-conforming Assembler routine, using an environment from a set of system-managed environments.

`,RTNNAME=rtnname`

`,RTNADDR=rtnaddr`

A required input parameter.

`,RTNNAME=rtname`

A parameter containing the 1-1024 character name of the routine to be called. The length of this name is provided via the RNAMELEN keyword. When DLLNAME is specified, then RTNNAME must be the name of a function which has been exported from the DLL. Otherwise, RTNNAME must be the name of a main() program or fetchable routine, and is limited to 8 characters in length.

To code: Specify the RS-type address, or address in register (2)-(12), of a character field.

`,RTNADDR=rtnaddr`

A parameter containing the address of the routine to be called.

To code: Specify the RS-type address, or address in register (2)-(12), of an eight-byte pointer field.

CELAAUTH

,RNAMELEN=*rnamelen*

When RTNNAME=*rtname* is specified, a required input parameter containing the length of the routine name specified on the RTNNAME keyword.

To code: Specify the RS-type address, or address in register (2)-(12), of a doubleword field. *rnamelen* must be in the range 1 through 1024.

,DLLNAME=*dllname*

When RTNNAME=*rtname* is specified, an optional input parameter containing the 1-8 character name, padded with blanks, of the DLL from which the function name specified on the RTNNAME keyword has been exported.

To code: Specify the RS-type address, or address in register (2)-(12), of an 8-character field.

,RTO=*rto*

An optional input parameter containing the runtime options to be associated with the environment when the specified main is called. The length of the runtime options cannot exceed 4096 characters. This value is ignored if the routine to be called is a subroutine.

To code: Specify the RS-type address, or address in register (2)-(12), of a character field.

,RTOLEN=*rtolen*

When RTO=*rto* is specified, a required input parameter containing the length of the runtime option string pointed to by RTO.

To code: Specify the RS-type address, or address in register (2)-(12), of a doubleword field.

,PARMLIST=*parmlist*

An optional input parameter containing the parameter list to be passed to the routine that is to be called.

- The parameter list is copied to the appropriate location in the stack frame.
- general purpose registers 1, 2 and 3 are loaded from the parameter list when the routine is executed.

The length of the parameter list is determined from the PPA1 of the routine. If the routine takes a variable length parameter list, the length of the parameter list is assumed to be 256 bytes. Floating point and complex values can only be passed by reference.

To code: Specify the RS-type address, or address in register (2)-(12), of a character field.

,RTNTOKEN=*rtntoken*

An optional input/output parameter containing the token that identifies the routine to be called. This token is built upon first invocation of the routine within this set of environments, and returned to the caller for use on subsequent calls using the same set of environments.

To code: Specify the RS-type address, or address in register (2)-(12), of a 16-character field.

,MENVID=*menvid*

A required input parameter containing the Managed Environment ID that identifies the set of environments to be used for this call. This is the same ID that had been provided during the call to CELAAUTH MNGDINIT call.

To code: Specify the RS-type address, or address in register (2)-(12), of an 8-character field.

,ETINDEX=etindex

A required input parameter containing the index of the Environment Definition Entry to be used when calling this routine. This value corresponds to one of the Environment Definition Entries that were defined within the Environment Definition Table that was an input on a previous CELAAUTH REQUEST=MNGDINIT call.

To code: Specify the RS-type address, or address in register (2)-(12), of a fullword field.

,RTNRETCODE=rtnretcode

A required output parameter that is to contain the routine return code after the routine has completed. For a main(), this is the enclave return code. For a subroutine, this is the value that the subroutine provided on the return statement that caused the subroutine to end. If the subroutine caused the enclave to terminate due to an unhandled condition or a call to exit(), then this is the enclave return code. For more information on the enclave return code, see *z/OS Language Environment Programming Guide*.

To code: Specify the RS-type address, or address in register (2)-(12), of a fullword field.

,RTNRSNCODE=rtnrsncode

A required output parameter that is to contain the routine reason code after the routine has completed. This value is 0 if the routine ended normally. If the enclave is terminated due to an unhandled condition or a call to exit(), then this is the enclave reason code. For more information on the enclave return code, see *z/OS Language Environment Programming Guide*.

To code: Specify the RS-type address, or address in register (2)-(12), of a fullword field.

,RTNFDBKCODE=rtnfdbkcode

A required output parameter that is to contain the condition token indicating why the application terminated. For normal completion of the routine, CEE000 is returned. If the enclave is terminated due to an unhandled condition or a call to exit(), this field contains the enclave feedback code for termination.

To code: Specify the RS-type address, or address in register (2)-(12), of a 16-character field.

,RECOVERY=EUTFRR

,RECOVERY=ESTAE

An optional parameter indicating the type of recovery routine to be set by CELAAUTH for this invocation.

,RECOVERY=EUTFRR

indicates that an EUTFRR will be set.

,RECOVERY=ESTAE

indicates that an ESTAE will be set. When invoked in SRB mode, an EUTFRR is always used.

,RETCODE=retcode

An optional output parameter into which the return code is to be copied from GPR 15.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,RSNCODE=rsncode

An optional output parameter into which the reason code is to be copied from GPR 0.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,PLISTVER=IMPLIED_VERSION
,PLISTVER=MAX
,PLISTVER=0

An optional input parameter that specifies the version of the macro. PLISTVER determines which parameter list the system generates. PLISTVER is an optional input parameter on all forms of the macro, including the list form. When using PLISTVER, specify it on all macro forms used for a request and with the same value on all of the macro forms. The values are:

- **IMPLIED_VERSION**, which is the lowest version that allows all parameters specified on the request to be processed. If you omit the PLISTVER parameter, IMPLIED_VERSION is the default.
- **MAX**, if you want the parameter list to be the largest size currently possible. This size might grow from release to release and affect the amount of storage that your program needs.

If you can tolerate the size change, IBM recommends that you always specify PLISTVER=MAX on the list form of the macro. Specifying MAX ensures that the list-form parameter list is always long enough to hold all the parameters you might specify on the execute form, when both are assembled with the same level of the system. In this way, MAX ensures that the parameter list does not overwrite nearby storage.

- **0**, if you use the currently available parameters.

To code: Specify one of the following:

- IMPLIED_VERSION
- MAX
- A decimal value of 0

,MF=S
,MF=(L,list addr)
,MF=(L,list addr,attr)
,MF=(L,list addr,0D)
,MF=(E,list addr)
,MF=(E,list addr,COMPLETE)

An optional input parameter that specifies the macro form.

Use MF=S to specify the standard form of the macro, which builds an inline parameter list and generates the macro invocation to transfer control to the service. MF=S is the default.

Use MF=L to specify the list form of the macro. Use the list form together with the execute form of the macro for applications that require reentrant code. The list form defines an area of storage that the execute form uses to store the parameters. Only the PLISTVER parameter may be coded with the list form of the macro.

Use MF=E to specify the execute form of the macro. Use the execute form together with the list form of the macro for applications that require reentrant code. The execute form of the macro stores the parameters into the storage area defined by the list form, and generates the macro invocation to transfer control to the service.

,list addr

The name of a storage area to contain the parameters. For MF=S and MF=E, this can be an RS-type address or an address in register (1)-(12).

,attr

An optional 1- to 60-character input string that you use to force boundary

alignment of the parameter list. Use a value of 0F to force the parameter list to a word boundary, or 0D to force the parameter list to a doubleword boundary. If you do not code *attr*, the system provides a value of 0D.

,COMPLETE

Specifies that the system is to check for required parameters and supply defaults for omitted optional parameters.

Syntax for REQUEST=MNGDUPDT

<i>name</i>	<i>name</i> : symbol. Begin <i>name</i> in column 1.
b	One or more blanks must precede CELAAUTH.
CELAAUTH	
b	One or more blanks must follow CELAAUTH.

REQUEST=MNGDUPDT	
,MENVID= <i>menvid</i>	<i>menvid</i> : RS-type address or address in register (2) - (12)
,ENVDEFN= <i>xenvdefn</i>	
,RECOVERY= EUTFRR	Default: RECOVERY=EUTFRR
,RECOVERY= ESTAE	
,RETCODE= <i>retcode</i>	<i>retcode</i> : RS-type address or register (2) - (12).
,RSNCODE= <i>rsncode</i>	<i>rsncode</i> : RS-type address or register (2) - (12).
,PLISTVER= IMPLIED_VERSION	Default: PLISTVER=IMPLIED_VERSION
,PLISTVER=MAX	
,PLISTVER=0	
,MF= S	Default: MF=S
,MF=(L, <i>list addr</i>)	<i>list addr</i> : RS-type address or register (1) - (12)
,MF=(L, <i>list addr</i> , <i>attr</i>)	
,MF=(L, <i>list addr</i> , 0D)	
,MF=(E, <i>list addr</i>)	
,MF=(E, <i>list addr</i> , COMPLETE)	

Parameters for REQUEST=MNGDUPDT

The parameters are explained as follows:

name

is an optional symbol, starting in column 1, that is the name on the CELAAUTH macro invocation. The name must conform to the rules for an ordinary assembler language symbol. The default is no name.

REQUEST=MNGDUPDT

REQUEST=MNGDUPDT indicates that Preinitialized Environments for Authorized Programs will update the characteristics of a system-managed environment set.

,MENVID=*xmenv id*

A required input parameter containing the Managed Environment ID that identifies the set of environments to be updated. This is the same ID that had been provided during the call to CELAAUTH MNGDINIT call.

To code: Specify the RS-type address, or address in register (2)-(12), of an 8-character field.

,ENVDEFN=*envdefn*

A required input parameter containing the address of the authorized environment definition table (AEDT). This table is built by the caller. It indicates which characteristics for each environment definition entry are to be updated. The table must contain the same number of entries as the one used to initialize the environment set being updated.

The following characteristics can be updated:

AEDE_MAX

FIXED(64) Maximum number of environments for this Environment Definition Entry. Minimum value is 1.

- If AEDE_MAX is equal to 0, or matches the current maximum number of environments, no update is performed.
- If AEDE_MAX is larger than the current maximum number of environments, Preinitialized Environments for Authorized Programs updates the maximum number.
- If AEDE_MAX is smaller than the current maximum number of environments, CELAAUTH returns a non-zero return code to indicate that the request update cannot be performed.

AEDE_RTO

PTR(64) Pointer to a field containing the runtime options to be used with the environments for this Environment Definition Entry. The length of the runtime options cannot exceed 4096 characters. When no runtime options are provided, this field should be set to zero.

AEDE_RTOLN

FIXED(64) Length of the runtime option string pointed to by AEDE_RTO.

To code: Specify the RS-type address, or address in register (2)-(12), of an eight-byte pointer field.

,RECOVERY=EUTFRR

,RECOVERY=ESTAE

An optional parameter indicating the type of recovery routine to be set by CELAAUTH for this invocation.

,RECOVERY=EUTFRR

indicates that an EUTFRR will be set.

,RECOVERY=ESTAE

indicates that an ESTAE will be set. When invoked in SRB mode, an EUTFRR is always used.

,RETCODE=retcode

An optional output parameter into which the return code is to be copied from GPR 15.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,RSNCODE=rsncode

An optional output parameter into which the reason code is to be copied from GPR 0.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,PLISTVER=IMPLIED_VERSION

,PLISTVER=MAX

,PLISTVER=0

An optional input parameter that specifies the version of the macro. PLISTVER determines which parameter list the system generates. PLISTVER is an optional input parameter on all forms of the macro, including the list form. When using PLISTVER, specify it on all macro forms used for a request and with the same value on all of the macro forms. The values are:

- **IMPLIED_VERSION**, which is the lowest version that allows all parameters specified on the request to be processed. If you omit the PLISTVER parameter, IMPLIED_VERSION is the default.
- **MAX**, if you want the parameter list to be the largest size currently possible. This size might grow from release to release and affect the amount of storage that your program needs.

If you can tolerate the size change, IBM recommends that you always specify PLISTVER=MAX on the list form of the macro. Specifying MAX ensures that the list-form parameter list is always long enough to hold all the parameters you might specify on the execute form, when both are assembled with the same level of the system. In this way, MAX ensures that the parameter list does not overwrite nearby storage.

- **0**, if you use the currently available parameters.

To code: Specify one of the following:

- IMPLIED_VERSION
- MAX
- A decimal value of 0

,MF=S

,MF=(L,list addr)

,MF=(L,list addr,attr)

,MF=(L,list addr,0D)

,MF=(E,list addr)

,MF=(E,list addr,COMPLETE)

An optional input parameter that specifies the macro form.

Use MF=S to specify the standard form of the macro, which builds an inline parameter list and generates the macro invocation to transfer control to the service. MF=S is the default.

Use MF=L to specify the list form of the macro. Use the list form together with the execute form of the macro for applications that require reentrant code. The list form defines an area of storage that the execute form uses to store the parameters. Only the PLISTVER parameter may be coded with the list form of the macro.

Use MF=E to specify the execute form of the macro. Use the execute form together with the list form of the macro for applications that require reentrant

CELAAUTH

code. The execute form of the macro stores the parameters into the storage area defined by the list form, and generates the macro invocation to transfer control to the service.

,list addr

The name of a storage area to contain the parameters. For MF=S and MF=E, this can be an RS-type address or an address in register (1)-(12).

,attr

An optional 1- to 60-character input string that you use to force boundary alignment of the parameter list. Use a value of 0F to force the parameter list to a word boundary, or 0D to force the parameter list to a doubleword boundary. If you do not code *attr*, the system provides a value of 0D.

,COMPLETE

Specifies that the system is to check for required parameters and supply defaults for omitted optional parameters.

Syntax for REQUEST=MNGDTERM

<i>name</i>	<i>name</i> : symbol. Begin <i>name</i> in column 1.
b	One or more blanks must precede CELAAUTH.
CELAAUTH	
b	One or more blanks must follow CELAAUTH.

REQUEST=MNGDTERM

<i>,MENVID=menvid</i>	<i>menvid</i> : RS-type address or address in register (2) - (12)
<i>,RECOVERY=<u>EUTFRR</u></i> <i>,RECOVERY=ESTAE</i>	Default: RECOVERY=EUTFRR
<i>,RETCODE=retcode</i>	<i>retcode</i> : RS-type address or register (2) - (12).
<i>,RSNCODE=rsncode</i>	<i>rsncode</i> : RS-type address or register (2) - (12).
<i>,PLISTVER=<u>IMPLIED_VERSION</u></i> <i>,PLISTVER=MAX</i> <i>,PLISTVER=0</i>	Default: PLISTVER=IMPLIED_VERSION
<i>,MF=<u>S</u></i> <i>,MF=(L,<i>list addr</i>)</i> <i>,MF=(L,<i>list addr</i>,<i>attr</i>)</i> <i>,MF=(L,<i>list addr</i>,<u>0D</u>)</i> <i>,MF=(E,<i>list addr</i>)</i> <i>,MF=(E,<i>list addr</i>,<u>COMPLETE</u>)</i>	Default: MF=S <i>list addr</i> : RS-type address or register (1) - (12)

Parameters for REQUEST=MNGDTERM

The parameters are explained as follows:

name

is an optional symbol, starting in column 1, that is the name on the CELAAUTH macro invocation. The name must conform to the rules for an ordinary assembler language symbol. The default is no name.

REQUEST=MNGDTERM

REQUEST=MNGDTERM ends a set of environments that were created using MNGDINIT.

,MENVID=*menvid*

A required input parameter containing the Managed Environment ID that identifies the set of environments to be terminated. This is the same ID that had been provided during the call to CELAAUTH MNGDINIT call.

To code: Specify the RS-type address, or address in register (2)-(12), of an 8-character field.

,RECOVERY=EUTFRR

,RECOVERY=ESTAE

An optional parameter indicating the type of recovery routine to be set by CELAAUTH for this invocation.

,RECOVERY=EUTFRR

indicates that an EUTFRR will be set.

,RECOVERY=ESTAE

indicates that an ESTAE will be set. When invoked in SRB mode, an EUTFRR is always used.

,RETCODE=*retcode*

An optional output parameter into which the return code is to be copied from GPR 15.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,RSNCODE=*rsncode*

An optional output parameter into which the reason code is to be copied from GPR 0.

To code: Specify the RS-type address of a fullword field, or register (2)-(12).

,PLISTVER=IMPLIED_VERSION

,PLISTVER=MAX

,PLISTVER=0

An optional input parameter that specifies the version of the macro. PLISTVER determines which parameter list the system generates. PLISTVER is an optional input parameter on all forms of the macro, including the list form. When using PLISTVER, specify it on all macro forms used for a request and with the same value on all of the macro forms. The values are:

- **IMPLIED_VERSION**, which is the lowest version that allows all parameters specified on the request to be processed. If you omit the PLISTVER parameter, IMPLIED_VERSION is the default.
- **MAX**, if you want the parameter list to be the largest size currently possible. This size might grow from release to release and affect the amount of storage that your program needs.

If you can tolerate the size change, IBM recommends that you always specify PLISTVER=MAX on the list form of the macro. Specifying MAX ensures that the list-form parameter list is always long enough to hold all

the parameters you might specify on the execute form, when both are assembled with the same level of the system. In this way, MAX ensures that the parameter list does not overwrite nearby storage.

- 0, if you use the currently available parameters.

To code: Specify one of the following:

- IMPLIED_VERSION
- MAX
- A decimal value of 0

,MF=S
,MF=(L, list addr)
,MF=(L, list addr, attr)
,MF=(L, list addr, 0D)
,MF=(E, list addr)
,MF=(E, list addr, COMPLETE)

An optional input parameter that specifies the macro form.

Use MF=S to specify the standard form of the macro, which builds an inline parameter list and generates the macro invocation to transfer control to the service. MF=S is the default.

Use MF=L to specify the list form of the macro. Use the list form together with the execute form of the macro for applications that require reentrant code. The list form defines an area of storage that the execute form uses to store the parameters. Only the PLISTVER parameter may be coded with the list form of the macro.

Use MF=E to specify the execute form of the macro. Use the execute form together with the list form of the macro for applications that require reentrant code. The execute form of the macro stores the parameters into the storage area defined by the list form, and generates the macro invocation to transfer control to the service.

,list addr

The name of a storage area to contain the parameters. For MF=S and MF=E, this can be an RS-type address or an address in register (1)-(12).

,attr

An optional 1- to 60-character input string that you use to force boundary alignment of the parameter list. Use a value of 0F to force the parameter list to a word boundary, or 0D to force the parameter list to a doubleword boundary. If you do not code *attr*, the system provides a value of 0D.

,COMPLETE

Specifies that the system is to check for required parameters and supply defaults for omitted optional parameters.

CELAAUTH general notes

- Each environment uses about 400 bytes of subpool 245 (common, fixed ESQA) storage. You should keep this in mind when considering the number of environments that your application creates.
- Preinitialized Environments for Authorized Programs forces POSIX(OFF) as the runtime option during execution.
- Preinitialized Environments for Authorized Programs does not support calls to z/OS UNIX System Services (USS).
- No locks can be held during calls to Preinitialized Environments for Authorized Programs.

- Preinitialized Environments for Authorized Programs services must only be used in full-function address spaces.

ABEND codes

None.

Return and reason codes

When the CELAAUTH macro returns control to your program:

- GPR 15 (and *retcode*, when you code RETCODE) contains a return code.
- When the value in GPR 15 is not zero, GPR 0 (and *rsncode*, when you code RSNCODE) contains reason code.

The CEEALRC macro provides constants for all of the CELAAUTH return and reason codes.

RC	Explanation
00	#ALRTN_SUCCESS - The call was successful.
04	#ALRTN_QUALIFIED_SUCCESS - The call was successful, but additional information was provided in the status code.
08	#ALRTN_INCORRECT_ENVIRONMENT - A problem was detected with the environment of the caller.
0C	#ALRTN_BAD_PARAMETERS - A problem was detected with one of the CELAAUTH parameters.
10	#ALRTN_RESOURCE_ERROR - A problem was detected with a resource needed by CELAAUTH Services.
14	#ALRTN_INTERNAL_ERROR - An internal CELAAUTH Services error occurred.

Table 89 on page 790 contains hexadecimal return and reason codes, the equate symbols associated with each reason code, and the meaning and suggested action for each return and reason code. Each reason code for the CELAAUTH macro is written in the format *mmmffrrr*, where:

mmm is the module ID where the reason code was set

ff is the function code of the CELAAUTH service in use

00: Internal processing

01: USERINIT - user managed initialization

02: USERCALL - user managed call

03: USERTERM - user managed termination

04: MNGDINIT - system managed initialization

05: MNGDCALL - system managed call

06: MNGDTERM - system managed termination

07: MNGDUPDT - system managed update

10: Internal processing

11: Internal processing

12: Internal processing

13: Internal processing

rrr is the value that identifies the reason for the non-zero return code

CELAAUTH

Table 89. Return and reason codes for the CELAAUTH macro

Reason Code (rrr)	Return Code	Meaning and Action
004	10	<p>Equate Symbol: #ALRC_SLE_MEMLIMIT_ZERO</p> <p>Explanation: CELAAUTH Services attempted to obtain library storage. However, memlimit was set to zero, so the call failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the MEMLIMIT setting for this address space is not zero.</p>
008	10	<p>Equate Symbol: #ALRC_SLE_NOISA</p> <p>Explanation: CELAAUTH Services attempted to obtain library storage, but the call failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Increase the MEMLIMIT size for the address space.</p>
00C	10	<p>Equate Symbol: #ALRC_SLE_NO_STACK</p> <p>Explanation: CELAAUTH Services attempted to obtain the initial stack and heap storage, but the call failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Increase the MEMLIMIT size for the address space.</p>
010	10	<p>Equate Symbol: #ALRC_SCSRG_ERR</p> <p>Explanation: When calling CELAAUTH's internal cellpool services to get storage for the runtime options, the service call failed. The module id and function code part of the reason code will help determine exactly where the service call failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough storage resources on the system.</p>
014	14	<p>Equate Symbol: #ALRC_SLE_TERM_FAILED</p> <p>Explanation: CELAAUTH attempted to terminate the preinitialized environment, but the call failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the system is running correctly at the time.</p>

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
100	10	<p>Equate Symbol: #ALRC_WRONG_KEY</p> <p>Explanation: CELAAUTH Services transferred control to the LE library, but the Language Environment library did not get control in the correct key. This is most likely be caused by CELQLIB, the LE 64-bit library, not in an authorized dataset.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure CELQLIB is placed in a dataset that is in the LPALST or LNKLST concatenation.</p>
104	14 or 0C	<p>Equate Symbol: #ALRC_9RCVY</p> <p>Explanation: While processing the request, CELAAUTH Services either program checked or abnormally ended, causing the recovery to get control. The recovery successfully recovered from the error, but the request failed because of the program check or abend.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: If return code is #ALRTN_BAD_PARAMETERS, ensure that the parameters are correct on the CELAAUTH call and that the storage for the parameters is in the correct key.</p>
108	04	<p>Equate Symbol: #ALRC_ENCLAVE_TERMINATED</p> <p>Explanation: On a CELAAUTH call subroutine or call dll subroutine, the enclave should not terminate. However, during this call, the enclave is terminated unexpectedly.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Make sure the called subroutine didn't cause any program checks or abends during its execution.</p>
10C	14	<p>Equate Symbol: #ALRC_UNHANDLED_CONDITION</p> <p>Explanation: An unhandled condition occurred during enclave initialization or enclave termination.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Contact IBM with details of the problem scenario.</p>

CELAAUTH

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
110	10 or 0C	<p>Equate Symbol: #ALRC_MODULE_LOAD_FAILED</p> <p>Explanation: If the return code is #ALRTN_RESOURCE_ERROR, that means there was an error calling the LOAD service. If the return code is #ALRTN_BAD_PARAMETERS, that means CELAAUTH's internal load service is called with bad parameters.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: If return code is #ALRTN_RESOURCE_ERROR, make sure there are enough system resources on the system. If return code is #ALRTN_BAD_PARAMETERS, take a dump when this return code is received, and contact IBM with the dump.</p>
114	10 or 0C	<p>Equate Symbol: #ALRC_MODULE_DELETE_FAILED</p> <p>Explanation: If the return code is #ALRTN_RESOURCE_ERROR, that means there was an error calling the DELETE service. If the return code is #ALRTN_BAD_PARAMETERS, that means CELAAUTH's internal delete service is called with bad parameters.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: If return code is #ALRTN_RESOURCE_ERROR, make sure there are enough system resources on the system. If return code is #ALRTN_BAD_PARAMETERS, take a dump when this return code is received, and contact IBM with the dump.</p>
118	0C	<p>Equate Symbol: #ALRC_MODULE_NOT_FOUND</p> <p>Explanation: On a CELAAUTH call by routine name call, CELAAUTH Services attempted a load of the module based on the routine name. However, the module with that name does not exist, the module load failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the routine name on the CELAAUTH call is correct.</p>
11C	0C	<p>Equate Symbol: #ALRC_DLLLOAD_FAILED</p> <p>Explanation: On a CELAAUTH call by dll, CELAAUTH Services attempted a load of a dll based on the given dll name. However, the dll with that name does not exist, the dll load failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the dll name on the CELAAUTH call is correct.</p>

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
120	0C	<p>Equate Symbol: #ALRC_DLLQUERYFN_FAILED</p> <p>Explanation: On a CELAAUTH call by dll subroutine, CELAAUTH Services attempted to query the dll subroutine address. However, either the subroutine is not exported, or it does not exist, the query failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the dll subroutine name on the CELAAUTH call is correct, and the subroutine is exported.</p>
124	14	<p>Equate Symbol: #ALRC_NEWMOD_FAILED</p> <p>Explanation: New module initialization failed for the module that contains the user routine.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Contact IBM with details of the problem scenario.</p>
128	14	<p>Equate Symbol: #ALRC_DLLINIT_FAILED</p> <p>Explanation: DLL static initialization failed for the module that contains the user routine.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Contact IBM with details of the problem scenario.</p>
12C	14	<p>Equate Symbol: #ALRC_STATCNST_FAILED</p> <p>Explanation: The running of C++ static constructors failed for the module that contains the user routine.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Contact IBM with details of the problem scenario.</p>
130	04	<p>Equate Symbol: #ALRC_APPL_ABEND</p> <p>Explanation: The user routine abended. The routine return code is set to the abend completion code from the SDWA (SDWAABCC). The routine reason code is set to the abend reason code from the SDWA (SDWACRC).</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Use the abend code and reason code to determine the problem.</p>

CELAAUTH

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
200	14	<p>Equate Symbol: #ALRC_LATCH_CREATE_FAILED</p> <p>Explanation: During CELAAUTH initialization, the service attempted to create internal latch sets to be used later. However, the latch set create failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough system resources on the system.</p>
300	0C	<p>Equate Symbol: #ALRC_INCORRECT_FUNCTION_CODE</p> <p>Explanation: The function code in the user parameter list is unknown.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Make sure CELAAUTH Services is called through the CELAAUTH or CELAUTHP macros.</p>
304	10	<p>Equate Symbol: #ALRC_INFRASTRUCT_STORAGE_OBTAIN_FAILED</p> <p>Explanation: CELAAUTH Services failed to get the storage for the central control block for CELAAUTH Services infrastructure.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough storage resources on the system.</p>
308	08	<p>Equate Symbol: #ALRC_IRB_SCHEDULE_FAILURE</p> <p>Explanation: CELAAUTH Services attempted to schedule an IRB to the ARO task, but the scheduling of the IRB failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough system resources on the system.</p>
30C	04	<p>Equate Symbol: #ALRC_ADDITIONAL_ENVIRONMENTS_EXIST</p> <p>Explanation: When CELAAUTH Services terminates a user managed or system managed environment, it checks to see if it should terminate the infrastructure as well. It found that there are still other environments initialized, so it skipped infrastructure termination. This reason code does not necessarily indicate an error, which is why the return code associated with it is #ALRTN_QUALIFIED_SUCCESS. This is only an error if the caller thinks that there shouldn't be any environment still initialized after the call.</p> <p>System Action: CELAAUTH returns this reason code back to the caller.</p> <p>Programmer Response: If the caller thinks this is an error, a dump should be taken, and contact IBM with the dump.</p>

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
310	08	<p>Equat Symbol: #ALRC_ENV_TOKEN_NOT_VALID</p> <p>Explanation: On a user managed call, the environment token passed in is not valid.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure a valid environment token is passed in.</p>
314	08	<p>Equat Symbol: #ALRC_AUTHLE_NOT_INITIALIZED</p> <p>Explanation: An attempt to call a CELAAUTH service is made; however, CELAAUTH Services infrastructure is not initialized.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure a USERINIT or MNGDINIT call is made successfully before any other CELAAUTH services are called.</p>
318	08	<p>Equat Symbol: #ALRC_AUTHLE_UNABLE_TO_INIT_IN_XMEM</p> <p>Explanation: Due to system restrictions in cross memory environment, CELAAUTH does not support initialization in cross memory environment. Any attempt to do so will receive this reason code back.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the CELAAUTH initialization call is not made in cross memory environment.</p>
31C	08	<p>Equat Symbol: #ALRC_AUTHLE_INIT_IN_PROGRESS_ERR</p> <p>Explanation: While the CELAAUTH Services infrastructure is initialized, no service requests are allowed until the infrastructure is fully initialized. Any attempt to call CELAAUTH while CELAAUTH Services initialization is in progress will result in this error reason code.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Wait a while before try calling CELAAUTH again. If the problem persists, take a dump, and contact IBM with the captured dump.</p>
320	08	<p>Equat Symbol: #ALRC_AUTHLE_ALREADY_INITIALIZED_ERR</p> <p>Explanation: If a system managed environment already exists, any attempt to initialize a system managed environment with the same environment id will result in this reason code.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Make sure the managed init should be done, and the environment id is correct. Or make sure the managed environment is terminated first before trying initialization again.</p>

CELAAUTH

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
324	08	<p>Equates Symbol: #ALRC_AUTHLE_TERM_IN_PROGRESS_ERR</p> <p>Explanation: If CELAAUTH Services is in the middle of terminating its infrastructure, any attempt to initialize CELAAUTH Services will result in this reason code.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Wait a while before try calling CELAAUTH Services initialization again. If the problem persists, take a dump, and contact IBM with the captured dump.</p>
328	10	<p>Equates Symbol: #ALRC_PET_ALLOCATE_ERROR</p> <p>Explanation: CELAAUTH Services attempted to allocate a Pause Element Token. However, the allocation failed. This is most likely due to a system resource shortage.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough system resources on the system.</p>
32C	14 or 10	<p>Equates Symbol: #ALRC_PET_PAUSE_ERROR</p> <p>Explanation: CELAAUTH Services attempted to pause on a Pause Element Token. However, the pause failed. This is most likely due to a system resource shortage.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough system resources on the system.</p>
330	14	<p>Equates Symbol: #ALRC_PET_RELEASE_ERROR</p> <p>Explanation: CELAAUTH Services attempted to release a Pause Element Token. However, the release failed. This is most likely due to a system resource shortage.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough system resources on the system.</p>
334	10	<p>Equates Symbol: #ALRC_PET_DEALLOCATE_ERROR</p> <p>Explanation: CELAAUTH Services attempted to deallocate a Pause Element Token. However, the deallocation failed. This is most likely due to a system resource shortage.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough system resources on the system.</p>

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
338	08	<p>Equates Symbol: #ALRC_LOCASCB_ERROR</p> <p>Explanation: While calling CELAAUTH in a cross memory environment, CELAAUTH Services attempted to locate the ASCB of the primary address space. However, the attempt failed. This is most likely the result of a system error.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the system is running correctly at the time.</p>
33C	0C	<p>Equates Symbol: #ALRC_ARO_TCB_NOT_VALID</p> <p>Explanation: During CELAAUTH Services infrastructure initialization, the caller passed in a Resource Owning Task TCB. However, the specified TCB is not valid.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the correct Resource Owning Task TCB is specified.</p>
340	0C	<p>Equates Symbol: #ALRC_ARO_TCB_NOT_BELOW_INIT</p> <p>Explanation: CELAAUTH Services initialization is not permitted on a task that's above the initiator task. Or CELAAUTH Services initialization is not permitted with a resource owning task that is above the initiator task. This is most likely cause by initializing CELAAUTH Services before the address space is fully initialized.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the address space is fully initialized, and the resource owning task is not above the initiator task.</p>
344	14	<p>Equates Symbol: #ALRC_WKRTASK_ATTACH_ERROR</p> <p>Explanation: During CELAAUTH Services infrastructure initialization, CELAAUTH attached a worker task to the CELAAUTH resource owning task. The attach of the worker task failed. This is most likely due to a shortage of system resources.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough system resource on the system.</p>
348	14	<p>Equates Symbol: #ALRC_CELQLIB_LOAD_FAILED</p> <p>Explanation: During CELAAUTH Services infrastructure initialization, CELAAUTH attempted to load the 64bit LE library. However, the load failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure CELQLIB is in the library load search path.</p>

