

z/OS
3.1

*XL C/C++
Programming Guide*



Note

Before using this information and the product it supports, read the information in [“Notices” on page 767](#).

This edition applies to IBM® z/OS® 3.1 (5655-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2024-06-24

© **Copyright International Business Machines Corporation 1996, 2024.**

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

Contents

Figures.....	xix
Tables.....	xxix
About this document.....	xxxv
Related z/OS XL C/C++ information.....	xliv
Where to find more information.....	xliv
z/OS Basic Skills in IBM Documentation.....	xliv
How to provide feedback to IBM.....	xlv
Summary of changes.....	xlvii
Summary of changes for z/OS 3.1.....	xlvii
Part 1. Introduction.....	1
Chapter 1. About IBM z/OS XL C/C++.....	3
Part 2. Input and Output.....	5
Chapter 2. Introduction to C and C++ input and output.....	7
Types of C and C++ input and output.....	7
Text streams.....	7
Binary streams.....	8
Record I/O.....	8
Blocked I/O.....	8
Chapter 3. Understanding models of C I/O.....	11
The record model for C I/O.....	11
Record formats.....	11
The byte stream model for C I/O.....	19
Mapping the C types of I/O to the byte stream model.....	20
Chapter 4. Using the Standard C++ Library I/O Stream Classes.....	21
Advantages to using the C++ I/O stream classes.....	21
Predefined streams for C++.....	21
How C++ I/O streams relate to C I/O streams.....	22
Mixing the Standard C++ I/O stream classes, USL I/O stream class library, and C I/O library functions.....	22
Specifying file attributes.....	22
Chapter 5. Buffering of C streams.....	23
Chapter 6. Using ASA text files.....	25
Example of writing to an ASA file.....	25
ASA file control.....	26
Chapter 7. z/OS XL C support for the double-byte character set.....	29
Opening files.....	30
Reading streams and files.....	30

Writing streams and files.....	31
Writing text streams.....	32
Writing binary streams.....	33
Flushing buffers.....	33
Flushing text streams.....	33
Flushing binary streams.....	34
ungetwc() considerations.....	34
Setting positions within files.....	34
Repositioning within text streams.....	34
Repositioning within binary streams.....	35
ungetwc() considerations.....	35
Closing files.....	35
Manipulating wide character array functions.....	35
Chapter 8. Performing OS I/O operations.....	37
Opening files.....	37
Using fopen() or freopen().....	37
Generation data group I/O.....	40
Regular and extended partitioned data sets.....	42
Partitioned and sequential concatenated data sets.....	43
In-stream data sets.....	45
SYSOUT data sets.....	46
Tapes.....	46
Multivolume data sets.....	47
Striped data sets.....	47
Large format sequential data sets.....	47
Other devices.....	48
Access method selection.....	48
fopen() and freopen() parameters.....	49
Buffering.....	52
Multiple buffering.....	53
DCB (Data Control Block) attributes.....	53
Reading from files.....	55
Reading from binary files.....	56
Reading from text files.....	56
Reading from record I/O files.....	57
Reading from blocked I/O files.....	57
Writing to files.....	57
Writing to binary files.....	58
Writing to text files.....	58
Writing to record I/O files.....	61
Writing to blocked I/O files.....	61
Flushing buffers.....	62
Updating existing records.....	62
Reading updated records.....	62
Writing new records.....	63
ungetc() considerations.....	63
Repositioning within files.....	64
ungetc() considerations.....	65
How long fgetpos() and ftell() values last.....	65
Using fseek() and ftell() in binary files.....	66
Using fseek() and ftell() in text files (ASA and Non-ASA).....	67
Using fseek() and ftell() in record files.....	67
Using fseek() and ftell() in blocked files.....	67
Porting old C code that uses fseek() or ftell().....	68
Closing files.....	68
Renaming and removing files.....	70
fldata() behavior.....	71

Chapter 9. Performing z/OS UNIX file system I/O operations.....	75
Creating files.....	75
Regular files.....	75
Link and symbolic link files.....	75
Directory files.....	76
Character special files.....	76
FIFO files.....	76
Opening files.....	76
Using fopen() or freopen().....	77
Reading from z/OS UNIX file system files.....	80
Opening and reading from z/OS UNIX file system directory files.....	81
Writing to z/OS UNIX file system files.....	81
Flushing records.....	82
Setting positions within files.....	82
Closing files.....	82
Deleting files.....	83
Pipe I/O.....	83
Using unnamed pipes.....	83
Using named pipes.....	84
Character special file I/O.....	87
Low-level z/OS UNIX I/O.....	88
Example of z/OS UNIX file system I/O functions.....	88
fldata() behavior.....	92
File tagging and conversion.....	93
Access control lists (ACLs).....	94
Chapter 10. Performing VSAM I/O operations.....	95
VSAM types (data set organization).....	95
Access method services.....	96
Choosing VSAM data set types.....	96
Keys, RBAs and RRNs.....	98
Summary of VSAM I/O operations.....	99
Opening VSAM data sets.....	100
Using fopen() or freopen().....	100
Buffering.....	104
Record I/O in VSAM.....	104
RRDS record structure.....	105
Reading record I/O files.....	105
Writing to record I/O files.....	106
Updating record I/O files.....	107
Deleting records.....	108
Repositioning within record I/O files.....	108
Flushing buffers.....	110
Summary of VSAM record I/O operations.....	110
VSAM record level sharing and transactional VSAM.....	111
Error reporting.....	112
VSAM extended addressability.....	113
Text and binary I/O in VSAM.....	114
Reading from text and binary I/O files.....	114
Writing to and updating text and binary I/O files.....	114
Deleting records in text and binary I/O files.....	114
Repositioning within text and binary I/O files.....	114
Flushing buffers.....	116
Summary of VSAM text I/O operations.....	116
Summary of VSAM binary I/O operations.....	117
Closing VSAM data sets.....	118
VSAM return codes.....	119

VSAM examples.....	119
KSDS example.....	119
RRDS example.....	127
fldata() behavior.....	129
Chapter 11. Performing terminal I/O operations.....	131
Opening files.....	131
Using fopen() and freopen().....	131
Buffering.....	133
Reading from files.....	133
Reading from binary files.....	134
Reading from text files.....	135
Reading from record I/O files.....	135
Writing to files.....	135
Writing to binary files.....	136
Writing to text files.....	136
Writing to record I/O files.....	137
Flushing records.....	137
Text streams.....	137
Binary streams.....	137
Record I/O.....	138
Repositioning within files.....	138
Closing files.....	138
fldata() behavior.....	138
Chapter 12. Performing memory file and hiperspace I/O operations.....	141
Using hiperspace operations.....	141
Opening files.....	141
Using fopen() or freopen().....	141
Simulating partitioned data sets.....	145
Buffering.....	147
Reading from files.....	147
Writing to files.....	148
Flushing records.....	149
ungetc() considerations.....	149
Repositioning within files.....	149
Closing files.....	150
Performance tips.....	150
Removing memory files.....	150
fldata() behavior.....	150
Example program.....	151
Chapter 13. Performing CICS Transaction Server I/O operations.....	153
Chapter 14. Language Environment Message file operations.....	155
Opening files.....	155
Reading from files.....	155
Writing to files.....	155
Flushing buffers.....	156
Repositioning within files.....	156
Closing files.....	156
Chapter 15. CELQIPI MSGRTN file operations.....	157
Opening files.....	157
Reading from files.....	157
Writing to files.....	157
Flushing buffers.....	157
Repositioning within files.....	157

Closing files.....	157
fldata() behavior.....	157
fldata() example.....	158
Chapter 16. Debugging I/O programs.....	161
Using the __amrc structure.....	161
Using the __amrc2 structure.....	164
Using __last_op codes.....	165
Using the SIGIOERR signal.....	168
File I/O trace.....	170
Locating the file I/O trace.....	171
Part 3. Interlanguage calls with z/OS XL C/C+.....	173
Chapter 17. Using linkage specifications in C or C+.....	175
Syntax for Linkage in C or C+.....	175
Syntax for linkage in C.....	175
Syntax for linkage in C+.....	175
Kinds of linkage used by C or C+ interlanguage programs.....	176
Using Linkage Specifications in C+.....	178
Part 4. Coding: Advanced topics.....	179
Chapter 18. Building and using Dynamic Link Libraries (DLLs).....	181
Support for DLLs.....	181
DLL concepts and terms.....	182
Loading a DLL.....	182
Loading a DLL implicitly.....	183
Loading a DLL explicitly.....	183
Managing the use of DLLs when running DLL applications.....	188
Loading DLLs.....	188
Sharing DLLs.....	189
Freeing DLLs.....	190
Creating a DLL or a DLL application.....	190
Building a simple DLL.....	190
Example of building a simple C DLL.....	190
Example of building a simple C+ DLL.....	191
Compiling your code.....	192
Binding your code.....	192
Building a simple DLL application.....	193
Steps for using an implicitly loaded DLL in your simple DLL application.....	193
Creating and using DLLs.....	194
DLL restrictions.....	196
Improving performance	197
Chapter 19. Building complex DLLs.....	199
Rules for compiling source code with XPLINK.....	200
XPLINK applications.....	200
Non-XPLINK applications.....	200
Compatibility issues between DLL and non-DLL code.....	202
Pointer assignment.....	203
Function pointers.....	203
DLL function pointer call in non-DLL code.....	205
C example	206
Non-DLL function pointer call in DLL(CBA) code.....	208
Non-DLL function pointer call in DLL code.....	210
Function pointer comparison in non-DLL code.....	211

Function pointer comparison in DLL code.....	213
Using DLLs that call each other.....	215
Chapter 20. z/OS 64-bit environment.....	221
Differences between the ILP32 and LP64 environments.....	221
ILP32 and LP64 addressing capabilities.....	221
ILP32 and LP64 data models and data type sizes.....	221
Advantages and disadvantages of the LP64 environment.....	222
LP64 application performance and program size.....	222
LP64 restrictions.....	223
Migrating applications from ILP32 to LP64.....	223
When to migrate applications to LP64.....	223
Checklist for ILP32-to-LP64 pre-migration activities.....	223
Checklist for ILP32-to-LP64 post-migration activities.....	224
Using compiler diagnostics to ensure portability of code.....	224
Using the INFO option to ensure that numbers are suffixed.....	225
Using the WARN64 option to identify potential portability problems.....	225
ILP32-to-LP64 portability issues.....	226
IPA(LINK) option and exploitation of 64-bit virtual memory.....	227
Availability of suboptions.....	227
Potential changes in structure size and alignment.....	227
Data type assignment differences under ILP32 and LP64.....	232
Pointer declarations when 32-bit and 64-bit applications share header files.....	236
Potential pointer corruption.....	237
Potential loss of data in constant expressions.....	238
Data alignment problems when structures are shared.....	239
Portability issues with unsuffixed numbers.....	241
Using a LONG_MAX macro in a printf subroutine.....	242
Programming for portability between ILP32 and LP64.....	243
Using header files to provide type definitions.....	243
Using suffixes and explicit types to prevent unexpected behavior.....	243
Defining pad members to avoid data alignment problems.....	243
Using prototypes to avoid debugging problems.....	245
Using a conditional compiler directive for preprocessor macro selection.....	245
Using converters under ILP32 or LP64.....	246
Using locales under ILP32 or LP64.....	246
Chapter 21. Using threads in z/OS UNIX applications.....	247
Models and requirements.....	247
Functions.....	247
Creating a thread.....	247
Synchronization primitives.....	248
Thread-specific data.....	252
Signals.....	253
Generating a signal.....	254
Thread cancellation.....	254
Cleanup for threads.....	256
Thread stack attributes.....	257
Behaviors and restrictions in z/OS UNIX applications.....	257
Using threads with MVS files.....	257
Multithreaded I/O.....	258
Thread-scoped functions.....	258
Unsafe thread functions.....	259
Fetched functions and writable statics.....	259
MTF and z/OS UNIX threading.....	259
Thread queuing function.....	259
Thread scheduling.....	259
iconv() family of functions.....	260

Threads and recoverable resources.....	260
MEMLIMIT for 64-bit multithreaded applications.....	260
Chapter 22. Reentrancy in z/OS XL C/C++.....	261
Natural or constructed reentrancy.....	262
Limitations of constructed reentrancy for C programs.....	262
Controlling external static in C programs.....	262
Controlling writable strings.....	263
Controlling the memory area in C++.....	263
Controlling where string literals exist in C++ code.....	264
Using writable static in Assembler code.....	264
Chapter 23. IEEE Floating-Point	267
Floating-point numbers.....	267
C/C++ compiler support.....	268
Using IEEE floating-point.....	268
Chapter 24. Handling error conditions, exceptions, and signals.....	273
Handling C software exceptions under C++.....	273
Handling hardware exceptions under C++.....	274
Tracebacks under C++.....	274
AMODE 64 exception handlers.....	276
Scope and nesting of exception handlers.....	277
Handling exceptions.....	277
Signal handlers.....	278
Handling signals with POSIX(OFF) using signal() and raise().....	278
Handling signals using Language Environment callable services.....	278
Handling signals using z/OS UNIX with POSIX(ON).....	279
Asynchronous signal delivery under z/OS UNIX.....	281
C signal handling features under z/OS XL C/C++.....	282
Chapter 25. Network communications under UNIX System Services.....	293
Understanding z/OS UNIX sockets and internetworking.....	293
Basics of network communication.....	293
Transport protocols for sockets.....	294
What is a socket?.....	294
z/OS UNIX Socket families.....	296
z/OS UNIX Socket types.....	296
Guidelines for using socket types.....	296
Addressing within sockets.....	297
The conversation.....	299
The server perspective.....	299
The client perspective.....	301
A typical TCP socket session.....	301
A typical UDP socket session.....	302
Locating the server's port.....	303
Network application example.....	303
Using common INET.....	308
Compiling and binding.....	309
Using TCP/IP APIs.....	310
Restrictions for using z/OS TCP/IP API with z/OS UNIX.....	310
Using z/OS UNIX sockets.....	311
Compiling under MVS batch for Berkeley sockets.....	312
Compiling under MVS batch for X/Open sockets.....	313
Understanding the X/Open Transport Interface (XTI).....	314
Transport endpoints.....	314
Transport providers for X/Open Transport Interface.....	315
General restrictions for z/OS UNIX.....	315

Chapter 26. Interprocess communication using z/OS UNIX.....	317
Message queues.....	317
Semaphores.....	317
Shared memory.....	318
Memory mapping.....	318
TSO commands from a shell.....	318
Chapter 27. Using templates in C++ programs.....	319
Using the TEMPINC compiler option.....	319
TEMPINC example.....	320
Regenerating the template instantiation file.....	323
TEMPINC considerations for shared libraries.....	323
Using the TEMPLATEDEPTH compiler option.....	323
Using the TEMPLATEREGISTRY compiler option.....	323
Recompiling related compilation units.....	323
Switching from TEMPINC to TEMPLATEREGISTRY.....	324
Using explicit instantiation declarations (C++11 only).....	324
Examples of explicit instantiation declarations.....	325
Chapter 28. Using environment variables.....	327
Working with environment variables.....	334
Naming conventions.....	335
Environment variables specific to the z/OS XL C/C++ library.....	335
_CEE_CONDWAIT_PAUSE.....	336
_CEE_DLLLOAD_XPCOMPAT.....	337
_CEE_DMPTARG.....	337
_CEE_ENVFILE.....	338
_CEE_ENVFILE_COMMENT.....	339
_CEE_ENVFILE_CONTINUATION.....	339
_CEE_ENVFILE_S.....	339
_CEE_HEAP_MANAGER.....	340
_CEE_REALLOC_CONTROL.....	341
_CEE_RUNOPTS.....	342
_EDC_ADD_ERRNO2.....	343
_EDC_ANSI_OPEN_DEFAULT.....	344
_EDC_AUTO_MAP64.....	344
_EDC_AUTOCVT_BINARY.....	344
_EDC_BYTE_SEEK.....	345
_EDC_CLEAR_SCREEN.....	345
_EDC_COMPAT.....	345
_EDC_CONTEXT_GUARD.....	346
_EDC_C99_NAN.....	346
_EDC_DLL_DIAG.....	347
_EDC_EOVERFLOW.....	348
_EDC_ERRNO_DIAG.....	348
_EDC_FLUSH_STDOUT_PIPE.....	349
_EDC_FLUSH_STDOUT_SOCKET.....	349
_EDC_GLOBAL_STREAMS.....	350
_EDC_IEEEV1_COMPATIBILITY_ENV.....	351
_EDC_IO_ABEND.....	351
_EDC_IO_TRACE.....	352
_EDC_OPEN_CONCAT.....	354
_EDC_POPEN.....	354
_EDC_PTHREAD_YIELD.....	354
_EDC_PTHREAD_YIELD_MAX.....	355
_EDC_PUTENV_COPY.....	355
_EDC_RRDS_HIDE_KEY.....	356

_EDC_SIG_DFLT.....	356
_EDC_STOR_INCREMENT.....	356
_EDC_STOR_INCREMENT_B.....	357
_EDC_STOR_INITIAL.....	357
_EDC_STOR_INITIAL_B.....	357
_EDC_STRPTM_STD.....	358
_EDC_SUSV3.....	358
_EDC_UMASK_DFLT.....	358
_EDC_ZERO_RECLEN.....	359
Propagating environment variables.....	359
Chapter 29. Using hardware built-in functions.....	361
General instructions.....	361
PLO - Perform Locked Operation available in ARCH(5).....	383
Decimal instructions.....	389
Floating-point support instructions.....	394
Decimal floating-point built-in functions.....	395
Macros for use with decimal floating-point built-in functions.....	403
Hexadecimal floating-point instructions.....	405
Binary floating-Point instructions.....	406
Built-in functions for transaction execution.....	409
Chapter 30. Using runtime check library.....	413
Chapter 31. XL C++ 98 applications and C99.....	415
Obtaining C99 behavior with XL C.....	415
Using C99 functions in XL C++ applications.....	415
Feature test macros that control C99 interfaces in XL C++ applications.....	415
Using C99 functions in C++ applications when ambiguous definitions exist.....	416
Chapter 32. Writing applications for Single UNIX Specification, Version 3.....	417
Announcing your intentions.....	417
Testing the environment	418
What is different in SUSv3.....	419
Symbols withdrawn in SUSv3.....	419
Candidates for removal in a future version.....	419
Implementation compliance.....	419
Chapter 33. Saved compile-time options information.....	421
Saved options information layout.....	421
Part 5. Performance optimization.....	425
Chapter 34. Improving program performance.....	427
Writing code for performance.....	427
Using C++ constructs in performance-critical code.....	427
Using explicit instantiation declarations (C++11 only).....	429
ANSI aliasing rules.....	429
Using ANSI aliasing rules.....	431
Using variables.....	432
Passing function arguments.....	433
Coding expressions.....	434
Coding conversions.....	434
Arithmetical considerations.....	435
Using loops and control constructs.....	435
Choosing a data type.....	436
Using library extensions.....	437

Using #pragmas.....	438
Using rvalue references (C++11).....	439
Using shared-memory parallelism (SMP).....	442
Chapter 35. Using built-in functions to improve performance.....	443
__builtin_expect.....	444
Platform-specific functions.....	444
Examples.....	445
Chapter 36. I/O Performance considerations.....	447
Accessing MVS data sets.....	447
Accessing UNIX file system files.....	448
Using memory files.....	449
Using the C++ I/O stream libraries.....	449
Chapter 37. Improving performance with compiler options.....	451
Using the OPTIMIZE option.....	451
Optimizations performed by the compiler.....	451
Aggressive optimizations with OPTIMIZE(3).....	452
The xlc utility optimization option levels.....	453
Using the ARCHITECTURE and TUNE options.....	454
Inlining.....	455
Selectively marking code to inline.....	455
Automatically choosing functions to inline.....	456
Modifying automatic inlining choices.....	456
Overriding inlining defaults.....	457
Inlining under IPA.....	457
Using the XPLINK option.....	457
When you should not use XPLINK.....	457
Using the HOT option.....	458
Using the IPA option.....	458
Types of procedural analysis.....	459
Program-directed feedback.....	460
Compiler processing flow.....	461
Additional options that affect performance.....	465
ANSIALIAS.....	465
AGGRCOPY.....	465
ASSERT(RESTRICT).....	465
COMPRESS.....	465
COMPACT.....	465
CVFT (C++ only).....	465
EXH (C++ only).....	465
EXPORTALL.....	465
FLOAT.....	465
HGPR.....	466
IGNERRNO.....	466
LANGLVL(NOCHECKPLACEMENTNEW).....	466
LIBANSI.....	466
NOCHECKNEW.....	466
OBJECTMODEL (C++ only).....	466
PREFETCH.....	467
RESTRICT.....	467
ROCONST.....	467
ROSTRING.....	467
RTTI.....	467
SPILL.....	467
STRICT.....	467
STRICT_INDUCTION.....	467

THREADED.....	467
UNROLL.....	468
VECTOR.....	468
Chapter 38. Parallelizing your programs.....	469
Using OpenMP directives.....	469
Shared and private variables in a parallel environment.....	470
OpenMP runtime functions for parallel processing.....	471
Chapter 39. Optimizing the system and Language Environment.....	481
Improving the performance of the Language Environment.....	481
Storing libraries and modules in system memory.....	481
Optimizing memory and storage.....	481
Optimizing runtime options.....	482
Tuning the system for efficient execution.....	482
Link pack areas.....	483
Library lookasides.....	483
Virtual lookasides.....	483
Chapter 40. Balancing compilation time and application performance.....	485
General tips.....	485
Programmer tips.....	486
System programmer tips.....	487
Chapter 41. Stepping through optimized code using the dbx debugger utility.....	489
Steps for setting up a stopping point for dbx in optimized code.....	489
Steps for setting up a stopping point for dbx in optimized code.....	489
Part 6. z/OS XL C/C++ environments.....	491
Chapter 42. Using the system programming C facilities.....	493
Using functions in the system programming C environment.....	494
System programming C facility considerations and restrictions.....	494
Creating freestanding applications.....	496
Creating modules without CEESTART.....	496
Including an alternative initialization routine under z/OS.....	496
Initializing a freestanding application without Language Environment.....	497
Initializing a freestanding application using C functions.....	497
Setting up a C environment with preallocated stack and heap.....	497
Determining ISA requirements.....	498
Building freestanding applications to run under z/OS.....	498
Parts used for freestanding applications.....	500
Creating system exit routines.....	501
Building system exit routines under z/OS.....	502
An example of a system exit.....	502
Creating and using persistent C environments.....	505
Building applications that use persistent C environments.....	506
An example of persistent C environments.....	506
Developing services in the service routine environment.....	509
Using application service routine control flow.....	509
Understanding the stub perspective.....	515
Establishing a server environment.....	522
Initiating a server request.....	522
Accepting a request for service.....	522
Returning control from service.....	523
Constructing user-server stub routines.....	523
Building user-server environments.....	523

Tailoring the system programming C environment.....	524
Generating abends.....	524
Getting storage.....	524
Getting page-aligned storage.....	525
Freeing storage.....	526
Loading a module.....	526
Deleting a module.....	526
Including a runtime message file.....	527
Additional library routines.....	528
Summary of application types.....	528
 Chapter 43. Library functions for system programming C.....	531
__xhotc() — Set Up a Persistent C Environment (No Library).....	531
__xhotl() — Set Up a Persistent C Environment (With Library).....	532
__xhott() — Terminate a Persistent C Environment.....	532
__xhotu()	533
__xregs() — Get Registers on Entry.....	533
__xsacc() — Accept Request for Service.....	534
__xsrv() — Return Control from Service.....	534
__xusr() - __xusr2() — Get Address of User Word.....	535
__24malc() — Allocate Storage below 16MB Line.....	535
__4kmalc() — Allocate Page-Aligned Storage.....	535
 Chapter 44. Using runtime user exits.....	537
Using runtime user exits in z/OS Language Environment.....	537
Understanding the basics.....	537
PL/I and C/370 compatibility.....	537
User exits supported under z/OS Language Environment.....	537
Order of processing of user exits.....	538
Using installation-wide or application-specific user exits.....	539
Using the Assembler user exit.....	539
Using sample Assembler user exits.....	540
Assembler user exit interface.....	541
Parameter values in the Assembler user exit.....	546
PL/I and C/370 compatibility.....	550
High level language user exit interface.....	550
 Chapter 45. Using the z/OS XL C MultiTasking Facility.....	555
Organizing a program with MTF.....	555
Ensuring computational independence.....	556
Running a C program without MTF.....	556
Running a C program with MTF.....	557
Running a C program with one parallel function.....	557
Running a C program with two different parallel functions.....	559
z/OS XL C with multiple instances of the same parallel function.....	560
Designing and coding applications for MTF.....	562
Step 1: Identifying computationally-independent code.....	562
Step 2: Creating parallel functions.....	562
Step 3: Inserting calls to parallel functions.....	564
Changing an application to use MTF.....	565
Compiling and linking programs that use MTF.....	569
Creating the main task program load module.....	569
Creating the parallel functions load module.....	569
Specifying the linkage-editor option.....	570
Modifying runtime options.....	570
Running programs that use MTF.....	570
STEPLIB DD statement.....	570
DD statements for standard streams.....	571

Example of JCL.....	571
Debugging programs that use MTF.....	571
Avoiding undesirable results when using MTF.....	571
Part 7. Programming with Other Products.....	573
Chapter 46. Using the CICS Transaction Server (CICS TS).....	575
Developing XL C/C++ programs for the CICS environment.....	575
Preparing CICS for use with z/OS Language Environment.....	575
Designing and coding for CICS.....	576
Using the CICS command-level interface.....	576
Using input and output.....	579
Using z/OS XL C/C++ library support.....	580
Storage management.....	583
Using ILC support.....	583
Exception handling.....	584
Example of error handling in CICS.....	585
ABEND codes and error messages under z/OS XL C/C++.....	588
Coding hints and tips.....	588
Translating and compiling for reentrancy.....	588
Options for translating CICS statements.....	588
Compiling XL C/C++ programs that were preprocessed by the standalone CICS translator.....	592
Linking all object modules.....	593
Defining and running the CICS program.....	594
Program processing.....	594
Link considerations for C programs.....	594
CSD considerations.....	594
Sample JCL to install z/OS XL C/C++ application programs.....	595
Chapter 47. Using Cross System Product (CSP).....	597
Common data types.....	597
Passing control.....	597
Running CSP under MVS.....	597
Calling CSP applications from z/OS XL C.....	598
Calling z/OS XL C from CSP.....	601
Running under CICS control.....	604
Example programs.....	604
Chapter 48. Using Data Window Services (DWS).....	609
Chapter 49. Using DB2 Universal Database.....	611
Preparing an XL C/C++ application to request DB2 services.....	611
Using the XL C/C++ DB2 coprocessor.....	611
Using the DB2 C/C++ precompiler.....	612
Using DB2 services and stored procedures with XPLINK.....	612
Examples of how to use XL C/C++ programs to request DB2 services.....	612
Chapter 50. Using Graphical Data Display Manager (GDDM).....	619
Examples.....	619
Chapter 51. Using the Information Management System (IMS).....	625
Handling errors.....	625
Other considerations.....	626
Examples.....	627
Chapter 52. Using the Query Management Facility (QMF).....	633
Example programs.....	633

Part 8. Internationalization: Locales and Character Sets.....	639
Chapter 53. Introduction to locale.....	641
Internationalization in programming languages.....	641
Elements of internationalization.....	641
z/OS XL C/C++ Support for internationalization.....	642
Locales and localization.....	642
Locale-sensitive interfaces.....	642
Chapter 54. Building a locale.....	645
Limitations of enhanced ASCII.....	645
Using the charmap file.....	646
The CHARMAP section.....	651
The CHARSETID section.....	652
Locale source files.....	653
LC_CTYPE category.....	655
LC_COLLATE category.....	658
LC_MONETARY category.....	664
LC_NUMERIC category.....	667
LC_TIME category.....	668
LC_MESSAGES category.....	670
LC_TOD category.....	671
LC_SYNTAX category.....	673
Method files.....	675
Using the localedef utility.....	678
Locale naming conventions.....	679
Chapter 55. Customizing a locale.....	691
Using the customized locale.....	692
Referring explicitly to a customized locale.....	693
Referring implicitly to a customized locale.....	694
Customizing your installation.....	696
Chapter 56. Customizing a time zone.....	697
Using the TZ or _TZ environment variable to specify time zone.....	697
Relationship between TZ or _TZ and LC_TOD.....	698
Chapter 57. Definition of S370 C, SAA C, and POSIX C locales.....	699
Differences between SAA C and POSIX C locales.....	705
Chapter 58. Code set conversion utilities.....	707
The genxlt utility.....	707
The iconv utility.....	707
Code conversion functions.....	708
_ICONV_MODE environmental variable.....	708
_ICONV_TECHNIQUE environmental variable.....	709
Code set converters supplied.....	709
Universal coded character set converters.....	720
Codeset conversion using UCS-2.....	727
UCMAP source format.....	727
Chapter 59. Coded character set considerations with locale functions.....	731
Variant character detail.....	731
Alternate code points.....	733
Coding without locale support by using a hybrid coded character set.....	733
Writing code using a hybrid coded character set.....	735

Converting hybrid code.....	735
Coded character set independence in developing applications.....	735
Coded character set in source code and header files.....	736
Converting coded character sets at compile time.....	739
Writing source code in coded character set IBM-1047.....	743
Exporting source code to other sites.....	744
Converting existing work.....	745
Considerations with other products and tools.....	745
Chapter 60. Bidirectional language support.....	747
Bidirectional languages.....	747
Overview of the layout functions.....	748
Using the layout functions.....	753
Appendix A. Mapping variant characters for z/OS XL C/C++.....	759
Specifying the appropriate code page for the compiler.....	759
Testing the display of variant characters.....	759
Inserting and viewing square brackets during an ISPF edit session.....	762
Appendix B. Accessibility.....	765
Notices.....	767
Terms and conditions for product documentation.....	768
IBM Online Privacy Statement.....	769
Policy for unsupported hardware.....	769
Minimum supported hardware.....	769
Permission Notice.....	770
Programming interface information.....	770
Trademarks.....	770
Standards.....	771
Bibliography.....	773
Index.....	777

Figures

1. Blocking fixed-length records.....	13
2. Variable-length records on z/OS.....	17
3. ASA Example.....	25
4. ungetwc() Example.....	34
5. Generation data group example for C.....	41
6. Generation data group example for C++.....	42
7. How the operating system completes the DCB.....	54
8. Example of reading updated records.....	62
9. Example of signal handler.....	70
10. fldata() Structure.....	72
11. Unnamed pipes example.....	84
12. Named pipes example.....	86
13. Example of z/OS UNIX stream input and output functions.....	88
14. Example of HFS stream input and output functions.....	91
15. fldata() structure.....	93
16. Types and advantages of VSAM data sets.....	97
17. VSAM example.....	99
18. KSDS example.....	120
19. KSDS example.....	125
20. RRDS example.....	128
21. fldata() structure.....	130
22. fldata() structure.....	139
23. Removing members of a PDS.....	146

24. Renaming members of a PDS.....	146
25. fldata() structure.....	151
26. Memory file example, part 1.....	152
27. Memory file example, part 2.....	152
28. fldata() structure.....	158
29. fldata() example.....	158
30. CELQPIPI MSGRTN example.....	159
31. __amrc structure.....	161
32. Example of printing the __amrc structure.....	164
33. __amrc2 structure.....	164
34. Example of using SIGIOERR.....	169
35. Sample File I/O Trace.....	171
36. Explicit use of a DLL in an application using the dllload() family of functions.....	185
37. Explicit use of a DLL in an application using the dlopen() family of functions.....	187
38. Using #pragma export to create a DLL executable module named BASICIO.....	191
39. Using #pragma export to create the triangle DLL executable module.....	191
40. Using _Export to create the triangle DLL executable module.....	191
41. Summary of DLL and DLL application preparation and usage.....	195
42. Referencing functions and external variables in DLL code.....	203
43. Pointer Assignment in DLL code.....	204
44. Pointer assignment in non-DLL code.....	205
45. File 1. Application A	206
46. File 2. Application A	206
47. File 3. Application B.....	206
48. File 4. DLL.....	207

49. DLL function pointer call in non-DLL code.....	208
50. DLL function pointer call in non-DLL code.....	209
51. C non-DLL code.....	210
52. C DLL code.....	210
53. C++ DLL code.....	211
54. C++ DLL code calling an Assembler function.....	211
55. Comparison of function pointers in non-DLL code.....	211
56. C DLL code.....	212
57. C non-DLL code.....	212
58. File 1 C DLL code.....	212
59. File 2 C DLL code.....	213
60. File 3 C non-DLL code.....	213
61. Comparison of two DLL function pointers in non-DLL code.....	213
62. Undefined comparison in DLL code (C or C++).....	214
63. Comparison of function pointers in DLL code (C or C++).....	214
64. Valid comparisons in DLL code (C or C++).....	215
65. Application CCNGA2.....	215
66. Application CCNGA2D1.....	216
67. Application CCNGA2D2.....	216
68. Application CCNGA2D3.....	216
69. CCNGA2M1.....	217
70. CCNGA2M2.....	218
71. Comparison of struct li, alignments under ILP32 and LP64.....	230
72. Comparison of struct lii alignments under ILP32 and LP64.....	231
73. Comparison of struct ili alignments under ILP32 and LP64.....	232

74. Example of potential alignment problems when a struct is shared or exchanged among 32-bit and 64-bit processes.....	241
75. Example of user-defined data padding for a structure that is shared or exchanged among 32-bit and 64-bit processes.....	245
76. Referring to thread-specific data.....	253
77. Controlling external static.....	262
78. Making strings constant.....	263
79. Example of controlling the memory area.....	264
80. How to Make String Literals Modifiable.....	264
81. Referencing objects in the writable static area, Part 1.....	265
82. Referencing objects in the writable static area, Part 2.....	266
83. Example illustrating C++ exception handling/traceback.....	275
84. Example illustrating C++ exception handling/traceback.....	276
85. Example illustrating signal handling.....	291
86. An electrical analogy showing the socket concept.....	295
87. A typical stream socket session.....	302
88. A typical datagram socket session.....	303
89. An application using socket().....	304
90. An application using bind().....	304
91. A bind() function using gethostbyname().....	305
92. An application using listen().....	305
93. An application using connect().....	305
94. An application using accept().....	306
95. An application using send() and recv().....	306
96. An application using sendto() and recvfrom().....	307
97. An application using select().....	307
98. An Application Using ioctl().....	308

99. An application using close().....	308
100. A conceptual overview of the compile, bind, and run steps.....	309
101. stackadd.cpp file (ccntmp3.cpp).....	321
102. stackops.cpp file (ccntmp4.cpp).....	321
103. stack.h file (ccntmp2.h).....	321
104. stack.c file (ccntmp1.c).....	322
105. stackops.h File (ccntmp5.h).....	322
106. JCL to compile source Files and TEMPINC destination.....	322
107. z/OS UNIX Syntax.....	322
108. Simple and typical use of explicit instantiation declarations.....	325
109. Erroneous use of explicit instantiation declarations.....	326
110. _CEE_RUNOPTS behaviour.....	343
111. Environment variables example-Part 1.....	359
112. Environment variables example-Part 2.....	360
113. z/Architecture FPC register-rounding mode definitions.....	404
114. Biased exponent type definitions.....	404
115. Test Data Class masks.....	404
116. Test Data Group masks.....	405
117. Example of code that requires the global namespace syntax.....	416
118. Numeric conversions example.....	435
119. Example of using OPTIMIZE(2).....	452
120. Flow of regular compiler processing.....	461
121. IPA compile step processing.....	462
122. IPA link step processing.....	463
123. Specifying alternative initialization at link edit.....	497

124. Sample Freestanding z/OS Routine.....	499
125. Link edit control statements used to build a freestanding z/OS routine.....	499
126. Compile and link using EDCCL.....	499
127. Sample reentrant freestanding z/OS routine.....	499
128. Building and running a reentrant freestanding z/OS routine.....	500
129. System exit example.....	503
130. Example of function used in a persistent C environment.....	506
131. Using a persistent C environment.....	507
132. Example of user routine.....	510
133. Example of application service routine.....	512
134. Example of server initialization stub.....	516
135. Example of server message stub-LIFO.....	517
136. Example of server message stub-FIFO.....	518
137. Example of server message stub-GET.....	520
138. Example of server message stub-TERM.....	521
139. Example of routine to generate abend.....	524
140. Example of routine to get storage.....	525
141. Example of routine to free storage.....	526
142. Location of user exits.....	538
143. Interface for Assembler user exits.....	542
144. CEEAUE_FLAGS format.....	544
145. Exit_list and hook_exit control blocks.....	552
146. Computational independence.....	556
147. Example of a C program running without MTF.....	557
148. Example of a C program running without MTF (Part 2).....	557

149. Processor usage with one parallel function.....	558
150. Sample program using one parallel function.....	559
151. Processor usage with two parallel function.....	559
152. Sample program using two parallel functions.....	560
153. Processor use with multiple instances of the same parallel function.....	561
154. Sample program using multiple instances of the same parallel function.....	561
155. Basic MTF layout.....	563
156. Identifying Computationally-Independent Code.....	565
157. Sample code as a parallel function.....	566
158. Scheduling instances of a parallel function.....	566
159. Main task program to call dot product function.....	567
160. Sample code to be changed to use MTF.....	567
161. Sample code (main routine).....	568
162. Sample code (routine to create parallel functions).....	568
163. Sample JCL to compile and link main task program.....	569
164. Sample JCL to compile and link parallel functions.....	570
165. Source code for EDCMTFS.....	570
166. Example runtime JCL.....	571
167. Example illustrating how to use EXEC CICS commands.....	577
168. Format of data written to a CICS data queue.....	580
169. Example illustrating error handling under CICS.....	585
170. Example illustrating how to use EXEC CICS commands.....	589
171. Child C program after translation.....	590
172. JCL to translate and compile a C program.....	593
173. JCL to translate and compile a C++ program.....	593

174. JCL to install z/OS XL C/C++ application programs.....	595
175. C/370 CALLing CSP under TSO.....	598
176. z/OS XL C transferring control to CSP under TSO using the XFER/DXFR statement.....	600
177. CSP CALLing z/OS XL C under TSO.....	601
178. CSP transferring control to z/OS XL C under TSO using the XFER statement.....	602
179. CSP CALLing z/OS XL C under CICS.....	604
180. CSP transferring control to z/OS XL C under CICS using the XFER statement.....	605
181. CSP Transferring Control to z/OS XL C under CICS Using the DXFR Statement.....	607
182. Example using DWS and C++.....	609
183. z/OS XL C/C++ Using Data Window Services.....	610
184. Using DB2 with C.....	613
185. Using DB2 with C/C++.....	615
186. Example using GDDM and C.....	620
187. Example using GDDM and C++.....	622
188. C++ Program using IMS.....	627
189. C Program using IMS.....	629
190. Header file for IMS example.....	630
191. QMF interface example.....	633
192. C++ Calling a C program that accesses QMF.....	635
193. C program that accesses QMF.....	636
194. Conceptual model of the locale build process.....	645
195. Example locale source containing header, body, and trailer.....	654
196. Example LC_CTYPE definition.....	658
197. Example LC_COLLATE definition.....	662
198. Example LC_MONETARY definition.....	667

199. Example LC_NUMERIC definition.....	668
200. Example LC_MESSAGES definition.....	671
201. Example LC_TOD definition.....	673
202. Example definition of LC_SYNTAX.....	675
203. Expected grammar for method files.....	676
204. Referring explicitly to a customized locale.....	694
205. Referring implicitly to a customized locale.....	695
206. Using environment variables to select a locale.....	695
207. Additional locale categories for POSIX C.....	700
208. Determining which locale is in effect.....	705
209. Supplied code set converters.....	710
210. Example of hybrid coded character set.....	734
211. Compile-edit, related to locale function.....	735
212. Example of __CODESET__ macro.....	738
213. Values of macros __FILETAG__, __LOCALE__, and __CODESET__.....	739
214. Using the CONVLIT compiler option.....	740
215. Example of output when locale option is used.....	742
216. Using the pragma convert directive.....	743
217. Example of using a layout string modifier.....	753
218. Example of using the m_setvalues_layout() function.....	753
219. Example of bidirectional layout API's.....	756
220. Variant characters.....	760
221. CCNGMV1: Displaying hexadecimal values.....	761
222. Sample ISPF macro for displaying square brackets.....	763

Tables

1. Syntax examples.....	xxxvi
2. z/OS XL C/C++ and related documents.....	xxxviii
3. Documents by task.....	xli
4. C control to ASA characters.....	14
5. C control to ASA characters translation table.....	25
6. Manipulating wide character arrays.....	36
7. PDSE and PDS differences.....	42
8. Rules for possible concatenations.....	44
9. Other devices supported for input and output.....	48
10. Parameters for the fopen() and freopen() functions for z/OS OS I/O.....	49
11. fopen() defaults for LRECL and BLKSIZE when creating OS files.....	54
12. C control to ASA characters.....	59
13. Parameters for the fopen() and freopen() functions for z/OS UNIX file system I/O.....	79
14. Summary of VSAM data set characteristics and allowable I/O operations.....	99
15. Keywords for the fopen() and freopen() functions for VSAM data sets.....	102
16. Summary of VSAM record I/O operations.....	110
17. AMODE31 application XADDR support.....	113
18. Summary of fseek() and ftell() parameters in text and binary.....	116
19. Summary of VSAM text I/O operations.....	117
20. Summary of VSAM binary I/O operations.....	117
21. Keywords for the fopen() and freopen() functions for terminal I/O.....	132
22. Keywords for the fopen() and freopen() functions for memory file I/O.....	143
23. __last_op codes and diagnosis information.....	165

24. Linkage used by C or C++ Interlanguage Programs.....	176
25. Summary of DLL concepts and terms.....	182
26. Example programs to demonstrate compiling options.....	201
27. Examples of how to compile two source modules and list result.....	201
28. Referencing functions and external variables.....	202
29. Comparison of ILP32 and LP64 addressing capabilities.....	221
30. Comparison of ILP32 and LP64 data models.....	221
31. ILP32 and LP64 type size comparisons for signed and unsigned data types.....	222
32. Example of diagnostic messages generated from code that is not ready to be migrated from ILP32 to LP64.....	225
33. Comparison of ILP32 and LP64 processing and runtime options.....	227
34. Comparison of data structure member lengths produced from the same code	228
35. Example of possible change of result after conversion from signed number to unsigned long.....	233
36. Example of possible change of result after conversion from unsigned int variable to signed long.....	233
37. Example of possible change of result after conversion from signed long long variable to unsigned long.....	234
38. Example of possible change of result after conversion from unsigned long long variable to unsigned long.....	234
39. Example of possible change of result after conversion from signed long long variable to signed long.....	235
40. Example of possible change of result after conversion from unsigned long long variable to signed long.....	236
41. Example of source code that explicitly converts an integer to a pointer.....	237
42. Example of truncation problem with a pointer cast conversion.....	238
43. Type of an integer constant.....	239
44. An attempt to share pointers between 32-bit and 64-bit processes.....	239
45. Example of unexpected behavior resulting from use of unsuffixed numbers.....	241
46. Example of using LONG_MAX macros in a printf subroutine.....	242

47. Example of source code that successfully shares pointers between ILP32 and LP64 programs.....	244
48. Functions used in creating multi-threaded applications.....	247
49. Functions to change default attributes.....	247
50. Functions used to control individual threads in a multi-threaded environment.....	248
51. Functions that allow for synchronization between threads.....	249
52. Functions used with thread-specific data.....	252
53. Cancellation point summary.....	255
54. Functions used to control cancellability.....	256
55. Functions used for cleanup purposes.....	257
56. C/C++ functions that support floating-point.....	269
57. Special purpose C/C++ functions that support floating-point.....	272
58. Functions that establish a signal handler.....	280
59. Other signal-related functions.....	280
60. Hardware exceptions - Default runtime messages and system actions.....	285
61. Software exceptions - Default runtime messages and system actions with POSIX(OFF).....	285
62. Default signal processing with POSIX(ON).....	287
63. Original versions of fdlibm functions	351
64. Standard general-instruction prototypes.....	361
65. Built-in general-instruction prototypes.....	364
66. PLO helper macros.....	385
67. Compare and load prototypes.....	386
68. Compare and swap prototypes.....	386
69. Double compare and swap prototypes.....	387
70. Compare and swap and store prototypes.....	387
71. Compare swap and double store prototypes.....	388

72. Compare and swap and triple store prototypes.....	388
73. Decimal instruction prototypes.....	389
74. Floating-point instruction prototypes.....	394
75. Decimal floating-point instruction prototypes for IEEE operations	395
76. Decimal floating-point instruction prototypes for IEEE . . . is operations	397
77. Decimal floating-point instruction prototypes for IBM Instructions	397
78. Instruction prototypes for conversions between decimal floating-point and zoned types.....	403
79. Hexadecimal floating-point instruction prototypes.....	405
80. Binary floating-point instruction prototypes.....	407
81. Built-in functions for transactional memory.....	410
82. SUSv3 options and option groups.....	418
83. Symbols not supported.....	420
84. Examples of acceptable alias types.....	429
85. Comparison of code generated with the ANSIALIAS and NOANSIALIAS options.....	430
86. Example of using temporaries to remove aliasing effects.....	432
87. Referencing data types.....	436
88. Pragmas that affect performance.....	438
89. SMP suboptions.....	442
90. C-library built-in functions.....	443
91. Platform-specific built-in functions.....	444
92. Optimization levels and options.....	453
93. Examples of optimization.....	455
94. Parts used for freestanding applications.....	501
95. Parts used by exit routines.....	505
96. Parts used by persistent environments.....	509

97. Parts used by or with application server routines.....	523
98. Abend and reason codes specific to system programming environments.....	527
99. Summary of types.....	528
100. User exits supported under z/OS Language Environment.....	538
101. Sample Assembler user exits for z/OS Language Environment.....	540
102. Interaction of Assembler user exits.....	550
103. Common data types between z/OS XL C and CSP.....	597
104. PCB generated for C program under TSO and IMS.....	626
105. PCB generated for C or C++ program under TSO and IMS.....	627
106. Characters in portable character set and corresponding symbolic names.....	646
107. Locale object prefix.....	680
108. Supported language-territory names and LT codes for EBCDIC locales.....	681
109. Supported language-territory names and LT codes for ASCII locales.....	684
110. Supported codeset names and CC codes.....	686
111. C macros used as synonyms for special locale names.....	689
112. Referencing data types.....	710
113. Coded character set conversion tables.....	711
114. UCS-2 converter names.....	721
115. Mappings of 13 PPCS variant characters.....	731
116. Mappings of Hex encoding of 13 PPCS variant characters.....	732
117. Layout attribute and values.....	749
118. Description of the values of SeenTail attribute.....	750

About this document

This document provides information about implementing programs that are written in C and C++. It contains advanced guidelines and information for developing C and C++ programs to run under z/OS.