CELAAUTH

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
34C	0C	<p>Equates Symbol: #ALRC_ENV_TOKEN_NOT_SPECIFIED</p> <p>Explanation: On a CELAAUTH user managed call or user managed term, the required parameter, environment token, is not specified.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the environment token is specified on the call.</p>
350	08	<p>Equates Symbol: #ALRC_AUTHLE_ENVIRONMENT_IN_USE</p> <p>Explanation: During CELAAUTH user managed call and user managed term, CELAAUTH Services attempted to exclusively lock the user managed environment, but the user managed environment is currently in use. Or during CELAAUTH system managed term, CELAAUTH Services attempted to exclusively lock the system managed environment, but the system managed environment is currently in use. Use the function code part of the reason code to determined the exact cause of this problem.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Wait a while for the environment to get freed up, before trying the service again. If the problem persists, take a dump, and contact IBM with the captured dump.</p>
354	0C	<p>Equates Symbol: #ALRC_RTOKEN_TOOBIG</p> <p>Explanation: The runtime option string passed in during initialization or call is too long.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Make sure the runtime options string length in the parameter list correctly reflect the length of the runtime option string. And the length of the runtime option string doesn't exceed the limit.</p>
358	08	<p>Equates Symbol: #ALRC_ENV_TOKEN_STALE</p> <p>Explanation: During a CELAAUTH user call or user term, the environment token specified belonged to a previous instance of the environment. This means the environment was terminated before this call was made, the environment token no longer refers to a valid environment.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the correct environment token is specified.</p>
35C	0C	<p>Equates Symbol: #ALRC_RTN_TOKEN_NOT_VALID</p> <p>Explanation: During a CELAAUTH user call or managed call, the routine token specified is not valid.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the correct routine token is specified.</p>

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
360	08	<p>Equate Symbol: #ALRC_RTN_TOKEN_STALE</p> <p>Explanation: During a CELAAUTH user call or manage call, the routine token specified belonged to a previous instance of the environment. This means the environment that this routine token belonged to was terminated before this call was made, the routine token no longer refers to a valid environment.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the correct routine token is specified.</p>
364	08	<p>Equate Symbol: #ALRC_RTN_ENV_TOKEN_MISMATCH</p> <p>Explanation: During a CELAAUTH user call, the routine token specified doesn't belong to the environment specified by the environment token.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the correct routine token and environment token are specified.</p>
368	0C	<p>Equate Symbol: #ALRC_RTN_NOT_MAIN_OR_FETCHABLE</p> <p>Explanation: On a call main or call sub, the module loaded does not contain a main or a fetchable subroutine.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the load module specified in the routine name field contains either a main or a fetchable routine.</p>
36C	0C	<p>Equate Symbol: #ALRC_RTN_NAME_LENGTH_ERROR</p> <p>Explanation: On a call main or call sub, the routine name length specified in the parameter is not valid.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the correct routine name length is specified.</p>
370	0C	<p>Equate Symbol: #ALRC_DLL_RTN_NAME_LENGTH_ERROR</p> <p>Explanation: On a call dll sub, the routine name length specified in the parameter is not valid.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure the correct routine name length is specified.</p>

CELAAUTH

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
374	10	<p>Equate Symbol: #ALRC_SCSRG_RTO_ERR</p> <p>Explanation: When calling CELAAUTH Services internal cellpool services to get storage for the runtime options, the service call failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough storage resources on the system.</p>
378	10	<p>Equate Symbol: #ALRC_SCSRG_RTO_GRP_ERR</p> <p>Explanation: When calling CELAAUTH Services internal cellpool services to get storage for a group of runtime options, the service call failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough storage resources on the system.</p>
37C	10	<p>Equate Symbol: #ALRC_SCSRG_ALEI_ERR</p> <p>Explanation: When calling CELAAUTH Services internal cellpool services to get storage for an environment information control block, the service call failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough storage resources on the system.</p>
380	10	<p>Equate Symbol: #ALRC_SCSRG_ALEI_GRP_ERR</p> <p>Explanation: When calling CELAAUTH Services internal cellpool services to get storage for a group of environment information control blocks, the service call failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough storage resources on the system.</p>
384	10	<p>Equate Symbol: #ALRC_SCSRG_AEDT_ERR</p> <p>Explanation: When calling CELAAUTH Services internal cellpool service to get the storage for an Authorized Environment Definition Table, the called service failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough storage resources on the system.</p>

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
388	10	<p>Equate Symbol: #ALRC_SCSRG_CP_CREATE_ERR</p> <p>Explanation: When calling CELAAUTH Services internal cellpool services to create the internal cellpools, the create failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough storage resources on the system.</p>
38C	10	<p>Equate Symbol: #ALRC_SCSRG_ALES_OBTAIN_ERR</p> <p>Explanation: When calling CELAAUTH Services internal cellpool services to get the storage for a System Managed control block, the called service failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure there are enough storage resources on the system.</p>
390	0C	<p>Equate Symbol: #ALRC_ENVNUM_MISMATCH</p> <p>Explanation: The init/incr/max environment numbers mismatch. There are 3 possibilities: 1) init = max and the incr ^= 0 2) init < max and the incr = 0 3) init > max</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Make sure the specified init/incr/max values in the AEDT are correct.</p>
394	0C	<p>Equate Symbol: #ALRC_RTO_MISMATCH</p> <p>Explanation: The RTO length specified in the AEDT is non-zero, while the RTO string in the AEDT is zero.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Make sure the RTO length and the RTO string are correct.</p>
398	0C	<p>Equate Symbol: #ALRC_NO_EDE_ENTRY</p> <p>Explanation: The user didn't specify any environment types in the parameter list on a CELAAUTH(MNGDINIT) call.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Make sure at least one AEDE entry is defined in the AEDT.</p>

CELAAUTH

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
39C	0C	<p>Equate Symbol: #ALRC_MENVID_NOT_FOUND</p> <p>Explanation: A set of environments with the managed environment ID specified on a CELAAUTH REQUEST(MNGDCALL) or REQUEST(MNGDTERM) could not be located.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that a set of environments for this managed environment ID has been created using CELAAUTH REQUEST(MNGDINIT).</p>
3A0	08	<p>Equate Symbol: #ALRC_ENV_SET_NOT_AVAILABLE</p> <p>Explanation: The requested environment set is not in a state in which it can be used by the current call. This probably indicates that the environment set is in the process of being terminated.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that no attempt is made to use an environment set while it is being terminated.</p>
3A4	10	<p>Equate Symbol: #ALRC_MALRI_SCSRG_ERR</p> <p>Explanation: Storage could not be obtained for a routine information control block.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Examine the information from the cell pool services call to determine the error.</p>
3A8	08	<p>Equate Symbol: #ALRC_ENV_SET_UNLOCK_FAILED</p> <p>Explanation: The requested environment set could not be properly unlocked.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Contact IBM with details of the problem scenario.</p>
3AC	0C	<p>Equate Symbol: #ALRC_BAD_MANAGED_ENV_ID</p> <p>Explanation: The managed environment id is invalid. Specify a valid env id.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Make sure the managed environment id not zero.</p>

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
3B0	10	<p>Equate Symbol: #ALRC_NO_AVAILABLE_ENVS</p> <p>Explanation: All existing environments within the requested environment type are in use, and no more environments can be created because the maximum number has been reached.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: If this reason code occurs frequently, consider increasing the maximum number of environments within the environment type entry (field AEDE_MAX in macro CEEAEDT).</p>
3B4	04	<p>Equate Symbol: #ALRC_NOT_ALL_ENVS_CREATED</p> <p>Explanation: The requested routine was successfully called. However, while attempting to create additional environments for the current environment type, one or more environments could not be created.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Examine the additional error information to determine the error.</p>
3B8	0C	<p>Equate Symbol: #ALRC_RTN_TOKEN_FAILURE</p> <p>Explanation: CELAAUTH Services attempted to use the routine token provided by the caller on a CELAAUTH REQUEST(USERCALL/MNGDCALL). A failure occurred while accessing this token.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that a valid routine token has been provided on the CELAAUTH call.</p>
3BC	0C	<p>Equate Symbol: #ALRC_AUTHLE_MVCSK_AEDE_NUM_ERR</p> <p>Explanation: While attempting to copy the AEDE number from the user passed in parameter list using the caller's key, CELAAUTH Services failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that a valid AEDT is specified on the CELAAUTH call.</p>
3C0	0C	<p>Equate Symbol: #ALRC_AUTHLE_MVCSK_AEDT_ERR</p> <p>Explanation: While attempting to copy the AEDT from the user passed in parameter list using the caller's key, CELAAUTH Services failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that a valid AEDT is specified on the CELAAUTH call.</p>

CELAAUTH

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
3C4	0C	<p>Equate Symbol: #ALRC_AUTHLE_MVCDK_ENVTOKEN_ERR</p> <p>Explanation: While attempting to store the environment token into the user passed parameter list, CELAAUTH Services failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that the environment token storage passed in on the CELAAUTH call is valid.</p>
3C8	0C	<p>Equate Symbol: #ALRC_AUTHLE_MVCSK_RTO_ERR</p> <p>Explanation: While attempting to copy the runtime option strings from user passed in parameter list using the caller's key, CELAAUTH Services failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that a valid runtime option string is specified on the CELAAUTH call.</p>
3CC	0C	<p>Equate Symbol: #ALRC_AUTHLE_MVCSK_RTNNNAME_ERR</p> <p>Explanation: While attempting to copy the routine name from the user parameter list, a failure occurred.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that a valid routine name is specified on the CELAAUTH call.</p>
3D0	0C	<p>Equate Symbol: #ALRC_AUTHLE_MVCSK_DLL_RTNNNAME_ERR</p> <p>Explanation: While attempting to copy the DLL routine name from the user parameter list, a failure occurred.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that a valid DLL routine name is specified on the CELAAUTH call.</p>
3D4	0C	<p>Equate Symbol: #ALRC_ENV_TOKEN_FAILURE</p> <p>Explanation: CELAAUTH Services attempted to use the environment token provided by the caller on a CELAAUTH REQUEST(USERCALL). A failure occurred while accessing this token.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that a valid environment token has been provided on the CELAAUTH call.</p>

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
3D8	14	<p>Equate Symbol: #ALRC_WRKRTASK_ATTACH_FAILURE</p> <p>Explanation: CELAAUTH Services attempted to attach the worker task using an IRB. The IRB failed while attempting the ATTACH. while accessing this token.</p> <p>System Action: A dump is taken to capture diagnostic information. CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Examine the dump to determine the reason for the ATTACH failure. If the failure does not appear to be the fault of the user's environment, contact IBM with the captured dump.</p>
3DC	14	<p>Equate Symbol: #ALRC_ALESTACK_OVERFLOW</p> <p>Explanation: CELAAUTH Services attempted to allocate an additional stack frame but the expansion overflowed the maximum boundary of the entire stack.</p> <p>System Action: An abend of 4088 is generated, a dump should be captured.</p> <p>Programmer Response: Contact IBM with the captured dump.</p>
3E0	0C	<p>Equate Symbol: #ALRC_PARMLIST_FAILURE</p> <p>Explanation: CELAAUTH Services attempted to access the user parameter list, but a failure occurred.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Ensure that the parameter list is allocated in storage that is accessible to the caller.</p>
3E4	14	<p>Equate Symbol: #ALRC_MODTABLE_FAILURE</p> <p>Explanation: CELAAUTH Services attempted to access its module table, but a failure occurred.</p> <p>System Action: The reason code is returned back to the Language Environment load service.</p> <p>Programmer Response: Contact IBM with details of the problem scenario.</p>
3E8	0C	<p>Equate Symbol: #ALRC_MODULE_EP_FAILURE</p> <p>Explanation: CELAAUTH Services attempted to access the user-provided module entrypoint, but a failure occurred.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: If RTNADDR was specified, ensure that the address that is provided is the CELQSTRT entrypoint for the routine. If RTNNAME was specified, ensure that the entrypoint for the corresponding load module is CELQSTRT.</p>

CELAAUTH

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
3EC	14	<p>Equat Symbol: #ALRC_WORKER_TASK_RM_RELEASED</p> <p>Explanation: While the caller is waiting for the worker task to process a request, the worker task is terminated, and the worker task resource manager got control, and released the caller from waiting.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup. This reason is most likely caused by an earlier error.</p> <p>Programmer Response: When this reason code is detected, the caller should exit its code as soon as possible, and let CELAAUTH Services do the necessary cleanup.</p>
3F0	10	<p>Equat Symbol: #ALRC_SCSRG_CP_DESTROY_ERR</p> <p>Explanation: When calling CELAAUTH Services internal cellpool services to destroy the internal cellpools, the destroy failed.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Contact IBM with details of the problem scenario.</p>
3F4	08	<p>Equat Symbol: #ALRC_WORKER_RESMGR_MEMTERM</p> <p>Explanation: During worker task resource manager processing, it terminated the address space because there are still units of work using the worker task resources.</p> <p>System Action: A dump with 4094 completion code and this reason code is generated.</p> <p>Programmer Response: Contact IBM with the captured dump.</p>
FFC	08	<p>Equat Symbol: #ALRC_AUTHLE_INTERNAL_ERR</p> <p>Explanation: CELAAUTH Services failed to determine its state. This could be caused by other unknown system errors, and/or because of storage overlays.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: If the system appears to be running fine, take a dump, and contact IBM support with the dump taken.</p>
3F8	0C	<p>Equat Symbol: #ALRC_AEDT_SIZE_MISMATCH</p> <p>Explanation: The AEDT_EDENUM value did not match the value used when the set of system managed environments was initialized. The AEDT specified for a MNGDUPDT request must contain the same number of environment definition entries (AEDE) as the AEDT that was used for the MNGDINIT request.</p> <p>System Action: CELAAUTH returns this reason code back to the caller, after cleanup.</p> <p>Programmer Response: Make sure the AEDT_EDENUM value in the AEDT is correct.</p>

Table 89. Return and reason codes for the CELAAUTH macro (continued)

Reason Code (rrr)	Return Code	Meaning and Action
3FC	0C, 04	<p>Equate Symbol: #ALRC_MAX_ENV_DECREASE</p> <p>Explanation: The AEDE_MAX value specified for at least one AEDE was less than the value currently in effect. The maximum number of environments can only be increased. The AEDE_MAX value must be greater than the value specified for the MNGDINIT request and any previous MNGDUPDT requests.</p> <p>System Action: CELAAUTH returns this reason code to the caller after cleanup. When the return code is #ALRTN_QUALIFIED_SUCCESS, the requests of other AEDE entries containing AEDE_MAX values greater than the value currently in effect are honored.</p> <p>Programmer Response: Make sure the AEDE_MAX value in each AEDE is correct.</p>
400	04	<p>Equate Symbol: #ALRC_NO_UPDATES</p> <p>Explanation: The AEDE_MAX value for every AEDE was either zero or the value currently in effect. No updates were performed. The maximum number of environments can only be increased. The AEDE_MAX value must be greater than the value specified for the MNGDINIT request and any previous MNGDUPDT requests for at least one AEDE.</p> <p>System Action: CELAAUTH returns this reason code to the caller after cleanup.</p> <p>Programmer Response: Make sure the AEDE_MAX value in each AEDE is correct.</p>

CELAUTH

Part 3. Appendixes

Appendix A. Options control block and supplementary options control block

The following sections describe the CEEOCB and CEESOCB macros, respectively.

Options control block

CEEOCB, the options control block (OCB), contains structures that describe the basic settings and parameters of each Language Environment runtime option. The following tables show the format of the OCB:

- Table 90 shows the type field definitions.
- Figure 154 on page 812 and following figures show the OCB field descriptions.
- Table 91 on page 856 shows the OCB constants.

Table 90. Options control block (OCB) and supplementary options control block (SOCB) type field definitions

Type	Definition
POINTER	A platform-dependent address pointer
BITSTRING	A string of bits of the defined length
CHARACTER	A string of characters (character array) of the defined length
DECIMAL	A two-byte or four-byte signed integer value
PTRINTOAREA	A two-byte or four-byte signed integer
SIGNED	A two-byte or four-byte signed integer
STRUCTURE	A mapping of a storage area; the displacement of a data item from the beginning of the OCB structure

CEEOCB Macro

1 CEEOCB

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB	
0	(0)	CHARACTER	1	CEEOCB_AREA_AREA(0)	
0	(0)	CHARACTER	8	CEEOCB_EYECATCHER	
8	(8)	SIGNED	2	CEEOCB_VERSION_RELEASE	
10	(A)	SIGNED	2	CEEOCB_LENGTH	
12	(C)	ADDRESS	4	*	
16	(10)	BITSTRING	1	CEEOCB_FORMAT	
			CEEOCB_FORMAT_31	"X'00' "
	1		CEEOCB_FORMAT_64	"X'01' "
17	(11)	BITSTRING	1	CEEOCB_IBM_SUPPLIED	
			CEEOCB_USER_SUPPLIED	"X'00' "
		1...		CEEOCB_IBM_SUPPLIED	"X'80' "
18	(12)	BITSTRING	1	*(2)	
20	(14)	CHARACTER	8	CEEOCB_RSVD1(0)	
20	(14)	BITSTRING	1	CEEOCB_RSVD1_BIT_FLAG	
		1...		CEEOCB_RSVD1_ON	"X'80' "
		.1..		CEEOCB_RSVD1_NOOVERRIDE	"X'40' "
	1		CEEOCB_RSVD1_ON_V	"X'01' "
21	(15)	BITSTRING	1	*	
22	(16)	SIGNED	2	CEEOCB_RSVD1_WHERE_SET	
24	(18)	ADDRESS	4	CEEOCB_RSVD1_SUB_OPTIONS	
28	(1C)	CHARACTER	8	CEEOCB_AIXBLD(0)	
28	(1C)	BITSTRING	1	CEEOCB_AIXBLD_BIT_FLAG	
		1...		CEEOCB_AIXBLD_ON	"X'80' "
		.1..		CEEOCB_AIXBLD_NOOVERRIDE	"X'40' "
	1		CEEOCB_AIXBLD_ON_V	"X'01' "
29	(1D)	BITSTRING	1	*	
30	(1E)	SIGNED	2	CEEOCB_AIXBLD_WHERE_SET	
32	(20)	ADDRESS	4	CEEOCB_AIXBLD_SUB_OPTIONS	
36	(24)	CHARACTER	8	CEEOCB_ALL31(0)	
36	(24)	BITSTRING	1	CEEOCB_ALL31_BIT_FLAG	
		1...		CEEOCB_ALL31_ON	"X'80' "
		.1..		CEEOCB_ALL31_NOOVERRIDE	"X'40' "
	1		CEEOCB_ALL31_ON_V	"X'01' "
37	(25)	BITSTRING	1	*	
38	(26)	SIGNED	2	CEEOCB_ALL31_WHERE_SET	
40	(28)	ADDRESS	4	CEEOCB_ALL31_SUB_OPTIONS	
44	(2C)	CHARACTER	8	CEEOCB_BELOWHEAP(0)	
44	(2C)	BITSTRING	1	CEEOCB_BELOWHEAP_BIT_FLAG	
		1...		CEEOCB_BELOWHEAP_ON	"X'80' "
		.1..		CEEOCB_BELOWHEAP_NOOVERRIDE	"X'40' "
	1		CEEOCB_BELOWHEAP_ON_V	"X'01' "
45	(2D)	BITSTRING	1	*	
46	(2E)	SIGNED	2	CEEOCB_BELOWHEAP_WHERE_SET	

Figure 154. Options control block (OCB) field descriptions (Part 1)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
48	(30)	ADDRESS	4	CEEOCB_BELOWHEAP_SUB_OPTIONS	
52	(34)	CHARACTER	8	CEEOCB_CHECK(0)	
52	(34)	BITSTRING	1	CEEOCB_CHECK_BIT_FLAG	
		1... ..		CEEOCB_CHECK_ON	"X'80'"
		.1.. ..		CEEOCB_CHECK_NOOVERRIDE	"X'40'"
	1		CEEOCB_CHECK_ON_V	"X'01'"
53	(35)	BITSTRING	1	*	
54	(36)	SIGNED	2	CEEOCB_CHECK_WHERE_SET	
56	(38)	ADDRESS	4	CEEOCB_CHECK_SUB_OPTIONS	
60	(3C)	CHARACTER	8	CEEOCB_PLITASKCOUNT(0)	
60	(3C)	BITSTRING	1	CEEOCB_PLITASKCOUNT_BIT_FLAG	
		1... ..		CEEOCB_PLITASKCOUNT_ON	"X'80'"
		.1.. ..		CEEOCB_PLITASKCOUNT_NOOVERRIDE	"X'40'"
	1		CEEOCB_PLITASKCOUNT_ON_V	"X'01'"
61	(3D)	BITSTRING	1	*	
62	(3E)	SIGNED	2	CEEOCB_PLITASKCOUNT_WHERE_SET	
64	(40)	ADDRESS	4	CEEOCB_PLITASKCOUNT_SUB_OPTIONS	
68	(44)	CHARACTER	8	CEEOCB_ABTERMENC(0)	
68	(44)	BITSTRING	1	CEEOCB_ABTERMENC_BIT_FLAG	
		1... ..		CEEOCB_ABTERMENC_ON	"X'80'"
		.1.. ..		CEEOCB_ABTERMENC_NOOVERRIDE	"X'40'"
	1		CEEOCB_ABTERMENC_ON_V	"X'01'"
69	(45)	BITSTRING	1	*	
70	(46)	SIGNED	2	CEEOCB_ABTERMENC_WHERE_SET	
72	(48)	ADDRESS	4	CEEOCB_ABTERMENC_SUB_OPTIONS	
76	(4C)	CHARACTER	8	CEEOCB_COUNTRY(0)	
76	(4C)	BITSTRING	1	CEEOCB_COUNTRY_BIT_FLAG	
		1... ..		CEEOCB_COUNTRY_ON	"X'80'"
		.1.. ..		CEEOCB_COUNTRY_NOOVERRIDE	"X'40'"
	1		CEEOCB_COUNTRY_ON_V	"X'01'"
77	(4D)	BITSTRING	1	*	
78	(4E)	SIGNED	2	CEEOCB_COUNTRY_WHERE_SET	
80	(50)	ADDRESS	4	CEEOCB_COUNTRY_SUB_OPTIONS	
84	(54)	CHARACTER	8	CEEOCB_DEBUG(0)	
84	(54)	BITSTRING	1	CEEOCB_DEBUG_BIT_FLAG	
		1... ..		CEEOCB_DEBUG_ON	"X'80'"
		.1.. ..		CEEOCB_DEBUG_NOOVERRIDE	"X'40'"
	1		CEEOCB_DEBUG_ON_V	"X'01'"
85	(55)	BITSTRING	1	*	
86	(56)	SIGNED	2	CEEOCB_DEBUG_WHERE_SET	

Figure 155. Options control block (OCB) field descriptions (Part 2)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
88	(58)	ADDRESS	4	CEEOCB_DEBUG_SUB_OPTIONS	
92	(5C)	CHARACTER	8	CEEOCB_ERRCOUNT(0)	
92	(5C)	BITSTRING	1	CEEOCB_ERRCOUNT_BIT_FLAG	
		1... ..		CEEOCB_ERRCOUNT_ON	"X'80'"
		.1.. ..		CEEOCB_ERRCOUNT_NOOVERRIDE	"X'40'"
	1		CEEOCB_ERRCOUNT_ON_V	"X'01'"
93	(5D)	BITSTRING	1	*	
94	(5E)	SIGNED	2	CEEOCB_ERRCOUNT_WHERE_SET	
96	(60)	ADDRESS	4	CEEOCB_ERRCOUNT_SUB_OPTIONS	
100	(64)	CHARACTER	8	CEEOCB_FILEHIST(0)	
100	(64)	BITSTRING	1	CEEOCB_FILEHIST_BIT_FLAG	
		1... ..		CEEOCB_FILEHIST_ON	"X'80'"
		.1.. ..		CEEOCB_FILEHIST_NOOVERRIDE	"X'40'"
	1		CEEOCB_FILEHIST_ON_V	"X'01'"
101	(65)	BITSTRING	1	*	
102	(66)	SIGNED	2	CEEOCB_FILEHIST_WHERE_SET	
104	(68)	ADDRESS	4	CEEOCB_FILEHIST_SUB_OPTIONS	
108	(6C)	CHARACTER	8	CEEOCB_ENVAR(0)	
108	(6C)	BITSTRING	1	CEEOCB_ENVAR_BIT_FLAG	
		1... ..		CEEOCB_ENVAR_ON	"X'80'"
		.1.. ..		CEEOCB_ENVART_NOOVERRIDE	"X'40'"
	1		CEEOCB_ENVAR_ON_V	"X'01'"
109	(6D)	BITSTRING	1	*	
110	(6E)	SIGNED	2	CEEOCB_ENVAR_WHERE_SET	
112	(70)	ADDRESS	4	CEEOCB_ENVAR_SUB_OPTIONS	
116	(74)	CHARACTER	8	CEEOCB_FLOWC(0)	
116	(74)	BITSTRING	1	CEEOCB_FLOWC_BIT_FLAG	
		1... ..		CEEOCB_FLOWC_ON	"X'80'"
		.1.. ..		CEEOCB_FLOWC_NOOVERRIDE	"X'40'"
	1		CEEOCB_FLOWC_ON_V	"X'01'"
117	(75)	BITSTRING	1	*	
118	(76)	SIGNED	2	CEEOCB_FLOWC_WHERE_SET	
120	(78)	ADDRESS	4	CEEOCB_FLOWC_SUB_OPTIONS	
124	(7C)	CHARACTER	8	CEEOCB_HEAP(0)	
124	(7C)	BITSTRING	1	CEEOCB_HEAP_BIT_FLAG	
		1... ..		CEEOCB_HEAP_ON	"X'80'"
		.1.. ..		CEEOCB_HEAP_NOOVERRIDE	"X'40'"
	1		CEEOCB_HEAP_ON_V	"X'01'"
125	(7D)	BITSTRING	1	*	
126	(7E)	SIGNED	2	CEEOCB_HEAP_WHERE_SET	

Figure 156. Options control block (OCB) field descriptions (Part 3)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
128	(80)	ADDRESS	4	CEEOCB_HEAP_SUB_OPTIONS	
132	(84)	CHARACTER	8	CEEOCB_INQPCOPN(0)	
132	(84)	BITSTRING	1	CEEOCB_INQPCOPN_BIT_FLAG	
		1... ..		CEEOCB_INQPCOPN_ON	"X'80'"
		.1.. ..		CEEOCB_INQPCOPN_NOOVERRIDE	"X'40'"
	1		CEEOCB_INQPCOPN_ON_V	"X'01'"
133	(85)	BITSTRING	1	*	
134	(86)	SIGNED	2	CEEOCB_INQPCOPN_WHERE_SET	
136	(88)	ADDRESS	4	CEEOCB_INQPCOPN_SUB_OPTIONS	
140	(8C)	CHARACTER	8	CEEOCB_INTERRUPT(0)	
140	(8C)	BITSTRING	1	CEEOCB_INTERRUPT_BIT_FLAG	
		1... ..		CEEOCB_INTERRUPT_ON	"X'80'"
		.1.. ..		CEEOCB_INTERRUPT_NOOVERRIDE	"X'40'"
	1		CEEOCB_INTERRUPT_ON_V	"X'01'"
141	(8D)	BITSTRING	1	*	
142	(8E)	SIGNED	2	CEEOCB_INTERRUPT_WHERE_SET	
144	(90)	ADDRESS	4	CEEOCB_INTERRUPT_SUB_OPTIONS	
148	(94)	CHARACTER	8	CEEOCB_LIBSTACK(0)	
148	(94)	BITSTRING	1	CEEOCB_LIBSTACK_BIT_FLAG	
		1... ..		CEEOCB_LIBSTACK_ON	"X'80'"
		.1.. ..		CEEOCB_LIBSTACK_NOOVERRIDE	"X'40'"
	1		CEEOCB_LIBSTACK_ON_V	"X'01'"
149	(95)	BITSTRING	1	*	
150	(96)	SIGNED	2	CEEOCB_LIBSTACK_WHERE_SET	
152	(98)	ADDRESS	4	CEEOCB_LIBSTACK_SUB_OPTIONS	
156	(9C)	CHARACTER	8	CEEOCB_MSGQ(0)	
156	(9C)	BITSTRING	1	CEEOCB_MSGQ_BIT_FLAG	
		1... ..		CEEOCB_MSGQ_ON	"X'80'"
		.1.. ..		CEEOCB_MSGQ_NOOVERRIDE	"X'40'"
	1		CEEOCB_MSGQ_ON_V	"X'01'"
157	(9D)	BITSTRING	1	*	
158	(9E)	SIGNED	2	CEEOCB_MSGQ_WHERE_SET	
160	(A0)	ADDRESS	4	CEEOCB_MSGQ_SUB_OPTIONS	
164	(A4)	CHARACTER	8	CEEOCB_MSGFILE(0)	
164	(A4)	BITSTRING	1	CEEOCB_MSGFILE_BIT_FLAG	
		1... ..		CEEOCB_MSGFILE_ON	"X'80'"
		.1.. ..		CEEOCB_MSGFILE_NOOVERRIDE	"X'40'"
	1		CEEOCB_MSGFILE_ON_V	"X'01'"
165	(A5)	BITSTRING	1	*	
166	(A6)	SIGNED	2	CEEOCB_MSGFILE_WHERE_SET	

Figure 157. Options control block (OCB) field descriptions (Part 4)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
168	(A8)	ADDRESS	4	CEEOCB_MSGFILE_SUB_OPTIONS	
172	(AC)	CHARACTER	8	CEEOCB_NATLANG(0)	
172	(AC)	BITSTRING	1	CEEOCB_NATLANG_BIT_FLAG	
		1... ..		CEEOCB_NATLANG_ON	"X'80'"
		.1.. ..		CEEOCB_NATLANG_NOOVERRIDE	"X'40'"
	1		CEEOCB_NATLANG_ON_V	"X'01'"
173	(AD)	BITSTRING	1	*	
174	(AE)	SIGNED	2	CEEOCB_NATLANG_WHERE_SET	
176	(B0)	ADDRESS	4	CEEOCB_NATLANG_SUB_OPTIONS	
180	(B4)	CHARACTER	8	CEEOCB_ERRUNIT(0)	
180	(B4)	BITSTRING	1	CEEOCB_ERRUNIT_BIT_FLAG	
		1... ..		CEEOCB_ERRUNIT_ON	"X'80'"
		.1.. ..		CEEOCB_ERRUNIT_NOOVERRIDE	"X'40'"
	1		CEEOCB_ERRUNIT_ON_V	"X'01'"
181	(B5)	BITSTRING	1	*	
182	(B6)	SIGNED	2	CEEOCB_ERRUNIT_WHERE_SET	
184	(B8)	ADDRESS	4	CEEOCB_ERRUNIT_SUB_OPTIONS	
188	(BC)	CHARACTER	8	CEEOCB_OCSTATUS(0)	
188	(BC)	BITSTRING	1	CEEOCB_OCSTATUS_BIT_FLAG	
		1... ..		CEEOCB_OCSTATUS_ON	"X'80'"
		.1.. ..		CEEOCB_OCSTATUS_NOOVERRIDE	"X'40'"
	1		CEEOCB_OCSTATUS_ON_V	"X'01'"
189	(BD)	BITSTRING	1	*	
190	(BE)	SIGNED	2	CEEOCB_OCSTATUS_WHERE_SET	
192	(C0)	ADDRESS	4	CEEOCB_OCSTATUS_SUB_OPTIONS	
196	(C4)	CHARACTER	8	CEEOCB_POSIX(0)	
196	(C4)	BITSTRING	1	CEEOCB_POSIX_BIT_FLAG	
		1... ..		CEEOCB_POSIX_ON	"X'80'"
		.1.. ..		CEEOCB_POSIX_NOOVERRIDE	"X'40'"
	1		CEEOCB_POSIX_ON_V	"X'01'"
197	(C5)	BITSTRING	1	*	
198	(C6)	SIGNED	2	CEEOCB_POSIX_WHERE_SET	
200	(C8)	ADDRESS	4	CEEOCB_POSIX_SUB_OPTIONS	
204	(CC)	CHARACTER	8	CEEOCB_RPTSTG(0)	
204	(CC)	BITSTRING	1	CEEOCB_RPTSTG_BIT_FLAG	
		1... ..		CEEOCB_RPTSTG_ON	"X'80'"
		.1.. ..		CEEOCB_RPTSTG_NOOVERRIDE	"X'40'"
	1		CEEOCB_RPTSTG_ON_V	"X'01'"
205	(CD)	BITSTRING	1	*	
206	(CE)	SIGNED	2	CEEOCB_RPTSTG_WHERE_SET	