This document contains reference information about implementing programs that are written in C and C++, which is specific to z/OS C/C++ runtime and z/OS.

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

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

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 IBM Documentation using a screen reader, syntax diagrams are provided in dotted decimal format.

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 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.

The following table provides syntax examples.

Table 1. Syntax examples

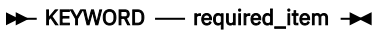
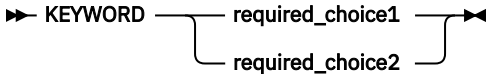
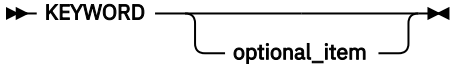
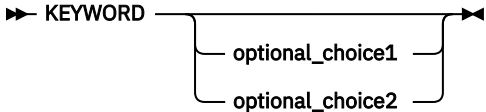
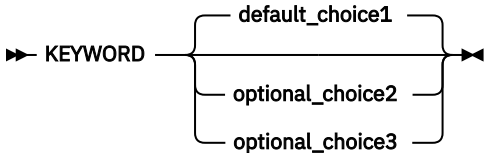
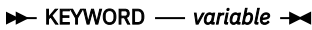
Item	Syntax example
Required item. Required items appear on the main path of the horizontal line. You must specify these items.	
Required choice. A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	
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.	

Table 1. Syntax examples (continued)

Item	Syntax example
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.	
Fragment. The fragment symbol indicates that a labeled 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.	

z/OS XL C/C++ and related documents

This topic summarizes the content of the z/OS XL C/C++ documents and shows where to find related information in other documents.

Table 2. z/OS XL C/C++ and related documents

Document Title and Number	Key Sections/Chapters in the Document
<u>z/OS XL C/C++ Programming Guide</u>	<p>Guidance information for:</p> <ul style="list-style-type: none"> • XL C/C++ input and output • Debugging z/OS XL C programs that use input/output • Using linkage specifications in C++ • Combining C and assembler • Creating and using DLLs • Using threads in z/OS UNIX System Services applications • Reentrancy • Handling exceptions, error conditions, and signals • Performance optimization • Network communications under z/OS UNIX • Interprocess communications using z/OS UNIX • Structuring a program that uses C++ templates • Using environment variables • Using System Programming C facilities • Library functions for the System Programming C facilities • Using runtime user exits • Using the z/OS XL C multitasking facility • Using other IBM products with z/OS XL C/C++ (IBM CICS® Transaction Server for z/OS, CSP, DWS, IBM DB2®, IBM GDDM, IBM IMS, ISPF, IBM QMF) • Globalization: locales and character sets, code set conversion utilities, mapping variant characters • POSIX character set • Code point mappings • Locales supplied with z/OS XL C/C++ • Charmap files supplied with z/OS XL C/C++ • Examples of charmap and locale definition source files • Converting code from coded character set IBM-1047 • Using built-in functions • Using vector programming support • Using runtime check library • Using high performance libraries • Programming considerations for z/OS UNIX C/C++

Table 2. z/OS XL C/C++ and related documents (continued)

Document Title and Number	Key Sections/Chapters in the Document
<u>z/OS XL C/C++ User's Guide</u>	<p>Guidance information for:</p> <ul style="list-style-type: none"> • z/OS XL C/C++ examples • Compiler options • Binder options and control statements • Specifying Language Environment® runtime options • Compiling, IPA Linking, binding, and running z/OS XL C/C++ programs • Utilities (Object Library, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH, c89, xlc) • Diagnosing problems • Cataloged procedures and IBM REXX EXECs • Customizing default options for the z/OS XL C/C++ compiler
<u>z/OS XL C/C++ Language Reference</u>	<p>Reference information for:</p> <ul style="list-style-type: none"> • The C and C++ languages • Lexical elements of z/OS XL C and C++ • Declarations, expressions, and operators • Implicit type conversions • Functions and statements • Preprocessor directives • C++ classes, class members, and friends • C++ overloading, special member functions, and inheritance • C++ templates and exception handling • z/OS XL C and C++ compatibility
<u>z/OS XL C/C++ Messages</u>	<p>Provides error messages and return codes for the compiler, and its related application interface libraries and utilities. For the z/OS C/C++ runtime library messages, refer to <u>z/OS Language Environment Runtime Messages</u>. For the c89 and xlc utility messages, refer to <u>z/OS UNIX System Services Messages and Codes</u>.</p>
<u>z/OS C/C++ Runtime Library Reference</u>	<p>Reference information for:</p> <ul style="list-style-type: none"> • header files • library functions

Table 2. z/OS XL C/C++ and related documents (continued)

Document Title and Number	Key Sections/Chapters in the Document
<u>z/OS C Curses</u>	<p>Reference information for:</p> <ul style="list-style-type: none"> • Curses concepts • Key data types • General rules for characters, renditions, and window properties • General rules of operations and operating modes • Use of macros • Restrictions on block-mode terminals • Curses functional interface • Contents of headers • The terminfo database
<u>z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer</u>	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> • Common migration questions • Application executable program compatibility • Source program compatibility • Input and output operations compatibility • Class library migration considerations • Changes between releases of z/OS • Pre-z/OS C and C++ compilers to current compiler migration • Other migration considerations
<u>z/OS Metal C Programming Guide and Reference</u>	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> • Metal C run time • Metal C programming • AR mode
<u>Standard C++ Library Reference</u>	<p>The documentation describes how to use the following three main components of the Standard C++ Library to write portable C/C++ code that complies with the ISO standards:</p> <ul style="list-style-type: none"> • ISO Standard C Library • ISO Standard C++ Library • Standard Template Library (C++) <p>The ISO Standard C++ library consists of 51 required headers. These 51 C++ library headers (along with the additional 18 Standard C headers) constitute a hosted implementation of the C++ library. Of these 51 headers, 13 constitute the Standard Template Library, or STL.</p>

Table 2. z/OS XL C/C++ and related documents (continued)

Document Title and Number	Key Sections/Chapters in the Document
z/OS Common Debug Architecture User's Guide	<p>This documentation is the user's guide for IBM's libddpi library. It includes:</p> <ul style="list-style-type: none"> • Overview of the architecture • Information on the order and purpose of API calls for model user applications and for accessing DWARF information • Information on using the Common Debug Architecture with C/C++ source <p>This user's guide is part of the Runtime Library Extensions documentation.</p>
z/OS Common Debug Architecture Library Reference	<p>This documentation is the reference for IBM's libddpi library. It includes:</p> <ul style="list-style-type: none"> • General discussion of Common Debug Architecture • Description of APIs and data types related to stacks, processes, operating systems, machine state, storage, and formatting <p>This reference is part of the Runtime Library Extensions documentation.</p>
DWARF/ELF Extensions Library Reference	<p>This documentation is the reference for IBM's extensions to the libdwarf and libelf libraries. It includes information on:</p> <ul style="list-style-type: none"> • Consumer APIs • Producer APIs <p>This reference is part of the Runtime Library Extensions documentation.</p>
IBM Developer for z Systems®	<p>The documentation for IBM Developer for z/OS (www.ibm.com/docs/en/adfz/developer-for-zos) provides guidance and reference information for debugging programs, using IBM Developer for z Systems in different environments, and language-specific information.</p>

Note: For complete and detailed information on linking and running with Language Environment services and using the Language Environment runtime options, refer to [z/OS Language Environment Programming Guide](#). For complete and detailed information on using interlanguage calls, refer to [z/OS Language Environment Writing Interlanguage Communication Applications](#).

The following table lists the z/OS XL C/C++ and related documents. The table groups the documents according to the tasks they describe.

Table 3. Documents by task

Tasks	Documents
Planning, preparing, and migrating to z/OS XL C/C++	<ul style="list-style-type: none"> • z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer • z/OS Language Environment Customization • z/OS Language Environment Runtime Application Migration Guide • z/OS UNIX System Services Planning • z/OS Planning for Installation

Table 3. Documents by task (continued)

Tasks	Documents
Installing	<ul style="list-style-type: none"> • z/OS Program Directory • z/OS Planning for Installation • z/OS Language Environment Customization
Option customization	<ul style="list-style-type: none"> • z/OS XL C/C++ User's Guide
Coding programs	<ul style="list-style-type: none"> • z/OS C/C++ Runtime Library Reference • z/OS XL C/C++ Language Reference • z/OS XL C/C++ Programming Guide • z/OS Metal C Programming Guide and Reference • z/OS Language Environment Concepts Guide • z/OS Language Environment Programming Guide • z/OS Language Environment Programming Reference
Coding and binding programs with interlanguage calls	<ul style="list-style-type: none"> • z/OS XL C/C++ Programming Guide • z/OS XL C/C++ Language Reference • z/OS Language Environment Programming Guide • z/OS Language Environment Writing Interlanguage Communication Applications • z/OS MVS Program Management: User's Guide and Reference • z/OS MVS Program Management: Advanced Facilities
Compiling, binding, and running programs	<ul style="list-style-type: none"> • z/OS XL C/C++ User's Guide • z/OS Language Environment Programming Guide • z/OS Language Environment Debugging Guide • z/OS MVS Program Management: User's Guide and Reference • z/OS MVS Program Management: Advanced Facilities
Compiling and binding applications in the z/OS UNIX (z/OS UNIX) environment	<ul style="list-style-type: none"> • z/OS XL C/C++ User's Guide • z/OS UNIX System Services User's Guide • z/OS UNIX System Services Command Reference • z/OS MVS Program Management: User's Guide and Reference • z/OS MVS Program Management: Advanced Facilities

Table 3. Documents by task (continued)

Tasks	Documents
Debugging programs	<ul style="list-style-type: none"> • README file • z/OS XL C/C++ User's Guide • z/OS XL C/C++ Messages • z/OS XL C/C++ Programming Guide • z/OS Language Environment Programming Guide • z/OS Language Environment Debugging Guide • z/OS Language Environment Runtime Messages • z/OS UNIX System Services Messages and Codes • z/OS UNIX System Services User's Guide • z/OS UNIX System Services Command Reference • z/OS UNIX System Services Programming Tools • IBM Developer for z/OS (www.ibm.com/docs/en/adfz/developer-for-zos) documentation
Developing debuggers and profilers	<ul style="list-style-type: none"> • z/OS Common Debug Architecture User's Guide • z/OS Common Debug Architecture Library Reference • DWARF/ELF Extensions Library Reference
Packaging XL C/C++ applications	<ul style="list-style-type: none"> • z/OS XL C/C++ Programming Guide • z/OS XL C/C++ User's Guide
Using shells and utilities in the z/OS UNIX environment	<ul style="list-style-type: none"> • z/OS XL C/C++ User's Guide • z/OS UNIX System Services Command Reference • z/OS UNIX System Services Messages and Codes
Using sockets library functions in the z/OS UNIX environment	<ul style="list-style-type: none"> • z/OS C/C++ Runtime Library Reference
Using the ISO Standard C++ Library to write portable C/C++ code that complies with ISO standards	<ul style="list-style-type: none"> • Standard C++ Library Reference
Performing diagnosis and submitting an Authorized Program Analysis Report (APAR)	<ul style="list-style-type: none"> • z/OS XL C/C++ User's Guide

Note: For information on using the prelinker, see the appendix on prelinking and linking z/OS XL C/C++ programs in [z/OS XL C/C++ User's Guide](#).

Softcopy documents

The z/OS XL C/C++ publications are supplied in PDF format and available for download from the [z/OS XL C/C++ documentation library \(www.ibm.com/software/awdtools/czos/library\)](http://www.ibm.com/software/awdtools/czos/library).

Note: To ensure that you can access cross-reference links to other z/OS XL C/C++ PDF documents, download each document into the same directory on your local machine and do not change the PDF file names.

To read a PDF file, use the Adobe Reader. If you do not have the Adobe Reader, you can download it (subject to Adobe license terms) from the [Adobe website \(www.adobe.com\)](http://www.adobe.com).

You can also browse the documents on the World Wide Web by visiting the [z/OS Internet Library \(www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary\)](http://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary).

Softcopy examples

Most of the larger examples in the following documents are available in machine-readable form:

- [z/OS XL C/C++ Language Reference](#)
- [z/OS XL C/C++ User's Guide](#)
- [z/OS XL C/C++ Programming Guide](#)

In the following documents, a label on an example indicates that the example is distributed as a softcopy file:

- [z/OS XL C/C++ Language Reference](#)
- [z/OS XL C/C++ Programming Guide](#)
- [z/OS XL C/C++ User's Guide](#)

The label is the name of a member in the CBC.SCCNSAM data set. The labels begin with the form CCN or CLB. Examples labelled as CLB appear only in the [z/OS XL C/C++ User's Guide](#), while examples labelled as CCN appear in all three documents, and are further distinguished by x following CCN, where x represents one of the following:

- R and X refer to [z/OS XL C/C++ Language Reference](#)
- G refers to [z/OS XL C/C++ Programming Guide](#)
- U refers to [z/OS XL C/C++ User's Guide](#)

Related z/OS XL C/C++ information

The following list shows where to find related z/OS XL C/C++ information:

- Softcopy examples and related documents information in [z/OS XL C/C++ User's Guide](#)
- Additional technical support is available at the [z/OS XL C/C++ Support page \(www.ibm.com/mysupport/s/topic/0TO0z0000006v6TGAQ/xl-cc?language=en_US&productId=01t0z000007g72LAAQ\)](#). This page provides a portal with search capabilities to a large selection of technical support FAQs and other support documents.
- For the latest information about z/OS XL C/C++, visit the [product page for z/OS XL C/C++ \(www.ibm.com/products/xl-cpp-compiler-zos\)](#).
- For information about boosting performance, productivity and portability, visit IBM Z and LinuxONE Community (community.ibm.com/community/user/ibmz-and-linuxone/groups/topic-home?CommunityKey=5805da79-8284-4015-97fb-5a19f6480452).

Where to find more information

For an overview of the information associated with z/OS, see [z/OS Information Roadmap](#).

z/OS Basic Skills in IBM Documentation

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

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

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

z/OS Basic Skills in IBM Documentation (www.ibm.com/docs/en/zos-basic-skills?topic=zosbasics/com.ibm.zos.zbasics/homepage.html) is available to all users (no login required).

How to provide feedback to IBM

We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information. For more information, see [How to send feedback to IBM](#).

Summary of changes

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

Note: IBM z/OS policy for the integration of service information into the z/OS product documentation library is documented on the z/OS Internet Library under [IBM z/OS Product Documentation Update Policy](http://www.ibm.com/docs/en/zos/latest?topic=zos-product-documentation-update-policy) (www.ibm.com/docs/en/zos/latest?topic=zos-product-documentation-update-policy).

Summary of changes for z/OS 3.1

The following content is new, changed, or no longer included in z/OS 3.1

New

The following content is new.

September 2023 release

- None.

Changed

The following content is changed.

September 2023 release

- None.

Deleted

The following content is deleted.

September 2023 release

- The *Using high performance libraries* chapter is moved to z/OS C/C++ Runtime Library Reference.

Part 1. Introduction

This part discusses introductory concepts on the z/OS XL C/C++ feature. Specifically, it discusses the following:

- [Chapter 1, “About IBM z/OS XL C/C++,” on page 3](#)

Chapter 1. About IBM z/OS XL C/C++

For an introduction to z/OS XL C/C++ and for information about changes to z/OS XL C/C++ for the current release, see [About IBM z/OS XL C/C++](#) in *z/OS XL C/C++ User's Guide*.

Part 2. Input and Output

This part describes the models of input and output available with IBM z/OS XL C/C++.

The C runtime functions are available if the corresponding C header files are used. C I/O can be used by C++ when the C runtime library functions are used.

The following references provide a description and examples of I/O streams:

- [Chapter 2, “Introduction to C and C++ input and output,” on page 7](#)
- [Chapter 3, “Understanding models of C I/O,” on page 11](#)
- [Chapter 4, “Using the Standard C++ Library I/O Stream Classes,” on page 21](#)
- [Chapter 5, “Buffering of C streams,” on page 23](#)
- [Chapter 6, “Using ASA text files,” on page 25](#)
- [Chapter 7, “z/OS XL C support for the double-byte character set,” on page 29](#)
- [Chapter 8, “Performing OS I/O operations,” on page 37](#)
- [Chapter 9, “Performing z/OS UNIX file system I/O operations,” on page 75](#)
- [Chapter 10, “Performing VSAM I/O operations,” on page 95](#)
- [Chapter 11, “Performing terminal I/O operations,” on page 131](#)
- [Chapter 12, “Performing memory file and hiperspace I/O operations,” on page 141](#)
- [Chapter 13, “Performing CICS Transaction Server I/O operations,” on page 153](#)
- [Chapter 14, “Language Environment Message file operations,” on page 155](#)
- [Chapter 15, “CELQPIPI MSGRTN file operations,” on page 157](#)
- [Chapter 16, “Debugging I/O programs,” on page 161](#)

Chapter 2. Introduction to C and C++ input and output

This chapter provides you with a general introduction to C and C++ input and output (I/O). Four types of C and C++ input and output are discussed in this chapter:

- [“Text streams” on page 7](#)
- [“Binary streams” on page 8](#)
- [“Record I/O” on page 8](#)
- [“Blocked I/O” on page 8](#)

Types of C and C++ input and output

A stream is a flow of data elements that are transmitted or intended for transmission in a defined format. A record is a set of data elements treated as a unit, and a file (or data set) is a named set of records that is stored or processed as a unit.

The z/OS XL C/C++ compiler supports four types of input and output: text streams, binary streams, record I/O, and blocked I/O. Text and binary streams are both ISO C/C++ standards; record I/O and blocked I/O are extensions for z/OS XL C. Record I/O and blocked I/O are not supported by either the USL I/O Stream Class Library or the Standard C++ I/O stream classes.

Note: If you have written data in one of these four types and try to read it as another type (for example, reading a binary file in text mode), you may not get the behavior that you expect.

Text streams

Text streams contain printable characters and, depending on the type of file, control characters. Text streams are organized into lines. Each line ends with a control character, usually a new-line. The last record in a text file may or may not end with a control character, depending on what kind of file you are using. Text files recognize the following control characters:

\a

Alarm.

\b

Backspace.

\f

Form feed.

\n

New-line.

\r

Carriage return.

\t

Horizontal tab character.

\v

Vertical tab character.

\x0E

DBCS shift-out character. Indicates the beginning of a DBCS string, if MB_CUR_MAX>1 in the definition of the locale that is in effect. For more information about __MBCURMAX, see [z/OS XL C support for the double-byte character set in z/OS XL C/C++ Programming Guide](#).

\x0F

DBCS shift-in character. Indicates the end of a DBCS string, if MB_CUR_MAX>1 in the definition of the locale that is in effect. For more information about __MBCURMAX, see [z/OS XL C support for the double-byte character set in z/OS XL C/C++ Programming Guide](#).

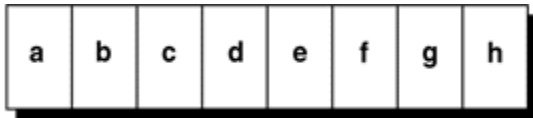
Control characters behave differently in terminal files (see [Chapter 11, “Performing terminal I/O operations,”](#) on page 131) and ASA files (see [Chapter 6, “Using ASA text files,”](#) on page 25).

Binary streams

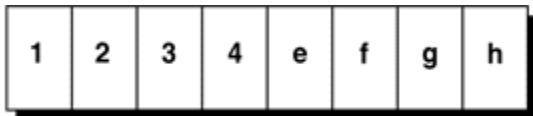
Binary streams contain a sequence of bytes. For binary streams, the library does not translate any characters on input or output. It treats them as a continuous stream of bytes, and ignores any record boundaries. When data is written out to a record-oriented file, it fills one record before it starts filling the next. Streams in the z/OS UNIX file system follow the binary model, regardless of whether they are opened for text, binary, record I/O, or blocked I/O. You can simulate record I/O by using new-line characters as record boundaries.

Record I/O

Record I/O is an extension to the ISO standard. For files opened in record format, z/OS XL C/C++ reads and writes one record at a time. If you try to write more data to a record than the record can hold, the data is truncated. For record I/O, z/OS XL C/C++ allows only the use of `fread()` and `fwrite()` to read and write to files. Any other functions (such as `fprintf()`, `fscanf()`, `getc()`, and `putc()`) will fail. For record-oriented files, records do not change size when you update them. If the new record has fewer bytes than the original record, the new data fills the first n bytes, where n is the number of bytes of the new data. The record will remain the same size, and the old bytes (those after n) are left unchanged. A subsequent update begins at the next boundary. For example, if you have the string "abcdefgh":



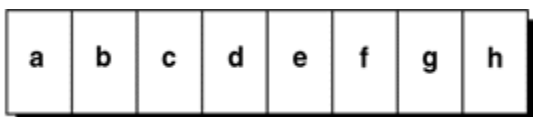
and you overwrite it with the string "1234", the record will look like this:



z/OS XL C/C++ record I/O is binary. That is, it does not interpret any of the data in a record file and therefore does not recognize control characters. The only exception is for file categories that do not support records, such as the UNIX file system (also known as POSIX I/O). For these files, z/OS XL C/C++ uses new-line characters as record boundaries.

Blocked I/O

Blocked I/O is an extension to the ISO standard. For files opened in block format, z/OS XL C/C++ reads and writes one block at a time. If you try to write more data to a block than the block can hold, the data is truncated. For blocked I/O, z/OS XL C/C++ allows only the use of `fread()` and `fwrite()` to read and write to files. Any other functions (such as `fprintf()`, `fscanf()`, `getc()`, and `putc()`) will fail. Blocks do not change size when you update them. If the new block has fewer bytes than the original block, the new data fills the first n bytes, where n is the number of bytes of the new data. The block will remain the same size, and the old bytes (those after n) are left unchanged. A subsequent update begins at the next boundary. For example, if you have the string "abcdefgh":



and you overwrite it with the string "1234", the block will look like this:

1	2	3	4	e	f	g	h
---	---	---	---	---	---	---	---

z/OS XL C/C++ blocked I/O is binary. That is, it does not interpret any of the data in a block file and therefore does not recognize control characters.

The `fflush()` function has no effect for blocked I/O files.

Chapter 3. Understanding models of C I/O

This chapter describes z/OS XL C/C++ support for the major models of C I/O:

- The record model
- The byte stream model

The next chapter (Chapter 4, “Using the Standard C++ Library I/O Stream Classes,” on page 21) describes a third major model, the *object-oriented model*.