Figure 158. Options control block (OCB) field descriptions (Part 5)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
208	(D0)	ADDRESS	4	CEEOCB_RPTSTG_SUB_OPTIONS	
212	(D4)	CHARACTER	8	CEEOCB_RTREUS(0)	
212	(D4)	BITSTRING	1	CEEOCB_RTREUS_BIT_FLAG	
		1... ..		CEEOCB_RTREUS_ON	"X'80'"
		.1.. ..		CEEOCB_RTREUS_NOOVERRIDE	"X'40'"
	1		CEEOCB_RTREUS_ON_V	"X'01'"
213	(D5)	BITSTRING	1	*	
214	(D6)	SIGNED	2	CEEOCB_RTREUS_WHERE_SET	
216	(D8)	ADDRESS	4	CEEOCB_RTREUS_SUB_OPTIONS	
220	(DC)	CHARACTER	8	CEEOCB_SIMVRD(0)	
220	(DC)	BITSTRING	1	CEEOCB_SIMVRD_BIT_FLAG	
		1... ..		CEEOCB_SIMVRD_ON	"X'80'"
		.1.. ..		CEEOCB_SIMVRD_NOOVERRIDE	"X'40'"
	1		CEEOCB_SIMVRD_ON_V	"X'01'"
221	(DD)	BITSTRING	1	*	
222	(DE)	SIGNED	2	CEEOCB_SIMVRD_WHERE_SET	
224	(E0)	ADDRESS	4	CEEOCB_SIMVRD_SUB_OPTIONS	
228	(E4)	CHARACTER	8	CEEOCB_STACK(0)	
228	(E4)	BITSTRING	1	CEEOCB_STACK_BIT_FLAG	
		1... ..		CEEOCB_STACK_ON	"X'80'"
		.1.. ..		CEEOCB_STACK_NOOVERRIDE	"X'40'"
	1		CEEOCB_STACK_ON_V	"X'01'"
229	(E5)	BITSTRING	1	*	
230	(E6)	SIGNED	2	CEEOCB_STACK_WHERE_SET	
232	(E8)	ADDRESS	4	CEEOCB_STACK_SUB_OPTIONS	
236	(EC)	CHARACTER	8	CEEOCB_STORAGE(0)	
236	(EC)	BITSTRING	1	CEEOCB_STORAGE_BIT_FLAG	
		1... ..		CEEOCB_STORAGE_ON	"X'80'"
		.1.. ..		CEEOCB_STORAGE_NOOVERRIDE	"X'40'"
	1		CEEOCB_STORAGE_ON_V	"X'01'"
237	(ED)	BITSTRING	1	*	
238	(EE)	SIGNED	2	CEEOCB_STORAGE_WHERE_SET	
240	(F0)	ADDRESS	4	CEEOCB_STORAGE_SUB_OPTIONS	
244	(F4)	CHARACTER	8	CEEOCB_AUTOTASK(0)	
244	(F4)	BITSTRING	1	CEEOCB_AUTOTASK_BIT_FLAG	
		1... ..		CEEOCB_AUTOTASK_ON	"X'80'"
		.1.. ..		CEEOCB_AUTOTASK_NOOVERRIDE	"X'40'"
	1		CEEOCB_AUTOTASK_ON_V	"X'01'"
245	(F5)	BITSTRING	1	*	
246	(F6)	SIGNED	2	CEEOCB_AUTOTASK_WHERE_SET	

Figure 159. Options control block (OCB) field descriptions (Part 6)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
248	(F8)	ADDRESS	4	CEEOCB_AUTOTASK_SUB_OPTIONS	
252	(FC)	CHARACTER	8	CEEOCB_TRACE(0)	
252	(FC)	BITSTRING	1	CEEOCB_TRACE_BIT_FLAG	
		1... ..		CEEOCB_TRACE_ON	"X'80'"
		.1.. ..		CEEOCB_TRACE_NOOVERRIDE	"X'40'"
	1		CEEOCB_TRACE_ON_V	"X'01'"
253	(FD)	BITSTRING	1	*	
254	(FE)	SIGNED	2	CEEOCB_TRACE_WHERE_SET	
256	(100)	ADDRESS	4	CEEOCB_TRACE_SUB_OPTIONS	
260	(104)	CHARACTER	8	CEEOCB_THREADHEAP(0)	
260	(104)	BITSTRING	1	CEEOCB_THREADHEAP_BIT_FLAG	
		1... ..		CEEOCB_THREADHEAP_ON	"X'80'"
		.1.. ..		CEEOCB_THREADHEAP_NOOVERRIDE	"X'40'"
	1		CEEOCB_THREADHEAP_ON_V	"X'01'"
261	(105)	BITSTRING	1	*	
262	(106)	SIGNED	2	CEEOCB_THREADHEAP_WHERE_SET	
264	(108)	ADDRESS	4	CEEOCB_THREADHEAP_SUB_OPTIONS	
268	(10C)	CHARACTER	8	CEEOCB_TEST(0)	
268	(10C)	BITSTRING	1	CEEOCB_TEST_BIT_FLAG	
		1... ..		CEEOCB_TEST_ON	"X'80'"
		.1.. ..		CEEOCB_TEST_NOOVERRIDE	"X'40'"
	1		CEEOCB_TEST_ON_V	"X'01'"
269	(10D)	BITSTRING	1	*	
270	(10E)	SIGNED	2	CEEOCB_TEST_WHERE_SET	
272	(110)	ADDRESS	4	CEEOCB_TEST_SUB_OPTIONS	
276	(114)	CHARACTER	8	CEEOCB_THREADSTACK(0)	
276	(114)	BITSTRING	1	CEEOCB_THREADSTACK_BIT_FLAG	
		1... ..		CEEOCB_THREADSTACK_ON	"X'80'"
		.1.. ..		CEEOCB_THREADSTACK_NOOVERRIDE	"X'40'"
	1		CEEOCB_THREADSTACK_ON_V	"X'01'"
277	(115)	BITSTRING	1	*	
278	(116)	SIGNED	2	CEEOCB_THREADSTACK_WHERE_SET	
280	(118)	ADDRESS	4	CEEOCB_THREADSTACK_SUB_OPTIONS	
284	(11C)	CHARACTER	8	CEEOCB_TRAP(0)	
284	(11C)	BITSTRING	1	CEEOCB_TRAP_BIT_FLAG	
		1... ..		CEEOCB_TRAP_ON	"X'80'"
		.1.. ..		CEEOCB_TRAP_NOOVERRIDE	"X'40'"
	1		CEEOCB_TRAP_ON_V	"X'01'"
285	(11D)	BITSTRING	1	*	
286	(11E)	SIGNED	2	CEEOCB_TRAP_WHERE_SET	

Figure 160. Options control block (OCB) field descriptions (Part 7)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
288	(120)	ADDRESS	4	CEEOCB_TRAP_SUB_OPTIONS	
292	(124)	CHARACTER	8	CEEOCB_UPSI(0)	
292	(124)	BITSTRING	1	CEEOCB_UPSI_BIT_FLAG	
		1... ..		CEEOCB_UPSI_ON	"X'80'"
		.1.. ..		CEEOCB_UPSI_NOOVERRIDE	"X'40'"
	1		CEEOCB_UPSI_ON_V	"X'01'"
293	(125)	BITSTRING	1	*	
294	(126)	SIGNED	2	CEEOCB_UPSI_WHERE_SET	
296	(128)	ADDRESS	4	CEEOCB_UPSI_SUB_OPTIONS	
300	(12C)	CHARACTER	8	CEEOCB_VCTRSAVE(0)	
300	(12C)	BITSTRING	1	CEEOCB_VCTRSAVE_BIT_FLAG	
		1... ..		CEEOCB_VCTRSAVE_ON	"X'80'"
		.1.. ..		CEEOCB_VCTRSAVE_NOOVERRIDE	"X'40'"
	1		CEEOCB_VCTRSAVE_ON_V	"X'01'"
301	(12D)	BITSTRING	1	*	
302	(12E)	SIGNED	2	CEEOCB_VCTRSAVE_WHERE_SET	
304	(130)	ADDRESS	4	CEEOCB_VCTRSAVE_SUB_OPTIONS	
308	(134)	CHARACTER	8	CEEOCB_PRTUNIT(0)	
308	(134)	BITSTRING	1	CEEOCB_PRTUNIT_BIT_FLAG	
		1... ..		CEEOCB_PRTUNIT_ON	"X'80'"
		.1.. ..		CEEOCB_PRTUNIT_NOOVERRIDE	"X'40'"
	1		CEEOCB_PRTUNIT_ON_V	"X'01'"
309	(135)	BITSTRING	1	*	
310	(136)	SIGNED	2	CEEOCB_PRTUNIT_WHERE_SET	
312	(138)	ADDRESS	4	CEEOCB_PRTUNIT_SUB_OPTIONS	
316	(13C)	CHARACTER	8	CEEOCB_XUFLOW(0)	
316	(13C)	BITSTRING	1	CEEOCB_XUFLOW_BIT_FLAG	
		1... ..		CEEOCB_XUFLOW_ON	"X'80'"
		.1.. ..		CEEOCB_XUFLOW_NOOVERRIDE	"X'40'"
	1		CEEOCB_XUFLOW_ON_V	"X'01'"
317	(13D)	BITSTRING	1	*	
318	(13E)	SIGNED	2	CEEOCB_XUFLOW_WHERE_SET	
320	(140)	ADDRESS	4	CEEOCB_XUFLOW_SUB_OPTIONS	
324	(144)	CHARACTER	8	CEEOCB_CBLOPTS(0)	
324	(144)	BITSTRING	1	CEEOCB_CBLOPTS_BIT_FLAG	
		1... ..		CEEOCB_CBLOPTS_ON	"X'80'"
		.1.. ..		CEEOCB_CBLOPTS_NOOVERRIDE	"X'40'"
	1		CEEOCB_CBLOPTS_ON_V	"X'01'"
325	(145)	BITSTRING	1	*	
326	(146)	SIGNED	2	CEEOCB_CBLOPTS_WHERE_SET	

Figure 161. Options control block (OCB) field descriptions (Part 8)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
328	(148)	ADDRESS	4	CEEOCB_CBLOPTS_SUB_OPTIONS	
332	(14C)	CHARACTER	8	CEEOCB_NONIPTSTACK(0)	
332	(14C)	BITSTRING	1	CEEOCB_NONIPTSTACK_BIT_FLAG	
		1... ..		CEEOCB_NONIPTSTACK_ON	"X'80' "
		.1..		CEEOCB_NONIPTSTACK_NOOVERRIDE	"X'40' "
	1		CEEOCB_NONIPTSTACK_ON_V	"X'01' "
333	(14D)	BITSTRING	1	*	
334	(14E)	SIGNED	2	CEEOCB_NONIPTSTACK_WHERE_SET	
336	(150)	ADDRESS	4	CEEOCB_NONIPTSTACK_SUB_OPTIONS	
340	(154)	CHARACTER	8	CEEOCB_RPTOPTS(0)	
340	(154)	BITSTRING	1	CEEOCB_RPTOPTS_BIT_FLAG	
		1... ..		CEEOCB_RPTOPTS_ON	"X'80' "
		.1..		CEEOCB_RPTOPTS_NOOVERRIDE	"X'40' "
	1		CEEOCB_RPTOPTS_ON_V	"X'01' "
341	(155)	BITSTRING	1	*	
342	(156)	SIGNED	2	CEEOCB_RPTOPTS_WHERE_SET	
344	(158)	ADDRESS	4	CEEOCB_RPTOPTS_SUB_OPTIONS	
348	(15C)	CHARACTER	8	CEEOCB_ANYHEAP(0)	
348	(15C)	BITSTRING	1	CEEOCB_ANYHEAP_BIT_FLAG	
		1... ..		CEEOCB_ANYHEAP_ON	"X'80' "
		.1..		CEEOCB_ANYHEAP_NOOVERRIDE	"X'40' "
	1		CEEOCB_ANYHEAP_ON_V	"X'01' "
349	(15D)	BITSTRING	1	*	
350	(15E)	SIGNED	2	CEEOCB_ANYHEAP_WHERE_SET	
352	(160)	ADDRESS	4	CEEOCB_ANYHEAP_SUB_OPTIONS	
356	(164)	CHARACTER	8	CEEOCB_ABPERC(0)	
356	(164)	BITSTRING	1	CEEOCB_ABPERC_BIT_FLAG	
		1... ..		CEEOCB_ABPERC_ON	"X'80' "
		.1..		CEEOCB_ABPERC_NOOVERRIDE	"X'40' "
	1		CEEOCB_ABPERC_ON_V	"X'01' "
357	(165)	BITSTRING	1	*	
358	(166)	SIGNED	2	CEEOCB_ABPERC_WHERE_SET	
360	(168)	ADDRESS	4	CEEOCB_ABPERC_SUB_OPTIONS	
364	(16C)	CHARACTER	8	CEEOCB_TERMTHDACT(0)	
364	(16C)	BITSTRING	1	CEEOCB_TERMTHDACT_BIT_FLAG	
		1... ..		CEEOCB_TERMTHDACT_ON	"X'80' "
		.1..		CEEOCB_TERMTHDACT_NOOVERRIDE	"X'40' "
	1		CEEOCB_TERMTHDACT_ON_V	"X'01' "
365	(16D)	BITSTRING	1	*	
366	(16E)	SIGNED	2	CEEOCB_TERMTHDACT_WHERE_SET	

Figure 162. Options control block (OCB) field descriptions (Part 9)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
368	(170)	ADDRESS	4	CEEOCB_TERMTHDACT_SUB_OPTIONS	
372	(174)	CHARACTER	8	CEEOCB_DEPTHCONDLMT(0)	
372	(174)	BITSTRING	1	CEEOCB_DEPTHCONDLMT_BIT_FLAG	
		1... ..		CEEOCB_DEPTHCONDLMT_ON	"X'80'"
		.1.. ..		CEEOCB_DEPTHCONDLMT_NOOVERRIDE	"X'40'"
	1		CEEOCB_DEPTHCONDLMT_ON_V	"X'01'"
373	(175)	BITSTRING	1	*	
374	(176)	SIGNED	2	CEEOCB_DEPTHCONDLMT_WHERE_SET	
376	(178)	ADDRESS	4	CEEOCB_DEPTHCONDLMT_SUB_OPTIONS	
380	(17C)	CHARACTER	8	CEEOCB_CBLPSHPOP(0)	
380	(17C)	BITSTRING	1	CEEOCB_CBLPSHPOP_BIT_FLAG	
		1... ..		CEEOCB_CBLPSHPOP_ON	"X'80'"
		.1.. ..		CEEOCB_CBLPSHPOP_NOOVERRIDE	"X'40'"
	1		CEEOCB_CBLPSHPOP_ON_V	"X'01'"
381	(17D)	BITSTRING	1	*	
382	(17E)	SIGNED	2	CEEOCB_CBLPSHPOP_WHERE_SET	
384	(180)	ADDRESS	4	CEEOCB_CBLPSHPOP_SUB_OPTIONS	
388	(184)	CHARACTER	8	CEEOCB_CBLQDA(0)	
388	(184)	BITSTRING	1	CEEOCB_CBLQDA_BIT_FLAG	
		1... ..		CEEOCB_CBLQDA_ON	"X'80'"
		.1.. ..		CEEOCB_CBLQDA_NOOVERRIDE	"X'40'"
	1		CEEOCB_CBLQDA_ON_V	"X'01'"
389	(185)	BITSTRING	1	*	
390	(186)	SIGNED	2	CEEOCB_CBLQDA_WHERE_SET	
392	(188)	ADDRESS	4	CEEOCB_CBLQDA_SUB_OPTIONS	
396	(18C)	CHARACTER	8	CEEOCB_PUNUNIT(0)	
396	(18C)	BITSTRING	1	CEEOCB_PUNUNIT_BIT_FLAG	
		1... ..		CEEOCB_PUNUNIT_ON	"X'80'"
		.1.. ..		CEEOCB_PUNUNIT_NOOVERRIDE	"X'40'"
	1		CEEOCB_PUNUNIT_ON_V	"X'01'"
397	(18D)	BITSTRING	1	*	
398	(18E)	SIGNED	2	CEEOCB_PUNUNIT_WHERE_SET	
400	(190)	ADDRESS	4	CEEOCB_PUNUNIT_SUB_OPTIONS	
404	(194)	CHARACTER	8	CEEOCB_RDRUNIT(0)	
404	(194)	BITSTRING	1	CEEOCB_RDRUNIT_BIT_FLAG	
		1... ..		CEEOCB_RDRUNIT_ON	"X'80'"
		.1.. ..		CEEOCB_RDRUNIT_NOOVERRIDE	"X'40'"
	1		CEEOCB_RDRUNIT_ON_V	"X'01'"
405	(195)	BITSTRING	1	*	
406	(196)	SIGNED	2	CEEOCB_RDRUNIT_WHERE_SET	

Figure 163. Options control block (OCB) field descriptions (Part 10)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
408	(198)	ADDRESS	4	CEEOCB_RDRUNIT_SUB_OPTIONS	
412	(19C)	CHARACTER	8	CEEOCB_RECPAD(0)	
412	(19C)	BITSTRING	1	CEEOCB_RECPAD_BIT_FLAG	
		1... ..		CEEOCB_RECPAD_ON	"X'80'"
		.1.. ..		CEEOCB_RECPAD_NOOVERRIDE	"X'40'"
	1		CEEOCB_RECPAD_ON_V	"X'01'"
413	(19D)	BITSTRING	1	*	
414	(19E)	SIGNED	2	CEEOCB_RECPAD_WHERE_SET	
416	(1A0)	ADDRESS	4	CEEOCB_RECPAD_SUB_OPTIONS	
420	(1A4)	CHARACTER	8	CEEOCB_USRHDLR(0)	
420	(1A4)	BITSTRING	1	CEEOCB_USRHDLR_BIT_FLAG	
		1... ..		CEEOCB_USRHDLR_ON	"X'80'"
		.1.. ..		CEEOCB_USRHDLR_NOOVERRIDE	"X'40'"
	1		CEEOCB_USRHDLR_ON_V	"X'01'"
421	(1A5)	BITSTRING	1	*	
422	(1A6)	SIGNED	2	CEEOCB_USRHDLR_WHERE_SET	
424	(1A8)	ADDRESS	4	CEEOCB_USRHDLR_SUB_OPTIONS	
428	(1AC)	CHARACTER	8	CEEOCB_NAMELIST(0)	
428	(1AC)	BITSTRING	1	CEEOCB_NAMELIST_BIT_FLAG	
		1... ..		CEEOCB_NAMELIST_ON	"X'80'"
		.1.. ..		CEEOCB_NAMELIST_NOOVERRIDE	"X'40'"
	1		CEEOCB_NAMELIST_ON_V	"X'01'"
429	(1AD)	BITSTRING	1	*	
430	(1AE)	SIGNED	2	CEEOCB_NAMELIST_WHERE_SET	
432	(1B0)	ADDRESS	4	CEEOCB_NAMELIST_SUB_OPTIONS	
436	(1B4)	CHARACTER	8	CEEOCB_PC(0)	
436	(1B4)	BITSTRING	1	CEEOCB_PC_BIT_FLAG	
		1... ..		CEEOCB_PC_ON	"X'80'"
		.1.. ..		CEEOCB_PC_NOOVERRIDE	"X'40'"
	1		CEEOCB_PC_ON_V	"X'01'"
437	(1B5)	BITSTRING	1	*	
438	(1B6)	SIGNED	2	CEEOCB_PC_WHERE_SET	
440	(1B8)	ADDRESS	4	CEEOCB_PC_SUB_OPTIONS	
This option is now obsolete - Do not use					
444	(1BC)	CHARACTER	8	CEEOCB_LIBRARY(0)	
444	(1BC)	BITSTRING	1	CEEOCB_LIBRARY_BIT_FLAG	
		1... ..		CEEOCB_LIBRARY_ON	"X'80'"
		.1.. ..		CEEOCB_LIBRARY_NOOVERRIDE	"X'40'"
	1		CEEOCB_LIBRARY_ON_V	"X'01'"

Figure 164. Options control block (OCB) field descriptions (Part 11)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
445	(1BD)	BITSTRING	1	*	
446	(1BE)	SIGNED	2	CEEOCB_LIBRARY_WHERE_SET	
448	(1C0)	ADDRESS	4	CEEOCB_LIBRARY_SUB_OPTIONS	
This option is now obsolete - Do not use					
452	(1C4)	CHARACTER	8	CEEOCB_VERSION(0)	
452	(1C4)	BITSTRING	1	CEEOCB_VERSION_BIT_FLAG	
		1... ..		CEEOCB_VERSION_ON	"X'80'"
		.1.. ..		CEEOCB_VERSION_NOOVERRIDE	"X'40'"
	1		CEEOCB_VERSION_ON_V	"X'01'"
453	(1C5)	BITSTRING	1	*	
454	(1C6)	SIGNED	2	CEEOCB_VERSION_WHERE_SET	
456	(1C8)	ADDRESS	4	CEEOCB_VERSION_SUB_OPTIONS	
This option is now obsolete - Do not use					
460	(1CC)	CHARACTER	8	CEEOCB_RTLS(0)	
460	(1CC)	BITSTRING	1	CEEOCB_RTLS_BIT_FLAG	
		1... ..		CEEOCB_RTLS_ON	"X'80'"
		.1.. ..		CEEOCB_RTLS_NOOVERRIDE	"X'40'"
	1		CEEOCB_RTLS_ON_V	"X'01'"
461	(1CD)	BITSTRING	1	*	
462	(1CE)	SIGNED	2	CEEOCB_RTLS_WHERE_SET	
464	(1D0)	ADDRESS	4	CEEOCB_RTLS_SUB_OPTIONS	
468	(1D4)	CHARACTER	8	CEEOCB_HEAPCHK(0)	
468	(1D4)	BITSTRING	1	CEEOCB_HEAPCHK_BIT_FLAG	
		1... ..		CEEOCB_HEAPCHK_ON	"X'80'"
		.1.. ..		CEEOCB_HEAPCHK_NOOVERRIDE	"X'40'"
	1		CEEOCB_HEAPCHK_ON_V	"X'01'"
469	(1D5)	BITSTRING	1	*	
470	(1D6)	SIGNED	2	CEEOCB_HEAPCHK_WHERE_SET	
472	(1D8)	ADDRESS	4	CEEOCB_HEAPCHK_SUB_OPTIONS	
476	(1DC)	CHARACTER	8	CEEOCB_PROFILE(0)	
476	(1DC)	BITSTRING	1	CEEOCB_PROFILE_BIT_FLAG	
		1... ..		CEEOCB_PROFILE_ON	"X'80'"
		.1.. ..		CEEOCB_PROFILE_NOOVERRIDE	"X'40'"
	1		CEEOCB_PROFILE_ON_V	"X'01'"
477	(1DD)	BITSTRING	1	*	
478	(1DE)	SIGNED	2	CEEOCB_PROFILE_WHERE_SET	
480	(1E0)	ADDRESS	4	CEEOCB_PROFILE_SUB_OPTIONS	
484	(1E4)	CHARACTER	8	CEEOCB_HEAPPOOLS(0)	
484	(1E4)	BITSTRING	1	CEEOCB_HEAPPOOLS_BIT_FLAG	

Figure 165. Options control block (OCB) field descriptions (Part 12)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
		1... ..		CEEOCB_HEAPPOLLS_ON	"X'80'"
		.1.. ..		CEEOCB_HEAPPOLLS_NOOVERRIDE	"X'40'"
	1		CEEOCB_HEAPPOLLS_ON_V	"X'01'"
485	(1E5)	BITSTRING	1	*	
486	(1E6)	SIGNED	2	CEEOCB_HEAPPOLLS_WHERE_SET	
488	(1E8)	ADDRESS	4	CEEOCB_HEAPPOLLS_SUB_OPTIONS	
492	(1EC)	CHARACTER	8	CEEOCB_INFOMSGFILTER(0)	
492	(1EC)	BITSTRING	1	CEEOCB_INFOMSGFILTER_BIT_FLAG	
		1... ..		CEEOCB_INFOMSGFILTER_ON	"X'80'"
		.1.. ..		CEEOCB_INFOMSGFILTER_NOOVERRIDE	"X'40'"
	1		CEEOCB_INFOMSGFILTER_ON_V	"X'01'"
493	(1ED)	BITSTRING	1	*	
494	(1EE)	SIGNED	2	CEEOCB_INFOMSGFILTER_WHERE_SET	
496	(1F0)	ADDRESS	4	CEEOCB_INFOMSGFILTER_SUB_OPTIONS	
500	(1F4)	CHARACTER	8	CEEOCB_XPLINK(0)	
500	(1F4)	BITSTRING	1	CEEOCB_XPLINK_BIT_FLAG	
		1... ..		CEEOCB_XPLINK_ON	"X'80'"
		.1.. ..		CEEOCB_XPLINK_NOOVERRIDE	"X'40'"
	1		CEEOCB_XPLINK_ON_V	"X'01'"
501	(1F5)	BITSTRING	1	*	
502	(1F6)	SIGNED	2	CEEOCB_XPLINK_WHERE_SET	
504	(1F8)	ADDRESS	4	CEEOCB_XPLINK_SUB_OPTIONS	
508	(1FC)	CHARACTER	8	CEEOCB_FILETAG(0)	
508	(1FC)	BITSTRING	1	CEEOCB_FILETAG_BIT_FLAG	
		1... ..		CEEOCB_FILETAG_ON	"X'80'"
		.1.. ..		CEEOCB_FILETAG_NOOVERRIDE	"X'40'"
	1		CEEOCB_FILETAG_ON_V	"X'01'"
509	(1FD)	BITSTRING	1	*	
510	(1FE)	SIGNED	2	CEEOCB_FILETAG_WHERE_SET	
512	(200)	ADDRESS	4	CEEOCB_FILETAG_SUB_OPTIONS	
516	(204)	CHARACTER	8	CEEOCB_HEAP64(0)	
516	(204)	BITSTRING	1	CEEOCB_HEAP64_BIT_FLAG	
		1... ..		CEEOCB_HEAP64_ON	"X'80'"
		.1.. ..		CEEOCB_HEAP64_NOOVERRIDE	"X'40'"
	1		CEEOCB_HEAP64_ON_V	"X'01'"
517	(205)	BITSTRING	1	*	
518	(206)	SIGNED	2	CEEOCB_HEAP64_WHERE_SET	

Figure 166. Options control block (OCB) field descriptions (Part 13)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
520	(208)	ADDRESS	4	CEEOCB_HEAP64_SUB_OPTIONS	
524	(20C)	CHARACTER	8	CEEOCB_HEAPPOOLS64(0)	
524	(20C)	BITSTRING	1	CEEOCB_HEAPPOOLS64_BIT_FLAG	
		1... ..		CEEOCB_HEAPPOOLS64_ON	"X'80'"
		.1.. ..		CEEOCB_HEAPPOOLS64_NOOVERRIDE	"X'40'"
	1		CEEOCB_HEAPPOOLS64_ON_V	"X'01'"
525	(20D)	BITSTRING	1	*	
526	(20E)	SIGNED	2	CEEOCB_HEAPPOOLS64_WHERE_SET	
528	(210)	ADDRESS	4	CEEOCB_HEAPPOOLS64_SUB_OPTIONS	
532	(214)	CHARACTER	8	CEEOCB_IOHEAP64(0)	
532	(214)	BITSTRING	1	CEEOCB_IOHEAP64_BIT_FLAG	
		1... ..		CEEOCB_IOHEAP64_ON	"X'80'"
		.1.. ..		CEEOCB_IOHEAP64_NOOVERRIDE	"X'40'"
	1		CEEOCB_IOHEAP64_ON_V	"X'01'"
533	(215)	BITSTRING	1	*	
534	(216)	SIGNED	2	CEEOCB_IOHEAP64_WHERE_SET	
536	(218)	ADDRESS	4	CEEOCB_IOHEAP64_SUB_OPTIONS	
540	(21C)	CHARACTER	8	CEEOCB_LIBHEAP64(0)	
540	(21C)	BITSTRING	1	CEEOCB_LIBHEAP64_BIT_FLAG	
		1... ..		CEEOCB_LIBHEAP64_ON	"X'80'"
		.1.. ..		CEEOCB_LIBHEAP64_NOOVERRIDE	"X'40'"
	1		CEEOCB_LIBHEAP64_ON_V	"X'01'"
541	(21D)	BITSTRING	1	*	
542	(21E)	SIGNED	2	CEEOCB_LIBHEAP64_WHERE_SET	
544	(220)	ADDRESS	4	CEEOCB_LIBHEAP64_SUB_OPTIONS	
548	(224)	CHARACTER	8	CEEOCB_STACK64(0)	
548	(224)	BITSTRING	1	CEEOCB_STACK64_BIT_FLAG	
		1... ..		CEEOCB_STACK64_ON	"X'80'"
		.1.. ..		CEEOCB_STACK64_NOOVERRIDE	"X'40'"
	1		CEEOCB_STACK64_ON_V	"X'01'"
549	(225)	BITSTRING	1	*	
550	(226)	SIGNED	2	CEEOCB_STACK64_WHERE_SET	
552	(228)	ADDRESS	4	CEEOCB_STACK64_SUB_OPTIONS	
556	(22C)	CHARACTER	8	CEEOCB_THREADSTACK64(0)	
556	(22C)	BITSTRING	1	CEEOCB_THREADSTACK64_BIT_FLAG	
		1... ..		CEEOCB_THREADSTACK64_ON	"X'80'"
		.1.. ..		CEEOCB_THREADSTACK64_NOOVERRIDE	"X'40'"
	1		CEEOCB_THREADSTACK64_ON_V	"X'01'"
557	(22D)	BITSTRING	1	*	
558	(22E)	SIGNED	2	CEEOCB_THREADSTACK64_WHERE_SET	

Figure 167. Options control block (OCB) field descriptions (Part 14)

CEEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
560	(230)	ADDRESS	4	CEEEOCB_THREADSTACK64_SUB_OPTIONS	
564	(234)	CHARACTER	8	CEEEOCB_DYNDUMP(0)	
564	(234)	BITSTRING	1	CEEEOCB_DYNDUMP_BIT_FLAG	
		1... ..		CEEEOCB_DYNDUMP_ON	"X'80'"
		.1... ..		CEEEOCB_DYNDUMP_NOOVERRIDE	"X'40'"
	1		CEEEOCB_DYNDUMP_ON_V	"X'01'"
565	(235)	BITSTRING	1	*	
566	(236)	SIGNED	2	CEEEOCB_DYNDUMP_WHERE_SET	
568	(238)	ADDRESS	4	CEEEOCB_DYNDUMP_SUB_OPTIONS	
572	(23C)	CHARACTER	8	CEEEOCB_CEEDUMP(0)	
572	(23C)	BITSTRING	1	CEEEOCB_CEEDUMP_BIT_FLAG	
		1... ..		CEEEOCB_CEEDUMP_ON	"X'80'"
		.1... ..		CEEEOCB_CEEDUMP_NOOVERRIDE	"X'40'"
	1		CEEEOCB_CEEDUMP_ON_V	"X'01'"
573	(23D)	BITSTRING	1	*	
574	(23E)	SIGNED	2	CEEEOCB_CEEDUMP_WHERE_SET	
576	(240)	ADDRESS	4	CEEEOCB_CEEDUMP_SUB_OPTIONS	
580	(244)	CHARACTER	8	CEEEOCB_PAGEFRAMESIZE(0)	
580	(244)	BITSTRING	1	CEEEOCB_PAGEFRAMESIZE_BIT_FLAG	
		1... ..		CEEEOCB_PAGEFRAMESIZE_ON	"X'80'"
		.1... ..		CEEEOCB_PAGEFRAMESIZE_NOOVERRIDE	"X'40'"
	1		CEEEOCB_PAGEFRAMESIZE_ON_V	"X'01'"
581	(245)	BITSTRING	1	*	
582	(246)	SIGNED	2	CEEEOCB_PAGEFRAMESIZE_WHERE_SET	
584	(248)	ADDRESS	4	CEEEOCB_PAGEFRAMESIZE_SUB_OPTIONS	
588	(24C)	BITSTRING	1	CEEEOCB_HEAPZONES_BIT_FLAG	
		1... ..		CEEEOCB_HEAPZONES_ON	"X'80'"
		.1... ..		CEEEOCB_HEAPZONES_NOOVERRIDE	"X'40'"
	1		CEEEOCB_HEAPZONES_ON_V	"X'01'"
589	(24D)	BITSTRING	1	*	
590	(24E)	SIGNED	2	CEEEOCB_HEAPZONES_WHERE_SET	
592	(250)	ADDRESS	4	CEEEOCB_HEAPZONES_SUB_OPTIONS	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEEOCB_ABTERMENC_SUB_OPTS	
0	(0)	BITSTRING	1	CEEEOCB_ABTERMENC_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEEOCB_ABTERMENC_EXITMODE	
0	(0)	STRUCTURE	0	CEEEOCB_BELOWHEAP_SUB_OPTS	
0	(0)	BITSTRING	1	CEEEOCB_BELOWHEAP_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEEOCB_BELOWHEAP_INIT_SIZE	
8	(8)	SIGNED	4	CEEEOCB_BELOWHEAP_INCR_SIZE	
12	(C)	BITSTRING	1	CEEEOCB_BELOWHEAP_SUB_BIT_FLAG	
		1... ..		CEEEOCB_BELOWHEAP_LOCATION	"X'80'"
		.1... ..		CEEEOCB_BELOWHEAP_DISPOSITION	"X'40'"
0	(0)	STRUCTURE	0	CEEEOCB_COUNTRY_SUB_OPTS	
0	(0)	BITSTRING	1	CEEEOCB_COUNTRY_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	CHARACTER	2	CEEEOCB_COUNTRY_CODE	
0	(0)	STRUCTURE	0	CEEEOCB_DEPTHCONDLMT_SUB_OPTS	
0	(0)	BITSTRING	1	CEEEOCB_DEPTHCONDLMT_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEEOCB_DEPTHCONDLMT_N	
0	(0)	STRUCTURE	0	CEEEOCB_ENVAR_SUB_OPTS	
0	(0)	BITSTRING	1	CEEEOCB_ENVAR_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	ADDRESS	4	CEEEOCB_ENVAR_STRING_0	

Figure 168. Options control block (OCB) field descriptions (Part 15)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_ENVAR_STRING_S	
0	(0)	CHARACTER	1	CEEOCB_ENVAR_STRING(0)	
0	(0)	SIGNED	2	CEEOCB_ENVAR_STRING_LENGTH	
2	(2)	CHARACTER	250	CEEOCB_ENVAR_STRING_STRING	
0	(0)	STRUCTURE	0	CEEOCB_ERRCOUNT_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_ERRCOUNT_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_ERRCOUNT_N	
0	(0)	STRUCTURE	0	CEEOCB_ERRUNIT_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_ERRUNIT_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_ERRUNIT_N	
0	(0)	STRUCTURE	0	CEEOCB_FLOWC_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_FLOWC_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_FLOWC_MAX_PROCEDURES	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_HEAP_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_HEAP_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_HEAP_INIT_SIZE	
8	(8)	SIGNED	4	CEEOCB_HEAP_INCR_SIZE	
12	(C)	BITSTRING	1	CEEOCB_HEAP_SUB_BIT_FLAG	
		1... ..		CEEOCB_HEAP_LOCATION	"X'80'"
		.1..		CEEOCB_HEAP_DISPOSITION	"X'40'"
13	(D)	BITSTRING	1	*(3)	
16	(10)	SIGNED	4	CEEOCB_HEAP_INITSZ24	
20	(14)	SIGNED	4	CEEOCB_HEAP_INCRSZ24	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_LIBSTACK_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_LIBSTACK_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_LIBSTACK_INIT_SIZE	
8	(8)	SIGNED	4	CEEOCB_LIBSTACK_INCR_SIZE	
12	(C)	BITSTRING	1	CEEOCB_LIBSTACK_SUB_BIT_FLAG	
		1... ..		CEEOCB_LIBSTACK_LOCATION	"X'80'"
		.1..		CEEOCB_LIBSTACK_DISPOSITION	"X'40'"

Figure 169. Options control block (OCB) field descriptions (Part 16)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_MSGFILE_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_MSGFILE_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	ADDRESS	4	CEEOCB_MSGFILE_DDNAME_0	
8	(8)	ADDRESS	4	CEEOCB_MSGFILE_RECFCM_0	
12	(C)	SIGNED	4	CEEOCB_MSGFILE_LRECL	
16	(10)	SIGNED	4	CEEOCB_MSGFILE_BLKSIZE	
0	(0)	STRUCTURE	0	CEEOCB_MSGFILE_DDNAME_S	
0	(0)	CHARACTER	1	CEEOCB_MSGFILE_DDNAME(0)	
0	(0)	SIGNED	2	CEEOCB_MSGFILE_DDNAME_LENGTH	
2	(2)	CHARACTER	8	CEEOCB_MSGFILE_DDNAME_STRING	
10	(A)	CHARACTER	1	*	
11	(B)	CHARACTER	1	CEEOCB_MSGFILE_DDNAME_ENQ	
0	(0)	STRUCTURE	0	CEEOCB_MSGFILE_RECFCM_S	
0	(0)	CHARACTER	1	CEEOCB_MSGFILE_RECFCM(0)	
0	(0)	SIGNED	2	CEEOCB_MSGFILE_RECFCM_LENGTH	
2	(2)	CHARACTER	4	CEEOCB_MSGFILE_RECFCM_STRING	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_NATLANG_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_NATLANG_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	CHARACTER	3	CEEOCB_NATLANG_NATIONAL_LANG	
7	(7)	BITSTRING	1	CEEOCB_NATLANG_SUB_BIT_FLAG	
		1... ..		CEEOCB_NATLANG_UENGLISH	"X'80'"

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_PRTUNIT_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_PRTUNIT_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_PRTUNIT_N	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_PUNUNIT_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_PUNUNIT_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_PUNUNIT_N	

Figure 170. Options control block (OCB) field descriptions (Part 17)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_RDRUNIT_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_RDRUNIT_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_RDRUNIT_N	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_RECPAD_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_RECPAD_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_RECPAD_LEVEL	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_STACK_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_STACK_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_STACK_INIT_SIZE	
8	(8)	SIGNED	4	CEEOCB_STACK_INCR_SIZE	
12	(C)	BITSTRING	1	CEEOCB_STACK_SUB_BIT_FLAG	
		1... ..		CEEOCB_STACK_LOCATION	"X'80'"
		.1..		CEEOCB_STACK_DISPOSITION	"X'40'"
13	(D)	BITSTRING	1	*(3)	
16	(10)	SIGNED	4	CEEOCB_STACK_DSINIT_SIZE	
20	(14)	SIGNED	4	CEEOCB_STACK_DSINCR_SIZE	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	12	CEEOCB_STORAGE_SUB_OPTS	
0	(0)	BITSTRING	4	CEEOCB_STORAGE_SUB_OPTS_V	
		1... ..		CEEOCB_STORAGE_HEAP_ALLOC_V	"X'80'"
		.1..		CEEOCB_STORAGE_HEAP_FREE_V	"X'40'"
		..1.		CEEOCB_STORAGE_DSA_ALLOC_V	"X'20'"
		...1		CEEOCB_STORAGE_RESERVE_SIZE_V	"X'10'"
1	(1)	BITSTRING POS	1	*(3)	
4	(4)	BITSTRING	1	CEEOCB_STORAGE_SUB_OPTS_FLAGS	
		1... ..		CEEOCB_STORAGE_HEAP_ALLOC_SET	"X'80'"
		.1..		CEEOCB_STORAGE_HEAP_FREE_SET	"X'40'"
		..1.		CEEOCB_STORAGE_DSA_ALLOC_SET	"X'20'"
		...1		CEEOCB_STORAGE_DSA_CLEAR_SET	"X'10'"
5	(5)	CHARACTER	1	CEEOCB_STORAGE_HEAP_ALLOC_VALUE	
6	(6)	CHARACTER	1	CEEOCB_STORAGE_HEAP_FREE_VALUE	
7	(7)	CHARACTER	1	CEEOCB_STORAGE_DSA_ALLOC_VALUE	
8	(8)	SIGNED	4	CEEOCB_STORAGE_RESERVE_SIZE	

Figure 171. Options control block (OCB) field descriptions (Part 18)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_AUTOTASK_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_AUTOTASK_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	ADDRESS	4	CEEOCB_AUTOTASK_LOADMOD_O	
8	(8)	SIGNED	4	CEEOCB_AUTOTASK_NTASKS	
0	(0)	STRUCTURE	0	CEEOCB_AUTOTASK_LOADMOD_S	
0	(0)	CHARACTER	1	CEEOCB_AUTOTASK_LOADMOD(0)	
0	(0)	SIGNED	2	CEEOCB_AUTOTASK_LOADMOD_LENGTH	
2	(2)	CHARACTER	8	CEEOCB_AUTOTASK_LOADMOD_STRING	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_TEST_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_TEST_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_TEST_CONTROL	
8	(8)	ADDRESS	4	CEEOCB_TEST_COMMAND_FILE_O	
12	(C)	ADDRESS	4	CEEOCB_TEST_INIT_COMMAND_O	
16	(10)	ADDRESS	4	CEEOCB_TEST_PREFERENCE_FILE_O	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_TEST_COMMAND_FILE_S	
0	(0)	CHARACTER	1	CEEOCB_TEST_COMMAND_FILE(0)	
0	(0)	SIGNED	2	CEEOCB_TEST_COMMAND_FILE_LEN	
2	(2)	CHARACTER	80	CEEOCB_TEST_COMMAND_FILE_STR	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_TEST_INIT_COMMAND_S	
0	(0)	CHARACTER	1	CEEOCB_TEST_INIT_COMMAND(0)	
0	(0)	SIGNED	2	CEEOCB_TEST_INIT_COMMAND_LEN	
2	(2)	CHARACTER	250	CEEOCB_TEST_INIT_COMMAND_STR	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_TEST_PREFERENCE_FILE_S	
0	(0)	CHARACTER	1	CEEOCB_TEST_PREFERENCE_FILE(0)	
0	(0)	SIGNED	2	CEEOCB_TEST_PREFERENCE_FILE_LEN	
2	(2)	CHARACTER	80	CEEOCB_TEST_PREFERENCE_FILE_STR	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_THREADSTACK_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_THREADSTACK_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_THREADSTACK_INIT_SIZE	
8	(8)	SIGNED	4	CEEOCB_THREADSTACK_INCR_SIZE	
12	(C)	BITSTRING	1	CEEOCB_THREADSTACK_SUB_BIT_FLAG	

Figure 172. Options control block (OCB) field descriptions (Part 19)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
		1... ..		CEEOCB_THREADSTACK_LOCATION	"X'80'"
		.1.. ..		CEEOCB_THREADSTACK_DISPOSITION	"X'40'"
13	(D)	BITSTRING	1	*(3)	
16	(10)	SIGNED	4	CEEOCB_THREADSTACK_DSINIT_SIZE	
20	(14)	SIGNED	4	CEEOCB_THREADSTACK_DSINCR_SIZE	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_TRACE_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_TRACE_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_TRACE_TBL_SIZE	
8	(8)	BITSTRING	4	CEEOCB_TRACE_GLOBAL	
12	(C)	BITSTRING	1	CEEOCB_TRACE_FLAGS(4)	
16	(10)	ADDRESS	4	CEEOCB_TRACE_LVL_V_O	
20	(14)	ADDRESS	4	CEEOCB_TRACE_LVL_S_O	
24	(18)	ADDRESS	4	CEEOCB_TRACE_LVL_O	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_TRACE_LVL_V	
0	(0)	BITSTRING	1	CEEOCB_TRACE_LVL_V_FLAGS(4)	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_TRACE_LVL_S	
0	(0)	BITSTRING	1	CEEOCB_TRACE_LVL_S_FLAGS(4)	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_TRACE_LVL	
0	(0)	BITSTRING	1	CEEOCB_TRACE_LEVELS(0)	
0	(0)	BITSTRING	4	*	
4	(4)	BITSTRING	4	CEEOCB_TRACE_CEL	
8	(8)	BITSTRING	4	*	
12	(C)	BITSTRING	4	CEEOCB_TRACE_C370	
16	(10)	BITSTRING	20	*	
36	(24)	BITSTRING	4	*	
40	(28)	BITSTRING	4	CEEOCB_TRACE_PLI	
44	(2C)	BITSTRING	4	*	
48	(30)	BITSTRING	4	CEEOCB_TRACE_SOCKET	
52	(34)	BITSTRING	20	*	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_UPSI_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_UPSI_N_V	

Figure 173. Options control block (OCB) field descriptions (Part 20)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
1	(1)	BITSTRING	1	*(3)	
4	(4)	CHARACTER	8	CEEOCB_UPSI_N	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_USRHDLR_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_USRHDLR_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	ADDRESS	4	CEEOCB_USRHDLR_ROUTINE_0	
8	(8)	ADDRESS	4	CEEOCB_USRHDLR_SUPERHDLR_0	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_USRHDLR_ROUTINE_S	
0	(0)	CHARACTER	1	CEEOCB_USRHDLR_ROUTINE(0)	
0	(0)	SIGNED	2	CEEOCB_USRHDLR_ROUTINE_LENGTH	
2	(2)	CHARACTER	8	CEEOCB_USRHDLR_ROUTINE_STRING	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_USRHDLR_SUPERHDLR_S	
0	(0)	CHARACTER	1	CEEOCB_USRHDLR_SUPERHDLR(0)	
0	(0)	SIGNED	2	CEEOCB_USRHDLR_SUPERHDLR_LENGTH	
2	(2)	CHARACTER	8	CEEOCB_USRHDLR_SUPERHDLR_STRING	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_NAMELIST_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_NAMELIST_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_NAMELIST_LEVEL	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_XUFLOW_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_XUFLOW_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_XUFLOW_LEVEL	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_ANYHEAP_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_ANYHEAP_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_ANYHEAP_INIT_SIZE	
8	(8)	SIGNED	4	CEEOCB_ANYHEAP_INCR_SIZE	
12	(C)	BITSTRING	1	CEEOCB_ANYHEAP_SUB_BIT_FLAG	
		1... ..		CEEOCB_ANYHEAP_LOCATION	"X'80' "

Figure 174. Options control block (OCB) field descriptions (Part 21)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
		.1..		CEEEOCB_ANYHEAP_DISPOSITION	"X'40'"

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEEOCB_MSGQ_SUB_OPTS	
0	(0)	BITSTRING	1	CEEEOCB_MSGQ_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEEOCB_MSGQ_N	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEEOCB_ABPerc_SUB_OPTS	
0	(0)	BITSTRING	1	CEEEOCB_ABPerc_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	BITSTRING	1	CEEEOCB_ABPerc_SUB_OPTS_FLAGS	
		1...		CEEEOCB_ABPerc_NONE	"X'80'"
		.1..		CEEEOCB_ABPerc_USER	"X'40'"
		..1.		CEEEOCB_ABPerc_SYST	"X'20'"
		...1		CEEEOCB_ABPerc_OTHR	"X'10'"
5	(5)	BITSTRING	1	*(3)	
8	(8)	SIGNED	4	CEEEOCB_ABPerc_ABNUM	@LI0021A
12	(C)	CHARACTER	8	CEEEOCB_ABPerc_ABCODE	@LI0021C

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEEOCB_TERMTHDACT_SUB_OPTS	
0	(0)	BITSTRING	1	CEEEOCB_TERMTHDACT_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEEOCB_TERMTHDACT_LEVEL	
8	(8)	BITSTRING	1	CEEEOCB_TERMTHDACT_CICSDEST	
9	(9)	BITSTRING	1	*(3)	NOT USED
12	(C)	SIGNED	2	CEEEOCB_TERMTHDACT_REGSTOR	
14	(E)	BITSTRING	1	*(2)	NOT USED

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEEOCB_THREADHEAP_SUB_OPTS	
0	(0)	BITSTRING	1	CEEEOCB_THREADHEAP_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEEOCB_THREADHEAP_INIT_SIZE	
8	(8)	SIGNED	4	CEEEOCB_THREADHEAP_INCR_SIZE	
12	(C)	BITSTRING	1	CEEEOCB_THREADHEAP_SUB_BIT_FLAG	
		1...		CEEEOCB_THREADHEAP_LOCATION	"X'80'"

Figure 175. Options control block (OCB) field descriptions (Part 22)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
		.1..		CEEOCB_THREADHEAP_DISPOSITION	"X'40'"

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_NONIPTSTACK_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_NONIPTSTACK_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_NONIPTSTACK_INIT_SIZE	
8	(8)	SIGNED	4	CEEOCB_NONIPTSTACK_INCR_SIZE	
12	(C)	BITSTRING	1	CEEOCB_NONIPTSTACK_SUB_BIT_FLAG	
		1...		CEEOCB_NONIPTSTACK_LOCATION	"X'80'"
		.1..		CEEOCB_NONIPTSTACK_DISPOSITION	"X'40'"

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_PLITASKCOUNT_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_PLITASKCOUNT_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_PLITASKCOUNT_TASKS	

This option is now obsolete - Do not use

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_LIBRARY_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_LIBRARY_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	ADDRESS	4	CEEOCB_LIBRARY_NAME_0	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_LIBRARY_NAME_S	
0	(0)	CHARACTER	1	CEEOCB_LIBRARY_NAME(0)	
0	(0)	SIGNED	2	CEEOCB_LIBRARY_NAME_LENGTH	
2	(2)	CHARACTER	8	CEEOCB_LIBRARY_NAME_STRING	

This option is now obsolete - Do not use

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_VERSION_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_VERSION_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	ADDRESS	4	CEEOCB_VERSION_NAME_0	
0	(0)	STRUCTURE	0	CEEOCB_VERSION_NAME_S	
0	(0)	CHARACTER	1	CEEOCB_VERSION_NAME(0)	
0	(0)	SIGNED	2	CEEOCB_VERSION_NAME_LENGTH	
2	(2)	CHARACTER	8	CEEOCB_VERSION_NAME_STRING	

Figure 176. Options control block (OCB) field descriptions (Part 23)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_HEAPCHK_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_HEAPCHK_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	4	CEEOCB_HEAPCHK_FREQUENCY	
8	(8)	SIGNED	4	CEEOCB_HEAPCHK_DELAY	
12	(C)	SIGNED	4	CEEOCB_HEAPCHK_CALL_LEVEL	
16	(10)	SIGNED	4	CEEOCB_HEAPCHK_POOL_CALL_LEVEL	
20	(14)	SIGNED	4	CEEOCB_HEAPCHK_POOL_ENTRIES	
24	(18)	SIGNED	4	CEEOCB_HEAPCHK_POOL_NUMBER	
28	(1C)	SIGNED	4	CEEOCB_HEAPCHK_POOL_ENTRIES31	
32	(20)	SIGNED	4	CEEOCB_HEAPCHK_POOL_NUMBER31	
=====					
OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_PROFILE_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_PROFILE_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	ADDRESS	4	CEEOCB_PROFILE_STRING_0	
=====					
OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_PROFILE_STRING_S	
0	(0)	CHARACTER	1	CEEOCB_PROFILE_STRING(0)	
0	(0)	SIGNED	2	CEEOCB_PROFILE_STRING_LENGTH	
2	(2)	CHARACTER	250	CEEOCB_PROFILE_STRING_STRING	
=====					
OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_HEAPPOOLS_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_HEAPPOOLS_SUB_OPTS_V(4)	
4	(4)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL1_SIZE	
8	(8)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL1_PRCNT	
12	(C)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL2_SIZE	
16	(10)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL2_PRCNT	
20	(14)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL3_SIZE	
24	(18)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL3_PRCNT	
28	(1C)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL4_SIZE	
32	(20)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL4_PRCNT	
36	(24)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL5_SIZE	
40	(28)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL5_PRCNT	
44	(2C)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL6_SIZE	
48	(30)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL6_PRCNT	
52	(34)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL7_SIZE	
56	(38)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL7_PRCNT	
60	(3C)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL8_SIZE	
64	(40)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL8_PRCNT	
68	(44)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL9_SIZE	
72	(48)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL9_PRCNT	
76	(4C)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL10_SIZE	
80	(50)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL10_PRCNT	
84	(54)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL11_SIZE	
88	(58)	SIGNED	4	CEEOCB_HEAPPOOLS_POOL11_PRCNT	

Figure 177. Options control block (OCB) field descriptions (Part 24)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
92	(5C)	SIGNED	4	CEEOCB_HEAPPOLLS_POOL12_SIZE	
96	(60)	SIGNED	4	CEEOCB_HEAPPOLLS_POOL12_PRCNT	
100	(64)	BITSTRING	1	CEEOCB_HEAPPOLLS_SUB_OPTS_V_2(4)	
104	(68)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL1_POOLS	
105	(69)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL2_POOLS	
106	(6A)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL3_POOLS	
107	(6B)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL4_POOLS	
108	(6C)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL5_POOLS	
109	(6D)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL6_POOLS	
110	(6E)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL7_POOLS	
111	(6F)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL8_POOLS	
112	(70)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL9_POOLS	
113	(71)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL10_POOLS	
114	(72)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL11_POOLS	
115	(73)	BITSTRING	1	CEEOCB_HEAPPOLLS_POOL12_POOLS	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_INFOMSGFILTER_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_INFOMSGFILTER_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	CHARACTER	1	CEEOCB_INFOMSGFILTER_ENV1	
5	(5)	CHARACTER	1	CEEOCB_INFOMSGFILTER_ENV2	
6	(6)	CHARACTER	1	CEEOCB_INFOMSGFILTER_ENV3	
7	(7)	CHARACTER	1	CEEOCB_INFOMSGFILTER_ENV4	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_TRAP_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_TRAP_SUB_OPTS_V	
		1... ..		CEEOCB_TRAP_SPIE_V	"X'80' "
1	(1)	BITSTRING	1	*(3)	
4	(4)	BITSTRING	1	CEEOCB_TRAP_FLAGS	
		1... ..		CEEOCB_TRAP_SPIE	"X'80' "
5	(5)	BITSTRING	1	*(3)	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_FILETAG_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_FILETAG_SUB_OPTS_V	
		1... ..		CEEOCB_FILETAG_AUTOCVT_V	"X'80' "
		.1..		CEEOCB_FILETAG_AUTOTAG_V	"X'40' "
1	(1)	BITSTRING	1	*(3)	
4	(4)	BITSTRING	1	CEEOCB_FILETAG_FLAGS	
		1... ..		CEEOCB_FILETAG_AUTOCVT	"X'80' "
		.1..		CEEOCB_FILETAG_AUTOTAG	"X'40' "
5	(5)	BITSTRING	1	*(3)	

Figure 178. Options control block (OCB) field descriptions (Part 25)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_HEAP64_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_HEAP64_SUB_OPTS_V(2)	9 valid bits
2	(2)	BITSTRING	1	*(2)	
4	(4)	SIGNED	8	CEEOCB_HEAP64_INIT_SIZE64	
12	(C)	SIGNED	8	CEEOCB_HEAP64_INCR_SIZE64	
20	(14)	BITSTRING	1	CEEOCB_HEAP64_SUB_BIT_FLAG64	
		.1..		CEEOCB_HEAP64_DISPOSITION64	"X'40'"
		..1.		CEEOCB_HEAP64_FILL64	"X'20'"
21	(15)	BITSTRING	1	*(3)	
24	(18)	SIGNED	4	CEEOCB_HEAP64_INIT_SIZE31	
28	(1C)	SIGNED	4	CEEOCB_HEAP64_INCR_SIZE31	
32	(20)	BITSTRING	1	CEEOCB_HEAP64_SUB_BIT_FLAG31	
		.1..		CEEOCB_HEAP64_DISPOSITION31	"X'40'"
33	(21)	BITSTRING	1	*(3)	
36	(24)	SIGNED	4	CEEOCB_HEAP64_INIT_SIZE24	
40	(28)	SIGNED	4	CEEOCB_HEAP64_INCR_SIZE24	
44	(2C)	BITSTRING	1	CEEOCB_HEAP64_SUB_BIT_FLAG24	
		.1..		CEEOCB_HEAP64_DISPOSITION24	"X'40'"
45	(2D)	BITSTRING	1	*(3)	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_HEAPPPOOLS64_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_HEAPPPOOLS64_SUB_OPTS_V(4)	32 valid bits
4	(4)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL1_SIZE	
8	(8)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL1_COUNT	
12	(C)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL2_SIZE	
16	(10)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL2_COUNT	
20	(14)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL3_SIZE	
24	(18)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL3_COUNT	
28	(1C)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL4_SIZE	
32	(20)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL4_COUNT	
36	(24)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL5_SIZE	
40	(28)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL5_COUNT	
44	(2C)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL6_SIZE	
48	(30)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL6_COUNT	
52	(34)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL7_SIZE	
56	(38)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL7_COUNT	
60	(3C)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL8_SIZE	
64	(40)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL8_COUNT	
68	(44)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL9_SIZE	
72	(48)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL9_COUNT	
76	(4C)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL10_SIZE	
80	(50)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL10_COUNT	
84	(54)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL11_SIZE	
88	(58)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL11_COUNT	
92	(5C)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL12_SIZE	
96	(60)	SIGNED	4	CEEOCB_HEAPPPOOLS64_POOL12_COUNT	
100	(64)	BITSTRING	1	CEEOCB_HEAPPPOOLS64_SUB_OPTS_V_2(4)	
104	(68)	BITSTRING	1	CEEOCB_HEAPPPOOLS64_POOL1_POOLS	
105	(69)	BITSTRING	1	CEEOCB_HEAPPPOOLS64_POOL2_POOLS	
106	(6A)	BITSTRING	1	CEEOCB_HEAPPPOOLS64_POOL3_POOLS	
107	(6B)	BITSTRING	1	CEEOCB_HEAPPPOOLS64_POOL4_POOLS	
108	(6C)	BITSTRING	1	CEEOCB_HEAPPPOOLS64_POOL5_POOLS	

Figure 179. Options control block (OCB) field descriptions (Part 26)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
109	(6D)	BITSTRING	1	CEEOCB_HEAPPOLLS64_POOL6_POOLS	
110	(6E)	BITSTRING	1	CEEOCB_HEAPPOLLS64_POOL7_POOLS	
111	(6F)	BITSTRING	1	CEEOCB_HEAPPOLLS64_POOL8_POOLS	
112	(70)	BITSTRING	1	CEEOCB_HEAPPOLLS64_POOL9_POOLS	
113	(71)	BITSTRING	1	CEEOCB_HEAPPOLLS64_POOL10_POOLS	
114	(72)	BITSTRING	1	CEEOCB_HEAPPOLLS64_POOL11_POOLS	
115	(73)	BITSTRING	1	CEEOCB_HEAPPOLLS64_POOL12_POOLS	
OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_IOHEAP64_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_IOHEAP64_SUB_OPTS_V(2)	9 valid bits
2	(2)	BITSTRING	1	*(2)	
4	(4)	SIGNED	8	CEEOCB_IOHEAP64_INIT_SIZE64	
12	(C)	SIGNED	8	CEEOCB_IOHEAP64_INCR_SIZE64	
20	(14)	BITSTRING	1	CEEOCB_IOHEAP64_SUB_BIT_FLAG64	
		.1..		CEEOCB_IOHEAP64_DISPOSITION64	"X'40' "
21	(15)	BITSTRING	1	*(3)	
24	(18)	SIGNED	4	CEEOCB_IOHEAP64_INIT_SIZE31	
28	(1C)	SIGNED	4	CEEOCB_IOHEAP64_INCR_SIZE31	
32	(20)	BITSTRING	1	CEEOCB_IOHEAP64_SUB_BIT_FLAG31	
		.1..		CEEOCB_IOHEAP64_DISPOSITION31	"X'40' "
33	(21)	BITSTRING	1	*(3)	
36	(24)	SIGNED	4	CEEOCB_IOHEAP64_INIT_SIZE24	
40	(28)	SIGNED	4	CEEOCB_IOHEAP64_INCR_SIZE24	
44	(2C)	BITSTRING	1	CEEOCB_IOHEAP64_SUB_BIT_FLAG24	
		.1..		CEEOCB_IOHEAP64_DISPOSITION24	"X'40' "
45	(2D)	BITSTRING	1	*(3)	
OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_LIBHEAP64_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_LIBHEAP64_SUB_OPTS_V(2)	9 valid bits
2	(2)	BITSTRING	1	*(2)	
4	(4)	SIGNED	8	CEEOCB_LIBHEAP64_INIT_SIZE64	
12	(C)	SIGNED	8	CEEOCB_LIBHEAP64_INCR_SIZE64	
20	(14)	BITSTRING	1	CEEOCB_LIBHEAP64_SUB_BIT_FLAG64	
		.1..		CEEOCB_LIBHEAP64_DISPOSITION64	"X'40' "
21	(15)	BITSTRING	1	*(3)	
24	(18)	SIGNED	4	CEEOCB_LIBHEAP64_INIT_SIZE31	
28	(1C)	SIGNED	4	CEEOCB_LIBHEAP64_INCR_SIZE31	
32	(20)	BITSTRING	1	CEEOCB_LIBHEAP64_SUB_BIT_FLAG31	
		.1..		CEEOCB_LIBHEAP64_DISPOSITION31	"X'40' "
33	(21)	BITSTRING	1	*(3)	
36	(24)	SIGNED	4	CEEOCB_LIBHEAP64_INIT_SIZE24	
40	(28)	SIGNED	4	CEEOCB_LIBHEAP64_INCR_SIZE24	
44	(2C)	BITSTRING	1	CEEOCB_LIBHEAP64_SUB_BIT_FLAG24	
		.1..		CEEOCB_LIBHEAP64_DISPOSITION24	"X'40' "
45	(2D)	BITSTRING	1	*(3)	

Figure 180. Options control block (OCB) field descriptions (Part 27)

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_STACK64_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_STACK64_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	8	CEEOCB_STACK64_INIT_SIZE	
12	(C)	SIGNED	8	CEEOCB_STACK64_INCR_SIZE	
20	(14)	SIGNED	8	CEEOCB_STACK64_MAX_SIZE	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_THREADSTACK64_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_THREADSTACK64_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	8	CEEOCB_THREADSTACK64_INIT_SIZE	
12	(C)	SIGNED	8	CEEOCB_THREADSTACK64_INCR_SIZE	
20	(14)	SIGNED	8	CEEOCB_THREADSTACK64_MAX_SIZE	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_DYNDUMP_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_DYNDUMP_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	ADDRESS	4	CEEOCB_DYNDUMP_HLQ_0	
8	(8)	BITSTRING	1	CEEOCB_DYNDUMP_4039_FLAGS	CEEOCB_DYNDUMP_4039_DYNAMIC "X'80"
		1... ..		CEEOCB_DYNDUMP_4039_NODYNAMIC	"X'40"
		.1..		CEEOCB_DYNDUMP_4039_FORCE	"X'20"
		...1		CEEOCB_DYNDUMP_4039_BOTH	"X'10"
9	(9)	BITSTRING	1	CEEOCB_DYNDUMP_40XX_FLAGS	CEEOCB_DYNDUMP_40XX_TDUMP "X'80"
		1... ..		CEEOCB_DYNDUMP_40XX_NOTDUMP	"X'40"
10	(A)	BITSTRING	1	*(2)	

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_DYNDUMP_HLQ_S	
0	(0)	CHARACTER	1	CEEOCB_DYNDUMP_HLQ(0)	
0	(0)	SIGNED	2	CEEOCB_DYNDUMP_HLQ_LENGTH	
2	(2)	CHARACTER	26	CEEOCB_DYNDUMP_HLQ_STRING	

Figure 181. Options control block (OCB) field descriptions (Part 28)

CEEOCB Macro

OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	16	CEEOCB_CEEDUMP_SUB_OPTS	
0	(0)	BIT(32)	4	CEEOCB_CEEDUMP_SUB_OPTS_V	
		1... ..		CEEOCB_CEEDUMP_PAGELEN_V	
		.1.. ..		CEEOCB_CEEDUMP_SYSOUT_CLASS_V	
		..1.		CEEOCB_CEEDUMP_SYSOUT_FNAME_V	
		...1		CEEOCB_CEEDUMP_FREE_V	
	 1...		CEEOCB_CEEDUMP_SPIN_V	
0	(0)	BIT(27) POS(6)	4	*	
4	(4)	UNSIGNED	4	CEEOCB_CEEDUMP_PAGELEN	
8	(8)	CHARACTER	4	CEEOCB_CEEDUMP_SYSOUT_FNAME	
12	(C)	CHARACTER	1	CEEOCB_CEEDUMP_SYSOUT_CLASS	
13	(D)	BIT(8)	1	CEEOCB_CEEDUMP_FREE	0: FREE=END 1: FREE=CLOSE
14	(E)	BIT(8)	1	CEEOCB_CEEDUMP_SPIN	0: SPIN=UNALLOC 1: SPIN=NO
15	(F)	CHARACTER	1	*	
OFFSET DECIMAL	OFFSET HEX	TYPE	LENGTH	NAME (DIM)	DESCRIPTION
0	(0)	STRUCTURE	0	CEEOCB_PAGEFRAMESIZE_SUB_OPTS	
0	(0)	BITSTRING	1	CEEOCB_PAGEFRAMESIZE_SUB_OPTS_V	
1	(1)	BITSTRING	1	*(3)	
4	(4)	SIGNED	2	CEEOCB_PAGEFRAMESIZE_HEAP	
6	(6)	SIGNED	2	CEEOCB_PAGEFRAMESIZE_ANYHEAP	
8	(8)	SIGNED	2	CEEOCB_PAGEFRAMESIZE_STACK	
10	(A)	SIGNED	2	*	

END OF CEEOCB

Figure 182. Options control block (OCB) field descriptions (Part 29)

The OCB cross reference information is shown in Figure 183 on page 841 through Figure 198 on page 856.