The record model for C I/O

Almost all the kinds of I/O that z/OS XL C/C++ supports use this model. The only ones that do not are z/OS UNIX file system, memory file, and Hiperspace I/O.

The record model consists of the following:

- A *record*, which is the unit of data transmitted to and from a program.
- A *block*, which is the unit of data transmitted to and from a device. Each block may contain one or more records.

In the record model of I/O, records and blocks have the following attributes:

RECFM

Specifies the format of the data or how the data is organized on the physical device.

LRECL

Specifies the length of logical records (as opposed to physical ones). Variable length records include a count field that is normally not available to the programmer.

BLKSIZE

Specifies the length of physical records (blocks on the physical device).

Record formats

Use the RECFM attribute to specify the record format. The records in a file using the record model have one of the following formats:

- Fixed-length (F)
- Variable-length (V)
- Undefined-length (U)

Note: z/OS XL C/C++ does not support Format-D files.

These formats support the following additional options for RECFM. The record formats and the options associated with them are discussed in the following sections.

A

Specifies that the file contains ASA control characters.

B

Specifies that a file is blocked. A blocked file can have more than one record in each block.

M

Specifies that the file contains machine control characters.

S

Specifies that a file is either in standard format (if it is fixed) or spanned (if it is variable). In a standard file, every block must be full before another one starts. In a spanned file, a record can be longer than a block. If it is, the record is divided into segments and stored in consecutive blocks.

Not all the I/O categories support all of these attributes. Depending on what category you are using, z/OS XL C/C++ ignores or simulates attributes that do not apply.

Fixed-format records

These are the formats you can specify for RECFM if you want to use a fixed-format file:

F

Fixed-length, unblocked

FA

Fixed-length, ASA print-control characters

FB

Fixed-length, blocked

FM

Fixed-length, machine print-control codes

FS

Fixed-length, unblocked, standard

FBA

Fixed-length, blocked, ASA print-control characters

FBM

Fixed-length, blocked, machine print-control codes

FBS

Fixed-length, blocked, standard

FSA

Fixed-length, unblocked, standard, ASA print-control characters

FSM

Fixed-length, unblocked, standard, machine print-control codes

FBSM

Fixed-length, blocked, standard, machine print-control codes

FBSA

Fixed-length, blocked, standard, ASA print-control characters.

In general, all references to files with record format FB also refer to FBM and FBA. The specific behavior of ASA files (such as FBA) is explained in [Chapter 6, “Using ASA text files,” on page 25](#).

Attention: z/OS XL C/C++ distinguishes between FB and FBS formats, because an FBS file contains no embedded short blocks (the last block may be short). FBS files give you much better performance. The use of standard (S) blocks optimizes the sequential processing of a file on a direct-access device. With a standard format file, the file pointer can be directly repositioned by calculating the exact position in that file of a given record rather than reading through the entire file.

If the records are FB, some blocks may contain fewer records than others, as shown in [Figure 1 on page 13](#).

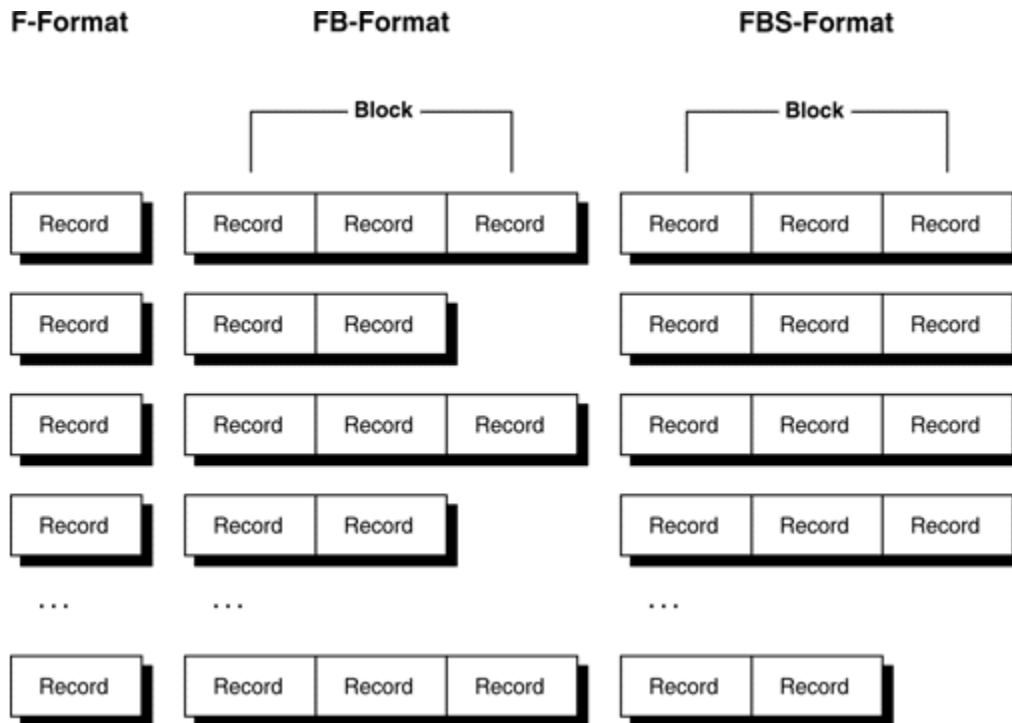


Figure 1. Blocking fixed-length records

Mapping C types to fixed format

This section describes the following formats:

- Binary
- Text (non-ASA)
- Text (ASA)
- Record
- Blocked

Binary

On binary input and output, data flows over record boundaries. Because all fixed-format records must be full, z/OS XL C/C++ completes any incomplete output record by padding it with nulls (' \0 ') when you close the file. Incomplete *blocks* are not padded. On input, nulls are visible and are treated as data.

For example, if record length is set to 10 and you are writing 25 characters of data, z/OS XL C/C++ will write two full records, each containing 10 characters, and then an incomplete record containing 5 characters. If you then close the file, z/OS XL C/C++ will complete the last record with 5 nulls. If you open the file for reading, z/OS XL C/C++ will read the records in order. z/OS XL C/C++ will not strip off the nulls at the end of the last record.

Text (non-ASA)

When writing in a text stream, you indicate the end of the data for a record by writing a new-line (' \n ') or carriage return (' \r ') to the stream. In a fixed-format file, the new-line or carriage return will not appear in the external file, and the record will be padded with blanks from the position of the new-line or carriage return to LRECL. (A carriage return is considered the same as a new-line because the ' \r ' is not written to the file.)

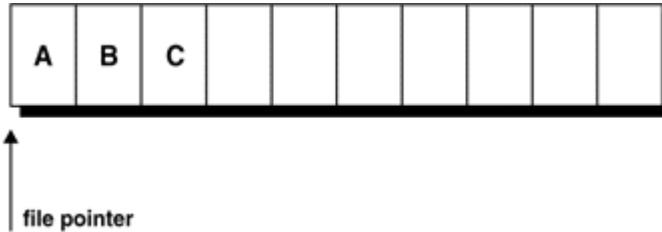
For example, if you have set LRECL to 10, and you write the string "ABC\n" to a fixed-format text file, z/OS XL C/C++ will write this to the physical file:



A record containing only a new-line is written to the file as LRECL blanks.

When reading in a text stream, the I/O functions place a new-line character ('\n') in the buffer to indicate the end of data for the record. In a fixed-format file, the new-line character is placed at the start of the blank padding at the end of the data.

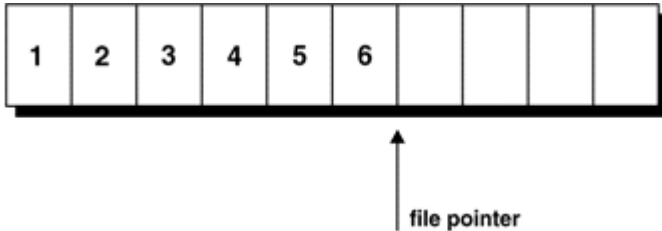
For example, if your file position points to the start of the following record in a fixed-format file opened as a text stream



and you call `fgets()` to read the line of text, `fgets()` places the string "ABC\n" in your input buffer.

Attention: Any blanks written immediately before a new-line or carriage return will be considered blank padding when the record is read back from the file. You cannot change the padding character.

When you are updating a fixed-format file opened as a text stream, you can update the amount of data in a record. The maximum length of the updated data is LRECL bytes plus the new-line character; the minimum length is zero data bytes plus the new-line character. Writing new data into an existing record replaces the old data. If the new data is longer or shorter than the old data, the number of blank padding characters in the record in the external file is changed. When you extend a record, thereby writing over the old new-line, there will be a new-line character implied after the new characters. For instance, if you were to overwrite the record mentioned in the previous example with the string "123456", the records in the physical file would then look like this:



The blanks at the end of the record imply a new-line at position 7. You can see this new-line by calling `fflush()` and then performing a read. The implied new-line is the first character returned from this read.

A fixed record can hold only LRECL characters. If you try to write more than that, z/OS XL C/C++ truncates the data unless you are using a standard stream or a terminal file. In this case, the output is split across multiple records. If truncation occurs, z/OS XL C/C++ raises SIGIOERR and sets both `errno` and the error flag.

Text (ASA)

For ASA files, the first character of each record is reserved for the ASA control character that represents a new-line, a carriage return, or a form feed. This control character represents what should happen before the record is written.

Table 4. C control to ASA characters		
C Control Character	ASA Character	Description
\n	' '	skip one line
\n\n	'0'	skip two lines

Table 4. C control to ASA characters (continued)		
C Control Character	ASA Character	Description
\n\n\n	' - '	skip three lines
\f	' 1 '	new page
\x	' + '	overstrike

A control character that ends a logical record is represented at the beginning of the following record in the external file. Since the ASA control character is in the first byte of each record, a record can hold only LRECL - 1 bytes of data. As with non-ASA text files described above, z/OS XL C/C++ adds blank padding to complete any record shorter than LRECL - 1 when it writes the record to the file. On input, z/OS XL C/C++ removes all trailing blanks. For example, if LRECL is 10, and you enter the string:

```
\nABC\nDEF
```

the record in the physical file will look like this:



On input, this string is read as follows:

```
\nABC\nDEF
```

You can lengthen and shorten records the same way as you can for non-ASA files. For more information about ASA, refer to [Chapter 6, “Using ASA text files,” on page 25](#).

Record

As with fixed-format text files, a record can hold LRECL bytes. Every call to `fwrite()` is considered to be writing a full record. If you write fewer than LRECL bytes, z/OS XL C/C++ completes the record with enough nulls to make it LRECL bytes long. If you try to write more than that, z/OS XL C/C++ truncates the data.

Blocked

Each call to `fwrite()` creates a block that must be shorter than or equal to the size established by `BLKSIZE`. When you write less than `BLKSIZE` bytes, if the request is to create a new block, a short block will be created; if it is to update an existing block, only requested part of the block will be updated. If you try to write more than `BLKSIZE` at one time, z/OS XL C/C++ truncates the data. z/OS XL C/C++ will not check the provided data. You might need to take the following cases into consideration:

- Because all fixed-format records must be full, any block you write must be multiple of a record. Otherwise, z/OS XL C/C++ will fail the write request.
- When updating an FBS short block at the end of a file, it could be updated to a full block or a longer short block.
- When writing or appending to an FBS short block, z/OS XL C/C++ will use the request buffer to replace the previous block, which might extend or shrink the short block.
- You must make sure that there is no short block in the middle of the FBS data set.

Variable-format records

In a file with variable-length records, each record may be a different length. The variable length formats permit both variable-length records and variable-length blocks. The first 4 bytes of each block are reserved for the Block Descriptor Word (BDW); the first 4 bytes of each record are reserved for the Record Descriptor Word (RDW), or, if you are using spanned files, the Segment Descriptor Word (SDW). Illustrations of variable-length records are shown in [Figure 2 on page 17](#).

Once you have set the LRECL for a variable-format file, you can write up to LRECL minus 4 characters in each record. z/OS XL C/C++ does not let you see RDWs, BDWs, or SDWs when you open a file as variable-format. To see the RDWs or SDWs and BDWs, open the variable file as undefined-format, as described in [“Undefined-format records”](#) on page 18.

The value of LRECL must be greater than 4 to accommodate the RDW or SDW. The value of BLKSIZE must be greater than or equal to the value of LRECL plus 4. You should not use a BLKSIZE greater than LRECL plus 4 for an unblocked data set. Doing so results in buffers that are larger than they need to be. The largest amount of data that any one record can hold is LRECL bytes minus 4.

For striped data sets, a block is padded out to its full BLKSIZE. This makes specifying an unnecessarily large BLKSIZE very inefficient.

Record format (RECFM)

You can specify the following formats for variable-length records:

V

Variable-length, unblocked

VA

Variable-length, ASA print control characters, unblocked

VB

Variable-length, blocked

VM

Variable-length, machine print-control codes, unblocked

VS

Variable-length, unblocked, spanned

VBA

Variable-length, blocked, ASA print control characters

VBM

Variable-length, blocked, machine print-control codes

VBS

Variable-length, blocked, spanned

VSA

Variable-length, spanned, ASA print control characters

VSM

Variable-length, spanned, machine print-control codes

VBSA

Variable-length, blocked, spanned, ASA print control characters

VBSM

Variable-length, blocked, spanned, machine print-control codes

Note: In general, all references in this guide to files with record format VB also refer to VBM and VBA. The specific behavior of ASA files (such as VBA) is explained in [Chapter 6, “Using ASA text files,”](#) on page 25.

V-format signifies unblocked variable-length records. Each record is treated as a block containing only one record.

VB-format signifies blocked variable-length records. Each block contains as many complete records as it can accommodate.

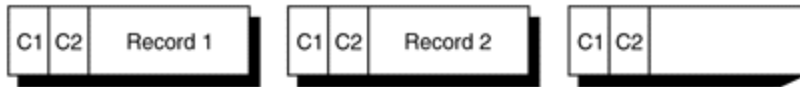
Spanned records

A spanned record is opened using both V and S in the format specifier. A spanned record is a variable-length record in which the length of the record can exceed the size of a block. If it does, the record is divided into segments and accommodated in two or more consecutive blocks. The use of spanned records allows you to select a block size, independent of record length, that will combine optimum use of auxiliary storage with the maximum efficiency of transmission.

VS-format specifies that each block contains only one record or segment of a record. The first 4 bytes of a block describe the block control information. The second 4 bytes contain record or segment control information, including an indication of whether the record is complete or is a first, intermediate, or last segment.

VBS-format differs from VS-format in that each block in VBS-format contains as many complete records or segments as it can accommodate, while each block in VS-format contains at most one record per block.

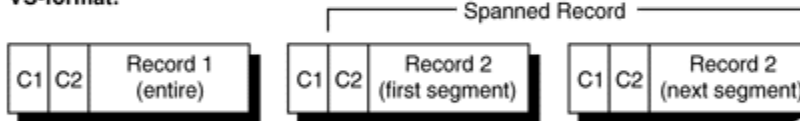
V-format:



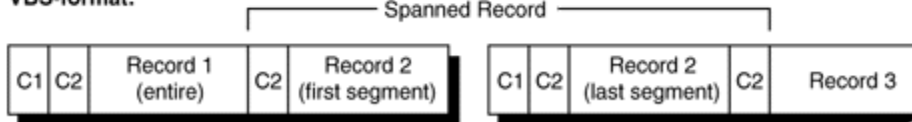
VB-format:



VS-format:



VBS-format:



C1: Block control information
C2: Record or segment control information

Figure 2. Variable-length records on z/OS

Mapping C types to variable format

Binary

On input and output, data flows over record boundaries. Any record will hold up to LRECL minus 4 characters of data. If you try to write more than that, your data will go to the next record, after the RDW or SDW. You will not be able to see the descriptor words when you read the file.

Note: If you need to see the BDWs, RDWs, or SDWs, you can open and read a V-format file as a U-format file. See [“Undefined-format records”](#) on page 18 for more information.

z/OS XL C/C++ never creates empty binary records for files opened in V-format. See [“Writing to binary files”](#) on page 58 for more information. An empty binary record is one that contains only an RDW, which is 4 bytes long. On input, empty records are ignored.

Text (non-ASA)

Record boundaries are used in the physical file to represent the position of the new-line character. You can indicate the end of a record by including a new-line or carriage return character in your data. In variable-format files, z/OS XL C/C++ treats the carriage return character as if it were a new-line. z/OS XL C/C++ does not write either of these characters to the physical file; instead, it creates a record boundary. When you read the file back, boundaries are read as new-lines.

If a record only contains a new-line character, the default behavior of z/OS XL C/C++ is to write a record containing a single blank to the file. Therefore, the string " \n" is treated the same way as the string "\n"; both are read back as "\n". All other blanks in your output are read back as is. Any empty (zero-length) record is ignored on input. However, if the environment variable `_EDC_ZERO_RECLLEN` was set to Y at the time the file was opened, a single new-line is written to the file as an empty record, and a single blank represents " \n". On input, an empty record is treated as a single new-line and is not ignored.

After a record has been written to a file, you cannot change its length. If you try to shorten a logical record by writing a new, smaller amount of data into it, the C I/O library will add blank characters until the record is full. Writing more data to a record than it can hold causes your data to be truncated unless you are writing to a standard stream or a terminal file. In this case, your output is split across multiple records. If truncation occurs, z/OS XL C/C++ raises SIGIOERR and sets both `errno` and the error flag.

Note: If you did not explicitly set the `_EDC_ZERO_RECLLEN` environment variable when you opened the file, you can update a record that contains a single blank to contain a non-blank character, thereby lengthening the logical record from ' \n ' to ' x\n ', where x is the non-blank character.

Text (ASA)

z/OS XL C/C++ treats variable-format ASA text files similarly to the way it treats fixed-format ones. Empty records are always ignored in ASA variable-format files; for a record to be recognized, it must contain at least one character as the ASA control character. For more information about ASA, refer to Chapter 6, "Using ASA text files," on page 25.

Record

Each call to `fwrite()` creates a record that must be shorter than or equal to the size established by `LRECL`. If you try to write more than `LRECL` bytes on one call to `fwrite()`, z/OS XL C/C++ will truncate your data. z/OS XL C/C++ never creates empty records using record I/O. On input, empty records are ignored unless you have set the `_EDC_ZERO_RECLLEN` environment variable to Y. In this case, empty records are treated as records with length 0.

If your application sets `_EDC_ZERO_RECLLEN` to Y, bear in mind that `fread()` returns back 0 bytes read, but does not set `errno`, and that both `feof()` and `ferror()` return 0 as well.

Blocked

Each call to `fwrite()` creates a block that must be shorter than or equal to the size established by `BLKSIZE`. When you write less than `BLKSIZE` bytes, if the request is to create a new block, a short block will be created; if it is to update an existing block, only requested part of the block will be updated. If you try to write more than `BLKSIZE` at one time, z/OS XL C/C++ truncates the data. z/OS XL C/C++ will not check the provided data. You must make sure that BDWs, RDWs, and SDWs in a block are correct.

Undefined-format records

Everything in an undefined-format file is treated as data, including control characters and record boundaries. Blocks in undefined-format records are variable-length; each block is considered a record.

It is impossible to have an empty record. Whatever you specify for `LRECL` has no effect on your data, but the value of `LRECL` must be less than or equal to the value you specify for `BLKSIZE`. Regardless of what you specify, z/OS XL C/C++ sets `LRECL` to zero when it creates an undefined-format file.

Reading a file in U-format enables you to read an entire block at once.

Record format (RECFM)

You can specify the following formats for undefined-length records:

U

Undefined-length

UA

Undefined-length, ASA print control characters

UM

Undefined-length, machine print-control codes

U, UA, and UM formats permit the processing of records that do not conform to F- and V-formats. The operating system treats each block as a record; your program must perform any additional blocking or deblocking.

You can read any file in U-format. This is useful if, for example, you want to see the BDWs and RDWs of a file that you have written in V-format.

Mapping C types to undefined format

Binary

When you are writing to an undefined-format file, binary data fills a block and then begins a new block.

Text (non-ASA)

Record boundaries (that is, block boundaries) are used in the physical file to represent the position of the new-line character. You can indicate the end of a record by including a new-line or carriage return character in your data. In undefined-format files, z/OS XL C/C++ treats the carriage return character as if it were a new-line. z/OS XL C/C++ does not write either of these characters to the physical file; instead, it creates a record boundary. When you read the file back, these boundaries are read as new-lines.

If a record contains only a new-line character, z/OS XL C/C++ writes a record containing a single blank to the file regardless of the setting of the `_EDC_ZERO_RECLEN` environment variable. Therefore, the string `' \n '` (a single blank followed by a new-line character) is treated the same way as `' \n '`; both are written out as a single blank. On input, both are read as `' \n '`. All other blank characters are written and read as you intended.

After a record has been written to a file, you cannot change its length. If you try to shorten a logical record by writing a new, smaller amount of data into it, the C I/O library adds blank characters until the record is full. Writing more data to a record than it can hold will cause your data to be truncated unless you are writing to a standard stream or a terminal file. In these cases, your output is split across multiple records. If truncation occurs, z/OS XL C/C++ raises SIGIOERR and sets both `errno` and the error flag.

Note: You can update a record that contains a single blank to contain a non-blank character, thereby lengthening the logical record from `' \n '` to `' x\n '`, where `x` is the non-blank character.

Text (ASA)

For a record to be recognized, it must contain at least one character as the ASA control character. For more information about ASA, refer to [Chapter 6, “Using ASA text files,” on page 25](#).

Record

Each call to `fwrite()` creates a record that must be shorter than or equal to the size established by `BLKSIZE`. If you try to write more than `BLKSIZE` bytes on one call to `fwrite()`, z/OS XL C/C++ truncates your data.

Blocked

Each call to `fwrite()` creates a block that must be shorter than or equal to the size established by `BLKSIZE`. When you write less than `BLKSIZE` bytes, if the request is to create a new block, a short block will be created; if it is to update an existing block, only requested part of the block will be updated. If you try to write more than `BLKSIZE` at one time, z/OS XL C/C++ truncates the data. z/OS XL C/C++ will not check the provided data.

The byte stream model for C I/O

The byte stream model differs from the record I/O model. In the byte stream model, a file is just a stream of bytes, with no record boundaries. New-line characters written to the stream appear in the external file.

If the file is opened in binary mode, any new-line characters previously written to the file are visible on input. z/OS XL C/C++ memory file I/O and Hiperspace memory file I/O are based on the byte stream

model (see [Chapter 12, “Performing memory file and hiperspace I/O operations,”](#) on page 141 for more information).

UNIX file system I/O, defined by POSIX, is also based on the byte stream model. See [Chapter 9, “Performing z/OS UNIX file system I/O operations,”](#) on page 75 for information about I/O with UNIX file system.

Mapping the C types of I/O to the byte stream model

Binary

In the byte stream model, files opened in binary mode do not contain any record boundaries. Data is written as is to the file.

Text

The byte stream model does not support ASA. New-lines, carriage returns, and other control characters are written as-is to the file.