1 CROSS REFERENCE

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB	0		1
CEEOCB_ABPERC	164		2
CEEOCB_ABPERC_ABCODE	C		2
CEEOCB_ABPERC_ABNUM	8		2
CEEOCB_ABPERC_BIT_FLAG	164		2
CEEOCB_ABPERC_NONE	4	80	2
CEEOCB_ABPERC_NOOVERRIDE	164	40	2
CEEOCB_ABPERC_ON	164	80	2
CEEOCB_ABPERC_ON_V	164	1	2
CEEOCB_ABPERC_OTHR	4	10	2
CEEOCB_ABPERC_SUB_OPTIONS	168		2
CEEOCB_ABPERC_SUB_OPTS	0		1
CEEOCB_ABPERC_SUB_OPTS_FLAGS	4		2
CEEOCB_ABPERC_SUB_OPTS_V	0		2
CEEOCB_ABPERC_SYST	4	20	2
CEEOCB_ABPERC_USER	4	40	2
CEEOCB_ABPERC_WHERE_SET	166		2
CEEOCB_ABTERMENC	44		2
CEEOCB_ABTERMENC_BIT_FLAG	44		2
CEEOCB_ABTERMENC_EXITMODE	4		2
CEEOCB_ABTERMENC_NOOVERRIDE	44	40	2
CEEOCB_ABTERMENC_ON	44	80	2
CEEOCB_ABTERMENC_ON_V	44	1	2
CEEOCB_ABTERMENC_SUB_OPTIONS	48		2
CEEOCB_ABTERMENC_SUB_OPTS	0		1
CEEOCB_ABTERMENC_SUB_OPTS_V	0		2
CEEOCB_ABTERMENC_WHERE_SET	46		2
CEEOCB_AIXBLD	1C		2
CEEOCB_AIXBLD_BIT_FLAG	1C		2
CEEOCB_AIXBLD_NOOVERRIDE	1C	40	2
CEEOCB_AIXBLD_ON	1C	80	2
CEEOCB_AIXBLD_ON_V	1C	1	2
CEEOCB_AIXBLD_SUB_OPTIONS	20		2
CEEOCB_AIXBLD_WHERE_SET	1E		2
CEEOCB_ALL31	24		2
CEEOCB_ALL31_BIT_FLAG	24		2
CEEOCB_ALL31_NOOVERRIDE	24	40	2
CEEOCB_ALL31_ON	24	80	2
CEEOCB_ALL31_ON_V	24	1	2
CEEOCB_ALL31_SUB_OPTIONS	28		2
CEEOCB_ALL31_WHERE_SET	26		2
CEEOCB_ANYHEAP	15C		2
CEEOCB_ANYHEAP_BIT_FLAG	15C		2
CEEOCB_ANYHEAP_DISPOSITION	C	40	2
CEEOCB_ANYHEAP_INCR_SIZE	8		2
CEEOCB_ANYHEAP_INIT_SIZE	4		2
CEEOCB_ANYHEAP_LOCATION	C	80	2
CEEOCB_ANYHEAP_NOOVERRIDE	15C	40	2
CEEOCB_ANYHEAP_ON	15C	80	2
CEEOCB_ANYHEAP_ON_V	15C	1	2
CEEOCB_ANYHEAP_SUB_BIT_FLAG	C		2
CEEOCB_ANYHEAP_SUB_OPTIONS	160		2
CEEOCB_ANYHEAP_SUB_OPTS	0		1
CEEOCB_ANYHEAP_SUB_OPTS_V	0		2
CEEOCB_ANYHEAP_WHERE_SET	15E		2
CEEOCB_AREA_AREA	0		2
CEEOCB_AUTOTASK	F4		2
CEEOCB_AUTOTASK_BIT_FLAG	F4		2
CEEOCB_AUTOTASK_LOADMOD	0		2
CEEOCB_AUTOTASK_LOADMOD_LENGTH	0		2
CEEOCB_AUTOTASK_LOADMOD_O	4		2
CEEOCB_AUTOTASK_LOADMOD_S	0		1
CEEOCB_AUTOTASK_LOADMOD_STRING	2		2
CEEOCB_AUTOTASK_NOOVERRIDE	F4	40	2
CEEOCB_AUTOTASK_NTASKS	8		2

Figure 183. Options control block (OCB) field descriptions (cross references 1)

CEEEOCB Macro

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEEOCB_AUTOTASK_ON	F4	80	2
CEEEOCB_AUTOTASK_ON_V	F4	1	2
CEEEOCB_AUTOTASK_SUB_OPTIONS	F8		2
CEEEOCB_AUTOTASK_SUB_OPTS	0		1
CEEEOCB_AUTOTASK_SUB_OPTS_V	0		2
CEEEOCB_AUTOTASK_WHERE_SET	F6		2
CEEEOCB_BELOWHEAP	2C		2
CEEEOCB_BELOWHEAP_BIT_FLAG	2C		2
CEEEOCB_BELOWHEAP_DISPOSITION	C	40	2
CEEEOCB_BELOWHEAP_INCR_SIZE	8		2
CEEEOCB_BELOWHEAP_INIT_SIZE	4		2
CEEEOCB_BELOWHEAP_LOCATION	C	80	2
CEEEOCB_BELOWHEAP_NOOVERRIDE	2C	40	2
CEEEOCB_BELOWHEAP_ON	2C	80	2
CEEEOCB_BELOWHEAP_ON_V	2C	1	2
CEEEOCB_BELOWHEAP_SUB_BIT_FLAG	C		2
CEEEOCB_BELOWHEAP_SUB_OPTIONS	30		2
CEEEOCB_BELOWHEAP_SUB_OPTS	0		1
CEEEOCB_BELOWHEAP_SUB_OPTS_V	0		2
CEEEOCB_BELOWHEAP_WHERE_SET	2E		2
CEEEOCB_CBLOPTS	144		2
CEEEOCB_CBLOPTS_BIT_FLAG	144		2
CEEEOCB_CBLOPTS_NOOVERRIDE	144	40	2
CEEEOCB_CBLOPTS_ON	144	80	2
CEEEOCB_CBLOPTS_ON_V	144	1	2
CEEEOCB_CBLOPTS_SUB_OPTIONS	148		2
CEEEOCB_CBLOPTS_WHERE_SET	146		2
CEEEOCB_CBLPSHPOP	17C		2
CEEEOCB_CBLPSHPOP_BIT_FLAG	17C		2
CEEEOCB_CBLPSHPOP_NOOVERRIDE	17C	40	2
CEEEOCB_CBLPSHPOP_ON	17C	80	2
CEEEOCB_CBLPSHPOP_ON_V	17C	1	2
CEEEOCB_CBLPSHPOP_SUB_OPTIONS	180		2
CEEEOCB_CBLPSHPOP_WHERE_SET	17E		2
CEEEOCB_CBLQDA	184		2
CEEEOCB_CBLQDA_BIT_FLAG	184		2
CEEEOCB_CBLQDA_NOOVERRIDE	184	40	2
CEEEOCB_CBLQDA_ON	184	80	2
CEEEOCB_CBLQDA_ON_V	184	1	2
CEEEOCB_CBLQDA_SUB_OPTIONS	188		2
CEEEOCB_CBLQDA_WHERE_SET	186		2
CEEEOCB_CEEDUMP	23C		2
CEEEOCB_CEEDUMP_BIT_FLAG	23C		3
CEEEOCB_CEEDUMP_FREE	D		2
CEEEOCB_CEEDUMP_FREE_V	0	10	3
CEEEOCB_CEEDUMP_NOOVERRIDE	23C	40	4
CEEEOCB_CEEDUMP_ON	23C	80	4
CEEEOCB_CEEDUMP_ON_V	23C	01	4
CEEEOCB_CEEDUMP_PAGELN	4		2
CEEEOCB_CEEDUMP_PAGELN_V	0	80	3
CEEEOCB_CEEDUMP_SPIN	E		2
CEEEOCB_CEEDUMP_SPIN_V	0	08	3
CEEEOCB_CEEDUMP_SUB_OPTIONS	240		3
CEEEOCB_CEEDUMP_SUB_OPTS	0		1
CEEEOCB_CEEDUMP_SUB_OPTS_V	0		2
CEEEOCB_CEEDUMP_SYSOUT_CLASS	C		2
CEEEOCB_CEEDUMP_SYSOUT_CLASS_V	0	40	3
CEEEOCB_CEEDUMP_SYSOUT_FNAME	8		2
CEEEOCB_CEEDUMP_SYSOUT_FNAME_V	0	20	3
CEEEOCB_CEEDUMP_WHERE_SET	23E		3

Figure 184. Options control block (OCB) field descriptions (cross references 2)

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_CHECK	34		2
CEEOCB_CHECK_BIT_FLAG	34		2
CEEOCB_CHECK_NOOVERRIDE	34	40	2
CEEOCB_CHECK_ON	34	80	2
CEEOCB_CHECK_ON_V	34	1	2
CEEOCB_CHECK_SUB_OPTIONS	38		2
CEEOCB_CHECK_WHERE_SET	36		2
CEEOCB_COUNTRY	4C		2
CEEOCB_COUNTRY_BIT_FLAG	4C		2
CEEOCB_COUNTRY_CODE	4		2
CEEOCB_COUNTRY_NOOVERRIDE	4C	40	2
CEEOCB_COUNTRY_ON	4C	80	2
CEEOCB_COUNTRY_ON_V	4C	1	2
CEEOCB_COUNTRY_SUB_OPTIONS	50		2
CEEOCB_COUNTRY_SUB_OPTS	0		1
CEEOCB_COUNTRY_SUB_OPTS_V	0		2
CEEOCB_COUNTRY_WHERE_SET	4E		2
CEEOCB_DEBUG	54		2
CEEOCB_DEBUG_BIT_FLAG	54		2
CEEOCB_DEBUG_NOOVERRIDE	54	40	2
CEEOCB_DEBUG_ON	54	80	2
CEEOCB_DEBUG_ON_V	54	1	2
CEEOCB_DEBUG_SUB_OPTIONS	58		2
CEEOCB_DEBUG_WHERE_SET	56		2
CEEOCB_DEPTHCONDLMT	174		2
CEEOCB_DEPTHCONDLMT_BIT_FLAG	174		2
CEEOCB_DEPTHCONDLMT_N	4		2
CEEOCB_DEPTHCONDLMT_NOOVERRIDE	174	40	2
CEEOCB_DEPTHCONDLMT_ON	174	80	2
CEEOCB_DEPTHCONDLMT_ON_V	174	1	2
CEEOCB_DEPTHCONDLMT_SUB_OPTIONS	178		2
CEEOCB_DEPTHCONDLMT_SUB_OPTS	0		1
CEEOCB_DEPTHCONDLMT_SUB_OPTS_V	0		2
CEEOCB_DEPTHCONDLMT_WHERE_SET	176		2
CEEOCB_DYNDUMP	234		2
CEEOCB_DYNDUMP_4039_BOTH	8	10	2
CEEOCB_DYNDUMP_4039_DYNAMIC	8	80	2
CEEOCB_DYNDUMP_4039_FLAGS	8		2
CEEOCB_DYNDUMP_4039_FORCE	8	20	2
CEEOCB_DYNDUMP_4039_NODYNAMIC	8	40	2
CEEOCB_DYNDUMP_40XX_FLAGS	9		2
CEEOCB_DYNDUMP_40XX_NOTDUMP	9	40	2
CEEOCB_DYNDUMP_40XX_TDUMP	9	80	2
CEEOCB_DYNDUMP_BIT_FLAG	234		2
CEEOCB_DYNDUMP_HLQ	0		2
CEEOCB_DYNDUMP_HLQ_O	4		2
CEEOCB_DYNDUMP_HLQ_LENGTH	0		2
CEEOCB_DYNDUMP_HLQ_S	0		1
CEEOCB_DYNDUMP_HLQ_STRING	2		2
CEEOCB_DYNDUMP_NOOVERRIDE	234	40	2
CEEOCB_DYNDUMP_ON	234	80	2
CEEOCB_DYNDUMP_ON_V	234	1	2
CEEOCB_DYNDUMP_SUB_OPTIONS	238		2
CEEOCB_DYNDUMP_SUB_OPTS	0		1
CEEOCB_DYNDUMP_SUB_OPTS_V	0		2
CEEOCB_DYNDUMP_WHERE_SET	236		2
CEEOCB_ENVAR	6C		2
CEEOCB_ENVAR_BIT_FLAG	6C		2
CEEOCB_ENVAR_ON	6C	80	2
CEEOCB_ENVAR_ON_V	6C	1	2
CEEOCB_ENVAR_STRING	0		2
CEEOCB_ENVAR_STRING_LENGTH	0		2
CEEOCB_ENVAR_STRING_O	4		2
CEEOCB_ENVAR_STRING_S	0		1
CEEOCB_ENVAR_STRING_STRING	2		2

Figure 185. Options control block (OCB) field descriptions (cross references 3)

CEEOCB Macro

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_ENVAR_SUB_OPTIONS	70		2
CEEOCB_ENVAR_SUB_OPTS	0		1
CEEOCB_ENVAR_SUB_OPTS_V	0		2
CEEOCB_ENVAR_WHERE_SET	6E		2
CEEOCB_ENVART_NOOVERRIDE	6C	40	2
CEEOCB_ERRCOUNT	5C		2
CEEOCB_ERRCOUNT_BIT_FLAG	5C		2
CEEOCB_ERRCOUNT_N	4		2
CEEOCB_ERRCOUNT_NOOVERRIDE	5C	40	2
CEEOCB_ERRCOUNT_ON	5C	80	2
CEEOCB_ERRCOUNT_ON_V	5C	1	2
CEEOCB_ERRCOUNT_SUB_OPTIONS	60		2
CEEOCB_ERRCOUNT_SUB_OPTS	0		1
CEEOCB_ERRCOUNT_SUB_OPTS_V	0		2
CEEOCB_ERRCOUNT_WHERE_SET	5E		2
CEEOCB_ERRUNIT	B4		2
CEEOCB_ERRUNIT_BIT_FLAG	B4		2
CEEOCB_ERRUNIT_N	4		2
CEEOCB_ERRUNIT_NOOVERRIDE	B4	40	2
CEEOCB_ERRUNIT_ON	B4	80	2
CEEOCB_ERRUNIT_ON_V	B4	1	2
CEEOCB_ERRUNIT_SUB_OPTIONS	B8		2
CEEOCB_ERRUNIT_SUB_OPTS	0		1
CEEOCB_ERRUNIT_SUB_OPTS_V	0		2
CEEOCB_ERRUNIT_WHERE_SET	B6		2
CEEOCB_EYECATCHER	0		2
CEEOCB_FILEHIST	64		2
CEEOCB_FILEHIST_BIT_FLAG	64		2
CEEOCB_FILEHIST_NOOVERRIDE	64	40	2
CEEOCB_FILEHIST_ON	64	80	2
CEEOCB_FILEHIST_ON_V	64	1	2
CEEOCB_FILEHIST_SUB_OPTIONS	68		2
CEEOCB_FILEHIST_WHERE_SET	66		2
CEEOCB_FILETAG	1FC		2
CEEOCB_FILETAG_AUTOCVT	4	80	2
CEEOCB_FILETAG_AUTOCVT_V	0	80	2
CEEOCB_FILETAG_AUTOTAG	4	40	2
CEEOCB_FILETAG_AUTOTAG_V	0	40	2
CEEOCB_FILETAG_BIT_FLAG	1FC		2
CEEOCB_FILETAG_FLAGS	4		2
CEEOCB_FILETAG_NOOVERRIDE	1FC	40	2
CEEOCB_FILETAG_ON	1FC	80	2
CEEOCB_FILETAG_ON_V	1FC	1	2
CEEOCB_FILETAG_SUB_OPTIONS	200		2
CEEOCB_FILETAG_SUB_OPTS	0		1
CEEOCB_FILETAG_SUB_OPTS_V	0		2
CEEOCB_FILETAG_WHERE_SET	1FE		2
CEEOCB_FLOWC	74		2
CEEOCB_FLOWC_BIT_FLAG	74		2
CEEOCB_FLOWC_MAX_PROCEDURES	4		2
CEEOCB_FLOWC_NOOVERRIDE	74	40	2
CEEOCB_FLOWC_ON	74	80	2
CEEOCB_FLOWC_ON_V	74	1	2
CEEOCB_FLOWC_SUB_OPTIONS	78		2
CEEOCB_FLOWC_SUB_OPTS	0		1
CEEOCB_FLOWC_SUB_OPTS_V	0		2
CEEOCB_FLOWC_WHERE_SET	76		2
CEEOCB_FORMAT	10		2
CEEOCB_FORMAT_31	10	0	2
CEEOCB_FORMAT_64	10	1	2

Figure 186. Options control block (OCB) field descriptions (cross references 4)

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_HEAP	7C		2
CEEOCB_HEAP_BIT_FLAG	7C		2
CEEOCB_HEAP_DISPOSITION	C	40	2
CEEOCB_HEAP_INCR_SIZE	8		2
CEEOCB_HEAP_INCRSZ24	14		2
CEEOCB_HEAP_INIT_SIZE	4		2
CEEOCB_HEAP_INITSZ24	10		2
CEEOCB_HEAP_LOCATION	C	80	2
CEEOCB_HEAP_NOOVERRIDE	7C	40	2
CEEOCB_HEAP_ON	7C	80	2
CEEOCB_HEAP_ON_V	7C	1	2
CEEOCB_HEAP_SUB_BIT_FLAG	C		2
CEEOCB_HEAP_SUB_OPTIONS	80		2
CEEOCB_HEAP_SUB_OPTS	0		1
CEEOCB_HEAP_SUB_OPTS_V	0		2
CEEOCB_HEAP_WHERE_SET	7E		2
CEEOCB_HEAPCHK	1D4		2
CEEOCB_HEAPCHK_BIT_FLAG	1D4		2
CEEOCB_HEAPCHK_CALL_LEVEL	C		2
CEEOCB_HEAPCHK_DELAY	8		2
CEEOCB_HEAPCHK_FREQUENCY	4		2
CEEOCB_HEAPCHK_NOOVERRIDE	1D4	40	2
CEEOCB_HEAPCHK_ON	1D4	80	2
CEEOCB_HEAPCHK_ON_V	1D4	1	2
CEEOCB_HEAPCHK_POOL_CALL_LEVEL	10		2
CEEOCB_HEAPCHK_POOL_ENTRIES	14		2
CEEOCB_HEAPCHK_POOL_ENTRIES31	1C		2
CEEOCB_HEAPCHK_POOL_NUMBER	18		2
CEEOCB_HEAPCHK_POOL_NUMBER31	20		2
CEEOCB_HEAPCHK_SUB_OPTIONS	1D8		2
CEEOCB_HEAPCHK_SUB_OPTS	0		1
CEEOCB_HEAPCHK_SUB_OPTS_V	0		2
CEEOCB_HEAPCHK_WHERE_SET	1D6		2
CEEOCB_HEAPPOLS	1E4		2
CEEOCB_HEAPPOLS_BIT_FLAG	1E4		2
CEEOCB_HEAPPOLS_NOOVERRIDE	1E4	40	2
CEEOCB_HEAPPOLS_ON	1E4	80	2
CEEOCB_HEAPPOLS_ON_V	1E4	1	2
CEEOCB_HEAPPOLS_POOL1_PRCNT	8		2
CEEOCB_HEAPPOLS_POOL1_SIZE	4		2
CEEOCB_HEAPPOLS_POOL10_PRCNT	50		2
CEEOCB_HEAPPOLS_POOL10_SIZE	4C		2
CEEOCB_HEAPPOLS_POOL11_PRCNT	58		2
CEEOCB_HEAPPOLS_POOL11_SIZE	54		2
CEEOCB_HEAPPOLS_POOL12_PRCNT	60		2
CEEOCB_HEAPPOLS_POOL12_SIZE	5C		2
CEEOCB_HEAPPOLS_POOL2_PRCNT	10		2
CEEOCB_HEAPPOLS_POOL2_SIZE	C		2
CEEOCB_HEAPPOLS_POOL3_PRCNT	18		2
CEEOCB_HEAPPOLS_POOL3_SIZE	14		2
CEEOCB_HEAPPOLS_POOL4_PRCNT	20		2
CEEOCB_HEAPPOLS_POOL4_SIZE	1C		2
CEEOCB_HEAPPOLS_POOL5_PRCNT	28		2
CEEOCB_HEAPPOLS_POOL5_SIZE	24		2
CEEOCB_HEAPPOLS_POOL6_PRCNT	30		2
CEEOCB_HEAPPOLS_POOL6_SIZE	2C		2
CEEOCB_HEAPPOLS_POOL7_PRCNT	38		2
CEEOCB_HEAPPOLS_POOL7_SIZE	34		2

Figure 187. Options control block (OCB) field descriptions (cross references 5)

CEEOCB Macro

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_HEAPPOLLS_POOL8_PRCNT	40		2
CEEOCB_HEAPPOLLS_POOL8_SIZE	3C		2
CEEOCB_HEAPPOLLS_POOL9_PRCNT	48		2
CEEOCB_HEAPPOLLS_POOL9_SIZE	44		2
CEEOCB_HEAPPOLLS_SUB_OPTIONS	1E8		2
CEEOCB_HEAPPOLLS_SUB_OPTS	0		1
CEEOCB_HEAPPOLLS_SUB_OPTS_V	0		2
CEEOCB_HEAPPOLLS_WHERE_SET	1E6		2
CEEOCB_HEAPPOLLS64	20C		2
CEEOCB_HEAPPOLLS64_BIT_FLAG	20C		2
CEEOCB_HEAPPOLLS64_NOOVERRIDE	20C	40	2
CEEOCB_HEAPPOLLS64_ON	20C	80	2
CEEOCB_HEAPPOLLS64_ON_V	20C	1	2
CEEOCB_HEAPPOLLS64_POOL1_COUNT	8		2
CEEOCB_HEAPPOLLS64_POOL1_SIZE	4		2
CEEOCB_HEAPPOLLS64_POOL10_COUNT	50		2
CEEOCB_HEAPPOLLS64_POOL10_SIZE	4C		2
CEEOCB_HEAPPOLLS64_POOL11_COUNT	58		2
CEEOCB_HEAPPOLLS64_POOL11_SIZE	54		2
CEEOCB_HEAPPOLLS64_POOL12_COUNT	60		2
CEEOCB_HEAPPOLLS64_POOL12_SIZE	5C		2
CEEOCB_HEAPPOLLS64_POOL2_COUNT	10		2
CEEOCB_HEAPPOLLS64_POOL2_SIZE	C		2
CEEOCB_HEAPPOLLS64_POOL3_COUNT	18		2
CEEOCB_HEAPPOLLS64_POOL3_SIZE	14		2
CEEOCB_HEAPPOLLS64_POOL4_COUNT	20		2
CEEOCB_HEAPPOLLS64_POOL4_SIZE	1C		2
CEEOCB_HEAPPOLLS64_POOL5_COUNT	28		2
CEEOCB_HEAPPOLLS64_POOL5_SIZE	24		2
CEEOCB_HEAPPOLLS64_POOL6_COUNT	30		2
CEEOCB_HEAPPOLLS64_POOL6_SIZE	2C		2
CEEOCB_HEAPPOLLS64_POOL7_COUNT	38		2
CEEOCB_HEAPPOLLS64_POOL7_SIZE	34		2
CEEOCB_HEAPPOLLS64_POOL8_COUNT	40		2
CEEOCB_HEAPPOLLS64_POOL8_SIZE	3C		2
CEEOCB_HEAPPOLLS64_POOL9_COUNT	48		2
CEEOCB_HEAPPOLLS64_POOL9_SIZE	44		2
CEEOCB_HEAPPOLLS64_SUB_OPTIONS	210		2
CEEOCB_HEAPPOLLS64_SUB_OPTS	0		1
CEEOCB_HEAPPOLLS64_SUB_OPTS_V	0		2
CEEOCB_HEAPPOLLS64_WHERE_SET	20E		2
CEEOCB_HEAP64	204		2
CEEOCB_HEAP64_BIT_FLAG	204		2
CEEOCB_HEAP64_DISPOSITION24	2C	40	2
CEEOCB_HEAP64_DISPOSITION31	20	40	2
CEEOCB_HEAP64_DISPOSITION64	14	40	2
CEEOCB_HEAP64_INCR_SIZE24	28		2
CEEOCB_HEAP64_INCR_SIZE31	1C		2
CEEOCB_HEAP64_INCR_SIZE64	C		2
CEEOCB_HEAP64_INIT_SIZE24	24		2
CEEOCB_HEAP64_INIT_SIZE31	18		2
CEEOCB_HEAP64_INIT_SIZE64	4		2
CEEOCB_HEAP64_NOOVERRIDE	204	40	2
CEEOCB_HEAP64_ON	204	80	2
CEEOCB_HEAP64_ON_V	204	1	2

Figure 188. Options control block (OCB) field descriptions (cross references 6)

NAME	HEX OFFSET	HEX VALUE	LEVEL
====	=====	=====	=====
CEEOCB_HEAP64_SUB_BIT_FLAG24	2C		2
CEEOCB_HEAP64_SUB_BIT_FLAG31	20		2
CEEOCB_HEAP64_SUB_BIT_FLAG64	14		2
CEEOCB_HEAP64_SUB_OPTIONS	208		2
CEEOCB_HEAP64_SUB_OPTS	0		1
CEEOCB_HEAP64_SUB_OPTS_V	0		2
CEEOCB_HEAP64_WHERE_SET	206		2
CEEOCB_INFOMSGFILTER	1EC		2
CEEOCB_INFOMSGFILTER_BIT_FLAG	1EC		2
CEEOCB_INFOMSGFILTER_ENV1	4		2
CEEOCB_INFOMSGFILTER_ENV2	5		2
CEEOCB_INFOMSGFILTER_ENV3	6		2
CEEOCB_INFOMSGFILTER_ENV4	7		2
CEEOCB_INFOMSGFILTER_NOOVERRIDE	1EC	40	2
CEEOCB_INFOMSGFILTER_ON	1EC	80	2
CEEOCB_INFOMSGFILTER_ON_V	1EC	1	2
CEEOCB_INFOMSGFILTER_SUB_OPTIONS	1F0		2
CEEOCB_INFOMSGFILTER_SUB_OPTS	0		1
CEEOCB_INFOMSGFILTER_SUB_OPTS_V	0		2
CEEOCB_INFOMSGFILTER_WHERE_SET	1EE		2
CEEOCB_INQPCOPN	84		2
CEEOCB_INQPCOPN_BIT_FLAG	84		2
CEEOCB_INQPCOPN_NOOVERRIDE	84	40	2
CEEOCB_INQPCOPN_ON	84	80	2
CEEOCB_INQPCOPN_ON_V	84	1	2
CEEOCB_INQPCOPN_SUB_OPTIONS	88		2
CEEOCB_INQPCOPN_WHERE_SET	86		2
CEEOCB_INTERRUPT	8C		2
CEEOCB_INTERRUPT_BIT_FLAG	8C		2
CEEOCB_INTERRUPT_NOOVERRIDE	8C	40	2
CEEOCB_INTERRUPT_ON	8C	80	2
CEEOCB_INTERRUPT_ON_V	8C	1	2
CEEOCB_INTERRUPT_SUB_OPTIONS	90		2
CEEOCB_INTERRUPT_WHERE_SET	8E		2
CEEOCB_IOHEAP64	214		2
CEEOCB_IOHEAP64_BIT_FLAG	214		2
CEEOCB_IOHEAP64_DISPOSITION24	2C	40	2
CEEOCB_IOHEAP64_DISPOSITION31	20	40	2
CEEOCB_IOHEAP64_DISPOSITION64	14	40	2
CEEOCB_IOHEAP64_INCR_SIZE24	28		2
CEEOCB_IOHEAP64_INCR_SIZE31	1C		2
CEEOCB_IOHEAP64_INCR_SIZE64	C		2
CEEOCB_IOHEAP64_INIT_SIZE24	24		2
CEEOCB_IOHEAP64_INIT_SIZE31	18		2
CEEOCB_IOHEAP64_INIT_SIZE64	4		2
CEEOCB_IOHEAP64_NOOVERRIDE	214	40	2
CEEOCB_IOHEAP64_ON	214	80	2
CEEOCB_IOHEAP64_ON_V	214	1	2
CEEOCB_IOHEAP64_SUB_BIT_FLAG24	2C		2
CEEOCB_IOHEAP64_SUB_BIT_FLAG31	20		2
CEEOCB_IOHEAP64_SUB_BIT_FLAG64	14		2
CEEOCB_IOHEAP64_SUB_OPTIONS	218		2
CEEOCB_IOHEAP64_SUB_OPTS	0		1
CEEOCB_IOHEAP64_SUB_OPTS_V	0		2
CEEOCB_IOHEAP64_WHERE_SET	216		2
CEEOCB_LENGTH	A		2

Figure 189. Options control block (OCB) field descriptions (cross references 7)

CEEOCB Macro

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_LIBHEAP64	21C		2
CEEOCB_LIBHEAP64_BIT_FLAG	21C		2
CEEOCB_LIBHEAP64_DISPOSITION24	2C	40	2
CEEOCB_LIBHEAP64_DISPOSITION31	20	40	2
CEEOCB_LIBHEAP64_DISPOSITION64	14	40	2
CEEOCB_LIBHEAP64_INCR_SIZE24	28		2
CEEOCB_LIBHEAP64_INCR_SIZE31	1C		2
CEEOCB_LIBHEAP64_INCR_SIZE64	C		2
CEEOCB_LIBHEAP64_INIT_SIZE24	24		2
CEEOCB_LIBHEAP64_INIT_SIZE31	18		2
CEEOCB_LIBHEAP64_INIT_SIZE64	4		2
CEEOCB_LIBHEAP64_NOOVERRIDE	21C	40	2
CEEOCB_LIBHEAP64_ON	21C	80	2
CEEOCB_LIBHEAP64_ON_V	21C	1	2
CEEOCB_LIBHEAP64_SUB_BIT_FLAG24	2C		2
CEEOCB_LIBHEAP64_SUB_BIT_FLAG31	20		2
CEEOCB_LIBHEAP64_SUB_BIT_FLAG64	14		2
CEEOCB_LIBHEAP64_SUB_OPTIONS	220		2
CEEOCB_LIBHEAP64_SUB_OPTS	0		1
CEEOCB_LIBHEAP64_SUB_OPTS_V	0		2
CEEOCB_LIBHEAP64_WHERE_SET	21E		2
CEEOCB_LIBRARY	1BC		2
CEEOCB_LIBRARY_BIT_FLAG	1BC		2
CEEOCB_LIBRARY_NAME	0		2
CEEOCB_LIBRARY_NAME_LENGTH	0		2
CEEOCB_LIBRARY_NAME_O	4		2
CEEOCB_LIBRARY_NAME_S	0		1
CEEOCB_LIBRARY_NAME_STRING	2		2
CEEOCB_LIBRARY_NOOVERRIDE	1BC	40	2
CEEOCB_LIBRARY_ON	1BC	80	2
CEEOCB_LIBRARY_ON_V	1BC	1	2
CEEOCB_LIBRARY_SUB_OPTIONS	1C0		2
CEEOCB_LIBRARY_SUB_OPTS	0		1
CEEOCB_LIBRARY_SUB_OPTS_V	0		2
CEEOCB_LIBRARY_WHERE_SET	1BE		2
CEEOCB_LIBSTACK	94		2
CEEOCB_LIBSTACK_BIT_FLAG	94		2
CEEOCB_LIBSTACK_DISPOSITION	C	40	2
CEEOCB_LIBSTACK_INCR_SIZE	8		2
CEEOCB_LIBSTACK_INIT_SIZE	4		2
CEEOCB_LIBSTACK_LOCATION	C	80	2
CEEOCB_LIBSTACK_NOOVERRIDE	94	40	2
CEEOCB_LIBSTACK_ON	94	80	2
CEEOCB_LIBSTACK_ON_V	94	1	2
CEEOCB_LIBSTACK_SUB_BIT_FLAG	C		2
CEEOCB_LIBSTACK_SUB_OPTIONS	98		2
CEEOCB_LIBSTACK_SUB_OPTS	0		1
CEEOCB_LIBSTACK_SUB_OPTS_V	0		2
CEEOCB_LIBSTACK_WHERE_SET	96		2
CEEOCB_MSGFILE	A4		2
CEEOCB_MSGFILE_BIT_FLAG	A4		2
CEEOCB_MSGFILE_BLKSIZE	10		2
CEEOCB_MSGFILE_DDNAME	0		2
CEEOCB_MSGFILE_DDNAME_ENQ	B		2
CEEOCB_MSGFILE_DDNAME_LENGTH	0		2
CEEOCB_MSGFILE_DDNAME_O	4		2
CEEOCB_MSGFILE_DDNAME_S	0		1
CEEOCB_MSGFILE_DDNAME_STRING	2		2
CEEOCB_MSGFILE_LRECL	C		2
CEEOCB_MSGFILE_NOOVERRIDE	A4	40	2
CEEOCB_MSGFILE_ON	A4	80	2
CEEOCB_MSGFILE_ON_V	A4	1	2

Figure 190. Options control block (OCB) field descriptions (cross references 8)

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEEOCB_MSGFILE_RECFM	0		2
CEEEOCB_MSGFILE_RECFM_LENGTH	0		2
CEEEOCB_MSGFILE_RECFM_O	8		2
CEEEOCB_MSGFILE_RECFM_S	0		1
CEEEOCB_MSGFILE_RECFM_STRING	2		2
CEEEOCB_MSGFILE_SUB_OPTIONS	A8		2
CEEEOCB_MSGFILE_SUB_OPTS	0		1
CEEEOCB_MSGFILE_SUB_OPTS_V	0		2
CEEEOCB_MSGFILE_WHERE_SET	A6		2
CEEEOCB_MSGQ	9C		2
CEEEOCB_MSGQ_BIT_FLAG	9C		2
CEEEOCB_MSGQ_N	4		2
CEEEOCB_MSGQ_NOOVERRIDE	9C	40	2
CEEEOCB_MSGQ_ON	9C	80	2
CEEEOCB_MSGQ_ON_V	9C	1	2
CEEEOCB_MSGQ_SUB_OPTIONS	A0		2
CEEEOCB_MSGQ_SUB_OPTS	0		1
CEEEOCB_MSGQ_SUB_OPTS_V	0		2
CEEEOCB_MSGQ_WHERE_SET	9E		2
CEEEOCB_NAMELIST	1AC		2
CEEEOCB_NAMELIST_BIT_FLAG	1AC		2
CEEEOCB_NAMELIST_LEVEL	4		2
CEEEOCB_NAMELIST_NOOVERRIDE	1AC	40	2
CEEEOCB_NAMELIST_ON	1AC	80	2
CEEEOCB_NAMELIST_ON_V	1AC	1	2
CEEEOCB_NAMELIST_SUB_OPTIONS	1B0		2
CEEEOCB_NAMELIST_SUB_OPTS	0		1
CEEEOCB_NAMELIST_SUB_OPTS_V	0		2
CEEEOCB_NAMELIST_WHERE_SET	1AE		2
CEEEOCB_NATLANG	AC		2
CEEEOCB_NATLANG_BIT_FLAG	AC		2
CEEEOCB_NATLANG_NATIONAL_LANG	4		2
CEEEOCB_NATLANG_NOOVERRIDE	AC	40	2
CEEEOCB_NATLANG_ON	AC	80	2
CEEEOCB_NATLANG_ON_V	AC	1	2
CEEEOCB_NATLANG_SUB_BIT_FLAG	7		2
CEEEOCB_NATLANG_SUB_OPTIONS	B0		2
CEEEOCB_NATLANG_SUB_OPTS	0		1
CEEEOCB_NATLANG_SUB_OPTS_V	0		2
CEEEOCB_NATLANG_UENGLISH	7	80	2
CEEEOCB_NATLANG_WHERE_SET	AE		2
CEEEOCB_NONIPTSTACK	14C		2
CEEEOCB_NONIPTSTACK_BIT_FLAG	14C		2
CEEEOCB_NONIPTSTACK_DISPOSITON	C	40	2
CEEEOCB_NONIPTSTACK_INCR_SIZE	8		2
CEEEOCB_NONIPTSTACK_INIT_SIZE	4		2
CEEEOCB_NONIPTSTACK_LOCATION	C	80	2
CEEEOCB_NONIPTSTACK_NOOVERRIDE	14C	40	2
CEEEOCB_NONIPTSTACK_ON	14C	80	2
CEEEOCB_NONIPTSTACK_ON_V	14C	1	2
CEEEOCB_NONIPTSTACK_SUB_BIT_FLAG	C		2
CEEEOCB_NONIPTSTACK_SUB_OPTIONS	150		2
CEEEOCB_NONIPTSTACK_SUB_OPTS	0		1
CEEEOCB_NONIPTSTACK_SUB_OPTS_V	0		2
CEEEOCB_NONIPTSTACK_WHERE_SET	14E		2