Record

If record I/O is supported by the kind of file you are using, z/OS XL C/C++ simulates it by treating new-line characters as record boundaries. New-lines are not treated as part of the record. A record written out with a new-line inside it is not read back as it was written, because z/OS XL C/C++ treats the new-line as a record boundary instead of data.

Files in z/OS UNIX file system support record I/O, but memory files do not.

As with all other record I/O, you can use only `fread()` and `fwrite()` to read from and write to files. Each call to `fwrite()` inserts a new-line in the byte stream; each call to `fread()` strips it off. For example, if you use one `fwrite()` statement to write the string ABC and the next to write DEF, the byte stream will look like this:

A	B	C	\n	D	E	F	\n		...
---	---	---	----	---	---	---	----	--	-----

There are no limitations on lengthening and shortening records. If you then rewind the file and write new data into it, z/OS XL C/C++ will replace the old data. For example, if you used the `rewind()` function on the stream in the previous example and then called `fwrite()` to place the string 12345 into it, the stream would look like this:

1	2	3	4	5	\n	F	\n		...
---	---	---	---	---	----	---	----	--	-----

If you are using files with this model, do not use new-line characters in your output. If you do, they will create extra record boundaries. If you are unsure about the data being written or are writing numeric data, use binary instead of text to avoid writing a byte that has the hex value of a new-line.

Chapter 4. Using the Standard C++ Library I/O Stream Classes

The object-oriented model for input and output (I/O) is a set of classes and header files that are provided by the Standard C++ Library. These classes implement and manage the stream buffers and the data held in the buffers. Stream buffers hold data sent to the program (input) and from the program (output), enabling the program to manipulate and format the data.

There are two base classes, `ios` and `streambuf`, from which all other I/O stream classes are derived. The `ios` class and its derivative classes are used to implement formatting of I/O and maintain error state information of stream buffers implemented with the `streambuf` class.

There are two shipped versions of the I/O stream classes:

- The UNIX Systems Laboratories C++ Language System Release (USL) I/O Stream Class Library

The UNIX Systems Laboratories C++ Language System Release (USL) I/O Stream Class Library is declared in the `iostream.h` header file. This version does not support ASCII and large files.

- The Standard C++ I/O stream classes

The Standard C++ I/O stream classes are declared in the `iostream` header file. This version supports ASCII and large files. For more detailed information on the I/O stream classes provided by the Standard C++ Library, see [`_LARGE_FILES` in *z/OS XL C/C++ Language Reference*](#).

The I/O stream classes use `OBJECTMODEL(CLASSIC)`. They cannot be used with other classes that use `OBJECTMODEL(IBM)`, within the same inheritance hierarchy. For more information, see [`OBJECTMODEL \(C++ only\)` in *z/OS XL C/C++ User's Guide*](#).

Advantages to using the C++ I/O stream classes

Although input and output are implemented with streams for both C and C++, the C++ I/O stream classes provide the same facilities for input and output as C `stdio.h`. The I/O stream classes in the Standard C++ Library have the following advantages:

- The input (`>>`) operator and output (`<<`) operator are typesafe. These operators are easier to use than `scanf()` and `printf()`.
- You can overload the input and output operators to define input and output for your own types and classes. This makes input and output across types, including your own, uniform.

Predefined streams for C++

z/OS XL C++ provides the following predefined streams:

cin

The standard input stream

cout

The standard output stream

cerr

The standard error stream, unit-buffered such that characters sent to this stream are flushed on each output operation

clog

The buffered error stream

All predefined streams are tied to `cout`. When you use `cin`, `cerr`, or `clog`, `cout` gets flushed sending the contents of `cout` to the ultimate consumer.

z/OS C standard streams create all I/O to I/O streams:

- Input to `cin` comes from `stdin` (unless `cin` is redirected)
- `cout` output goes to `stdout` (unless `cout` is redirected)
- `cerr` output goes to `stderr` (unit-buffered) (unless `cerr` is redirected)
- `clog` output goes to `stderr` (unless `clog` is redirected)

When redirecting or intercepting a C standard stream, the corresponding C++ standard stream becomes redirected. This applies unless you redirect an I/O stream.

How C++ I/O streams relate to C I/O streams

Typically, USL I/O Stream Class Library file I/O is implemented in terms of z/OS XL C file I/O, and is buffered from it.

Note: The only exception is that `cerr` is unit-buffered (that is, `ios::unitbuf` is set).

A `filebuf` object is associated with each `ifstream`, `ofstream`, and `fstream` object. When the `filebuf` is flushed, it writes to the underlying C stream, which has its own buffer. The `filebuf` object follows every `fwrite()` to the underlying C stream with an `fflush()`.

Mixing the Standard C++ I/O stream classes, USL I/O stream class library, and C I/O library functions

It is not recommended to mix the usage of the Standard C++ I/O stream classes, USL I/O Stream Class Library, and C I/O library functions.

The USL I/O stream class library uses a separate buffer, which means that you would need to flush the buffer after each call to `cout` either by setting `ios::unitbuf` or by calling `sync_with_stdio()`.

You should avoid switching between the formatted extraction functions of the C++ I/O stream classes and C `stdio` library functions whenever possible. You should also avoid switching between versions of these classes.

Specifying file attributes

The `fstream`, `ifstream`, and `ofstream` classes specialize stream input and output for files.

For z/OS XL C++, overloaded `fstream`, `ifstream`, and `ofstream` constructors, and `open()` member functions, with an additional parameter, are provided so you can specify z/OS XL C `fopen()` mode values. You can use this additional parameter to specify any z/OS XL C `fopen()` mode value except `type=record` or `type=blocked`. If you choose to use a constructor without this additional parameter, you will get the default z/OS XL C `fopen()` file characteristics.

Chapter 5. Buffering of C streams

This chapter describes buffering modes used by z/OS XL C/C++ library functions available to control buffering and methods of flushing buffers.

z/OS XL C/C++ uses buffers to map C I/O to system-level I/O. When z/OS XL C/C++ performs I/O operations, it uses one of the following buffering modes:

- *Line buffering* - characters are transmitted to the system as a block when a new-line character is encountered. Line buffering is meaningful only for text streams and UNIX file system files.
- *Full buffering* - characters are transmitted to the system as a block when a buffer is filled.
- *No buffering* - characters are transmitted to the system as they are written. Only regular memory files and UNIX file system files support the no buffering mode.

The buffer mode affects the way the buffer is flushed. You can use the `setvbuf()` and `setbuf()` library functions to control buffering, but you cannot change the buffering mode after an I/O operation has used the buffer, as all read, write, and reposition operations do. In some circumstances, repositioning alters the contents of the buffer. It is strongly recommended that you only use `setbuf()` and `setvbuf()` before *any* I/O, to conform with ISO C/C++, and to avoid any dependency on the current implementation. If you use `setvbuf()`, z/OS XL C/C++ may or may not accept your buffer for its internal use. For a hyperspace memory file, if the size of the buffer specified to `setvbuf()` is 8K or more, it will affect the number of hyperspace blocks read or written on each call to the operating system; the size is rounded down to the nearest multiple of 4K.

Full buffering is the default except in the following cases:

- If you are using an interactive terminal, z/OS XL C/C++ uses line buffering.
- If you are running under CICS, z/OS XL C/C++ also uses line buffering.
- `stderr` is line-buffered by default.
- If you are using a memory file, z/OS XL C/C++ does not use any buffering.

For terminals, because I/O is always unblocked, line buffering is equivalent to full buffering.

For record I/O files, buffering is meaningful only for blocked files or for record I/O files in z/OS UNIX file system using full buffering. For unblocked files, the buffer is full after every write and is therefore written immediately, leaving nothing to flush. For blocked files or fully-buffered UNIX file system files, however, the buffer can contain one or more records that have not been flushed and that require a flush operation for them to go to the system.

For blocked I/O files, buffering is always meaningless.

You can flush buffers to the system in several different ways.

- If you are using full buffering, z/OS XL C/C++ automatically flushes a buffer when it is filled.
- If you are using line buffering for a text file or a UNIX file system file, z/OS XL C/C++ flushes a buffer when you complete it with a control character. Except for UNIX file system files, specifying line buffering for a record I/O, blocked I/O, or binary file has no effect; z/OS XL C/C++ treats the file as if you had specified full buffering.
- z/OS XL C/C++ flushes buffers to the system when you close a file or end a program.
- z/OS XL C/C++ flushes buffers to the system when you call the `fflush()` library function, with the following restrictions:
 - A file opened in text mode does not flush data if a record has not been completed with a new-line.
 - A file opened in fixed format does not flush incomplete records to the file.
 - An FBS file does not flush out a short block unless it is a DISK file opened without the `NOSSEEK` parameter.

- All streams are flushed across non-POSIX `system()` calls. Streams are *not* flushed across POSIX `system()` calls. For a POSIX system call, we recommend that you do a `fflush()` before the `system()` call.

Note: This is not supported for VSAM files.

You may not see output if a program that is using input and output fails, and the error handling routines cannot close all the open files.

Chapter 6. Using ASA text files

This chapter describes the American Standards Association (ASA) text files, the control characters used in ASA files, how z/OS XL C/C++ translates the control characters, and how z/OS XL C/C++ treats ASA files during input and output. The first column of each record in an ASA file contains a control character (' ', '0', '-', '1', or '+') when it appears in the external medium.

z/OS XL C/C++ translates control characters in ASA files opened for text processing (r, w, a, r+, w+, a+ functions). On input, z/OS XL C/C++ translates ASA characters to sequences of control characters, as shown in Table 5 on page 25. On output, z/OS XL C/C++ performs the reverse translation. The following sequences of control characters are translated, and the resultant ASA character becomes the first character of the following record.

Table 5. C control to ASA characters translation table		
C Control Character Sequence	ASA Character	Description
\n	' '	skip one line
\n\n	'0'	skip two lines
\n\n\n	'-'	skip three lines
\f	'1'	new page
\r	'+'	overstrike

If you are writing to the first record or byte of the file and the output data does not start with a translatable sequence of C control characters, the ' ' ASA control character is written to the file before the specified data.

z/OS XL C/C++ does not translate or verify control characters when you open an ASA file for binary, record I/O, or blocked I/O.

Example of writing to an ASA file

Sample program CCNGAS1, shown in Figure 3 on page 25, demonstrates how a program can write to an ASA file.

```
/* this example shows how to write to an ASA file */

#include <stdio.h>
#define MAX_LEN 80

int main(void) {
    FILE *fp;
    int i;
    char s[MAX_LEN+1];
    fp = fopen("asa.file", "w, recfm=fba");
    if (fp != NULL) {
        fputs("\n\nabcdef\x345\n\n", fp);
        fputs("\n\n9034\n", fp);
        fclose(fp);

        return(0);
    }

    fp = fopen("asa.file", "r");
    for (i = 0; i < 5; i++) {
        fscanf(fp, "%s", s[0]);
        printf("string = %s\n", s);
    }
}
```

Figure 3. ASA Example

The program writes five records to the file `asa.file`, as follows. Note that the last record is 9034. The last single `'\n'` does not create a record with a single control character (`' '`). If this same file is opened for read, and the `getc()` function is called to read the file 1 byte at a time, the same characters as those that were written out by `fputs()` in the first program are read.

```
0abcdef
1
+345
-
9034
```

ASA file control

ASA files are treated as follows:

- If the first record written does not begin with a control character, then a single new-line is written and then followed by data; that is, the ASA character defaults to a space when none is specified.
- In ASA files, control characters are treated the same way that they are treated in other text files, with the following exceptions (see [Table 5 on page 25](#) for more information):

'\f' – form feed

Defines a record boundary and determines the ASA character of the following record

'\n' – new-line

Does either of these:

- Define a record boundary and determines the ASA character of the following record.
- Modify the preceding ASA character if the current position is directly after an ASA character of `' '` or `'0'`.

'\r' – carriage return

Defines a record boundary and determines the ASA character of the following record.

- Records are terminated by writing a new-line (`'\n'`), carriage return (`'\r'`), or form feed (`'\f'`) character.
- An ASA character can be updated to any other ASA character.

Updates made to any of the C control characters that make up an ASA character cause the ASA character to change.

If the file is positioned directly after a `' '` or `'0'` ASA character, writing a `'\n'` character changes the ASA character to a `'0'` or `'-'` respectively. However, if the ASA character is a `'-'`, `'1'` or `'+'`, the `'\n'` truncates the record (that is, it adds blank padding to the end of the record), and causes the following record's ASA character to be written as a `' '`. Writing a `'\f'` or `'\r'` terminates the record and start a new one, but writing a normal data character simply overwrites the first data character of the record.

- You cannot overwrite the ASA character with a normal data character. The position at the start of a record (at the ASA character) is the logical end of the previous record. If you write normal data there, you are writing to the end of the previous record. z/OS XL C/C++ truncates data for the following files, except when they are standard streams:

- Variable-format files
- Undefined-format files
- Fixed-format files in which the previous record is full of data

When truncation occurs, z/OS XL C/C++ raises `SIGIOERR` and sets both `errno` and the error flag.

- Even when you update an ASA control character, seeking to a previously recorded position still succeeds. If the recorded position was at a control character that no longer exists (because of an update), the reposition is to the next character. Often, this is the first data character of the record.

For example, if you have the following string, you have saved the position of the third new-line:

`\n\n\nHELLO WORLD`

↑

`x = ftell()`

If you then update the ASA character to a form feed (`'\f'`), the logical ASA position `x` no longer exists:

`\fHELLO WORLD`

If you call `fseek()` with the logical position `x`, it repositions to the next valid character, which is the letter 'H':

`\fHELLO WORLD`

↑

`fseek() to pos x`

- If you try to shorten a record when you are updating it, z/OS XL C/C++ adds enough blank padding to fill the record.
- The ASA character can represent up to three new-lines, which can increase the logical record length by 1 or 2 bytes.
- Extending a fixed logical record on update implies that the logical end of the line follows the last written non-blank character.
- If an undefined text record is updated, the length of the physical records does not change. If the replacement record is:
 - *Longer* - data characters beyond the record boundary are truncated. At the point of truncation, the User error flag is set and SIGIOERR is raised (if the signal is not set up to be ignored). Truncation continues until you do one of these:
 1. Write a new-line character, carriage return, or form feed to complete the current record
 2. Close the file explicitly or implicitly at termination
 3. Reposition to another position in the file.
 - *Shorter* - the blank character is used to overwrite the rest of the record.
- If you close an ASA file that has a new-line as its last character, z/OS XL C/C++ does not write the new-line to the physical file. The next time you read from the file or update it, z/OS XL C/C++ returns the new-line to the end of the file. An exception to this rule happens when you write only a new-line to a new file. In this case, z/OS XL C/C++ does not truncate the new-line; it writes a single blank to the file. On input, however, you will read two new-lines.
- Using ASA format to read a file that contains zero-length records results in undefined behavior.
- You may have trouble updating a file if two ASA characters are next to each other in the file. For example, if there is a single-byte record (containing only an ASA character) immediately followed by the ASA character of the next record, you are positioned at or within the first ASA character. If you then write a sequence of `'\n'` characters intended to update both ASA characters, the `'\n'` s will be absorbed by the first ASA character before overflowing to the next record. This absorption may affect the crossing of record boundaries and cause truncation or corruption of data.

At least one normal intervening data character (for example, a space) is required between `'\n'` and `'\n'` to differentiate record boundaries.

Note: Be careful when you update an ASA file with data containing more than one consecutive new-line: the result of the update depends on how the original ASA records were structured.

- If you are writing data to a non-blocked file without intervening flush or reposition requests, each record is written to the system on completion (that is, when a `'\n'`, `'\r'` or `'\f'` character is written or when the file is closed).

If you are writing data to a blocked file without intervening flush or reposition requests, and the file is opened in full buffering mode, the block is written to the system on completion of the record that fills the block. If the blocked file is line buffered, each record is written to the system on completion.

If you are writing data to a spanned file without intervening flush or reposition requests, and the record spans multiple blocks, each block is written to the system once it is full and the user writes an additional byte of data.

- If a flush occurs while an ASA character indicating more than one new-line is being updated, the remaining new-lines will be discarded and a read will continue at the first data character. For example, if '\n\n\n' is updated to be '\n\n' and a flush occurs, then a '0' will be written out in the ASA character position.

Chapter 7. z/OS XL C support for the double-byte character set

The number of characters in some languages such as Japanese or Korean is larger than 256, the number of distinct values that can be encoded in a single byte. The characters in such languages are represented in computers by a sequence of bytes, and are called multibyte characters. This chapter explains how the z/OS XL C compiler supports multibyte characters.

Note: The z/OS XL C++ compiler does not have native support for multibyte characters. The support described here is what z/OS XL C provides; for C++, you can take advantage of this support by using interlanguage calls to C code. Please refer to [Chapter 17, “Using linkage specifications in C or C++,”](#) on [page 175](#) for more information.

The z/OS XL C compiler supports the IBM EBCDIC encoding of multibyte characters, in which each natural language character is uniquely represented by one to four bytes. The number of bytes that encode a single character depends on the *global shift state information*. If a stream is in initial shift state, one multibyte character is represented by a byte or sequence of bytes that has the following characteristics:

- It starts with the byte containing the shift-out (0x0e) character.
- The shift-out character is followed by 2 bytes that encode the value of the character.
- These bytes may be followed by a byte containing the shift-in (0x0f) character.

If the sequence of bytes ends with the shift-in character, the state remains initial, making this sequence represent a 4-byte multibyte character. Multibyte characters of various lengths can be normalized by the set of z/OS XL C library functions and encoded in units of one length. Such normalized characters are called wide characters; in z/OS XL C they are represented by two bytes. Conversions between multibyte format and wide character format can be performed by string conversion functions such as `wcstombs()`, `mbstowcs()`, `wcsrtombs()`, and `mbsrtowcs()`, as well by the family of the wide character I/O functions. `MB_CUR_MAX` is defined in the `stdlib.h` header file. Depending on its value, either of the following happens:

- When `MB_CUR_MAX` is 1, all bytes are considered single-byte characters; shift-out and shift-in characters are treated as data as well.
- When `MB_CUR_MAX` is 4:
 - On input, the wide character I/O functions read the multibyte character from the streams, and convert them to the wide characters.
 - On output, they convert wide characters to multibyte characters and write them to the output streams.

Both binary and text streams have *orientation*. Streams opened with `type=record` or `type=blocked` do not. There are three possible orientations of a stream:

Non-oriented

A stream that has been associated with an open file before any I/O operation is performed. The first I/O operation on a non-oriented stream will set the orientation of the stream. The `fwide()` function may be used to set the orientation of a stream before any I/O operation is performed. You can use the `setbuf()` and `setvbuf()` functions only when I/O has not yet been performed on a stream. When you use these functions, the orientation of the stream is not affected. When you perform one of the wide character input/output operations on a non-oriented stream, the stream becomes *wide-oriented*. When you perform one of the byte input/output operations on a non-oriented stream, the stream becomes *byte-oriented*.

Wide-oriented

A stream on which any wide character input/output functions are guaranteed to operate correctly. Conceptually, wide-oriented streams are sequences of wide characters. The external file associated with a wide-oriented stream is a sequence of *multibyte* characters. Using byte I/O functions on a

wide-oriented stream results in undefined behavior. A stream opened for record I/O or blocked I/O cannot be wide-oriented.

Byte-oriented

A stream on which any byte input/output functions are guaranteed to operate properly. Using wide character I/O functions on a byte input/output stream results in undefined behavior. Byte-oriented streams have minimal support for multibyte characters.

Calls to the `clearerr()`, `feof()`, `ferror()`, `fflush()`, `fgetpos()`, or `ftell()` functions do not change the orientation. Other functions that do not change the orientation are `ftello()`, `fsetpos()`, `fseek()`, `fseeko()`, `rewind()`, `fldata()`, and `fileno()`. Also, the `perror()` function does not affect the orientation of the `stderr` stream.

Once you have established a stream's orientation, the only way to change it is to make a successful call to the `freopen()` function, which removes a stream's orientation.

The `wchar.h` header file declares the `WEOF` macro and the functions that support wide character input and output. The macro expands to a constant expression of type `wint_t`. Certain functions return `WEOF` type when the end-of-file is reached on the stream.

Note: The behavior of the wide character I/O functions is affected by the `LC_CTYPE` category of the current locale, and the setting of `MB_CUR_MAX`. Wide-character input and output should be performed under the same `LC_CTYPE` setting. If you change the setting between when you read from a file and when you write to it, or vice versa, you may get undefined behavior. If you change it back to the original setting, however, you will get the behavior that is documented. See the introduction of this chapter for a discussion of the effects of `MB_CUR_MAX`.

Opening files

You can use the `fopen()` or `freopen()` library functions to open I/O files that contain multibyte characters. You do not need to specify any special parameters on these functions for wide character I/O.

Reading streams and files

Wide character input functions read multibyte characters from the stream and convert them to wide characters. The conversion process is performed in the same way that the `mbrtowc()` function performs conversions. The following z/OS XL C library functions support wide character input:

- `fgetwc()`
- `fgetwc_unlocked()`
- `fgetws()`
- `fgetws_unlocked()`
- `fwscanf()`
- `fwscanf_unlocked()`
- `getwc()`
- `getwc_unlocked()`
- `getwchar()`
- `getwchar_unlocked()`
- `vfwscanf()`
- `vfwscanf_unlocked()`
- `vwscanf()`
- `vwscanf_unlocked()`
- `wscanf()`
- `wscanf_unlocked()`

In addition, the following byte-oriented functions support handling multibyte characters by providing conversion specifiers to handle the `wchar_t` data type:

- `fscanf()`
- `fscanf_unlocked()`
- `scanf()`
- `scanf_unlocked()`
- `vfscanf()`
- `vfscanf_unlocked()`
- `vscanf()`
- `vscanf_unlocked()`

All other byte-oriented input functions treat input as single-byte.

For a detailed description of unformatted and formatted I/O functions, see [z/OS C/C++ Runtime Library Reference](#).

The wide-character input/output functions maintain global shift state for multibyte character streams they read or write. For each multibyte character they read, wide-character input functions change global shift state as the `mbrtowc()` function would do. Similarly, for each multibyte character they write, wide-character output functions change global shift state as the `wcrtomb()` function would do.

When you are using wide-oriented input functions, multibyte characters are converted to wide characters according to the current shift state. Invalid double-byte character sequences cause conversion errors on input. As z/OS XL C uses wide-oriented functions to read a stream, it updates the shift state when it encounters shift-out and shift-in characters. Wide-oriented functions always read complete multibyte characters. Byte-oriented functions do not check for complete multibyte characters, nor do they maintain information about the shift state. Therefore, they should not be used to read multibyte streams.

For binary streams, no validation is performed to ensure that records start or end in initial shift state. For text streams, however, all records must start and end in initial shift state.

Writing streams and files

Wide character output functions convert wide characters to multibyte characters and write the result to the stream. The conversion process is performed in the same way that the `wcrtomb()` function performs conversions.

The following z/OS XL C functions support wide character output:

- `fputwc()`
- `fputwc_unlocked()`
- `fputws()`
- `fputws_unlocked()`
- `fwprintf()`
- `fwprintf_unlocked()`
- `putwc()`
- `putwc_unlocked()`
- `putwchar()`
- `putwchar_unlocked()`
- `vfwprintf()`
- `vfwprintf_unlocked()`
- `vwprintf()`
- `vwprintf_unlocked()`

- `wprintf()`
- `wprintf_unlocked()`

In addition, the following byte-oriented functions support handling multibyte characters by providing conversion specifiers to handle the `wchar_t` data type:

- `fprintf()`
- `fprintf_unlocked()`
- `printf()`
- `printf_unlocked()`
- `vfprintf()`
- `vfprintf_unlocked()`
- `vprintf()`
- `vprintf_unlocked()`

All other output functions do not support the `wchar_t` data type. However, all of the output functions support multibyte character output for text streams if `MB_CUR_MAX` is 4.

For a detailed description of unformatted and formatted I/O functions, see [z/OS C/C++ Runtime Library Reference](#).

Writing text streams

When you are using wide-oriented output functions, wide characters are converted to multibyte characters. For text streams, all records must start and end in initial shift state. The wide-character functions add shift-out and shift-in characters as they are needed. When the file is closed, a shift-out character may be added to complete the file in initial shift state.

When you are using byte-oriented functions to write out multibyte data, z/OS XL C starts each record in initial shift state and makes sure you complete each record in initial shift state before moving to the next record. When a string starts with a shift-out, all data written is treated as multibyte, not single-byte. This means that you cannot write a single-byte control character (such as a new-line) until you complete the multibyte string with a shift-in character.

Attempting to write a second shift-out character before a shift-in is not allowed. z/OS XL C truncates the second shift-out and raises `SIGIOERR` if `SIGIOERR` is not set to `SIG_IGN`.

When you write a shift-in character to an incomplete multibyte character, z/OS XL C completes the multibyte character with a padding character (`0xfe`) before it writes the shift-in. The padding character is not counted as an output character in the total returned by the output function; you will never get a return code indicating that you wrote more characters than you provided. If z/OS XL C adds a padding character, however, it does raise `SIGIOERR`, if `SIGIOERR` is not set to `SIG_IGN`.

Control characters written before the shift-in are treated as multibyte data and are not interpreted or validated.

When you close the file, z/OS XL C ensures that the file ends in initial shift state. This may require adding a shift-in and possibly a padding character to complete the last multibyte character, if it is not already complete. If padding is needed in this case, z/OS XL C does not raise `SIGIOERR`.

Multibyte characters are never split across record boundaries. In addition, all records end and start in initial shift state. When a shift-out is written to the file, either directly or indirectly by wide-oriented functions, z/OS XL C calculates the maximum number of complete multibyte characters that can be contained in the record with the accompanying shift-in. If multibyte output (including any required shift-out and shift-in characters) does not fit within the current record, the behavior depends on what type of file it is (a memory file has no record boundaries and so never has this particular problem). For a standard stream or terminal file, data is wrapped from one record to the next. Shift characters may be added to ensure that the first record ends in initial shift state and that the second record starts in the required shift state.

For files that are not standard streams, terminal files, or memory files, any attempt to write data that does not fit into the current record results in data truncation. In such a case, the output function returns an error code, raises SIGIOERR, and sets `errno` and the error flag. Truncation continues until initial state is reached and a new-line is written to the file. An entire multibyte stream may be truncated, including the shift-out and shift-in, if there are not at least two bytes in the record. For a wide-oriented stream, truncation stops when a `wchar_t` new-line character is written out.

Updating a wide-oriented file or a file containing multibyte characters is strongly discouraged, because your update may overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The data after the shift-out is meaningless when it is treated in initial shift state. Appending new data to the end of the file is safe.

Writing binary streams

When you are using wide-oriented output functions, wide characters are converted to multibyte characters. No validation is performed to ensure that records start or end in initial shift state. When the file is closed, any appends are completed with a shift-in character, if it is needed to end the stream in initial shift state. If you are updating a record when the stream is closed, the stream is flushed. See [“Flushing buffers” on page 33](#) for more information.

Byte-oriented output functions do not interpret binary data. If you use them for writing multibyte data, ensure that your data is correct and ends in initial shift state.

Updating a wide-oriented file or a file containing multibyte characters is strongly discouraged, because your update may overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The data after the shift-out is meaningless when it is treated in initial shift state. Appending new data to the end of the file is safe for a wide-oriented file.

If you update a record after you call `fgetpos()`, the shift state may change. Using the `fpos_t` value with the `fsetpos()` function may cause the shift state to be set incorrectly.

Flushing buffers

You can use the library function `fflush()` to flush streams to the system. For more information about `fflush()`, see the [z/OS C/C++ Runtime Library Reference](#).

The action taken by the `fflush()` library function depends on the buffering mode associated with the stream and the type of stream. If you call one z/OS XL C program from another z/OS XL C program by using the `ANSI system()` function, all open streams are flushed before control is passed to the callee. A call to the `POSIX system()` function does not flush any streams to the system. For a POSIX system call, we recommend that you do a `fflush()` before the system call.

Flushing text streams

When you call `fflush()` after updating a text stream, `fflush()` calculates your current shift state. If you are not in initial shift state, z/OS XL C looks forward in the record to see whether a shift-in character occurs before the end of the record or any shift-out. If not, z/OS XL C adds a shift-in to the data if it will not overwrite a shift-out character. The shift-in is placed such that there are complete multibyte characters between it and the shift-out that took the data out of initial state. z/OS XL C may accomplish this by skipping over the next byte in order to leave an even number of bytes between the shift-out and the added shift-in.

Updating a wide-oriented or byte-oriented multibyte stream is strongly discouraged. In a byte-oriented stream, you may have written only half of a multibyte character when you call `fflush()`. In such a case, z/OS XL C adds a padding byte before the shift-out. For both wide-oriented and byte-oriented streams, the addition of any shift or padding character does not move the current file position.

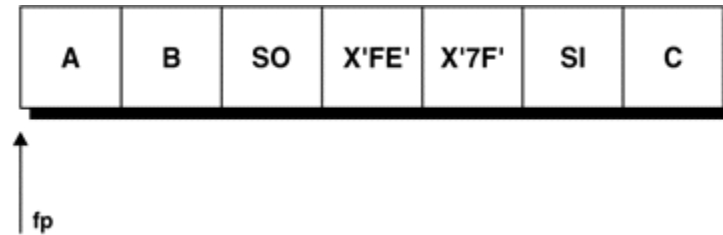
Calling `fflush()` has no effect on the current record when you are writing new data to a wide-oriented or byte-oriented multibyte stream, because the record is incomplete.

Flushing binary streams

In a wide-oriented stream, calling `fflush()` causes z/OS XL C to add a shift-in character if the stream does not already end in initial shift state. In a byte-oriented stream, calling `fflush()` causes no special behavior beyond what a call to `fflush()` usually does.

ungetwc() considerations

`ungetwc()` pushes wide characters back onto the input stream for binary and text files. You can use it to push one wide character onto the `ungetwc()` buffer. Never use `ungetc()` on a wide-oriented file. After you call `ungetwc()`, calling `fflush()` backs up the file position by one wide character and clears the pushed-back wide character from the stream. Backing up by one wide character skips over shift characters and backs up to the start of the previous character (whether single-byte or double-byte). For text files, z/OS XL C counts the new-lines added to the records as single-byte characters when it calculates the file position. For example, if you have the following stream,



you can run the code fragment shown in [Figure 4 on page 34](#).

```
fgetwc(fp);      /* Returns X'00C1' (the hexadecimal      */
                 /* wchar representation of A)      */
fgetwc(fp);      /* Returns X'00C2' (the hexadecimal      */
                 /* wchar representation of B)      */
fgetwc(fp);      /* Returns X'7FFE' (the hexadecimal      */
                 /* wchar representation of the DBCS   */
                 /* character) between the SO and SI   */
                 /* characters; leaves file position at C */
ungetwc('Z',fp); /* Logically inserts Z before SI character */
fflush(fp);      /* Backs up one wchar, leaving position at   */
                 /* beginning of X'7FFE' DBCS char   */
                 /* and DBCS state in double-byte mode; */
                 /* clears Z from the logical stream */
```

Figure 4. `ungetwc()` Example

You can set the `_EDC_COMPAT` environment variable before you open the file, so that `fflush()` ignores any character pushed back with `ungetwc()` or `ungetc()`, and leaves the file position where it was when `ungetwc()` or `ungetc()` was first issued. Any characters pushed back are still cleared. For more information about `_EDC_COMPAT`, see [Chapter 28, “Using environment variables,” on page 327](#).

Setting positions within files

The following conditions apply to text streams and binary streams.

Repositioning within text streams

When you use the `fseek()` or `fsetpos()` function to reposition within files, z/OS XL C recalculates the shift state.

If you update a record after a successful call to the `fseek()` function or the `fsetpos()` function, a partial multibyte character can be overwritten. Calling a wide character function for data after the written character can result in undefined behavior.

Use the `fseek()` or `fsetpos()` functions to reposition only to the start of a multibyte character. If you reposition to the middle of a multibyte character, undefined behavior can occur.

Repositioning within binary streams

When you are working with a wide-oriented file, keep in mind the state of the file position that you are repositioning to. If you call `ftell()`, you can seek with `SEEK_SET` and the state will be reset correctly. You cannot use such an `ftell()` value across a program boundary unless the stream has been marked wide-oriented. A seek specifying a relative offset (`SEEK_CUR` or `SEEK_END`) will change the state to initial state. Using relative offsets is strongly discouraged, because you may be seeking to a point that is not in initial state, or you may end up in the middle of a multibyte character, causing wide-oriented functions to give you undefined behavior. These functions expect you to be at the beginning or end of a multibyte character in the correct state. Using your own offset with `SEEK_SET` also does the same. For a wide-oriented file, the number of valid bytes or records that `ftell()` supports is cut in half.

When you use the `fsetpos()` function to reposition within a file, the shift state is set to the state saved by the function. Use this function to reposition to a wide character that is not in the initial state.

ungetwc() considerations

For text files, the library functions `fgetpos()` and `ftell()` take into account the character you have pushed back onto the input stream with `ungetwc()`, and move the file position back by one wide character. The starting position for an `fseek()` call with a whence value of `SEEK_CUR` also takes into account this pushed-back wide character. Backing up one wide character means backing up either a single-byte character or a multibyte character, depending on the type of the preceding character. The implicit new-lines at the end of each record are counted as wide characters.

For binary files, the library functions `fgetpos()` and `ftell()` also take into account the character you have pushed back onto the input stream with `ungetwc()`, and adjust the file position accordingly. However, the `ungetwc()` must push back the same type of character just read by `fgetwc()`, so that `ftell()` and `fgetpos()` can save the state correctly. An `fseek()` with an offset of `SEEK_CUR` also accounts for the pushed-back character. Again, the `ungetwc()` must unget the same type of character for this to work properly. If the `ungetwc()` pushes back a character in the opposite state, you will get undefined behavior.

You can make only one call to `ungetwc()`. If the current logical file position is already at or before the first `wchar` in the file, a call to `ftell()` or `fgetpos()` after `ungetwc()` fails.

When you are using `fseek()` with a whence value of `SEEK_CUR`, the starting point for the reposition also accounts for the presence of `ungetwc()` characters and compensates as `ftell()` and `fgetpos()` do. Specifying a relative offset other than 0 is not supported and results in undefined behavior.

You can set the `_EDC_COMPAT` environment variable to specify that `ungetwc()` should not affect `fgetpos()` or `fseek()`. (It will still affect `ftell()`.) If the environment variable is set, `fgetpos()` and `fseek()` ignore any pushed-back wide character. See [Chapter 28, “Using environment variables,” on page 327](#) for more information about `_EDC_COMPAT`.

If a repositioning operation fails, z/OS XL C attempts to restore the original file position by treating the operation as a call to `fflush()`. It does not account for the presence of `ungetwc()` characters, which are lost.

Closing files

z/OS XL C expects files to end in initial shift state. For binary byte-oriented files, you must ensure that the ending state of the file is initial state. Failure to do so results in undefined behavior if you reaccess the file again. For wide-oriented streams and byte-oriented text streams, z/OS XL C tracks new data that you add. If necessary, z/OS XL C adds a padding byte to complete any incomplete multibyte character and a shift-in to end the file in initial state.

Manipulating wide character array functions

To manipulate wide character arrays in your program, you can use the functions shown in Table 6 on page 36. For more information about these functions, refer to [z/OS C/C++ Runtime Library Reference](#).

Table 6. Manipulating wide character arrays

Function	Purpose
<code>wmemcmp()</code>	Compare wide character
<code>wmemchr()</code>	Locate wide character
<code>wmemcpy()</code>	Copy wide character
<code>wmemmove()</code>	Move wide character
<code>wmemset()</code>	Set wide character
<code>wcrtomb()</code>	Convert a wide character to a multibyte character
<code>wscat()</code>	Append to wide-character string
<code>wcschr()</code>	Search for wide-character substring
<code>wscmp()</code>	Compare wide-character strings

Chapter 8. Performing OS I/O operations

This topic describes using OS I/O, which includes support for the following:

- Regular sequential DASD (including striped data sets)
- Partitioned DASD (PDS and PDSE)
- Tapes
- SYSOUT
- Printers
- In-stream JCL

Notes:

1. z/OS XL C/C++ does not support BDAM, ISAM, or non-VSAM keyed data sets. Attempting to open a non-VSAM keyed data set for read or append fails; attempting to open an existing non-VSAM keyed data set for write results in unpredictable results.
2. z/OS XL C/C++ provides complete read and write support for large format sequential data sets when seek is requested (not specifying the noseek keyword) and when noseek is requested and honored.
3. z/OS XL C/C++ provides support for extended format sequential data sets in the extended addressing space on extended address volumes (EAVs).
4. z/OS XL C/C++ supports opening DDNAMEs that have been dynamically allocated with the XTIO, UCB nocapture, and DSAB-above-the-line options specified in the SVC99 parameters (S99TIOEX, S99ACUCB, S99DSABA flags).

OS I/O supports text, binary, record I/O, and blocked I/O, in record formats: fixed (F), variable (V), and undefined (U). For information about using wide-character I/O with z/OS XL C/C++, see [Chapter 7, “z/OS XL C support for the double-byte character set,”](#) on page 29.

This topic describes C I/O stream functions as they can be used within C++ programs. If you want to use the C++ I/O stream classes instead, see [Chapter 4, “Using the Standard C++ Library I/O Stream Classes,”](#) on page 21 for general information. For more detailed information, see [Standard C++ Library Reference](#), which discusses the Standard C++ I/O stream classes

Opening files

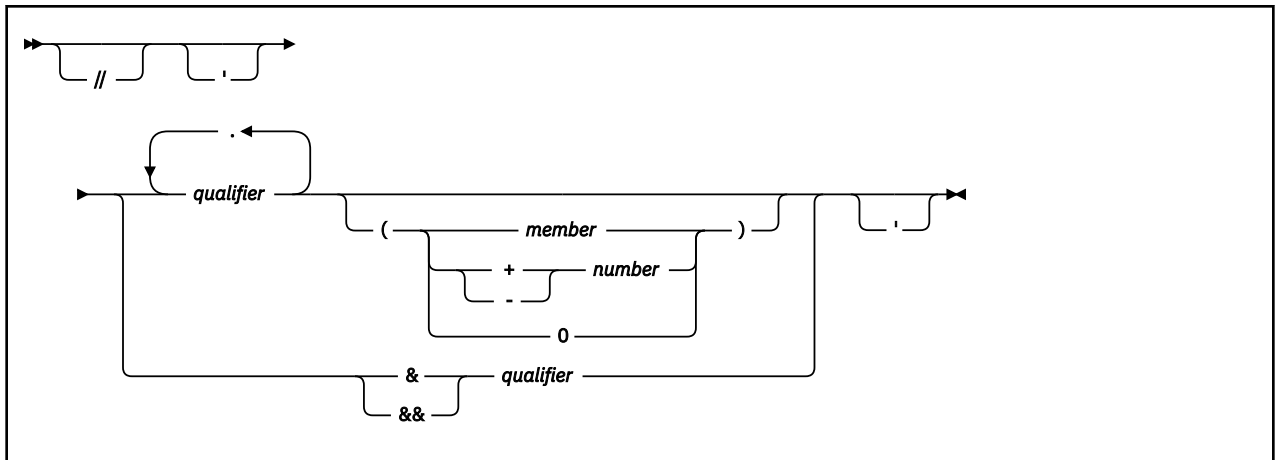
To open an OS file, you can use the Standard C functions `fopen()` or `freopen()`. These are described in general terms in [z/OS C/C++ Runtime Library Reference](#). Details about them specific to all z/OS XL C/C++ I/O are discussed in the "Opening Files" section. This section describes considerations for using `fopen()` and `freopen()` with OS files.

Using `fopen()` or `freopen()`

When you open a file using `fopen()` or `freopen()`, you must specify the filename (a data set name) or a ddname. **Restriction:** It is not possible to open a file for writing if there is already an open file with the same data set name on a different volume

Using a data set name

Files are opened with a call to `fopen()` or `freopen()` in the format `fopen("filename", "mode")`. The following diagram shows the syntax for the *filename* argument on your `fopen()` or `freopen()` call:



Note: The single quotation marks in the *filename* syntax diagram must be matched; if you use one, you must use the other.

A sample construct is shown below:

```
'qualifier1.qualifier2(member)'
```

//

Specifying these slashes indicates that the filename refers to a non-POSIX file or data set.

qualifier

Each qualifier is a 1- to 8-character name. These characters may be alphanumeric, national (\$, #, @), or the hyphen. The first character should be either alphabetic or national. Do not use hyphens in names for RACF®-protected data sets.

You can join qualifiers with periods. The maximum length of a data set name is as follows:

- Generally, 44 characters, including periods.
- For a generation data group, 35 characters, including periods.

These numbers do not include a member name or GDG number and accompanying parentheses.

Specifying one or two ampersands before a single qualifier opens a temporary data set. Multiple qualifiers are not valid after ampersands, because the system generates additional qualifiers. Opening two temporary data sets with the same name creates two distinct files. If you open a second temporary data set using the same name as the first, you get a distinct data set. For example, the following statements open two temporary data sets:

```
fp = fopen("//&&myfile","wb+");
fp2 = fopen("//&&myfile","wb+");
```

You cannot fully qualify a temporary data set name. The file is created at open time and is empty. When you close a temporary data set, the system removes it.

(member)

If you specify a *member*, the data set you are opening must be a PDS or a PDSE. For more information about PDSs and PDSEs, see [“Regular and extended partitioned data sets”](#) on page 42. For members, the member name (including trailing blanks) can be up to 8 characters long. A member name cannot begin with leading blanks. The characters in a member name may be alphanumeric, national (\$, #, @), the hyphen, or the character X'CO'. The first character should be either alphabetic or national.

+number

-number

0

You specify a Generation Data Group (GDG) by using a plus (+) or minus (-) to precede the version number, or by using a 0. For more information about GDGs, see [“Generation data group I/O”](#) on page 40.

The Resource Access Control Facility (RACF) expects the data set name to have a high-level qualifier that is defined to RACF. RACF uses the entire data set name when it protects a tape data set.

When you enclose a name in single quotation marks, the name is *fully qualified*. The file opened is the one specified by the name inside the quotation marks. If the name is not fully qualified, z/OS XL C/C++ does one of the following:

- If your system does not use RACF, z/OS XL C/C++ does not add a high-level qualifier to the name you specified.
- If you are running under TSO (batch or interactive), z/OS XL C/C++ appends the TSO user prefix to the front of the name. For example, the statement `fopen("a.b", "w")`; opens a data set `tsoid.A.B`, where `tsoid` is the user prefix. If the name is fully qualified, z/OS XL C/C++ does not append a user prefix. You can set the user prefix by using the TSO PROFILE command with the PREFIX parameter.
- If you are running under z/OS batch or IMS (batch or online), z/OS XL C/C++ appends the RACF user ID to the front of the name.

If you want your code to be portable between the VM/CMS and z/OS systems and between memory files and disk files, use a name of the format `name1.name2`, where `name1` and `name2` are up to 8 characters and are delimited by a period, or use a ddname. You can also add a member name.

For example, the following piece of code can run under Language Environment for VM and z/OS Language Environment:

```
FILE *stream;
stream = fopen("parts.instock", "r");
```

Using a DDname

The DD statement enables you to write C or C++ source programs that are independent of the files and input/output devices they use. You can modify the parameters of a file or process different files without recompiling your program.

Use ddnames if you want to use non-DASD devices.

If you specify DISP=MOD on a DD statement and `w` or `wb` mode on the `fopen()` call, z/OS XL C/C++ treats the file as if you had opened it in append mode instead of write mode.

To open a file by ddname under z/OS batch, you must define the ddname first. You can do this in any of the following ways:

- In batch (z/OS, TSO, or IMS), you can write a JCL DD statement. For the declaration shown above for the C or C++ file PARTS.INSTOCK, you write a JCL DD statement similar to the following:

```
//STOCK DD DSN=USERID.PARTS.INSTOCK,DISP=SHR
```

When defining DD, do not use `DD ... FREE=CLOSE` for unallocating DD statements. The C library may close files to perform some file operations such as `freopen()`, and the DD statement will be unallocated.

If you use SPACE=RLSE on a DD statement, z/OS XL C/C++ releases space only if all of the following are true:

- The file is open in `w`, `wb`, `a`, or `ab` mode
- It is not simultaneously open for read
- No positioning functions (`fseek()`, `ftell()`, `rewind()`, `fgetpos()`, `fsetpos()`) have been performed.

For more information on writing DD statements, refer to the job control language (JCL) manuals listed in [z/OS Information Roadmap](#).

- Under TSO (interactive and batch), you can issue an ALLOCATE command. The DD definition shown above for the C file STOCK has an equivalent TSO ALLOCATE command, as follows:

```
ALLOCATE FILE(STOCK) DATASET(PARTS.INSTOCK) SHR
```

See [z/OS Information Roadmap](#) for manuals containing information on TSO ALLOCATE.

- In the z/OS environment, you can use the `svc99()` or `dynalloc()` library functions to define ddnames. For information about these functions, refer to [z/OS C/C++ Runtime Library Reference](#).

DCB parameter

The DCB (data control block) parameter of the DD statement allows you to describe the characteristics of the data in a file and the way it will be processed at run time. The other parameters of the DD statement deal chiefly with the identity, location, and disposition of the file. The DCB parameter specifies information required for the processing of the records themselves. The subparameters of the DCB parameter are described in [z/OS MVS JCL User's Guide](#).

The DCB parameter contains subparameters that describe:

- The organization of the file and how it will be accessed. Parameters supplied on `fopen()` override those specified in DCB.
- Device-dependent information such as the recording technique for magnetic tape or the line spacing for a printer (for example: CODE, DEN, FUNC, MODE, OPTCD=J, PRTSP, STACK, SPACE, UNIT and TRTCH subparameters).
- The data set format (for example: BLKSIZE, LRECL, and RECFM subparameters).