Figure 191. Options control block (OCB) field descriptions (cross references 9)

CEEOCB Macro

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_OCSTATUS	BC		2
CEEOCB_OCSTATUS_BIT_FLAG	BC		2
CEEOCB_OCSTATUS_NOOVERRIDE	BC	40	2
CEEOCB_OCSTATUS_ON	BC	80	2
CEEOCB_OCSTATUS_ON_V	BC	1	2
CEEOCB_OCSTATUS_SUB_OPTIONS	C0		2
CEEOCB_OCSTATUS_WHERE_SET	BE		2
CEEOCB_PAGEFRAMESIZE	244		2
CEEOCB_PAGEFRAMESIZE_ANYHEAP	6		2
CEEOCB_PAGEFRAMESIZE_ANYHEAP_V	0	40	3
CEEOCB_PAGEFRAMESIZE_BIT_FLAG	244		3
CEEOCB_PAGEFRAMESIZE_HEAP	4		2
CEEOCB_PAGEFRAMESIZE_HEAP_V	0	80	3
CEEOCB_PAGEFRAMESIZE_NOOVERRIDE	244	40	4
CEEOCB_PAGEFRAMESIZE_ON	244	80	4
CEEOCB_PAGEFRAMESIZE_ON_V	244	01	4
CEEOCB_PAGEFRAMESIZE_STACK	8		2
CEEOCB_PAGEFRAMESIZE_STACK_V	0	20	3
CEEOCB_PAGEFRAMESIZE_SUB_OPTIONS	248		3
CEEOCB_PAGEFRAMESIZE_SUB_OPTS	0		1
CEEOCB_PAGEFRAMESIZE_SUB_OPTS_V	0		2
CEEOCB_PAGEFRAMESIZE_WHERE_SET	246		3
CEEOCB_PC	1B4		2
CEEOCB_PC_BIT_FLAG	1B4		2
CEEOCB_PC_NOOVERRIDE	1B4	40	2
CEEOCB_PC_ON	1B4	80	2
CEEOCB_PC_ON_V	1B4	1	2
CEEOCB_PC_SUB_OPTIONS	1B8		2
CEEOCB_PC_WHERE_SET	1B6		2
CEEOCB_PLITASKCOUNT	3C		2
CEEOCB_PLITASKCOUNT_BIT_FLAG	3C		2
CEEOCB_PLITASKCOUNT_NOOVERRIDE	3C	40	2
CEEOCB_PLITASKCOUNT_ON	3C	80	2
CEEOCB_PLITASKCOUNT_ON_V	3C	1	2
CEEOCB_PLITASKCOUNT_SUB_OPTIONS	40		2
CEEOCB_PLITASKCOUNT_SUB_OPTS	0		1
CEEOCB_PLITASKCOUNT_SUB_OPTS_V	0		2
CEEOCB_PLITASKCOUNT_TASKS	4		2
CEEOCB_PLITASKCOUNT_WHERE_SET	3E		2
CEEOCB_POSIX	C4		2
CEEOCB_POSIX_BIT_FLAG	C4		2
CEEOCB_POSIX_NOOVERRIDE	C4	40	2
CEEOCB_POSIX_ON	C4	80	2
CEEOCB_POSIX_ON_V	C4	1	2
CEEOCB_POSIX_SUB_OPTIONS	C8		2
CEEOCB_POSIX_WHERE_SET	C6		2
CEEOCB_PROFILE	1DC		2
CEEOCB_PROFILE_BIT_FLAG	1DC		2
CEEOCB_PROFILE_NOOVERRIDE	1DC	40	2
CEEOCB_PROFILE_ON	1DC	80	2
CEEOCB_PROFILE_ON_V	1DC	1	2
CEEOCB_PROFILE_STRING	0		2
CEEOCB_PROFILE_STRING_LENGTH	0		2
CEEOCB_PROFILE_STRING_0	4		2
CEEOCB_PROFILE_STRING_S	0		1
CEEOCB_PROFILE_STRING_STRING	2		2
CEEOCB_PROFILE_SUB_OPTIONS	1E0		2
CEEOCB_PROFILE_SUB_OPTS	0		1
CEEOCB_PROFILE_SUB_OPTS_V	0		2
CEEOCB_PROFILE_WHERE_SET	1DE		2

Figure 192. Options control block (OCB) field descriptions (cross references 10)

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_PRTUNIT	134		2
CEEOCB_PRTUNIT_BIT_FLAG	134		2
CEEOCB_PRTUNIT_N	4		2
CEEOCB_PRTUNIT_NOOVERRIDE	134	40	2
CEEOCB_PRTUNIT_ON	134	80	2
CEEOCB_PRTUNIT_ON_V	134	1	2
CEEOCB_PRTUNIT_SUB_OPTIONS	138		2
CEEOCB_PRTUNIT_SUB_OPTS	0		1
CEEOCB_PRTUNIT_SUB_OPTS_V	0		2
CEEOCB_PRTUNIT_WHERE_SET	136		2
CEEOCB_PUNUNIT	18C		2
CEEOCB_PUNUNIT_BIT_FLAG	18C		2
CEEOCB_PUNUNIT_N	4		2
CEEOCB_PUNUNIT_NOOVERRIDE	18C	40	2
CEEOCB_PUNUNIT_ON	18C	80	2
CEEOCB_PUNUNIT_ON_V	18C	1	2
CEEOCB_PUNUNIT_SUB_OPTIONS	190		2
CEEOCB_PUNUNIT_SUB_OPTS	0		1
CEEOCB_PUNUNIT_SUB_OPTS_V	0		2
CEEOCB_PUNUNIT_WHERE_SET	18E		2
CEEOCB_RDRUNIT	194		2
CEEOCB_RDRUNIT_BIT_FLAG	194		2
CEEOCB_RDRUNIT_N	4		2
CEEOCB_RDRUNIT_NOOVERRIDE	194	40	2
CEEOCB_RDRUNIT_ON	194	80	2
CEEOCB_RDRUNIT_ON_V	194	1	2
CEEOCB_RDRUNIT_SUB_OPTIONS	198		2
CEEOCB_RDRUNIT_SUB_OPTS	0		1
CEEOCB_RDRUNIT_SUB_OPTS_V	0		2
CEEOCB_RDRUNIT_WHERE_SET	196		2
CEEOCB_RECPAD	19C		2
CEEOCB_RECPAD_BIT_FLAG	19C		2
CEEOCB_RECPAD_LEVEL	4		2
CEEOCB_RECPAD_NOOVERRIDE	19C	40	2
CEEOCB_RECPAD_ON	19C	80	2
CEEOCB_RECPAD_ON_V	19C	1	2
CEEOCB_RECPAD_SUB_OPTIONS	1A0		2
CEEOCB_RECPAD_SUB_OPTS	0		1
CEEOCB_RECPAD_SUB_OPTS_V	0		2
CEEOCB_RECPAD_WHERE_SET	19E		2
CEEOCB_RPTOPTS	154		2
CEEOCB_RPTOPTS_BIT_FLAG	154		2
CEEOCB_RPTOPTS_NOOVERRIDE	154	40	2
CEEOCB_RPTOPTS_ON	154	80	2
CEEOCB_RPTOPTS_ON_V	154	1	2
CEEOCB_RPTOPTS_SUB_OPTIONS	158		2
CEEOCB_RPTOPTS_WHERE_SET	156		2
CEEOCB_RPTSTG	CC		2
CEEOCB_RPTSTG_BIT_FLAG	CC		2
CEEOCB_RPTSTG_NOOVERRIDE	CC	40	2
CEEOCB_RPTSTG_ON	CC	80	2
CEEOCB_RPTSTG_ON_V	CC	1	2
CEEOCB_RPTSTG_SUB_OPTIONS	D0		2
CEEOCB_RPTSTG_WHERE_SET	CE		2

Figure 193. Options control block (OCB) field descriptions (cross references 11)

CEEOCB Macro

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_RSVD1	14		2
CEEOCB_RSVD1_BIT_FLAG	14		2
CEEOCB_RSVD1_NOOVERRIDE	14	40	2
CEEOCB_RSVD1_ON	14	80	2
CEEOCB_RSVD1_ON_V	14	1	2
CEEOCB_RSVD1_SUB_OPTIONS	18		2
CEEOCB_RSVD1_WHERE_SET	16		2
CEEOCB_RTREUS	D4		2
CEEOCB_RTREUS_BIT_FLAG	D4		2
CEEOCB_RTREUS_NOOVERRIDE	D4	40	2
CEEOCB_RTREUS_ON	D4	80	2
CEEOCB_RTREUS_ON_V	D4	1	2
CEEOCB_RTREUS_SUB_OPTIONS	D8		2
CEEOCB_RTREUS_WHERE_SET	D6		2
CEEOCB_RTLS	1CC		2
CEEOCB_RTLS_BIT_FLAG	1CC		2
CEEOCB_RTLS_NOOVERRIDE	1CC	40	2
CEEOCB_RTLS_ON	1CC	80	2
CEEOCB_RTLS_ON_V	1CC	1	2
CEEOCB_RTLS_SUB_OPTIONS	1D0		2
CEEOCB_RTLS_WHERE_SET	1CE		2
CEEOCB_SIMVRD	DC		2
CEEOCB_SIMVRD_BIT_FLAG	DC		2
CEEOCB_SIMVRD_NOOVERRIDE	DC	40	2
CEEOCB_SIMVRD_ON	DC	80	2
CEEOCB_SIMVRD_ON_V	DC	1	2
CEEOCB_SIMVRD_SUB_OPTIONS	E0		2
CEEOCB_SIMVRD_WHERE_SET	DE		2
CEEOCB_STACK	E4		2
CEEOCB_STACK_BIT_FLAG	E4		2
CEEOCB_STACK_DISPOSITION	C	40	2
CEEOCB_STACK_DSINCR_SIZE	14		2
CEEOCB_STACK_DSINIT_SIZE	10		2
CEEOCB_STACK_INCR_SIZE	8		2
CEEOCB_STACK_INIT_SIZE	4		2
CEEOCB_STACK_LOCATION	C	80	2
CEEOCB_STACK_NOOVERRIDE	E4	40	2
CEEOCB_STACK_ON	E4	80	2
CEEOCB_STACK_ON_V	E4	1	2
CEEOCB_STACK_SUB_BIT_FLAG	C		2
CEEOCB_STACK_SUB_OPTIONS	E8		2
CEEOCB_STACK_SUB_OPTS	0		1
CEEOCB_STACK_SUB_OPTS_V	0		2
CEEOCB_STACK_WHERE_SET	E6		2
CEEOCB_STACK64	224		2
CEEOCB_STACK64_BIT_FLAG	224		2
CEEOCB_STACK64_INCR_SIZE	C		2
CEEOCB_STACK64_INIT_SIZE	4		2
CEEOCB_STACK64_MAX_SIZE	14		2
CEEOCB_STACK64_NOOVERRIDE	224	40	2
CEEOCB_STACK64_ON	224	80	2
CEEOCB_STACK64_ON_V	224	1	2
CEEOCB_STACK64_SUB_OPTIONS	228		2
CEEOCB_STACK64_SUB_OPTS	0		1
CEEOCB_STACK64_SUB_OPTS_V	0		2
CEEOCB_STACK64_WHERE_SET	226		2

Figure 194. Options control block (OCB) field descriptions (cross references 12)

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_STORAGE	EC		2
CEEOCB_STORAGE_BIT_FLAG	EC		3
CEEOCB_STORAGE_DSA_ALLOC_SET	4	20	3
CEEOCB_STORAGE_DSA_ALLOC_V	0	20	3
CEEOCB_STORAGE_DSA_ALLOC_VALUE	7		2
CEEOCB_STORAGE_DSA_CLEAR_SET	4	10	3
CEEOCB_STORAGE_HEAP_ALLOC_SET	4	80	3
CEEOCB_STORAGE_HEAP_ALLOC_V	0	80	3
CEEOCB_STORAGE_HEAP_ALLOC_VALUE	5		2
CEEOCB_STORAGE_HEAP_FREE_SET	4	40	3
CEEOCB_STORAGE_HEAP_FREE_V	0	40	3
CEEOCB_STORAGE_HEAP_FREE_VALUE	6		2
CEEOCB_STORAGE_NOOVERRIDE	EC	40	4
CEEOCB_STORAGE_ON	EC	80	4
CEEOCB_STORAGE_ON_V	EC	01	4
CEEOCB_STORAGE_RESERVE_SIZE	8		2
CEEOCB_STORAGE_RESERVE_SIZE_V	0	10	3
CEEOCB_STORAGE_SUB_OPTIONS	F0		3
CEEOCB_STORAGE_SUB_OPTS	0		1
CEEOCB_STORAGE_SUB_OPTS_FLAGS	4		2
CEEOCB_STORAGE_SUB_OPTS_V	0		2
CEEOCB_STORAGE_WHERE_SET	EE		3
CEEOCB_TERMTHDACT	16C		2
CEEOCB_TERMTHDACT_BIT_FLAG	16C		2
CEEOCB_TERMTHDACT_CICSDEST	8		2
CEEOCB_TERMTHDACT_LEVEL	4		2
CEEOCB_TERMTHDACT_NOOVERRIDE	16C	40	2
CEEOCB_TERMTHDACT_ON	16C	80	2
CEEOCB_TERMTHDACT_ON_V	16C	1	2
CEEOCB_TERMTHDACT_REGSTOR	C		2
CEEOCB_TERMTHDACT_SUB_OPTIONS	170		2
CEEOCB_TERMTHDACT_SUB_OPTS	0		1
CEEOCB_TERMTHDACT_SUB_OPTS_V	0		2
CEEOCB_TERMTHDACT_WHERE_SET	16E		2
CEEOCB_TEST	10C		2
CEEOCB_TEST_BIT_FLAG	10C		2
CEEOCB_TEST_COMMAND_FILE	0		2
CEEOCB_TEST_COMMAND_FILE_LEN	0		2
CEEOCB_TEST_COMMAND_FILE_O	8		2
CEEOCB_TEST_COMMAND_FILE_S	0		1
CEEOCB_TEST_COMMAND_FILE_STR	2		2
CEEOCB_TEST_CONTROL	4		2
CEEOCB_TEST_INIT_COMMAND	0		2
CEEOCB_TEST_INIT_COMMAND_LEN	0		2
CEEOCB_TEST_INIT_COMMAND_O	C		2
CEEOCB_TEST_INIT_COMMAND_S	0		1
CEEOCB_TEST_INIT_COMMAND_STR	2		2
CEEOCB_TEST_NOOVERRIDE	10C	40	2
CEEOCB_TEST_ON	10C	80	2
CEEOCB_TEST_ON_V	10C	1	2
CEEOCB_TEST_PREFERENCE_FILE	0		2
CEEOCB_TEST_PREFERENCE_FILE_LEN	0		2
CEEOCB_TEST_PREFERENCE_FILE_O	10		2
CEEOCB_TEST_PREFERENCE_FILE_S	0		1
CEEOCB_TEST_PREFERENCE_FILE_STR	2		2
CEEOCB_TEST_SUB_OPTIONS	110		2
CEEOCB_TEST_SUB_OPTS	0		1
CEEOCB_TEST_SUB_OPTS_V	0		2
CEEOCB_TEST_WHERE_SET	10E		2

Figure 195. Options control block (OCB) field descriptions (cross references 13)

CEEOCB Macro

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_THREADHEAP	104		2
CEEOCB_THREADHEAP_BIT_FLAG	104		2
CEEOCB_THREADHEAP_DISPOSITION	C	40	2
CEEOCB_THREADHEAP_INCR_SIZE	8		2
CEEOCB_THREADHEAP_INIT_SIZE	4		2
CEEOCB_THREADHEAP_LOCATION	C	80	2
CEEOCB_THREADHEAP_NOOVERRIDE	104	40	2
CEEOCB_THREADHEAP_ON	104	80	2
CEEOCB_THREADHEAP_ON_V	104	1	2
CEEOCB_THREADHEAP_SUB_BIT_FLAG	C		2
CEEOCB_THREADHEAP_SUB_OPTIONS	108		2
CEEOCB_THREADHEAP_SUB_OPTS	0		1
CEEOCB_THREADHEAP_SUB_OPTS_V	0		2
CEEOCB_THREADHEAP_WHERE_SET	106		2
CEEOCB_THREADSTACK	114		2
CEEOCB_THREADSTACK_BIT_FLAG	114		2
CEEOCB_THREADSTACK_DISPOSITION	C	40	2
CEEOCB_THREADSTACK_DSINCR_SIZE	14		2
CEEOCB_THREADSTACK_DSINIT_SIZE	10		2
CEEOCB_THREADSTACK_INCR_SIZE	8		2
CEEOCB_THREADSTACK_INIT_SIZE	4		2
CEEOCB_THREADSTACK_LOCATION	C	80	2
CEEOCB_THREADSTACK_NOOVERRIDE	114	40	2
CEEOCB_THREADSTACK_ON	114	80	2
CEEOCB_THREADSTACK_ON_V	114	1	2
CEEOCB_THREADSTACK_SUB_BIT_FLAG	C		2
CEEOCB_THREADSTACK_SUB_OPTIONS	118		2
CEEOCB_THREADSTACK_SUB_OPTS	0		1
CEEOCB_THREADSTACK_SUB_OPTS_V	0		2
CEEOCB_THREADSTACK_WHERE_SET	116		2
CEEOCB_THREADSTACK64	22C		2
CEEOCB_THREADSTACK64_BIT_FLAG	22C		2
CEEOCB_THREADSTACK64_INCR_SIZE	C		2
CEEOCB_THREADSTACK64_INIT_SIZE	4		2
CEEOCB_THREADSTACK64_MAX_SIZE	14		2
CEEOCB_THREADSTACK64_NOOVERRIDE	22C	40	2
CEEOCB_THREADSTACK64_ON	22C	80	2
CEEOCB_THREADSTACK64_ON_V	22C	1	2
CEEOCB_THREADSTACK64_SUB_OPTIONS	230		2
CEEOCB_THREADSTACK64_SUB_OPTS	0		1
CEEOCB_THREADSTACK64_SUB_OPTS_V	0		2
CEEOCB_THREADSTACK64_WHERE_SET	22E		2
CEEOCB_TRACE	FC		2
CEEOCB_TRACE_BIT_FLAG	FC		2
CEEOCB_TRACE_CEL	4		2
CEEOCB_TRACE_C370	C		2
CEEOCB_TRACE_FLAGS	C		2
CEEOCB_TRACE_GLOBAL	8		2
CEEOCB_TRACE_LEVELS	0		2
CEEOCB_TRACE_LVL	0		1
CEEOCB_TRACE_LVL_0	18		2
CEEOCB_TRACE_LVL_S	0		1
CEEOCB_TRACE_LVL_S_FLAGS	0		2
CEEOCB_TRACE_LVL_S_0	14		2
CEEOCB_TRACE_LVL_V	0		1
CEEOCB_TRACE_LVL_V_FLAGS	0		2
CEEOCB_TRACE_LVL_V_0	10		2
CEEOCB_TRACE_NOOVERRIDE	FC	40	2
CEEOCB_TRACE_ON	FC	80	2
CEEOCB_TRACE_ON_V	FC	1	2
CEEOCB_TRACE_PLI	28		2
CEEOCB_TRACE_SOCKET	30		2
CEEOCB_TRACE_SUB_OPTIONS	100		2
CEEOCB_TRACE_SUB_OPTS	0		1
CEEOCB_TRACE_SUB_OPTS_V	0		2
CEEOCB_TRACE_TBL_SIZE	4		2
CEEOCB_TRACE_WHERE_SET	FE		2

Figure 196. Options control block (OCB) field descriptions (cross references 14)

CEEEOCB Macro

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEEOCB_TRAP	11C		2
CEEEOCB_TRAP_BIT_FLAG	11C		2
CEEEOCB_TRAP_FLAGS	4		2
CEEEOCB_TRAP_NOOVERRIDE	11C	40	2
CEEEOCB_TRAP_ON	11C	80	2
CEEEOCB_TRAP_ON_V	11C	1	2
CEEEOCB_TRAP_SPIE	4	80	2
CEEEOCB_TRAP_SPIE_V	0	80	2
CEEEOCB_TRAP_SUB_OPTIONS	120		2
CEEEOCB_TRAP_SUB_OPTS	0		1
CEEEOCB_TRAP_SUB_OPTS_V	0		2
CEEEOCB_TRAP_WHERE_SET	11E		2
CEEEOCB_UPSI	124		2
CEEEOCB_UPSI_BIT_FLAG	124		2
CEEEOCB_UPSI_N	4		2
CEEEOCB_UPSI_N_V	0		2
CEEEOCB_UPSI_NOOVERRIDE	124	40	2
CEEEOCB_UPSI_ON	124	80	2
CEEEOCB_UPSI_ON_V	124	1	2
CEEEOCB_UPSI_SUB_OPTIONS	128		2
CEEEOCB_UPSI_SUB_OPTS	0		1
CEEEOCB_UPSI_WHERE_SET	126		2
CEEEOCB_USRHDLR	1A4		2
CEEEOCB_USRHDLR_BIT_FLAG	1A4		2
CEEEOCB_USRHDLR_NOOVERRIDE	1A4	40	2
CEEEOCB_USRHDLR_ON	1A4	80	2
CEEEOCB_USRHDLR_ON_V	1A4	1	2
CEEEOCB_USRHDLR_ROUTINE	0		2
CEEEOCB_USRHDLR_ROUTINE_LENGTH	0		2
CEEEOCB_USRHDLR_ROUTINE_O	4		2
CEEEOCB_USRHDLR_ROUTINE_S	0		1
CEEEOCB_USRHDLR_ROUTINE_STRING	2		2
CEEEOCB_USRHDLR_SUB_OPTIONS	1A8		2
CEEEOCB_USRHDLR_SUB_OPTS	0		1
CEEEOCB_USRHDLR_SUB_OPTS_V	0		2
CEEEOCB_USRHDLR_SUPERHDLR	0		2
CEEEOCB_USRHDLR_SUPERHDLR_LENGTH	0		2
CEEEOCB_USRHDLR_SUPERHDLR_O	8		2
CEEEOCB_USRHDLR_SUPERHDLR_S	0		1
CEEEOCB_USRHDLR_SUPERHDLR_STRING	2		2
CEEEOCB_USRHDLR_WHERE_SET	1A6		2
CEEEOCB_VCTRSAVE	12C		2
CEEEOCB_VCTRSAVE_BIT_FLAG	12C		2
CEEEOCB_VCTRSAVE_NOOVERRIDE	12C	40	2
CEEEOCB_VCTRSAVE_ON	12C	80	2
CEEEOCB_VCTRSAVE_ON_V	12C	1	2
CEEEOCB_VCTRSAVE_SUB_OPTIONS	130		2
CEEEOCB_VCTRSAVE_WHERE_SET	12E		2
CEEEOCB_VERSION	1C4		2
CEEEOCB_VERSION_BIT_FLAG	1C4		2
CEEEOCB_VERSION_NAME	0		2
CEEEOCB_VERSION_NAME_LENGTH	0		2
CEEEOCB_VERSION_NAME_O	4		2
CEEEOCB_VERSION_NAME_S	0		1
CEEEOCB_VERSION_NAME_STRING	2		2
CEEEOCB_VERSION_NOOVERRIDE	1C4	40	2
CEEEOCB_VERSION_ON	1C4	80	2
CEEEOCB_VERSION_ON_V	1C4	1	2
CEEEOCB_VERSION_RELEASE	8		2
CEEEOCB_VERSION_SUB_OPTIONS	1C8		2
CEEEOCB_VERSION_SUB_OPTS	0		1
CEEEOCB_VERSION_SUB_OPTS_V	0		2
CEEEOCB_VERSION_WHERE_SET	1C6		2

Figure 197. Options control block (OCB) field descriptions (cross references 15)

CEEOCB Macro

NAME	HEX OFFSET	HEX VALUE	LEVEL
=====	=====	=====	=====
CEEOCB_XPLINK	1F4		2
CEEOCB_XPLINK_BIT_FLAG	1F4		2
CEEOCB_XPLINK_NOOVERRIDE	1F4	40	2
CEEOCB_XPLINK_ON	1F4	80	2
CEEOCB_XPLINK_ON_V	1F4	1	2
CEEOCB_XPLINK_SUB_OPTIONS	1F8		2
CEEOCB_XPLINK_WHERE_SET	1F6		2
CEEOCB_XUFLOW	13C		2
CEEOCB_XUFLOW_BIT_FLAG	13C		2
CEEOCB_XUFLOW_LEVEL	4		2
CEEOCB_XUFLOW_NOOVERRIDE	13C	40	2
CEEOCB_XUFLOW_ON	13C	80	2
CEEOCB_XUFLOW_ON_V	13C	1	2
CEEOCB_XUFLOW_SUB_OPTIONS	140		2
CEEOCB_XUFLOW_SUB_OPTS	0		1
CEEOCB_XUFLOW_SUB_OPTS_V	0		2
CEEOCB_XUFLOW_WHERE_SET	13E		2

Figure 198. Options control block (OCB) field descriptions (cross references 16)

Table 91. Options control block (OCB) constants

Len	Type	Value	Name	Description
4	DECIMAL	2792	OPTIONS_CONTROL_BLOCK_LENGTH	
1	HEX	00	CEEOCB_FORMAT_31	
1	HEX	01	CEEOCB_FORMAT_64	
0	BIT	0	KEEP	
0	BIT	1	FREE	
0	BIT	0	ANYWHERE	
0	BIT	1	BELOW	
4	DECIMAL	1	RET_EXIT	
4	DECIMAL	2	ABD_EXIT	
2	DECIMAL	1	NONE_CONDITION	test/notest
2	DECIMAL	2	ERROR_CONDITION	
2	DECIMAL	4	ALL_CONDITION	
2	DECIMAL	1	DUMP_CONDITION	termthdact
2	DECIMAL	2	TRACE_CONDITION	
2	DECIMAL	4	MSG_CONDITION	
2	DECIMAL	8	QUIET_CONDITION	
2	DECIMAL	16	UADUMP_CONDITION	
2	DECIMAL	32	UAONLY_CONDITION	
2	DECIMAL	64	UAIMM_CONDITION	
2	DECIMAL	128	UATRACE_CONDITION	
2	DECIMAL	64	CICSDDS_CONDITION	
2	DECIMAL	128	CESE_CONDITION	
2	DECIMAL	1	OLD_NAMELIST	namelist
2	DECIMAL	2	F90_NAMELIST	
2	DECIMAL	1	VAR_RECPAD	recpad
2	DECIMAL	2	ON_RECPAD	
2	DECIMAL	2	ALL_RECPAD	
2	DECIMAL	4	OFF_RECPAD	
2	DECIMAL	4	NONE_RECPAD	
2	DECIMAL	1	AUTO_XUFLOW	xuflow

Table 91. Options control block (OCB) constants (continued)

Len	Type	Value	Name	Description
2	DECIMAL	2	ON_XUFLOW	
2	DECIMAL	4	OFF_XUFLOW	
2	DECIMAL	128	DYN_DYNAMIC	
2	DECIMAL	64	DYN_NODYNAMIC	
2	DECIMAL	32	DYN_FORCE	
2	DECIMAL	16	DYN_BOTH	
2	DECIMAL	128	DYN_TDUMP	
2	DECIMAL	64	DYN_NOTDUMP	
7	CHARACTER	DYNAMIC	DYN_DYNAMIC_C	
9	CHARACTER	NODYNAMIC	DYN_NODYNAMIC_C	
5	CHARACTER	FORCE	DYN_FORCE_C	
4	CHARACTER	BOTH	DYN_BOTH_C	
5	CHARACTER	TDUMP	DYN_TDUMP_C	
7	CHARACTER	NOTDUMP	DYN_NOTDUMP_C	
2	DECIMAL	50	DEFAULT_SETTING	
2	DECIMAL	100	IBM_SUPPLIED_DEFAULTS	
2	DECIMAL	200	PROGRAMMER_DEFAULTS	
2	DECIMAL	300	ASSEMBLER_USER_EXIT	
2	DECIMAL	400	PROGRAM_INVOCATION	
2	DECIMAL	500	REGION_DEFAULTS	
2	DECIMAL	600	STGTUNE_USER_EXIT	
2	DECIMAL	700	OVER_RIDE	
2	DECIMAL	800	IG_NORED	
2	DECIMAL	900	MAP_PED	
2	DECIMAL	1000	CICS_CLER_TRANS	
2	DECIMAL	1100	CICS_AUTO_TUNE	
2	DECIMAL	23386	MAX_WHERE_SET	
4	DECIMAL	1	ALLOW_ONLY_31BIT_RTO	
4	DECIMAL	2	ALLOW_ONLY_64BIT_RTO	
4	DECIMAL	3	ALLOW_ALL_RTO	
4	DECIMAL	0	ERRCOUNT_DEFAULT_64	
4	DECIMAL	2	ABTERMENC_DEFAULT_64	
4	DECIMAL	10	DEPTHCONDLMT_DEFAULT_64	
4	DECIMAL	15	MSGQ_DEFAULT_64	
2	DECIMAL	1	PAGE_4K	
2	DECIMAL	2	PAGE_1M	

Supplementary options control block

The following three tables show the format of the SOCB, which is the supplementary options control block:

- Table 90 on page 811 shows the type field definitions.
- Table 92 on page 858 shows the SOCB field descriptions.
- Table 93 on page 859 shows the SOCB constants.
- Table 94 on page 859 shows the SOCB cross reference information.