You cannot use the DCB parameter to override information already established for the file in your C or C++ program (by the file attributes declared and the other attributes that are implied by them). DCB subparameters that attempt to change information already supplied by `fopen()` or `freopen()` are ignored. An example of the DCB parameter is:

```
DCB=(RECFM=FB, BLKSIZE=400, LRECL=40)
```

It specifies that fixed-length records, 40 bytes in length, are to be grouped in a block 400 bytes long. You can copy attributes from another data set by either setting the DCB parameter to `DCB=(dsname)` or using the SVC 99 services provided by the `svc99()` and `dynalloc()` library functions.

Generation data group I/O

A Generation Data Group (GDG) is a group of related cataloged data sets. Each data set within a generation data group is called a generation data set. Generation data sets have sequentially ordered absolute and relative names that represent their age. The absolute generation name is the representation used by the catalog management routines in the catalog. The relative name is a signed integer used to refer to the latest (0), the next to the latest (-1), and so forth, generation. The relative number can also be used to catalog a new generation (+1). For more information on GDGs, see [z/OS DFSMS Using Data Sets](#).

If you want to open a generation data set by data set name with `fopen()` or `freopen()`, you will require a model. This model specifies parameters for the group, including the maximum number of generations (the generation index). You can define such a model by using the Access Method Services `DEFINE` command. For more information on the `DEFINE` command, see [z/OS DFSMS Access Method Services Commands](#). Note also that `fopen()` does not support a `DCB=` parameter. If you want to change the parameters, alter the JCL that describes the model and open it in `w` mode.

z/OS uses an absolute generation and version number to catalog each generation. The generation and version numbers are in the form `GxxxxVyy`, where `xxxx` is an unsigned 4-digit decimal generation number (0001 through 9999) and `yy` is an unsigned 2-digit decimal version number (00 through 99). For example:

- A.B.C.G0001V00 is generation data set 1, version 0, in generation data group A.B.C.
- A.B.C.G0009V01 is generation data set 9, version 1, in generation data group A.B.C.

The number of generations kept depends on the size of the generation index.

When you open a GDG by relative number, z/OS XL C/C++ returns the relative generation in the `__dsname` field of the structure returned by the `fldata()` function. You cannot use the `rename()` library function to rename GDGs by relative generation number; rename GDG data sets by using their absolute names.

The example shown in [Figure 5 on page 41](#) is valid only for C. The sample program (CCNGOS1) defines a GDG. The `fopen()` fails because it tries to change the RECFM of the data set.

```
/*-----  
/* This example demonstrates GDG I/O  
/*-----  
/* Create GDG model MYGDG.MODEL and GDG name MYGDG  
/*-----  
//MODEL      EXEC PGM=IDCAMS  
//DD1        DD DSN=userid.MYGDG.MODEL,DISP=(NEW,CATLG),  
//            UNIT=SYSDA,SPACE=(TRK,(0)),  
//            DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB)  
//SYSPRINT   DD SYSOUT=*  
//SYSIN      DD *  
DEFINE GDG -  
    (NAME(userid.MYGDG) -  
    EMPTY -  
    SCRATCH -  
    LIMIT(255))  
/*  
/*-----  
/* Create GDG data set MYGDG(+1)  
/*-----  
//DATASET    EXEC PGM=IEFBR14  
//DD1        DD DSN=userid.MYGDG(+1),DISP=(NEW,CATLG),  
//            SPACE=(CYL,(1,1)),UNIT=SYSDA,  
//            DCB=userid.MYGDG.MODEL  
//SYSPRINT   DD SYSOUT=*  
//SYSIN      DD DUMMY  
/*-----  
/* Compile, link, and run an inlined C program.  
/* This program attempts to open the GDG data set MYGDG(+1) but  
/* should fail as it is opening the data set with a RECFM that is  
/* different from that of the GDG model (F versus FB).  
/*-----  
//C          EXEC EDCCLG,  
//            CPARM='NOSEQ,NOMARGINS'  
//COMPILE.SYSIN DD DATA,DLM='>'  
#include <stdio.h>  
#include <errno.h>  
  
int main(void)  
{  
    FILE *fp;  
  
    fp = fopen("MYGDG(+1)", "a,recfm=F");  
  
    if (fp == NULL)  
    {  
        printf("Error...Unable to open file\n");  
        printf("errno ... %d\n",errno);  
        perror("perror ... ");  
    }  
  
    printf("Finished\n");  
}  
>
```

Figure 5. Generation data group example for C

The example shown in [Figure 6 on page 42](#) (CCNGOS2) is valid only C++.

```

/*-----
/* This example demonstrates GDG I/O
/*-----
/* Create GDG model MYGDG.MODEL and GDG name MYGDG
/*-----
//MODEL      EXEC PGM=IDCAMS
//DD1        DD DSN=userid.MYGDG.MODEL,DISP=(NEW,CATLG),
//            UNIT=SYSDA,SPACE=(TRK,(0)),
//            DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB)
//SYSPRINT   DD SYSOUT=*
//SYSIN      DD *
//DEFINE GDG -
//            (NAME(userid.MYGDG) -
//            EMPTY -
//            SCRATCH -
//            LIMIT(255))
/*
/*-----
/* Create GDG data set MYGDG(+1)
/*-----
//DATASET    EXEC PGM=IEFBR14
//DD1        DD DSN=userid.MYGDG(+1),DISP=(NEW,CATLG),
//            SPACE=(CYL,(1,1)),UNIT=SYSDA,
//            DCB=userid.MYGDG.MODEL
//SYSPRINT   DD SYSOUT=*
//SYSIN      DD DUMMY
/*-----
/* Compile, bind, and run an inlined C++ program.
/* This program attempts to open the GDG data set MYGDG(+1) but
/* should fail as it is opening the data set with a RECFM that is
/* different from that of the GDG model (F versus FB).
/*-----
/*
//DOCLG1     EXEC CBCCBG,
//            CPARM='NOSEQ,NOMARGINS'
//COMPILE.SYSIN DD DATA,DLM='<>'
#include <stdio.h>
#include <errno.h>
int main(void)
{
    FILE *fp;

    fp = fopen("MYGDG(+1)", "a,recfm=F");

    if (fp == NULL)
    {
        printf("Error...Unable to open file\n");
        printf("errno ... %d\n",errno);
        perror("perror ... ");
    }

    printf("Finished\n");
}

```

Figure 6. Generation data group example for C++

A relative number used in the JCL refers to the same generation throughout a job. The (+1) used in the example above exists for the life of the entire job and not just the step, so that `fopen()`'s reference to (+1) did not create another new data set but accessed the same data set as in previous steps.

Note: You cannot use `fopen()` to create another generation data set because `fopen()` does not fully support the DCB parameter.

Regular and extended partitioned data sets

Partitioned data sets (PDS) and partitioned data sets extended (PDSE) are DASD data sets divided into sections known as *members*. Each member can be accessed individually by its unique 1- to 8-character name. As [Table 7 on page 42](#) shows, PDSEs are similar to PDSs, but contain a number of enhancements.

Table 7. PDSE and PDS differences

PDSE Characteristics	PDS Characteristics
Data set has a 123-extent limit	Data set has a 16-extent limit
Directory is open-ended and indexed by member name; faster to search directory	Fixed-size directory is searched sequentially

Table 7. PDSE and PDS differences (continued)

PDSE Characteristics	PDS Characteristics
PDSEs are device-independent: records are reblockable	Block sizes are device-dependent
Uses dynamic space allocation and reclamation	Must use IEBCOPY COMPRESS to reclaim space
Supports creation of more than one member at a time*	Supports creation of only one member at a time

Note: *z/OS XL C/C++ allows you to open two separate members of a PDSE for writing at the same time. However, you cannot open a single member for writing more than once.

You specify a member by enclosing its name in parentheses and placing it after the data set name. For example, the following JCL refers to member A of the data set MY . DATA:

```
//MYDD DD DSN=userid.MY.DATA(A),DISP=SHR
```

You can specify members on calls to `fopen()` and `freopen()`. You can specify members when you are opening a data set by its data set name or by a ddname. When you use a ddname and a member name, the definition of the ddname must not also specify a member. For example, using the DD statement above, the following will fail:

```
fp = fopen("dd:MYDD(B)", "r");
```

You cannot open a PDS or PDSE member using the modes `a`, `ab`, `a+`, `a+b`, `w+`, `w+b`, or `wb+`. If you want to perform the equivalent of the `w+` or `wb+` mode, you must first open the file as `w` or `wb`, write to it, and then close it. Then you can perform updates by reopening the file in `r+` or `rb+` mode. You can use the C library functions `ftell()` or `fgetpos()` to obtain file positions for later updates to the member. Normally, opening a file in `r+` or `rb+` mode enables you to extend a file by writing to the end; however, with these modes you cannot extend a member. To do so, you must copy the contents of the old member plus any extensions to a new member. You can remove the old member by using the `remove()` function and then rename the new member to the old name by using `rename()`.

All members have identical attributes for RECFM, LRECL, and BLKSIZE. For PDSs, you cannot add a member with different attributes or specify a RECFM of FBS, FBSA, or FBSM. z/OS XL C/C++ verifies any attributes you specify.

For PDSEs, z/OS XL C/C++ checks to make sure that any attributes you specify are compatible with those of the existing data set. Compatible attributes are those that specify the same record format (F, V, or U) and the same LRECL. Compatibility of attributes enables you to choose whether to specify blocked or unblocked format, because PDSEs reblock all the records. For example, you can create a PDSE as FB LRECL=40 BLKSIZE=80, and later open it for read as FB LRECL=40 BLKSIZE=1600 or F LRECL=40 BLKSIZE=40. The LRECL cannot change, and the BLKSIZE must be compatible with the RECFM and LRECL. Also, you cannot change the basic format of the PDSE from F to V or vice versa. If the PDS or PDSE already exists, you do not need to specify any attributes, because z/OS XL C/C++ uses the previously existing ones as its defaults.

At the start of each partitioned data set is its directory, a series of records that contain the member names and starting locations for each member within the data set. You can access the directory by specifying the PDS or PDSE name without specifying a member. You can open the directory only for read; update and write modes are not allowed. The only RECFM that you can specify for reading the directory is RECFM=U. However, you do not need to specify the RECFM, because z/OS XL C/C++ uses U as the default.

[z/OS DFSMS Using Data Sets](#) contains more detailed explanations about how to use PDSs and PDSEs.

Partitioned and sequential concatenated data sets

There are two forms of concatenated data sets: partitioned and sequential. You can open concatenated data sets only by ddname, and only for read or update. Specifying any of the write, or append modes fails. As with PDS members, you cannot extend a concatenated data set.

Partitioned concatenation consists of specifying multiple PDSs or PDSEs under one ddname. When you access the concatenation, it acts as one large PDS or PDSE, from which you can access any member. If two or more partitioned data sets in the concatenation contain a member with the same name, using the concatenation ddname to specify that member refers to the first member with that name found in the entire concatenation. You cannot use the ddname to access subsequent members. For example, if you have a PDS named PDS1, with members A, B, and C, and a second PDS named PDS2, with members C, D, and E, and you concatenate the two data sets as follows:

```
//MYDD DD userid.PDS1,DISP=SHR
// DD userid.PDS2,DISP=SHR
```

and perform the following:

```
fp = fopen("DD:MYDD(C)","r");
fp2 = fopen("DD:MYDD(D)","r");
```

the first call to `fopen()` finds member C from PDS1, even though there is also a member C in PDS2. The second call finds member D from PDS2, because PDS2 is the first PDS in the concatenation that contains this member. The member C in PDS2 is inaccessible.

When you are concatenating partitioned data sets, be aware of the DCB attributes for them. The concatenation is treated as a single data set with the following attributes:

- RECFM= the RECFM of the first data set in the concatenation
- LRECL= the LRECL of the first data set in the concatenation
- BLKSIZE= the largest BLKSIZE of any data set in the concatenation

Table 8 on page 44 describes the rules for compatible concatenations.

Table 8. Rules for possible concatenations

RECFM of first data set	RECFM of subsequent data sets	LRECL of subsequent data sets
RECFM=F	RECFM=F	Same as that of first one
RECFM=FB	RECFM=F or RECFM=FB	Same as that of first one
RECFM=V	RECFM=V	Less than or equal to that of first one
RECFM=VS	RECFM=V or RECFM=VS	Less than or equal to that of first one
RECFM=VB	RECFM=V or RECFM=VB	Less than or equal to that of first one
RECFM=VBS	RECFM=V, RECFM=VB, RECFM=VS, or RECFM=VBS	Less than or equal to that of first one
RECFM=U	RECFM=U or RECFM=F (see note)	

Note: You can use a data set in V-format, but when you read it, you will see all of the BDWs and RDWs or SDWs with the data.

If the first data set is in ASA format, all subsequent data sets must be ASA as well. The preceding rules apply to ASA files if you add an A to the RECFMs specified.

If you do not follow these rules, undefined behavior occurs. For example, trying to read a fixed-format member as RECFM=V could cause an exception or abend.

Repositioning is supported as it is for regular PDSs and PDSEs. If you try to read the directory, you will be able to read only the first one.

Sequential concatenation consists of treating multiple sequential data sets or partitioned data set members as one long sequential data set.

```
//MYDD DD userid.PDS1(A),DISP=SHR
//      DD userid.PDS2(E),DISP=SHR
//      DD userid.DATA,DISP=SHR
```

creates a concatenation that contains two members and a regular sequential data set. You can read or update all of these in order. In partitioned concatenations, you can read only one member at a time.

z/OS XL C/C++ does not support concatenating data sets that do not have compatible DCB attributes. The rules for compatibility are the same as those for partitioned concatenations.

If all the data sets in the concatenation support repositioning, you can reposition within a concatenation by using the functions `fseek()`, `ftell()`, `fgetpos()`, `fsetpos()`, and `rewind()`. If the first one does not, all of the repositioning functions except `rewind()` fail for the entire concatenation. If the first data set supports repositioning but a subsequent one does not, you must specify the `noseek()` parameter on the `fopen()` or `freopen()` call. If you do not, `fopen()` or `freopen()` opens the file successfully; however, an error occurs when the read position gets to the data set that does not support repositioning.

You can use sequential concatenation (DSORG=PS in DCB) to sequentially read directories of PDSs and PDSEs. For more information about reading directories of PDSs and PDSEs, refer to [z/OS DFSMS Using Data Sets](#).

Note: Concatenated and multivolume data sets only tolerate single buffering mode.

In-stream data sets

An *in-stream data set* is a data set contained within a set of JCL statements. In-stream data sets (also called inline data sets) begin with a `DD *` or `DD DATA` statement. These DD statements can have any valid ddname, including `SYSIN`. If you omit a DD statement before the input data, the system provides a `DD *` statement with the ddname of `SYSIN`. This example shows you how to indicate an in-stream data set:

```
//MYDD DD *
record 1
record 2
record 3
/*
```

The `//` at the beginning of the data set starts in column 1. The statement `fopen("DD:MYDD", "rb")`; opens a data set with `lrecl=80`, `blksize=80`, and `recfm=FB`. In this example, the delimiter indicating the end of the data set is `/*`. In some cases, your data may contain this string. For example, if you are using C source code that contains comments, z/OS XL C/C++ treats the beginning of the first comment as the end of the in-stream data set. To avoid this occurrence, you can change the delimiter by specifying `DLM=nn`, where `nn` is a two-character delimiter, on the DD statement that identifies the file. For example:

```
//MYDD DD *,DLM=@@
#include <stdio.h>
/* Hello, world program */
int main() {printf("Hello, world\n"); }
@@
```

For more information about in-stream data sets, see [z/OS MVS JCL User's Guide](#).

To open an in-stream data set, call the `fopen()` or `freopen()` library function and specify the ddname of the data set. You can open an in-stream data set only for reading. Specifying any of the update, write, or append modes fails. Once you have opened an in-stream data set, you cannot acquire or change the file position except by rewinding. This means that calls to the `fseek()`, `ftell()`, `fgetpos()`, and `fsetpos()` for in-stream data sets fail. Calling `rewind()` causes z/OS XL C/C++ to reopen the file, leaving the file position at the beginning.

You can concatenate regular sequential data sets and in-stream data sets. If you do so, note the following:

- If the first data set is in-stream, you cannot acquire or change the file position for the entire concatenation.
- If the first data set is not in-stream and supports repositioning, you must specify the `noseek` parameter on the `fopen()` or `freopen()` call that opens the concatenation. If you do not, `fopen()` or `freopen()` opens the file successfully; however, an error occurs when the read position gets to the in-stream.
- The in-stream data set is treated as FB 80 and the concatenation rules for sequential concatenation apply.

SYSOUT data sets

You can specify a SYSOUT data set by using the SYSOUT parameter on a DD statement. z/OS XL C/C++ supports opening SYSOUT data sets in two ways:

1. Specifying a ddname that has the SYSOUT parameter. For information about defining ddnames, see [“Using a DDname”](#) on page 39.
2. Specifying a data set name of * on a call to `fopen()` or `freopen()` while you are running under z/OS batch or IMS online or batch.

On a DD statement, you specify `SYSOUT=x`, where `x` is the output class. If the class matches the JOB statement `MSGCLASS`, the output appears with the job log. You can specify a SYSOUT data set and get the job `MSGCLASS` by specifying `SYSOUT=*`. If you want to create a job stream within your program, you can specify `INTRDR` on the DD statement. This sends your SYSOUT data set to the internal reader to be read as an input job stream. For example,

```
//MYDD DD SYSOUT=(A,INTRDR)
```

For more details about the SYSOUT parameter, refer to [z/OS MVS JCL User's Guide](#).

You can specify DCB attributes for a SYSOUT data set on a DD statement or a call to `fopen()` or `freopen()`. If you do not, z/OS XL C/C++ uses the following defaults:

Binary, Record I/O, or Blocked I/O

RECFM=VB LRECL=137 BLKSIZE=882

Text I/O

RECFM=VBA LRECL=137 BLKSIZE=882

Tapes

z/OS XL C/C++ supports standard label (SL) tapes. If you are creating tape files, you can only open them by ddname. z/OS XL C/C++ provides support for opening tapes in read, write, or append mode, but not update. When you open a tape for read or append, any data set control block (DCB) characteristics you specify must match those of the existing data set exactly. The repositioning functions are available only when you have opened a tape for read. For tapes opened for write or append, calling `rewind()` has no effect; calls to any of the other repositioning functions fail. To open a tape file for write, you must open it by ddname.

Opening FBS-format tape files with append-only mode is not supported.

When you open a tape file for output, the data set name you specify in the JCL must match the data set name specified in the tape label, even if the existing tape file is empty. If this is not the case, you must either change the JCL to specify the correct data set name or write to another tape file, or reinitialize the tape to remove the tape label and the data. You can use IEBGENER with the following JCL to create an empty tape file before passing it to the subsequent steps:

```
//ALLOC EXEC PGM=IEBGENER
//SYSUT1 DD *
/*
//SYSUT2 DD DSN=name-of-OUTPUT-tape-file,UNIT=xxxx,LABEL=(x,SL),
// DISP=(NEW,PASS),(DCB=LRECL=xx,BLKSIZE=xx,RECFM=xx),
// VOL=SER=xxx
```

```
//SYSIN DD DUMMY
//SYSPRINT DD SYSOUT=*
```

Note: For tapes, the value for UNIT= can be TAPE or CART.

Because the C library does not create tape files, you can append only to a tape file that already exists. Attempting to append to a file that does not already exist on a tape will cause an error. You can create an empty data set on a tape by using the utility IEBGENER.

Multivolume data sets

z/OS XL C/C++ supports data sets that span more than one volume of DASD or tape. You can open multivolume DASD data sets for read (r, rb), write (w, wb), update (r+, rb+, w+, wb+), or append (a, a+, ab, ab+) by dsname and ddname. Multivolume data sets can be extended only in read/update mode (r+, rb+).

The repositioning functions are available when you have opened a multivolume data set for r, r+, rb, rb+, w+, wb+, a+, ab+. Repositioning multivolume data sets opened for w, wb, a, ab is not allowed because it would be meaningless. For multivolume data sets opened for write, calling rewind() has no effect; calls to any of the other repositioning functions fail.

Here is an example of a multivolume data set declaration:

```
//MYDD DD DSNAME=TEST.TWO,DISP=(NEW,CATLG),
//      VOLUME=(, ,3,SER=(333001,333002,333003)),
//      SPACE=(TRK,(9,10)),UNIT=(3390,P)
```

This creates a data set that may span up to three volumes. For more information about the VOLUME parameter on DD statements, refer to [z/OS MVS JCL User's Guide](#).

Notes:

1. Simultaneous readers (files that can support sharing by a writer and one or more readers) are not supported for multivolume data sets.
2. Concatenated and multivolume data sets only tolerate single buffering mode.

Striped data sets

z/OS XL C/C++ supports extended format sequential data sets. Extended format data sets must be SMS-managed. Optionally, extended format data sets can be striped. Striping spreads a data set over a specified number of volumes such that I/O parallelism can be exploited. Unlike a multivolume data set in which physical record n follows record n-1, a striped data set has physical records n and n-1 on separate volumes. This enables asynchronous I/O to perform parallel operations, making requests for multiple reads and writes faster.

Striped data sets also facilitate repositioning once the relative block number is known. z/OS XL C/C++ exploits this capability when it uses fseek() to reposition. This can result in substantial savings for applications that use ftell() and fseek() with data sets that have RECFMs of V, U, and FB (not FBS). data sets. When a data set is striped, an fseek() can seek directly to the specified block just as an fsetpos() or rewind() can. For a normal data set with the aforementioned RECFMs, z/OS XL C/C++ has to read forward or rewind the data set to get to the desired position. Depending on how large the data set is, this can be quite inefficient compared to a direct reposition. Note that for such data sets, striping pads blocks to their maximum size. Therefore, you may be wasting space if you have short records.

Large format sequential data sets

A large format sequential data set is a modification to traditional sequential data sets that allows for more than 65535 tracks of data per volume. Large format sequential data sets can be single or multivolume, and can reside on SMS managed or non-SMS managed direct access storage devices.

A large format sequential data set is specified using the DSNTYPE=LARGE keyword on a JCL DD statement or using the dynamic allocation equivalent. z/OS XL C/C++ does not support the allocation of a large format sequential data set using `fopen()` or `freopen()`.

z/OS XL C/C++ provides complete read and write support for:

- Large format sequential data sets when seek is requested (not specifying the noseek keyword).
- Large format sequential data sets when noseek is requested and honored. See [“Access method selection”](#) on page 48 for more information on when noseek is requested but not honored.

Note: Restrictions associated with traditional sequential single and multivolume data sets, or concatenations of such data sets, continue to apply when the data set is large format.

Other devices

z/OS XL C/C++ supports several other devices for input and output. You can open these devices only by ddname. Table 9 on page 48 lists a number of these devices and describes which record formats are valid for them. None of the devices listed can be opened for update except the DUMMY data set.

Table 9. Other devices supported for input and output

Device	Valid open modes	Repositioning?	fldata().__device
Printer	w, wb, a, ab	No	__PRINTER
Card reader	r, rb	rewind() only	__OTHER
Card punch	w, wb, a, ab	No	__OTHER
Optical reader	r, rb	rewind() only	__OTHER
DUMMY data set	r, rb, r+, rb+, r+b, w, wb, w+, wb+ w+b, a, ab, a+, ab+, a+b	rewind() only	__DUMMY
SUBSYS=	r, rb	No	__OTHER

Note: For all devices above that support open modes a or ab, the modes are treated as if you had specified w or wb.

z/OS XL C/C++ queries each device to find out its maximum BLKSIZE.