CEESOCB Macro

Table 92. Supplementary options control block (SOCB) field descriptions

Offsets		Type	Len	Name (Dim) (*= Reserved)
Dec	Hex			
0	(0)	STRUCTURE	44	CEESOCB
0	(0)	SIGNED	2	CEESOCB_VERSION_RELEASE
2	(2)	SIGNED	2	CEESOCB_LENGTH
4	(4)	CHARACTER	8	CEESOCB_EXECOPS
4	(4)	BITSTRING	1	CEESOCB_EXECOPS_BIT_FLAG
		1...		CEESOCB_EXECOPS_ON
		.1..		CEESOCB_EXECOPS_NOOVERRIDE
		..11 111.		*
	1		CEESOCB_EXECOPS_ON_V
5	(5)	BITSTRING	1	*
6	(6)	SIGNED	2	CEESOCB_EXECOPS_WHERE_SET
8	(8)	PTR INTOAREA	4	CEESOCB_EXECOPS_SUB_OPTIONS
12	(C)	CHARACTER	8	CEESOCB_REDIR
12	(C)	BITSTRING	1	CEESOCB_REDIR_BIT_FLAG
		1...		CEESOCB_REDIR_ON
		.1..		CEESOCB_REDIR_NOOVERRIDE
		..11 111.		*
	1		CEESOCB_REDIR_ON_V
13	(D)	BITSTRING	1	*
14	(E)	SIGNED	2	CEESOCB_REDIR_WHERE_SET
16	(10)	PTR INTOAREA	4	CEESOCB_REDIR_SUB_OPTIONS
20	(14)	CHARACTER	8	CEESOCB_ARGPARSE
20	(14)	BITSTRING	1	CEESOCB_ARGPARSE_BIT_FLAG
		1...		CEESOCB_ARGPARSE_ON
		.1..		CEESOCB_ARGPARSE_NOOVERRIDE
		..11 111.		*
	1		CEESOCB_ARGPARSE_ON_V
21	(15)	BITSTRING	1	*
22	(16)	SIGNED	2	CEESOCB_ARGPARSE_WHERE_SET
24	(18)	PTR INTOAREA	4	CEESOCB_ARGPARSE_SUB_OPTIONS
28	(1C)	CHARACTER	8	CEESOCB_ENV
28	(1C)	BITSTRING	1	CEESOCB_ENV_BIT_FLAG
		1...		CEESOCB_ENV_ON
		.1..		CEESOCB_ENV_NOOVERRIDE
		..11 111.		*
	1		CEESOCB_ENV_ON_V
29	(1D)	BITSTRING	1	*
30	(1E)	SIGNED	2	CEESOCB_ENV_WHERE_SET
32	(20)	PTR INTOAREA	4	CEESOCB_ENV_SUB_OPTIONS
36	(24)	CHARACTER	8	CEESOCB_PLIST
36	(24)	BITSTRING	1	CEESOCB_PLIST_BIT_FLAG
		1...		CEESOCB_PLIST_ON
		.1..		CEESOCB_PLIST_NOOVERRIDE
		..11 111.		*

Table 92. Supplementary options control block (SOCB) field descriptions (continued)

Offsets		Type	Len	Name (Dim) (*= Reserved)
Dec	Hex			
	1		CEESOCB_PLIST_ON_V
37	(25)	BITSTRING	1	*
38	(26)	SIGNED	2	CEESOCB_PLIST_WHERE_SET
40	(28)	PTR INTOAREA	4	CEESOCB_PLIST_SUB_OPTIONS
End of fixed portion				
0	(0)	STRUCTURE	8	CEESOCB_ENV_SUB_OPTS
0	(0)	BITSTRING	4	CEESOCB_ENV_SUB_OPTS_V
		1...		CEESOCB_ENV_OP_V
0	(0)	BITSTRING	3	*
4	(4)	SIGNED	4	CEESOCB_ENV_OP
0	(0)	STRUCTURE	8	CEESOCB_PLIST_SUB_OPTS
0	(0)	BITSTRING	4	CEESOCB_PLIST_SUB_OPTS_V
		1...		CEESOCB_PLIST_FORMAT_V
0	(0)	BITSTRING	3	*
4	(4)	SIGNED	4	CEESOCB_PLIST_FORMAT

Table 93. Supplementary options control block (SOCB) constants

Len	Type	Value	Name	Description
4	DECIMAL	60	SOCB_LENGTH	
4	DECIMAL	1	CEESOCB_PLIST_CMS	
4	DECIMAL	2	CEESOCB_PLIST_HOST	
4	DECIMAL	3	CEESOCB_PLIST_MVS	
4	DECIMAL	4	CEESOCB_PLIST_TSO	
4	DECIMAL	5	CEESOCB_PLIST_CICS	
4	DECIMAL	6	CEESOCB_PLIST_IMS	
4	DECIMAL	7	CEESOCB_PLIST_OS	
4	DECIMAL	1	CEESOCB_ENV_CMS	
4	DECIMAL	2	CEESOCB_ENV_MVS	
4	DECIMAL	3	CEESOCB_ENV_IMS	

Table 94. Supplementary options control block (SOCB) cross reference

Name	Hex Offset	Hex Value
CEESOCB	0	
CEESOCB_ARGPARSE	14	
CEESOCB_ARGPARSE_BIT_FLAG	14	
CEESOCB_ARGPARSE_NOOVERRIDE	14	40
CEESOCB_ARGPARSE_ON	14	80
CEESOCB_ARGPARSE_ON_V	14	01
CEESOCB_ARGPARSE_SUB_OPTIONS	18	
CEESOCB_ARGPARSE_WHERE_SET	16	
CEESOCB_ENV	1C	
CEESOCB_ENV_BIT_FLAG	1C	
CEESOCB_ENV_NOOVERRIDE	1C	40
CEESOCB_ENV_ON	1C	80

CEESOCB Macro

Table 94. Supplementary options control block (SOCB) cross reference (continued)

Name	Hex Offset	Hex Value
CEESOCB_ENV_ON_V	1C	01
CEESOCB_ENV_OP	4	
CEESOCB_ENV_OP_V	0	80
CEESOCB_ENV_SUB_OPTIONS	20	
CEESOCB_ENV_SUB_OPTS	0	
CEESOCB_ENV_SUB_OPTS_V	0	
CEESOCB_ENV_WHERE_SET	1E	
CEESOCB_EXECOPS	4	
CEESOCB_EXECOPS_BIT_FLAG	4	
CEESOCB_EXECOPS_NOOVERRIDE	4	40
CEESOCB_EXECOPS_ON	4	80
CEESOCB_EXECOPS_ON_V	4	01
CEESOCB_EXECOPS_SUB_OPTIONS	8	
CEESOCB_EXECOPS_WHERE_SET	6	
CEESOCB_LENGTH	2	
CEESOCB_PLIST	24	
CEESOCB_PLIST_BIT_FLAG	24	
CEESOCB_PLIST_FORMAT	4	
CEESOCB_PLIST_FORMAT_V	0	80
CEESOCB_PLIST_NOOVERRIDE	24	40
CEESOCB_PLIST_ON	24	80
CEESOCB_PLIST_ON_V	24	01
CEESOCB_PLIST_SUB_OPTIONS	28	
CEESOCB_PLIST_SUB_OPTS	0	
CEESOCB_PLIST_SUB_OPTS_V	0	
CEESOCB_PLIST_WHERE_SET	26	
CEESOCB_REDIR	C	
CEESOCB_REDIR_BIT_FLAG	C	
CEESOCB_REDIR_NOOVERRIDE	C	40
CEESOCB_REDIR_ON	C	80
CEESOCB_REDIR_ON_V	C	01
CEESOCB_REDIR_SUB_OPTIONS	10	
CEESOCB_REDIR_WHERE_SET	E	
CEESOCB_VERSION_RELEASE	0	

Appendix B. CALL linkage argument examples

This section provides examples of linkage of FASTLINK CALL and XPLINK CALL linkage arguments.

FASTLINK CALL linkage argument examples

The following sections provide different types of examples of FASTLINK CALL linkage arguments.

Notational shorthand used

In the argument list and diagram examples shown in the following sections, all arguments denote direct, by-value arguments. The following notation is used.

- l** denotes fixed bin (31)
- d** denotes double precision float
- dxu** denotes leftmost word of double precision (dxl-lower)
- f** denotes single precision float
- e** denotes extended precision float
- s** denotes fixed bin(15)
- c** denotes fixed bin(7) (or signed char in "C"), which is passed right justified in a word
- s2-14-f2** denotes a structure with leaf elements fixed bin(15), fixed bin(31), fixed bin(31), and single precision within structures half words are aligned on half word boundaries and full word binary is aligned on word boundaries and space is skipped if necessary because of alignment requirements of previous structure members. This is not meant to imply any commonality in alignment or structure mapping rules across languages in Language Environment but is just used for purposes of illustration
- ...** indicates where value would be placed if it must be represented in storage
- ** indicates sign extension for signed values and zero for unsigned ones
- ///** indicates structure padding (undefined contents)

Argument list examples

All examples shown are for non-extended-mode enabled routines.

Argument Examples

Example 1A: call Suba(11,d,12)

WILL BE PASSED IN:	STORAGE MAPPING OF ARG AREA ON THE STACK		
R1	0	11...	P1
FP0	4	du...	P2
	8	d1...	P3
STACK	12	12	P4

Note that the compiler does not initialize the related argument slots in the argument area of the stack, although they are still allocated in case the callee needs them. Only one word is used in the corresponding argument slot in the argument area.

Example 1B: call Suba(d,11,12)

WILL BE PASSED IN:	STORAGE MAPPING OF ARG AREA ON THE STACK		
FP0	0	du...	P1
	4	d1...	P2
GPR3	8	11...	P3
STACK	12	12	P4

Example 1C: call Suba(&d,11,&12,&13)

WILL BE PASSED IN:	STORAGE MAPPING OF ARG AREA ON THE STACK		
GPR1	0	address of d	P1
GPR2	4	11	P2
GPR3	8	address of 12	P3
STACK	12	address of 13	P4

The next examples illustrate the passing of argument areas by reference parameters.

Example 2: call Suba(e,1)

WILL BE PASSED IN:	STORAGE MAPPING OF ARG AREA ON THE STACK		
FP0/FP2	0	euu...	P1
	4	eu1...	P2
	8	e1u...	P3
	12	e11...	P4
STACK	16	1	P5

Example 3: call Suba(d1,d2,1)

WILL BE PASSED IN:	STORAGE MAPPING OF ARG AREA ON THE STACK		
FP0	0	d1u...	P1
	4	d11...	P2
STACK	8	d2	P3
	12		P4
STACK	16	1	P5

Example 4: call Suba(11,12,d1,d2,13)

WILL BE PASSED IN:	STORAGE MAPPING OF ARG AREA ON THE STACK		
R1	0	11...	P1
R2	4	12...	P2
FP0	8	d1u...	P3
	12	d11...	P4
STACK	16	d2	P5
	20		P6
STACK	24	13	P7

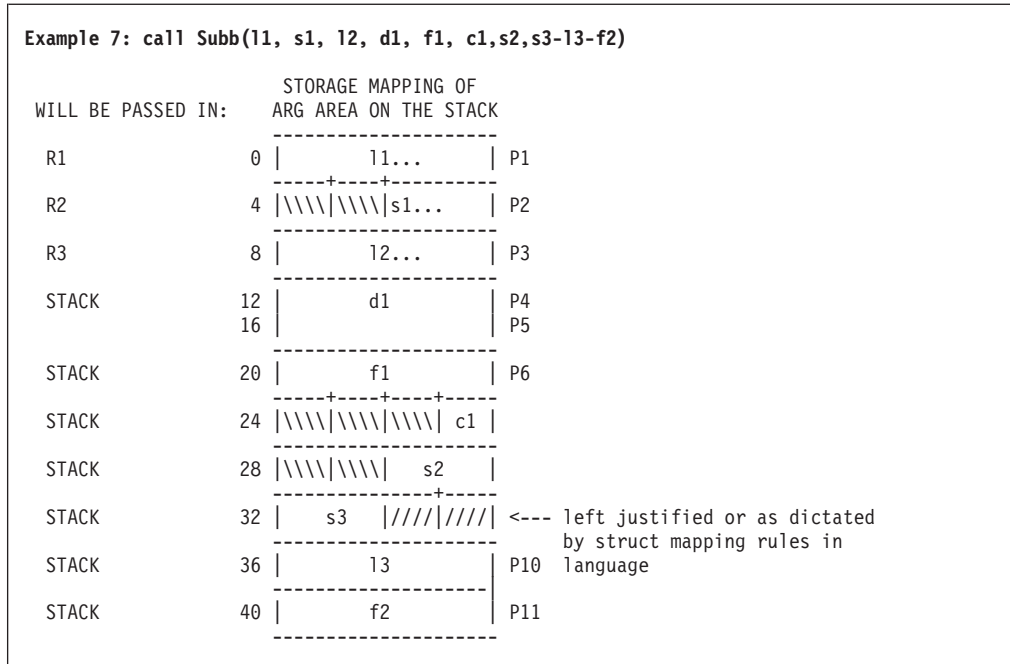
Argument Examples

Example 5: call Suba(s,11,e,12)

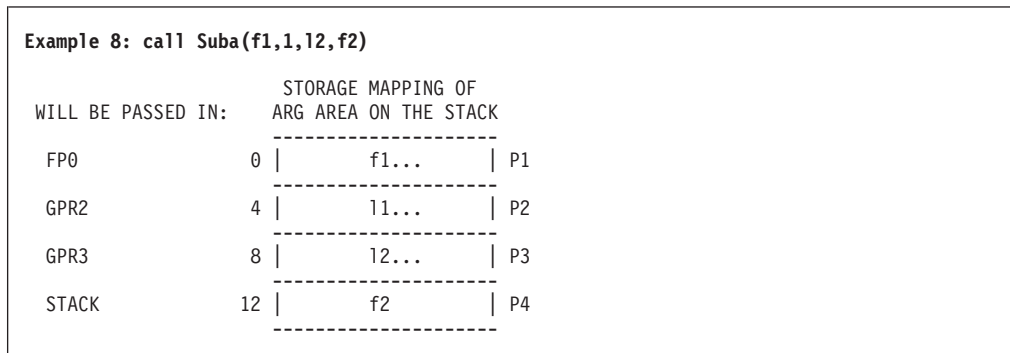
WILL BE PASSED IN:	STORAGE MAPPING OF ARG AREA ON THE STACK	
R1	0	\\\ \\\ s... P1
R2	4	11... P2
FP0/FP2	8	euu... P3
	12	eu1... P4
	16	e1u... P5
	20	e11... P6
STACK	24	12 P7

Example 6: call Subc (s1-11-d1,12,f2,d2)

WILL BE PASSED IN:	STORAGE MAPPING OF ARG AREA ON THE STACK		
R1	0	s1 \\\ \\\ <--- left justified or as dictated by struc mapping rules in language	
R2	4	11... P2	
R3	8	d1u... P3	
STACK	12	d11 P4	
STACK	16	12 P5	
STACK	20	f2 P6	
STACK	24	d2 P8	
	38	P9	



Note that you can not pass single precision floating point to a language like C++, which, upon call, promotes all single precision value arguments to double precision, and get it to work reliably. The first floating argument might happen to work because it is in a register.



Note that C++ on 390 always performs a promote of short floating values to long floating point. Thus, in some cases, C++ on 390 will not actually work as described above.

Function results

The handling of result values is very symmetric to the way parameters are passed into the function. Values are loaded into the same registers. The only difference is that a structure/string return value that does not fit in the first three GPRs is not returned in the argument area; rather, it is returned in an area passed by the caller as a hidden (first) parameter. (The same alignment rules are used as for arguments.)

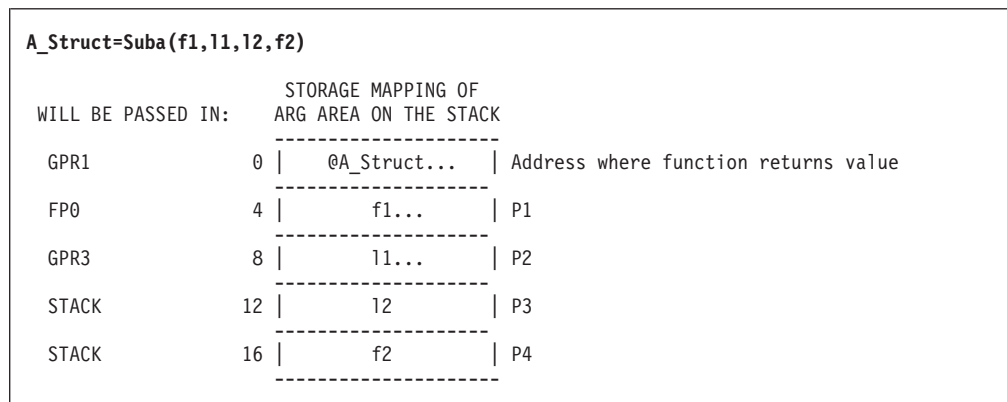
Result value	Register type
boolean (less or equal to 32 bits)	GPR1
integer byte, halfword, fullword	GPR1 (sign extended appropriately)

Argument Examples

Result value	Register type
floating point short, long	FPR0
floating point extended	FPR0 and FPR2
complex short, long	FPR0 and FPR2
complex extended	FPR0 through FPR6
character (byte), halfword (Kanji)	GPR1
pointer	GPR1
first 3 words of structures/strings	GPR1 through GPR3

Otherwise, the result will be returned in allocated storage whose address is passed as the first (hidden) argument. The caller must provide the required storage and pass its address as if it were the first argument. In FASTLINK this address, is always passed in GPR1, and one less GPR is available to pass user arguments. If the size of the return value is less than or equal to 3 words then the caller does not pass a hidden parameter for the return value. Structure return values longer than three words are passed partly in storage and partly in the GPRs following the same rules as for structure value arguments. Note that C++ does not return arrays but only a pointer to an array.

The following figure illustrates the argument list layout when a function is invoked that returns a structure whose length is larger than 3 words.



FASTLINK passes more return values in registers than does C linkage. In these cases, the simulated Code epilog may have to relocate values from registers to storage.

XPLINK CALL linkage argument examples

Restriction: “Parameter Adjust” is not used for AMODE 64 applications only.

The following example shows “by reference” parameters. In this example, “Parameter Adjust” is always zero and arguments are never passed in floating point registers. The value of the high-order bit on the last, or any, reference parameter is not defined here; this is left to the implementation, possibly specified by language constructs such as #pragma in C.

Argument Examples

Prototype:	f0(int&,	float&,	double&,	struct { /*... */ }&,	int&)	
Offset in argument list		+0	+4	+8	+12	+16	
Stored in argument list		No	No	No	Yes	Yes	
Passed in Registers		GPR1	GPR2	GPR3			
Parameter Adjust		000000/000000/000000/000000					

The remaining examples show “by value” semantics in parameter lists. “Parameter Adjust” is zero except where shown.

Prototype:	f1(int,	int,	int,	int,	int)	
Offset in argument list		+0	+4	+8	+12	+16	
Stored in argument list		No	No	No	Yes	Yes	
Passed in Registers		GPR1	GPR2	GPR3			

Prototype:	f2(char,	short,	int,	long long)		
Offset in argument list		+0	+4	+8	+12		
Stored in argument list		No	No	No	Yes		
Passed in Registers		GPR1	GPR2	GPR3			

Prototype:	f3(long long,	int,	int)			
Offset in argument list		+0	+8	+12			
Stored in argument list		No	No	Yes			
Passed in Registers		GPR1/ GPR2	GPR3				

Prototype:	f4(struct {int,	int },	int,	int)		
Offset in argument list		+0	+4	+8	+12		
Stored in argument list		No	No	No	Yes		
Passed in Registers		GPR1	GPR2	GPR3			

Prototype:	f5(struct {float,	double },	int,	int)		
Offset in argument list		+0	+8	+16	+20		
Stored in argument list		No	No	Yes	Yes		
Passed in Registers		GPR1	GPR3				

Prototype:	f6(struct {double,	float },	int,	int)		
Offset in argument list		+0	+8	+16	+20		
Stored in argument list		No	No	Yes	Yes		
Passed in Registers		GPR1/2	GPR3				

Argument Examples

Prototype:	f7(double,	long double,	double)			
Offset in argument list		+0	+8	+24			
Stored in argument list		No	No	Yes			
Passed in Registers		FPR0	FPR4/6				
Parameter Adjust		100000/000000/100000/100000					

Prototype:	f8(int,	long double,	int,	double,	int,	double)
Offset in argument list		+0	+4	+20	+24	+32	+36
Stored in argument list		No	No	Yes	No	Yes	No
Passed in Registers		GPR1	FPR0/2		FPR4		FPR6
Parameter Adjust		100001/100000/100001/100001					

Prototype:	f9(double,	double,	double,	long double)		
Offset in argument list		+0	+8	+16	+24		
Stored in argument list		No	No	No	Yes		
Passed in Registers		FPR0	FPR2	FPR4			
Parameter Adjust		100000/100000/100000/000000					

Prototype:	f10(double,	double,	double)			
Offset in argument list		+0	+8	+16			
Stored in argument list		No	No	No			
Passed in Registers		FPR0	FPR2	FPR4			
Parameter Adjust		100000/100000/100000/000000					

Prototype:	f11(double,	double,	double,	struct { double,	double})	
Offset in argument list		+0	+8	+16	+24	+32	
Stored in argument list		No	No	No	No	Yes	
Passed in Registers		FPR0	FPR2	FPR4	FPR6		
Parameter Adjust		100000/100000/100000/100000					

Prototype:	f12(int,	double,	...)			
Actual Parameters				int	double		
Offset in argument list		+0	+4	+12	+16		
Stored in argument list		No	No	Yes	Yes		
Passed in Registers		GPR1	FPR0				
Parameter Adjust		100001/000000/000000/000000					

Prototype:	f13(double,	...)			
Actual Parameters			double			
Offset in argument list	+0	+8	+12			
Stored in argument list	No	Yes	Yes			
Passed in Registers	FPR0	GPR3				
Parameter Adjust	100000/000000/000000/000000					

The following two figures show how a C/C++ structure containing two doubles is used to mimic the native COMPLEX(16) type in PLI (shown here passed by value).

Prototype:	f14(double	struct { double, double})			
Offset in argument list	+0	+8	+16			
Stored in argument list	No	No	No			
Passed in Registers	FPR0	FPR2	FPR4			
Parameter Adjust	100000/100000/100000/000000					

Prototype:	DCL F15 ENTRY(FLOAT (16)	COMPLEX (16)			
Offset in argument list	+0	+8	+24			
Stored in argument list	No	No	No			
Passed in Registers	FPR0	FPR2	FPR4			
Parameter Adjust	100000/100000/100000/000000					

The following two figures show how a C/C++ structure containing two long doubles is used to mimic the native COMPLEX(33) type in PLI.

Prototype:	f16(double,	struct { long double, long double})			
Offset in argument list	+0	+8	+24			
Stored in argument list	No	No	Yes			
Passed in Registers	FPR0	FPR4/6				
Parameter Adjust	100000/000000/100000/100000					

Prototype:	DCL F17 ENTRY(FLOAT (16)	COMPLEX (33)			
Offset in argument list	+0	+8	+24			
Stored in argument list	No	No	Yes			
Passed in Registers	FPR0	FPR4/6				
Parameter Adjust	100000/000000/100000/100000					

The following figures show how unprototyped calls match the conventions expected by both vararg and non-vararg functions.

Argument Examples

Prototype:	(none)				
Actual Parameters	int	int	double		
Offset in argument list	+0	+4	+8	+12	
Stored in argument list	No	No	Yes	Yes	
Passed in Registers	GPR1	GPR2	GPR3		
			FPR0		
Parameter Adjust	(none)				

Prototype:	f18(int,	...)		
Actual Parameters		int	double		
Offset in argument list	+0	+4	+8	+12	
Stored in argument list	No	No	Yes	Yes	
Passed in Registers	GPR1	GPR2	GPR3		

Prototype:	f19(int,	int,	double)		
Offset in argument list	+0	+4	+8			
Stored in argument list	No	No	No			
Passed in Registers	GPR1	GPR2	FPR0			
Parameter Adjust	100010/000000/000000/000000					

Prototype:	(none)				
Actual Parameters	int	int	double		float (IEEE)
Offset in argument list	+0	+4	+8	+12	+16
Stored in argument list	No	No	Yes	Yes	Yes
Passed in Registers	GPR1	GPR2	GPR3		FPR2
			FPR0		
Parameter Adjust	(none)				

Prototype:	f20(int,	...)		
Actual Parameters		int	double		float (IEEE)
Offset in argument list	+0	+4	+8	+12	+16
Stored in argument list	No	No	Yes	Yes	Yes
Passed in Registers	GPR1	GPR2	GPR3		

Prototype:	f21(int,	int,	double,	float (IEEE))	
Offset in argument list	+0	+4	+8	+16		
Stored in argument list	No	No	No	No		
Passed in Registers	GPR1	GPR2	FPR0	FPR2		
Parameter Adjust	100010/010000/000000/000000					

Argument Examples

Prototype:	(none)					
Actual Parameters	int	float (IEEE)	double		long double	
Offset in argument list	+0	+4	+8	+12	+16	
Stored in argument list	No	No	Yes	Yes	Yes	
Passed in Registers	GPR1	GPR2 FPR0	GPR3	FPR0	FPR4 FPR6	
Parameter Adjust	(none)					

Prototype:	f22(int,	...)			
Actual Parameters			float (IEEE)	double		long double
Offset in argument list	+0	+4	+8	+12	+16	
Stored in argument list	No	No	Yes	Yes	Yes	
Passed in Registers	GPR1	GPR2	GPR3			

Prototype:	f23(int,	float (IEEE),	double,	long double)		
Offset in argument list	+0	+4	+8	+16			
Stored in argument list	No	No	No	No			
Passed in Registers	GPR1	FPR0	FPR2	FPR4 FPR6			
Parameter Adjust	010001/100000/100000/100000						

Prototype:	(none)					
Actual Parameters	int	float (Hex)	int	long double		
Offset in argument list	+0	+4	+12	+16		
Stored in argument list	No	No	Yes	Yes		
Passed in Registers	GPR1	GPR2/3 FPR0		FPR4 FPR6		
Parameter Adjust	(none)					

Prototype:	f24(int,	float (Hex),	int,	long double)		
Offset in argument list	+0	+4	+12	+16			
Stored in argument list	No	No	Yes	No			
Passed in Registers	GPR1	FPR0		FPR4 FPR6			
Parameter Adjust	100001/000000/100001/100000						

Prototype:	(none)					
Actual Parameters	int	float (IEEE)	int	long double		

Argument Examples

Prototype:	(none)					
Offset in argument list	+0	+4	+8	+12		
Stored in argument list	No	No	No	Yes		
Passed in Registers	GPR1	GPR2 FPR0	GPR3	FPR4 FPR6		
Parameter Adjust	(none)					

Prototype:	f25(int,	float (IEEE),	...)		
Actual Parameters				int	long double	
Offset in argument list	+0	+4	+8	+12		
Stored in argument list	No	No	No	Yes		
Passed in Registers	GPR1	FPR0	GPR3			

Prototype:	f26(int,	float (IEEE),	int,	long double)	
Offset in argument list	+0	+4	+8	+12		
Stored in argument list	No	No	No	No		
Passed in Registers	GPR1	FPR0	GPR3	FPR4 FPR6		
Parameter Adjust	010001/000000/100001/100000					

Prototype:	f27(int,	float (Hex),	...)		
Actual Parameters				int	long double	
Offset in argument list	+0	+4	+12	+16		
Stored in argument list	No	No	Yes	Yes		
Passed in Registers	GPR1	FPR0				

Appendix C. Accessibility

Accessible publications for this product are offered through the z/OS Information Center. If you experience difficulty with the accessibility of any z/OS information, send an email to mhvrcfs@us.ibm.com or write to the following address:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size.

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the z/OS Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax

elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Note:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
 2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
 3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.
- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

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

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted

for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (<http://www.ibm.com/software/support/systemsz/lifecycle/>)
- For information about currently-supported IBM hardware, contact your IBM representative.

Permission Notice

This book includes information about certain callable service stub and linkage-assist (stub) routines contained in specific data sets that are intended to be bound or link-edited with code and run on z/OS systems. In connection with your authorized use of z/OS, you may bind or link-edit these stubs into your modules and distribute your modules with the included stubs for the purposes of developing, using, marketing and distributing programs conforming to the documented programming interfaces for z/OS, provided that each stub is included in its entirety, including any IBM copyright statements. These stubs have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply the reliability, serviceability, or function of these stub programs. The stubs referred to in this book are contained in one or more of the following data sets:

- CEE.SAFHFORT
- CEE.SCEEBIND
- CEE.SCEEBND2
- CEE.SCEECPP
- CEE.SCEELKED
- CEE.SCEELKEX
- CEE.SCEE OBJ
- CEE.SCEESPC
- CEE.SIBMAM24
- CEE.SIBMCALL
- CEE.SIBMCAL2
- CEE.SIBMMATH
- CEE.SIBMTASK

Programming interface information

This document describes intended Programming Interfaces that allow the customer to write programs to obtain the services of Language Environment in z/OS.

It is to be expected that programs written using this technical information, because of their dependencies on the detailed design and implementation of Language

Environment, might need to be changed in order to run with new Language Environment product releases or versions, or as a result of maintenance.

Trademarks

IBM, the IBM logo, and `ibm.com` are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>.

Adobe, Acrobat, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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

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

Index

Special characters

`__ae_autoconvert_state()` 249
`__ae_thread_setmode()` 246
`__ae_thread_swapmode()` 247
`__alcxpf()` 217, 714
`__bldxfd()` 326
`__CEE_DEBUG_FILENAME31` 352
`__CEE_DEBUG_FILENAME64` 353, 739
`__chkbfp()` 627
`__cttbl()` 244
`__dsa_prev()` 269, 725
`__ep_find ()` 34
`__ep_find()` 728
`__far_jump()` 272, 730
`__fnwsa()` 332
`__fp_btoh()` 628
`__fp_cast()` 629
`__fp_htob()` 630
`__fp_level()` 631
`__fp_read_rnd()` 631
`__fp_setmode()` 632
`__fp_swap_rnd()` 634
`__fp_swapmode()` 633
`__fpc_rd()` 635
`__fpc_rs()` 636
`__fpc_rw()` 637
`__fpc_sm()` 638
`__fpc_wr()` 639
`__isASCII()` 248
`__isBFP()` 639
`__le_condition_token_build()` 724
`__le_debug_set_resume_mch()`
for AMODE 64 applications 735
`__le_msg_add_insert()` 724
`__le_msg_get_and_write()` 723
`__le_msg_get()` 724
`__le_msg_write()` 723
`__set_stack_softlimit()` 274
`__setHookEvents` 339
for AMODE 64 applications 736
`__stack_info()` 221
`__static_reinit()` 334
`__vhm_event()` 216, 713
`_BPX_SHAREAS` 601
`_CEE_ENVFILE` 215
`_CEE_HEAP_MANAGER` 215, 713
`_EDC_IEEEV1_COMPATIBILITY`
`_ENV` 627
`_to_xx()` 640

Numerics

32-bit PPA3 layout, COBOL V5 29
32-bit PPA4 layout, C/C++ DWARF 29
32-bit PPA4 layout, COBOL V5 30
64-bit PPA4 layout, C/C++ DWARF 660

A

abend shunt routine 278
abend summary 179, 708
abends 286, 734
access register
conventions 5
accessibility 873
contact IBM 873
features 873
accessing original fdlibm functions 627
add new members to enclave CWI 161
ADDRESS 811
AMODE 4, 647
AMODE 64 applications
storage management 711
anchor 40, 662, 666
anchor considerations 477
anchor lookup, CEEARLU 477
anchor support 475, 751
API
`__vhm_event()` 216, 713
argument examples 861
argument/parameter list architecture 91, 677
access modes 91, 677
argument passing semantics 91, 677
ASCII
character mode
determine 248
set 246
swap 247
ASCII/EBCDIC 244
assistive technologies 873
atterm event 501
automatic destructor event 529

B

base locator table 31
BDI (block debug information) 11, 650
binary floating-point (see IEEE floating-point) 625
BITSTRING 811
block debug information 11, 650
bootstrap routine, member 168

C

C/C++ special purpose interfaces for IEEE floating-point 625
C/C++-specific vendor interfaces 244
`__ae_autoconvert_state()` 249
`__ae_thread_setmode()` 246
`__ae_thread_swapmode()` 247
`__cttbl()` 244
`__isASCII()` 248
CAA (common anchor area)
constants 46
cross reference 46
field descriptions 40

CAA (common anchor area) *(continued)*
format of 40, 666

CALL
conventions 91
for AMODE 64 applications 677
FASTLINK linkage 102
linkages 91, 677
standard linkage 92
XPLINK linkage 114, 678
CALL linkage argument examples 861
CALL or function invocation
argument/parameter list 91, 677
access modes 91, 677
argument passing semantics 91, 677

callable services

CEE3DMP 401
CEE3ERP 251
CEE3RSUM 252
CEE3SGLN 255
CEE3SGLT 256
CEE3SMO 243
CEEBDMP 419
CEEHDMP 418
CEELDMP 414
CEEMRCM 266
CEETDLI 473
CEEVDMP 415
CEEYDSAF 267
change the MSGFILE ddname 239
conversion services 386
dump services 399
dynamic storage services 201
interactive debug services 339, 735
math services 369
PLIST manipulation CWI 171
program mask 5
register content
at entry 4
at exit 5
user-created services 210
vendor heap manager interface 213, 712
XPLINK Compatibility Stack
Swapping Services 218
XPLINK DSA Extension Services 216
calling the preinitialization environment in SRB mode 196
cancel load module event 528
cancel/release load module CWI 155
case 230, 721
CDI (compile debug information) 11, 650
CEE3ADDM 161
CEE3CBTS 342
CEE3CDO 422
CEE3CRE 163
CEE3CSYS 167
CEE3DDBC 171
CEE3DMP 401
CEE3ERP 251

CEE3MBR	168	CEEOPCPO	552	CEERCB_ZFREEST	206
CEE3PLST	171	CEEOPCPU	553	CEERCB_ZGETST	205
CEE3RSUM	252	CEEOPCS	586	CEERELU	173
CEE3SGLN	255	CEEOPCT	587	CEESDMP	401
CEE3SGLT	256	CEEOPCW	590	CEESNAP	424
CEE3SMO	243	CEEOPDD	592	CEESTART	140
CEE3SMS	257	CEEOPDG	593	CEETBCK	404
CEE3SMS2	259	CEEOPDI	594	CEETDLI	473, 474
CEE3SRT	365	CEEOPDS	595	CEETGCAA	350
CEEARLU	477	CEEOPPE	541	CEETGTFN	330
CEEATTRM	176	CEEOPPEQ	542	CEETHLOC	324
CEEBCRM	155	CEEOPGS	548	CEETLOC	323
CEEBDMP	419	CEEOPJ	543	CEETLOR	328
CEEBENV CWI	605	CEEOPKC	549	CEETRCB	401
CEEBETBL CSECT	148	CEEOPKD	550	CEETREC	174
CEEBFBC	344	CEEOPMD	555	CEETREN	175
CEEBLLST CSECT	149	CEEOPMF	236	CEETSFB	350
CEEBSENM	156	CEEOPMI	557	CEETSFC	351
CEEBSHL	182	CEEOPML	559	CEEUOPT	179
CEEBSIOP	146	CEEOPML2	561	CEEURTB	424
CEEBSRM	157	CEEOPMT	562	CEEV#FRS	208
CEECAAGETS	93	CEEOPMU	563	CEEV#GTS	207
CEECCICS	442, 445, 446, 450, 451, 452, 456, 457, 460, 467, 468, 469, 472	CEEOPMU2	564	CEEVDMP	415
CEECHMF	239	CEEOPU	544	CEEVGTSB	206
CEECIB	78, 283, 733	CEEOPRL	565	CEEVGTUN	98
CEECLAS	235	CEEOPRL2	567	CEEVH2OS	220
CEECMIB	240	CEEOPRT	567	CEEVHRPT	209
CEECOPP	179, 708	CEEOPRU	569	CEEVROND	218
CEECTCB	471	CEEOPRU2	570	CEEVRONU	219
CEECTOK	226	CEEOPS	545	CEEVSSEG	99
CEEDLLF	335	CEEOPSS	551	CEEVUHR	210
CEEEVDBG entry point	352	CEEOPWL	571	CEEVUHFR	212
CEEEVPRF entry point	361, 746	CEEOPWL2	573	CEEVUHGT	211
CEEFMAIN	145	CEEOPWT	573	CEEVUHRP	212
CEEGETFB	230	CEEOPXD	575	CEEVXPAL	217
CEEGIN	172	CEEOPXG	576	CEEXETBL macro	148
CEEGOTO	261	CEEOPXI	578	CEEXVSEL macro	184
CEEHDHDL	265	CEEOPXS	580	CEEYCVHE	387
CEEHDMP	418	CEEOSPWN	601	CEEYCVHF	389
CEEINT	139, 153, 155	CEEOXEXE	604	CEEYCVHI	394
CEEINT interface	153	CEEPB_DELETE	299	CEEYDSAF	267
CEEKTRC	428	CEEPB_LOAD	300	CEEYEPAF	33
CEEKRGPM	345	CEEPB_ZFREEST	204	CEEYPPAF	36
CEEKSNP	423	CEEPB_ZGETST	203	CEEZDEL	290
CEELDMP	414	CEEPDEL	296	CEEZDEL2	292
CEEMAIN	145	CEEPDEL2	297	CEEZLOD	290
CEEMCH	81	CEEPDEL3	303	CEEZLDR	291
CEEMFNDM	241	CEEPFDE	318	CELAAUTH	759
CEEMRCM	266	CEEPFWSA	331	CELQBST	699
CEEODMF	235	CEEPGFD	159	format	702
CEEODEXEC	597	CEEPIPI	193	CELQETBL CSECT	703
CEEODEXIT	604	CEEPLDE	316	CELQFMAN	702
CEEOFORK	599	CEEPLOD	293	CELQLLST CSECT	704
CEEOKILL()	546	CEEPLOD2	294	CELQMAIN	
CEEOPAD	534	CEEPLODT	301	format	701
CEEOPAGD	535	CEEPLVE	310	CELQSTRT	699
CEEOPAGS	535	CEEPLVI	309	CELQUOPT	708
CEEOPAGW	536	CEEPLVT	311	CELQVDBG entry point	353, 739
CEEOPAI	533	CEEPPOS	312	central control blocks, member	
CEEOPASD	537	CEEPQDF	319	language	480
CEEOPASS	538	CEEPQDV	320	CHARACTER	811
CEEOPASW	539	CEEPQLD	298	CIB (condition information block)	78
CEEOPC	540	CEEPQFD	160	CICS	
CEEOPCB	582	CEEQDMF	237	background information	431
CEEOPCD	584	CEEQFBC	347	extended runtime language	
CEEOPCI	585	CEEQLD	348	interface	437
		CEEQUMF	238	languages supported	436

CICS (*continued*)
 running an application program under CICS 433
 terminology 431
 thread initialization 449
 thread termination 451
 cleanup routine processing 261
 clearenv() 605
 close ddname 235
 COBOL
 call/cancel routine 611
 extract side file routine 617
 file and runtime information query routine 613
 library load/delete exit 609
 COBOL-specific vendor interfaces 609
 command
 syntax diagrams xix
 command line equivalent event 521
 common anchor area 40, 475, 666, 751
 common naming conventions 4, 648
 compilation units 3
 compile debug information 11, 650
 Component Broker Connector (CBC) 342
 condition handling 251, 725
 math service library 369
 condition code definitions 383
 condition information block 78, 283, 733
 condition manager 251
 abends 286, 734
 for AMODE 64 applications 725
 interface
 to error processing 280
 interfaces
 to shunt routine 277, 732
 condition representation
 for AMODE 64 applications 717
 model 225, 717
 objectives 225, 717
 condition token 718
 Condition_ID
 Case 1 228, 719
 case 2 229, 720
 control 230, 664, 721
 control block dump service 419
 conventions
 access register 5
 CALL linkage 91, 677
 FASTLINK linkage 102
 floating-point register 5
 HLL condition handling 285
 Language Environment service 3, 647
 program mask 5
 standard linkage 92
 XPLINK linkage 114, 678
 conversion services 386, 387, 389, 394
 terminology 386
 COUNTRY
 description of 233
 country code
 country code change event 499
 country code change event 499
 created enclaves
 CEE3CRE 163
 creating a new enclave 163
 creating nested enclaves CWI 167
 CWI (compiler-writer interface)
 __alcxp() 217
 for AMODE 64 applications 714
 __bldxfd() 326
 __dsa_prev() 269, 725
 __ep_find() 728
 __far_jump() 272, 730
 __fnwsa() 332
 __set_stack_softlimit() 274
 __setHookEvents 339
 __stack_info 221
 __static_reinit() 334
 CEE3ADDM 161
 CEE3CBTS 342
 CEE3CDO 422
 CEE3CRE 163
 CEE3CSYS 167
 CEE3DDBC 171
 CEE3ERP 251
 CEE3MBR 168
 CEE3PLST 171
 CEE3RSUM 252
 CEE3SGLN 255
 CEE3SGLT 256
 CEE3SMS 257
 CEE3SMS2 259
 CEE3SRT 365
 CEEARLU 477
 CEEATTRM 176
 CEEBCRLM 155
 CEEBDMP 419
 CEEBENV 605
 CEEBFBC 344
 CEEBSENL 156
 CEEBSHL 182
 CEEBSIOP 146
 CEEBSRCM 157
 CEECHMF 239
 CEECLOS 235
 CEECMIB 240
 CEECOPP 179, 708
 CEECTCB 471
 CEEGETFB 230
 CEEGIN 172
 CEEGOTO 261
 CEEHDHDL 265
 CEEHDMP 418
 CEEINT 139
 CEEKCTRC 428
 CEEKRGPM 345
 CEEKSNP 423
 CEELDMP 414
 CEEMFNDM 241
 CEEMRCM 266
 CEEODMF 235
 CEEOEXEC 597
 CEEOEXIT 604
 CEEOFORK 599
 CEEOKILL() 546
 CEEOPAD 534
 CEEOPAGD 535
 CEEOPAGS 535
 CEEOPAGW 536
 CEEOPAI 533
 CEEOPASD 537
 CEEOPASS 538
 CWI (compiler-writer interface) (*continued*)
 CEEOPASW 539
 CEEOPC 540
 CEEOPCB 582
 CEEOPCD 584
 CEEOPCI 585
 CEEOPCPO 552
 CEEOPCPU 553
 CEEOPCS 586
 CEEOPCT 587
 CEEOPCW 590
 CEEOPDD 592
 CEEOPDG 593
 CEEOPDI 594
 CEEOPDS 595
 CEEOPE 541
 CEEOPEQ 542
 CEEOPGS 548
 CEEOPJ 543
 CEEOPKC 549
 CEEOPKD 550
 CEEOPMD 555
 CEEOPMF 236
 CEEOPMI 557
 CEEOPML 559
 CEEOPML2 561
 CEEOPMT 562
 CEEOPMU 563
 CEEOPMU2 564
 CEEOPO 544
 CEEOPRL 565
 CEEOPRL2 567
 CEEOPRT 567
 CEEOPRU 569
 CEEOPRU2 570
 CEEOPS 545
 CEEOPSS 551
 CEEOPWL 571
 CEEOPWL2 573
 CEEOPWT 573
 CEEOPXD 575
 CEEOPXG 576
 CEEOPXI 578
 CEEOPXS 580
 CEEOSPWN 601
 CEEPCB_DELETE 299
 CEEPCB_LOAD 300
 CEEPCB_ZFREEST 204
 CEEPCB_ZGETST 203
 CEEPDEL 296
 CEEPDEL2 297
 CEEPDEL3 303
 CEEPFDE 318
 CEEPFWSA 331
 CEEPGFD 159
 CEEPLDE 316
 CEEPLOD 293
 CEEPLOD2 294
 CEEPLODT 301
 CEEPLVE 310
 CEEPLVI 309
 CEEPLVT 311
 CEEPPOS 312
 CEEPQDF 319
 CEEPQDV 320
 CEEPQLD 298

CWI (compiler-writer interface)
(continued)

CEEPRFD 160
CEEQDMF 237
CEEQFBC 347
CEEQLOD 348
CEEQUMF 238
CEERCB_ZFREEST 206
CEERCB_ZGETST 205
CEESDMP 401
CEETBCK 404
CEETGCAA 350
CEETGTFN 330
CEETHLOC 324
CEETLOC 323
CEETLOR 328
CEETRCB 401
CEETREC 174
CEETREN 175
CEETSFB 350
CEETSFC 351
CEEURTB 424
CEEV#FRS 208
CEEV#GTS 207
CEEVDMP 415
CEEVGTSB 206
CEEVGUN 98
CEEVH2OS 220
CEEVHRPT 209
CEEVROND 218
CEEVRONU 219
CEEVSSEG 99
CEEVUHCR 210
CEEVUHFR 212
CEEVUHGT 211
CEEVUHRP 212
CEEVXPAL 217
CEEYCVHE 387
CEEYCVHF 389
CEEYCVHI 394
CEEYDSAF 267
CEEYEPAF 33
CEEYPPAF 34, 36, 728
CEEZDEL 290
CEEZDEL2 292
CEEZLDR 290
CEEZLDR2 291
change the MSGFILE ddname 239
close ddname 235
conversion routine 394
dump services 423
message services 424
obtain program's invocation name 172
PLIST manipulation CWI 171
process-level FREESTORE 203
process-level GETSTORE 203
region-level FREESTORE 204
region-level GETSTORE 204
set interrupt option 146
set return save area 170
snap dump services 423
user routine traceback service 424
CWI for scalar math routines 370
CWI services 290, 291, 292

D

data area
 member list 83
data type definitions 86
data types
 CEECTOK 226
 CHARn 87
 COMPLEX16 87
 COMPLEX8 87
 condition token 718
 entry 87
 ENTRY 87
 FEED_BACK 87
 FLOAT4 87
 FLOAT8 87
 HCURSOR 87
 INT2 87
 INT4 87
 label 88
 LABEL 87
 POINTER 87
 RCURSOR 87
 VSTRING 87
ddname
 CEECHMF, change MSGFILE
 CWI 239
 CEECLOS, close CWI 235
 CEEODMF, open an input ddname
 CWI 235
 CEEOPMF, open MSGFILE CWI 236
 CEEQDMF, query an input ddname
 CWI 237
 CEEQUMF, query MSGFILE
 CWI 238
 CEESNAP 424
debug event handler 352, 353, 739
debug flags, PPA4 30, 660
debug services 339, 735
Debug Tool 339, 735
Debug Tool event 502
debugger interfaces area 668
debugger interfaces area (DIA)
 format of 668
DECIMAL 811
default options event 522
degree input/output trig functions 381
destructor function processing 261
determine enclave work area lengths
 event 518
determine working storage (CICS only)
 event 519
DFSORT 365
dll initialization event 509
downward-growing stack frame
 CEEVROND 218
 run on 218
downward-growing stack segment
 CEEVROND 218
 run on 218
DSA (dynamic storage area XPLINK) 39,
 661
DSA (dynamic storage area) 37
 dummy DSA 38
 layout 38
 managing library stack 96
 example 97
 managing user stack 93, 95

DSA (dynamic storage area) (continued)

 examples 94, 95
 zeroth DSA 38
DSA Classification 491
DSA exit routine 498
DSA Ownership 488
dummy DSA block chain, setting 171
dump event handler event 492
dump services 399, 423
 CEE3DMP 401
 CEEBDMP 419
 CEEHDMP 418
 CEEKSNP 423
 CEELDMP 414
 CEESDMP 399
 CEETRCB 401
 CEEVDMP 415
 member language dump exit 414
 snap dump services 423
dynamic load libraries (DLL) 316
dynamic storage area 37
dynamic storage area XPLINK 39, 661
dynamic storage services 201

E

EBCDIC
 character mode
 determine 248
 set 246
 swap 247
EDB (enclave data block)
 constants 64
 cross reference 64
 field descriptions 61
 format of 61
edcwccwi.h 712
enclave data block 61
enclave data block (EDB)
 format of 671
enclave initialization 137, 697
enclave initialization event 503
enclave level delete CWI services
 CEEPDEL 296
 CEEPDEL2 297
enclave level load CWI services
 CEEPLOD 293
enclave level LOAD CWI services
 CEEPLOD2 294
enclave services, program manager 292
enclave termination 138, 698
enclave termination event 506
enclaves
 CEE3CRE 163
 created enclaves 163
 creating a new enclave 163
Entry Point and Compile Unit
 Identification 488
environment variables
 __CEE_DEBUG_FILENAME31 352
 __CEE_DEBUG_FILENAME64 353,
 739
 _BPX_SHAREAS 601
 _CEE_ENVFILE 215, 713
 _CEE_HEAP_MANAGER 215, 216,
 713

environment variables (*continued*)
 _EDC_IEEEV1_COMPATIBILITY
 _ENV 627
 LIBPATH 296, 352
 POSIX 605
 error handling 367, 789
 error processing 280
 error recovery, user-provided
 CEE3ERP 251
 CEE3RSUM 252
 event codes
 atterm 178, 501
 automatic destructor 529
 cancel load module 528
 condition handling
 condition enablement 275, 481
 condition handling for given stack
 frame 275, 481
 stack frame zero handling 275,
 481
 Debug Tool 502
 dump events 414
 enclave initialization 177
 enclave termination 177
 get function pointer 527
 GOTO target DSA 275, 497
 interrupt received 525
 main-opts event 487
 member needs options
 processing 520
 normal target DSA 524
 POSIX event code 511
 process initialization 177
 process termination 177
 release function pointer 527
 release load module 528
 runtime options 177, 486, 521, 522,
 523
 static constructor 515
 static destructor 522
 utility event 487
 event handler routine, member
 language 481
 event handler, with member list 84
 events
 atterm event 501
 automatic destructor 529
 cancel load module 528
 command line equivalent event 521
 country code change event 499
 Debug Tool event 502
 default options event 522
 determine enclave work area lengths
 event 518
 determine working storage (CICS
 only) event 519
 dll initialization event 509
 DSA exit routines event 498
 dump event handler event 492
 enclave initialization event 503
 enclave termination event 506
 get function pointer 527
 handle condition according to
 language defaults event 485
 handle condition represented by the
 CIB event 481

events (*continued*)
 identify module entry point
 event 517
 interrupt received event 525
 main routine invocation event 146,
 500
 main-opts event 487
 member needs options processing
 event 520
 member program mask 530
 national language change event 499
 new condition event 496
 new load module event 495
 normal target DSA 524
 perform enablement for this stack
 frame event 483
 perform GOTO validation (CICS only)
 event 520
 POSIX events event 511
 preallocated storage event 523
 process initialization event 502
 process termination event 508
 query/build feedback code
 event 507
 region initialization event 516
 region termination event 517
 release function pointer 527
 release load module 528
 resume from a condition handler
 event 497
 runtime options event 486
 stack frame zero processing
 event 510
 static destructor event 522
 static object constructor event 514
 utility event 487
 examples
 CALL linkage argument 861
 exception handling, handler errors 283,
 733
 exit
 dump 414
 load/delete 609
 exit routine 276, 498
 explicit DLL reference
 CEEPFDE 318
 CEEPLDE 316
 CEEPQDF 319
 CEEPQDV 320
 extended runtime language
 interface 437
 external names, Language
 Environment 4, 647
 externals table 148, 703
F
 Facility_ID 229
 FACILITY_ID 720
 FDCB (function descriptor control block)
 format of 325
 feedback code 230, 721
 feedback code routine
 build 344
 query 347
 translate 351

fetch
 bootstrap behavior 146
 fetch anchor routine 475
 Find previous DSA
 CEEYDSAF 267
 flags, PPA4 debug 30, 660
 flags, PPA4 program 31, 661
 floating-point
 register conventions 5
 format of
 CAA 40
 CEESTART 145
 EDB 61
 LIBVEC descriptor 307
 Non-XPLINK CEESTART 140
 OCB 811
 PCB 69
 RCB 74
 SOCB 858
 XPLINK CEESTART 140
 function descriptor control block 325
 function invocation of old code 330
 function prototypes 85

G

get function pointer CWI 159
 get function pointer event 527
 get next CAA pointer CWI 350
 getenv() 605

H

handle condition according to language
 defaults event 485
 handle condition represented by the CIB
 event 481
 Header 244
 header file
 edcwccwi.h 712
 heap management 201, 711
 heap services 201
 user-created services 210
 vendor heap manager interface 213
 for AMODE 64 applications 712
 XPLINK Compatibility Stack
 Swapping Services 218
 XPLINK DSA Extension Services 216
 hex storage dump service 418
 HLL condition handling
 conventions 285
 HLL condition handling
 information 286
 HLL condition handling routine 275,
 483, 485

I

IARV64 DUMPPRIORITY 715
 IARV64 USERTKN 715
 IBMPXSF 621
 identify module entry point event 517
 IEEE decimal floating-point 626
 IEEE floating-point functions 627
 __chkbf() 627
 __fp_bto() 628

- IEEE floating-point functions (*continued*)
 - __fp_cast() 629
 - __fp_htob() 630
 - __fp_level() 631
 - __fp_read_rnd() 631
 - __fp_setmode() 632
 - __fp_swap_rnd() 634
 - __fp_swapmode() 633
 - __fpc_rd() 635
 - __fpc_rs() 636
 - __fpc_rw() 637
 - __fpc_sm() 638
 - __fpc_wr() 639
 - __isBFP() 639
 - __to_xx() 640
 - IEEE floating-point, C/C++ special purpose interfaces 625
 - IGZCXCC 611
 - IGZCXSF 617
 - IGZXAPI 613
 - ILBOLLDX 609
 - ILC (interlanguage communication)
 - conventions 3
 - epilog code 31
 - load module 3
 - member identifier
 - Language Environment-enabled language member identifiers 19
 - PPA1 10, 650
 - PPA2 11, 650
 - program flags, Language Environment 15, 16, 17
 - prolog code
 - control level 11, 650
 - prolog information blocks 10, 650
 - routine layout 6, 648
 - service interface 3
 - ILC within SORT exits 367
 - implicit DLL reference
 - CEETGTFN 330
 - CEETLOC 323
 - CEETLOR 328
 - IMS 472
 - IMS (information management system)
 - implementation 474
 - IMS-to-Language Environment interface 472
 - Language Environment-to-IMS interface 473
 - information management system 472
 - init/term overview 139, 698
 - initialization 137, 697
 - initialization member event codes 177
 - initialization member interfaces
 - enclave initialization components
 - CEEINT 139
 - CELLIST 139
 - CEESGnnn 139
 - CEESTART 139
 - CELQBST 699
 - CELQSTRT 699
 - initialization parameter list 150, 705
 - initialization/termination application interfaces 179
 - initializing the preinitialization environment 196
 - instance specific information (ISI) 230, 721
 - interactive debug services 339, 735
 - Language Environmentactions for 359, 745
 - Language Environmentdata areas 359, 745
 - interactive test services 339, 735
 - interface
 - to shunt routine 277, 732
 - interface validation exit 183
 - CEEXVSEL macro 184
 - high-level selection criteria 184
 - language-specific
 - arguments passed to 187
 - example of 191
 - reference list 189
 - structure of 184
 - interface, Language Environment service 3, 533, 647
 - interlanguage communication 3
 - with IMS interface 472
 - internal names 4, 647
 - interrupt
 - CEEBSIOP 146
 - set interrupt option CWI 146
 - interrupt received event 525
 - invalid resume request
 - CEE3SGLN 255
 - ISI (instance specific information) 230, 721
 - IVE (interface validation exit) 183
- ## K
- keyboard
 - navigation 873
 - PF keys 873
 - shortcut keys 873
- ## L
- LAA (library anchor area)
 - format of 662
 - Language Environment
 - abend summary 179, 708
 - abends 286, 734
 - initialization 155
 - storage management 98
 - termination 176, 708
 - CICS interface 436
 - common anchor area (CAA) 40, 475, 666, 751
 - conversion services 386, 387, 389, 394
 - terminology 386
 - ddname
 - CEECHMF 239
 - CEECLOS 235
 - CEEQDMF 237
 - CEEQUMF 238
 - CEESNAP 424
 - debugger interfaces area (DIA) 668
 - enclave data block (EDB) 61, 671
 - function control block (FCB) 325
 - interface validation exit 183
 - library anchor area (LAA) 662
 - Language Environment (*continued*)
 - library control area (LCA) 664
 - non-XPLINK stack storage model 92
 - Preinitialized Environments for Authorized Programs 753
 - process control block (PCB) 69, 672
 - region control block (PCB) 673
 - region control block (RCB) 73
 - shell, exit from/re-entry to,
 - CEEBSHL 182
 - variable control block (VDCB) 329
 - XPLINK stack storage model 679
 - XPLINKstack storage model 115
 - Language Environment-enabled language member identifiers 19
 - language list 149, 704
 - language support 233
 - AMODE 64 applications 723
 - language-specific interface validation exit 187
 - LCA (library control area)
 - format of 664
 - Leaf Routines 110
 - LIBPACK
 - CSECT definition 306
 - description 305
 - LIBPATH 296, 352
 - library anchor area 662
 - library control area 664
 - library stack
 - allocate/return storage 96
 - managing 96
 - library subroutine access 303
 - LIBVEC 304
 - CWI to low-level services 309
 - descriptor format 307
 - direct access instructions 305
 - indirect access instructions 305
 - initialization 308
 - library subroutine access table 304
 - LIBVEC
 - CWI to low-level services 309
 - description 304
 - descriptor format 307
 - direct access instructions 305
 - indirect access instructions 305
 - initialization 308
 - termination 311
 - verify load/delete 310
 - linkage
 - CEEVH2OS 220
 - OS linkage 220
 - linkage, CALL 91, 677
 - linkage, FASTLINK 102
 - linkage, standard 92
 - linkage, XPLINK 114, 678
 - load module
 - *name support
 - CICS 289
 - MVS 289
 - z/OS UNIX 290
 - definition of 3
 - Locates a field in the PPA1 optional area CEEYPPAF 36
 - Locates XPLINK/non-XPLINK entry point
 - CEEYEPAF 33

locator table, base 31
look up anchor routine 477
looking up RCBs 173

M

machine state block 81
macro
 CEEOCB, options control block 811
 CEESOCB, supplementary options control block 857
 CEEXVSEL, high-level selection criteria 184
main routine invocation event 146, 500
main routine parameter list processing 171
main-opts event 487
mask, program 5
math library
 callable services 369
 condition code definitions 383
 conversion services 387, 389
 conversion routine 394
 CWI for scalar math routines 370
 math services 372
 message text 383
math services
 degree input/output trig functions 381
 scalar bit manipulation routines 381
 scalar math services 373
 value of inserts 384
MCH (machine state block) 81
member bootstrap routine 168
member identifier
 Language Environment-enabled language member identifiers 19
member interfaces
 for initialization 177
 for termination 177
member language
 about 479
 central control blocks 480
 event handler 481
 restricted use of OS services 479
member list 83
member needs options processing event 520
member program mask event 530
member termination interfaces 174, 707
memory object dump priority 715
memory object user tokens 715
message
 single-line service 414
message handling 233
 country code change event 499
 find message insert 242
 introduction 234
 national language change event 499
message handling services 240
message services 243, 424
 CEECMIB 240
 CEEMFNDM 241
 for AMODE 64 applications 723
 introduction 723
message text, math library 383

Mixed Mode Support for Enhanced ASCII
 C-RTL
 ASCII/EBCDIC 244
 Header information 245
 Overview 244
 Usage example 245
Msg_No 229, 720
MSGFILE
 related CWIs 235

N

NAB (next available byte) 98
national language
 national language change event 499
national language change event 499
navigation
 keyboard 873
nested enclaves, creating 167
new condition event 496
new load module event 495
next available byte 98
NLS (national language support)
 AMODE 64 applications 723
 country code change event 499
 message handling services 233
 national language change event 499
Non-XPLINK CEESTART 140
normal resume event 524
normal resume in DSA event 524
normal target DSA 524
Notices 877

O

obtaining program's invocation name 172
OCB (options control block)
 constants 856
 field descriptions 812
OCB (options control block) and SOCB (supplementary options control block)
 type field definitions 811
options control block 811
OS/VS COBOL
 call/cancel routine 611
 extract side file routine 617
 file and runtime information query routine 613
 library load/delete exit 609
out-of-storage condition 711

P

parameter list
 main routine parameter list processing 171
 with IMS interface 472
parameter/parameter list architecture 91, 677
 access modes 91, 677
 argument passing semantics 91, 677
PCB
 constants 70
 cross reference 71
 field descriptions 69

PCB (*continued*)
 format of 69
perform enablement for this stack frame event 483
perform GOTO validation (CICS only) event 520
performance analysis services
 profile tool event handler 361, 746
PL/I
 extract side file routine 621
PL/I-specific vendor interfaces 621
POSIX event code 511
POSIX function
 clearenv() 605
 getenv() 605
 setenv() 605
PPA1 flag 1
 program flags, Language Environment 652
PPA1 Flag 1
 program flags, Language Environment 21
PPA1 flag 2
 program flags, Language Environment 654
PPA1 Flag 2
 program flags, Language Environment 22
PPA1 flag 3
 program flags, Language Environment 654
PPA1 Flag 3
 program flags, Language Environment 22
PPA1 flag 4
 program flags, Language Environment 655
PPA1 Flag 4
 program flags, Language Environment 23
PPA1 Word
 program flags, Language Environment 25, 658
PPA3 layout, COBOL V5 32-bit 29
PPA4 debug flags 30, 660
PPA4 layout, C/C++ DWARF 32-bit 29
PPA4 layout, C/C++ DWARF 64-bit 660
PPA4 layout, COBOL V5 32-bit 30
PPA4 program flags 31, 661
preallocated storage event 523
preinitialization interfaces 193
preinitialization service routines 196
Preinitialized Environments for Authorized Programs 753
 CELAAUTH 759
 creating an environment 753
 system-managed 754
 user-managed 754
 tasks 755
procedure prototypes 85
process control block 69
process control block (PCB)
 format of 672
process initialization 137, 697
process initialization event 502
process services, program manager 290
process termination 139, 698

- process termination event 508
- profiler tools 361, 746
- program flags, PPA4 31, 661
- program invocation name, obtaining 172
- program manager 289
 - CWI services 290, 291, 292
 - enclave level services 292
 - load module name support
 - CICS 289
 - MVS 289
 - z/OS UNIX 290
 - process level services 290
 - region-level services 291
 - responsibilities 289
- program mask
 - callable services 5
 - conventions 5
- PTR INTOAREA 811

Q

- query/build feedback code event 507
- quick options 520

R

- RCB (region control block)
 - constants 75
 - cross reference 75
 - field descriptions 74
 - format of 73
- RCB look up 173
- recursive Language Environment
 - code 3, 647
- reentrancy, Language Environment 3, 647
- reference list, interface validation
 - exit 189
- region control block 73
- region control block (RCB)
 - format of 673
- region initialization event 516
- region services, program manager 291
- region termination event 517
- register event handler 176
- release function pointer CWI 160
- release function pointer event 527
- release load module event 528
- REQUEST=MNGDCALL
 - parameters 779
 - syntax 778
- REQUEST=MNGDINIT
 - parameters 773
 - syntax 772
- REQUEST=MNGDTERM
 - parameters 787
 - syntax 786
- REQUEST=MNGDUPDT
 - parameters 783
 - syntax 783
- REQUEST=USERCALL
 - parameters 766
 - syntax 765
- REQUEST=USERINIT
 - parameters 762
 - syntax 761

- REQUEST=USERTERM
 - parameters 770
 - syntax 770
- resource-owning TCB 755
- resume from a condition handler
 - event 497
- return save area, setting 170
- returns the address of the entry point of
 - the function owning the dsa_p DSA.
 - __ep_find () 34, 728
- RMODE 4, 647
- routine
 - extract side file 621
- routines, Language Environment
 - AMODE 4, 647
 - linking 3, 647
 - RMODE 4, 647
- RUNOPTS 179, 708
- runtime language interface,
 - extended 437
- runtime options
 - compiler service, CEECOPP 179, 708
 - TRACE 426
- runtime options event 486

S

- scalar bit manipulation routines 381
- scalar math services 373
- segment
 - CEEVROND 218
 - CEEVRONU 219
 - CEEVSSEG 99
 - downward-growing stack
 - segment 218
 - stack segment bounds 99
 - upward-growing stack segment 219
- sending comments to IBM xxiii
- services, Language Environment
 - conventions 3, 647
 - interfaces 3, 647
- set anchor routine 476
- set dummy DSA block chain 171
- set enclave name CWI 156
- set enclave return code modifier
 - CWI 157
- set return save area CWI 170
- setenv() 605
- severity 228, 230, 719, 721
- shortcut keys 873
- shunt routine 277, 732
- shunt routine interface
 - abend 278
 - establishing 277, 732
- signal
 - CEE3SGLN 255
 - CEE3SGLT 256
- SIGNED 811
- snap dump services 423
- SOCB (supplementary options control
 - block)
 - constants 859
 - cross reference 859
 - field descriptions 858
- SORT interface 365
- specify execute hook events for
 - target 339, 736

- stack frame
 - CEEVSSEG 99
 - segment bounds 99
- stack frame zero processing event 510
- stack management 94, 95
- stack pointer, saving 715
- stack segment
 - CEEVSSEG 99
 - segment bounds 99
- stack segment ranges 221
- Stack Swapping Services
 - XPLINK Compatibility 218
- Statement Identification 490
- static destructor event 522
- static object constructor event 514
- stderr 723
- storage management 201, 711
 - abend
 - reason codes 98
 - dynamic storage services 201
 - heap 201, 711
 - library stack 96
 - NAB locator 98
 - stack 94, 95
 - user stack 93, 95
 - vendor heap manager 216, 713
 - XPLINK DSA extension 217
 - for AMODE 64 applications 714
- STRUCTURE 811
- stub
 - Call by Name 324
 - CEETHLOC 324
- stub for trigger load
 - CEETHLOC 324
 - on XPLINK call by name 324
 - XPLINK 324
- Summary of changes xxv
- supplementary options control
 - block 857
- suppress printing of messages 243
- synchronous condition handling
 - HLL condition handling
 - conventions 285
 - HLL condition handling
 - information 286
- syntax diagrams
 - how to read xix
- system-managed Preinitialized
 - Environments for Authorized Programs
 - environment 754

T

- T_I_S condition 176
- table, base locator 31
- terminate without raising T_I_S 175
- termination 137, 138, 697, 698
 - ABEND 4094 176, 708
 - CEE3SGLT 256
- termination member event codes 177
- termination member interfaces 174, 707
- test services 339, 735
- The preinitialization environment and
 - SRB mode 196
- thread initialization 137, 697
- thread level load/delete CWI services
 - CEEPDELT 303

thread level load/delete CWI services
(*continued*)
 CEEPLODT 301
TRACE runtime option 426
Traceback utility 404
tracing services
 add a trace table entry,
 CEEKCTRC 428
 global tracing 427
 member-specific tracing 427
trigger load on call 323
trigger load on reference 328
type field definitions
 ADDRESS 811
 BITSTRING 811
 CHARACTER 811
 DECIMAL 811
 PTR INTOAREA 811
 SIGNED 811
 STRUCTURE 811

U

upward-growing stack frame
 CEEVRONU 219
 run on 219
upward-growing stack segment
 CEEVRONU 219
 run on 219
user interface
 ISPF 873
 TSO/E 873
user routine traceback service 424
user stack
 allocate/extend/return storage 93
 managing 94
 obtain a DSA 95
user-created services 210
user-managed Preinitialized
 Environments for Authorized Programs
 environment 754
user-provided error recovery
 CEE3ERP 251
 CEE3RSUM 252
utility event 487

V

value of inserts, math services 384
variable descriptor control block 329
variable dump service 415
VDCB (variable descriptor control block)
 format of 329
vendor heap manager 216, 713
vendor heap manager interface 213
 for AMODE 64 applications 712
vendor interfaces, C/C++-specific 244
vendor interfaces, COBOL-specific 609
vendor interfaces, PL/I-specific 621

W

worker task 755
writable static area (WSA) 331

X

XPLINK CEESTART 140
XPLINK Compatibility Stack Swapping
 Services 218
XPLINK DSA extension 217, 714
XPLINK DSA Extension Services 216
XPLINK to OS linkage on
 upward-growing stack frame
 CEEVH2OS 220
 XPLINK to 220
XPLINK to OS linkage on
 upward-growing stack segment
 CEEVH2OS 220
 XPLINK to 220

Z

z/OS UNIX support 533



Printed in USA