The DUMMY data set is not truly a device, although z/OS XL C/C++ treats it as one. To use the DUMMY data set, specify DD DUMMY in your JCL. On input, the DUMMY data set always returns EOF; on output, it is always successful. This is the way to specify a DUMMY data set:

```
//MYDD DD DUMMY
```

For more information on DUMMY data sets, see [z/OS MVS JCL User's Guide](#).

z/OS XL C/C++ provides minimal support for subsystem (SUBSYS=) managed data sets. Support is limited to opening for read using the dd:ddname(member) syntax with the `fopen()` filename. The ddname in the JCL stream must specify the SUBSYS= parameter. In this case, `fopen()` will open the DCB as DSORG=PS, required for subsystem managed data sets, but will perform the BLDL/FIND sequence to allow the subsystem to manage processing of the desired member.

Access method selection

The `fopen()` and `freopen()` keyword noseek requests QSAM (queued sequential access method) be used to process the data set. This request also indicates that the repositioning functions will not be used by the application. This access method generally provides the best performance. Omitting the keyword noseek selects BSAM (basic sequential access method) with NOTE and POINT macros requested, allowing repositioning functions to be used (where applicable) on the stream.

The following scenarios exist where QSAM (noseek) is requested, but the z/OS C/C++ runtime library switches to BSAM with NOTE and POINT macros requested (seek):

- The data set is opened for update (`r+`, `rb+`, `w+`, `wb+`, `a+`, `ab+`)
- The data set is already opened for write (or update) in the same C process
- The data set is RECFM=FBS opened for append (`a`, `ab`, `a+`, `ab+`)
- The data set is LRECL=X
- The data set is the directory of a partitioned data set (PDS or PDSE)
- The data set is a member of a partitioned data set where the member was not specified at allocation, but rather specified at `fopen()` or `freopen()`

Note: Repositioning is not allowed when `noseek` is requested, even if there was a switch to `seek`.

fopen() and freopen() parameters

Table 10 on page 49 lists the parameters that are available on the `fopen()` and `freopen()` functions, tells you which ones are allowed and applicable for OS I/O, and lists the option values that are valid for the applicable ones. Detailed descriptions of these options follow the table.

Parameter	Allowed?	Applicable?	Notes
<code>recfm=</code>	Yes	Yes	Any of the 27 record formats available under z/OS XL C/C++, plus * and A are valid.
<code>lrecl=</code>	Yes	Yes	0, any positive integer up to 32760, or X is valid. See the parameter list below.
<code>blksize=</code>	Yes	Yes	0 or any positive integer up to 32760 is valid.
<code>space=</code>	Yes	Yes	Valid only if you are opening a new data set by its data set name. See the parameter list below.
<code>type=</code>	Yes	Yes	May be omitted. If you do specify it, <code>type=record</code> or <code>type=blocked</code> can be the valid values.
<code>acc=</code>	Yes	No	Not used for OS I/O.
<code>password=</code>	Yes	No	Not used for OS I/O.
<code>asis</code>	Yes	No	Used to specify mixed-case filenames. Not recommended.
<code>byteseek</code>	Yes	Yes	Used for binary files to specify that the seeking functions should use relative byte offsets instead of encoded offsets.
<code>noseek</code>	Yes	Yes	Used to disable seeking functions for improved performance.
<code>OS</code>	Yes	No	Ignored.
<code>abend=</code>	Yes	Yes	See note below.

recfm=

z/OS XL C/C++ allows you to specify any of the 27 possible RECFM types (listed in “Fixed-format records” on page 12, “Variable-format records” on page 15, and “Undefined-format records” on page 18), as well as the z/OS XL C/C++ RECFMs * and A.

When you are opening an existing file for read or append (or for write, if you have specified `DISP=MOD`), any RECFM that you specify must match that of the existing file, except that you may specify `recfm=U` to open any file for read, and you may specify `recfm=FBS` for a file created as `recfm=FB`. Specifying `recfm=FBS` indicates to z/OS XL C/C++ that there are no short blocks within the file. If there are, undefined behavior results.

For variable-format OS files, the RDW, SDW, and BDW contain the length of the record, segment, and block as well as their own lengths. If you open a file for read with `recfm=U`, z/OS XL C/C++ treats each physical block as an undefined-format record. For files created with `recfm=V`, z/OS XL C/C++ does not strip off block descriptor words (BDWs) or record descriptor words (RDWs), and for blocked files,

it does not deblock records. Using `recfm=U` is helpful for viewing variable-format files or seeing how records are blocked in the file.

When you are opening an existing PDS or PDSE for write and you specify a RECFM, it must be compatible with the RECFM of the existing data set. FS and FBS formats are invalid for PDS members. For PDSs, you must use exactly the same RECFM. For PDSEs, you may choose to change the blocked attribute (B), because PDSEs perform their own blocking. If you want to read a PDS or PDSE directory and you specify a RECFM, it must be `recfm=U`.

Specifying `recfm=A` indicates that the file contains ASA control characters. If you are opening an existing file and you specify that ASA characters exist (`>recfm=A`) when they do not, the call to `fopen()` or `freopen()` fails. If you create a file by opening it for write or append, the A attribute is added to the default RECFM. For more information about ASA, see [Chapter 6, “Using ASA text files,”](#) on page 25.

Specifying `recfm=*` causes the compiler to fill in any attributes that you do not specify, taking the attributes from the existing data set. This is useful if you want to create a new version of a data set with the same attributes as the previous version. If you open a data set for write and the data set does not exist, the compiler uses the default attributes specified in `fopen()` defaults. This parameter has no effect when you are opening for read or append, and when you use it for non-DASD files.

`recfm=+` is identical to `recfm=*` with the following exceptions:

- If there is no record format for the existing DASD data set, the defaults are assigned as if the data set did not exist.
- When append mode is used, the `fopen()` fails.

lrecl= and blksize=

The LRECL that you specify on the `fopen()` call defines the maximum record length that the C library allows. Records longer than the maximum record length are not written to the file. The first 4 bytes of each block and the first 4 bytes of each record of variable-format files are used for control information. For more information, see [“Variable-format records”](#) on page 15.

The maximum LRECL supported for fixed, undefined, or variable-blocked-spanned format sequential disk files is 32760. For other variable-length format disk files the maximum LRECL is 32756. Sequential disk files for any format have a maximum BLKSIZE of 32760. The record length can be any size when opening a spanned file and specifying `lrecl=X`. You can now specify `lrecl=X` on the `fopen()` or `freopen()` call for spanned files. If you are updating an existing file, the file must have been originally opened with `lrecl=X` for the open to succeed. `lrecl=X` is useful only for text, record I/O, and blocked I/O.

When you are opening an existing file for read or append (or for write, if you have specified `DISP=MOD`), any LRECL or BLKSIZE that you specify must match that of the existing file, except when you open an F or FB format file on a disk device without specifying the `noseek` parameter. In this case, you can specify the S attribute to indicate to z/OS XL C/C++ that the file has no imbedded short blocks. Files without short blocks improve the performance of z/OS XL C/C++.

`BLKSIZE=0` will be ignored for an existing data set opened for read or append.

When you are opening an existing PDS or PDSE for write and you specify an LRECL or BLKSIZE, it must be compatible with the LRECL or BLKSIZE of the existing data set. For PDSs, you must use exactly the same values. For PDSEs, the LRECL must be the same, but the BLKSIZE may be different if you have changed the blocking attribute as described under the RECFM parameter above. You can change the blocking attribute, because PDSEs perform their own blocking. The BLKSIZE you choose should be compatible with the RECFM and LRECL. When you open the directory of a PDS or PDSE, do not specify LRECL or BLKSIZE; z/OS XL C/C++ uses the defaults. See [Table 11 on page 54](#) for more information.

space=(units, (primary, secondary, directory) [rlse | norlse])

This keyword enables you to specify the space parameters for the allocation of a z/OS data set. It applies only to z/OS data sets that you open by filename and do not already exist. If you open a data set by ddname, this parameter has no effect. You cannot specify any whitespace inside the value for the space keyword. You must specify at least one value with this parameter. Any parameter that you

specify will be validated for syntax. If that validation fails, then the `fopen()` or `freopen()` will fail even if the parameter would have been ignored.

The supported values for *units* are as follows:

- Any positive integer indicating BLKSIZE
- CYL (mixed case)
- TRK (mixed case)

The primary quantity, the secondary quantity, and the directory quantity all must be positive integers. The primary quantity is always required.

If you specify values only for *units* and *primary*, you do not have to specify the inside set of parentheses. You can use a comma to indicate a quantity is to take the default value. For example:

```
space=(cyl,(100,,10)) - default secondary value
space=(trk,(100,,))   - default secondary and directory value
space=(500,(100,))    - default secondary, no directory
```

The last parameter, *rlse* or *norlse*, controls the disposition of the unused space. If you open a new file for write and specify the *space* keyword, by default, any unused space will be released when the file is closed. You can preserve the allocated space by specifying *norlse*. For example:

```
space=(cyl,(100,,10),norlse) - does not release unused space
space=(trk,(100,,),rlse)     - releases unused space
pace=(500,(100,))            - releases unused space
```

You can specify only the values indicated on this parameter. If you specify any other values, `fopen()` or `freopen()` fails. Any values not specified are omitted on the allocation. These values are filled by the system during SVC 99 processing.

type=

You can omit this parameter. If you specify it, the valid values for OS I/O are *type=record* which opens a file for record I/O, and *type=blocked* which opens a file for blocked I/O.

acc=

This parameter is not valid for OS I/O. If you specify it, z/OS XL C/C++ ignores it.

password=

This parameter is not valid for OS I/O. If you specify it, z/OS XL C/C++ ignores it.

asis

If you use this parameter, z/OS XL C/C++ does not convert your filenames to upper case. The use of the *asis* parameter is strongly discouraged, because most of the I/O services used by z/OS XL C/C++ require uppercase filenames.

byteseek

When you specify this parameter and open a file in binary mode, all repositioning functions (such as `fseek()` and `ftell()`) use relative byte offsets from the beginning of the file instead of encoded offsets. In previous releases of z/OS XL C/C++, byte seeking was performed only for fixed format binary files. To have the *byteseek* parameter set as the default for all your calls to `fopen()` or `freopen()`, you can set the environment variable `_EDC_BYTE_SEEK` to Y. See [Chapter 28, “Using environment variables,”](#) on page 327 for more information.

noseek

Specifying this parameter on the `fopen()` call disables the repositioning functions `ftell()`, `fseek()`, `fgetpos()`, and `fsetpos()` for as long as the file is open. When you have specified *NOSEEK* and have opened a disk file for read only, the only repositioning function allowed on the file is `rewind()`, if the device supports rewinding. Otherwise, a call to `rewind()` sets `errno` and raises `SIGIOERR`, if `SIGIOERR` is not set to `SIG_IGN`. Calls to `ftell()`, `fseek()`, `fsetpos()`, or `fgetpos()` return `EOF`, set `errno`, and set the stream error flag on.

The use of the `noseek` parameter may improve performance when you are reading and writing data sets.

Note: If you specify the `NOSEEK` parameter when you open a file for writing, you must specify `NOSEEK` on any subsequent `fopen()` call that simultaneously opens the file for reading; otherwise, you will get undefined behavior.

OS

If you specify this parameter, z/OS XL C/C++ ignores it.

abend= abend | recover

This parameter is ignored for SPC applications. The z/OS XL C/C++ runtime library uses the DCB ABEND exit to intercept abend conditions that occur during OS I/O operations. When an abend condition occurs, DFSMS issues a write-to-programmer message and gives control to the DCB ABEND exit. Within the information provided to the exit, a flag indicates if the abend condition can be ignored. It is not predictable when an abend condition can be ignored, and sometimes the same abend condition (completion code and reason code) can be ignored while at other times it cannot.

When the abend condition can be ignored, the runtime library instructs DFSMS to do so. This allows the runtime library to return gracefully back to the user code with a failing return value, `errno` set to 92 (meaning an I/O abend was trapped), and diagnostic information in the `__amrc` structure. DFSMS will have stopped processing the DCB. Only the `fdata()`, `fclose()`, and `freopen()` functions are permitted on the stream after an abend condition is trapped.

When the abend condition cannot be ignored, DFSMS issues the abend and Language Environment condition handling semantics take effect. If condition handling is active, the abend is converted to a Language Environment condition or SIGABND signal. In the absence of a condition handler or signal handler, the default behavior is to terminate the enclave. An application can write its own condition handler or SIGABND handler and try to recover from the error. As is the case when an abend condition can be ignored, DFSMS will have stopped processing the DCB and only the `fdata()`, `fclose()`, and `freopen()` functions are permitted on the stream.

abend

instructs the runtime library to ignore abend conditions that can be ignored. No attempt is made to recover from abend conditions that cannot be ignored.

recover

instructs the runtime library to attempt to recover from an abend issued during certain low-level I/O operations (WRITE / CHECK sequence, CLOSE, GET, and PUT). If recovery is possible, control will be returned to the user code with a failing return value, `errno` set to 92, and diagnostic information in the `__amrc` structure.

The `abend` keyword specifies the behavior only for the stream being opened. It overrides the setting of the `_EDC_IO_ABEND` environment variable. See Chapter 28, “Using environment variables,” on page 327 for more information. Also, see Chapter 16, “Debugging I/O programs,” on page 161 for more information on the `__amrc` structure and handling errors during I/O operations.

Buffering

z/OS XL C/C++ uses buffers to map C I/O to system-level I/O. When z/OS XL C/C++ performs I/O operations, it uses one of the following buffering modes:

- *Line buffering* — characters are transmitted to the system when a new-line character is encountered. Line buffering is meaningless for binary, record I/O, and blocked I/O files.
- *Full buffering* — characters are transmitted to the system when a buffer is filled.

C/C++ provides a third buffering mode, unbuffered I/O, which is not supported for OS files.

You can use the `setvbuf()` and `setbuf()` library functions to set the buffering mode before you perform any I/O operation to the file. `setvbuf()` fails if you specify unbuffered I/O. It also fails if you try to specify line buffering for an FBS data set opened in text mode, where the device does not support repositioning. This failure happens because z/OS XL C/C++ cannot deliver records at line boundaries

without violating FBS format. Do not try to change the buffering mode after you have performed any I/O operation to the file.

For all files except `stderr`, full buffering is the default, but you can use `setvbuf()` to specify line buffering. For binary files, record I/O files, and unblocked text files, a block is written out as soon as it is full, regardless of whether you have specified line buffering or full buffering. For blocked I/O files, a block is written out as soon as `fwrite()` completes. Line buffering is different from full buffering only for blocked text files.

Multiple buffering

Multiple buffering (or asynchronous I/O) is supported for z/OS data sets. Multiple buffering is not supported for a data set opened for read at the same time that another file pointer has it opened for write or append. When you open files for multiple buffering, blocks are read into buffers before they are needed, eliminating the delay caused by waiting for I/O to complete. Multiple buffering may make I/O less efficient if you are seeking within or writing to a file, because seeking or writing may discard blocks that were read into buffers but never used.

To specify multiple buffering, code either the `NCP=xx` or `BUFNO=yy` subparameter of the DCB parameter on the JCL DD statement (or allocation), where `xx` is an integer number between 02 and 99, and `yy` is an integer number normally between 02 and 255. Whether z/OS XL C/C++ uses `NCP` or `BUFNO` depends on whether you are using BSAM or QSAM, respectively. `NCP` is supported under BSAM; `BUFNO` is supported under QSAM. BSAM and QSAM are documented in [z/OS DFSMS Using Data Sets](#). If you specify `noseek`, z/OS XL C/C++ uses QSAM if possible. If z/OS XL C/C++ is using BSAM and you specify a value for `BUFNO`, z/OS XL C/C++ maps this value to `NCP`. If z/OS XL C/C++ is using QSAM and you specify a value for `NCP`, z/OS XL C/C++ maps this value to `BUFNO`.

If you specify both `NCP` and `BUFNO`, z/OS XL C/C++ takes the greater of the two values, up to the maximum for the applicable value. For example, if you specify a `BUFNO` of 120 and you are using BSAM, which uses `NCP` instead, z/OS XL C/C++ will use `NCP=99`.

If you do not specify either, z/OS XL C/C++ defaults to single buffering, except in the following cases, where z/OS XL C/C++ uses the system's default `BUFNO` and performs multiple buffering for both reading and writing:

- If you open a device that does not support repositioning, and specify read-only or write-only mode (`r`, `rb`, `w`, `wb`, `a`, `ab`).
- If you specify the `NOSEEK` parameter on the call to `fopen()` or `freopen()`, and specify read-only or write-only mode. When you specify `NOSEEK`, you get multiple buffering for both reads and writes.

Here is an example of how to specify `BUFNO`:

```
//DD5 DD DSN=TORONTO.BLUEJAYS,DISP=SHR,DCB=(BUFNO=5)
```

You may need to update code from previous releases that relies on z/OS XL C/C++ ignoring `NCP` or `BUFNO` parameters.

Note: Multiple buffering is ignored for concatenated and multivolume data sets.

DCB (Data Control Block) attributes

For OS files, the C runtime library creates a skeleton data control block (DCB) for the file when you open it. File attributes are determined from the following sources in this order:

1. The `fopen()` or `freopen()` function call
2. Attributes for a `ddname` specified previously (if you are opening by `ddname`)
3. Existing file attributes (if you specify `reclm=*` or you are opening an existing file for read or append)
4. Defaults from `fopen()` or `freopen()` for creating a new file.

If you do not specify `RECFM` when you are creating a new file, z/OS XL C/C++ uses the following defaults:

- If `recfm` is not specified in a `fopen()` call for an output binary file, `recfm` defaults to:
 - `recfm=VB` for spool (printer) files,
 - `recfm=FB` otherwise.
- If `recfm` is not specified in a `fopen()` call for an output text file, `recfm` defaults to:
 - `recfm=F` if `_EDC_ANSI_OPEN_DEFAULT` is set to Y and no `LRECL` or `BLKSIZE` specified. In this case, `LRECL` and `BLKSIZE` are both defaulted to 254.
 - `recfm=VBA` for spool (printer) files.
 - `recfm=U` for terminal files
 - `recfm=V` if the `LRECL` or `BLKSIZE` is specified
 - `recfm=VB` for all other OS files.

If `recfm` is not specified for a record I/O or blocked I/O file, you will get the default of `recfm=VB`. Table 11 on page 54 shows the defaults for `LRECL` and `BLKSIZE` when the z/OS XL C/C++ compiler creates an OS file. Information from the C or C++ program overrides that from the DD statement and the tape label. Information from the DD statement overrides that from the data set label.

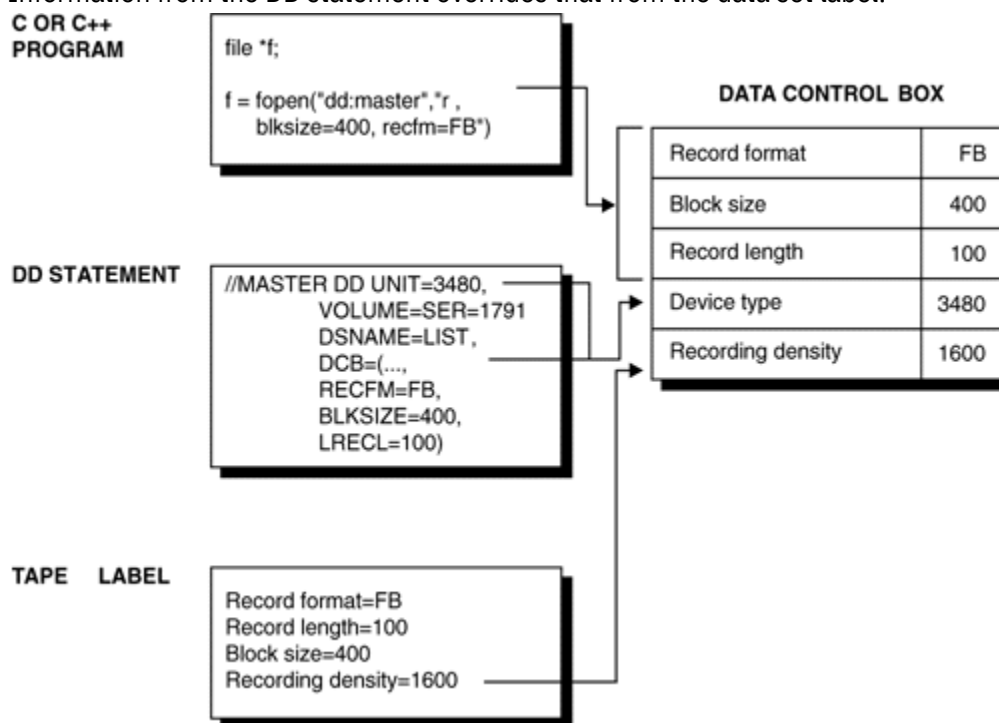


Figure 7. How the operating system completes the DCB

Table 11. `fopen()` defaults for `LRECL` and `BLKSIZE` when creating OS files

lrecl specified?	blksize specified?	RECFM	LRECL	BLKSIZE
no	no	All F	80	80
		All FB	80	maximum integral multiple of 80 less than or equal to <i>max</i>
		All V, VB, VS, or VBS	minimum of 1028 or <i>max-4</i>	<i>max</i>
		All U	0	<i>max</i>

Table 11. *fopen()* defaults for LRECL and BLKSIZE when creating OS files (continued)

lrecl specified?	blksize specified?	RECFM	LRECL	BLKSIZE
yes	no	All F	<i>lrecl</i>	<i>lrecl</i>
		All FB	<i>lrecl</i>	maximum integral multiple of <i>lrecl</i> less than or equal to <i>max</i>
		All V	<i>lrecl</i>	<i>lrecl</i> +4
		All U	0	<i>lrecl</i>
no	yes	All F or FB	<i>blksize</i>	<i>blksize</i>
		All V, VB, VS, or VBS	minimum of 1028 or <i>blksize</i> -4	<i>blksize</i>
		All U	0	<i>blksize</i>

Note: All includes the standard (S) specifier for fixed formats, the ASA (A) specifier, and the machine control character (M) specifier.

In Table 11 on page 54, the value *max* represents the maximum reasonable block size for the device. These are the current default maximum block sizes for several devices that z/OS XL C/C++ supports:

Device

Default maximum block size

DASD

6144

3203 Printer

132

3211 Printer

132

4245 Printer

132

2540 Reader

80

2540 Punch

80

2501 Reader

80

3890 Document Processor

80

TAPE

32760

For more information about specific default block sizes as returned by the DEVTYPE macro, refer to [z/OS DFSMS Using Data Sets](#).

You can perform multiple buffering under z/OS. See [“Multiple buffering” on page 53](#) for details.

Reading from files

You can use the following library functions to read from a file:

- `fread()`
- `fread_unlocked()`

- `fgetc()`
- `fgetc_unlocked()`
- `fgets()`
- `fgets_unlocked()`
- `fscanf()`
- `fscanf_unlocked()`
- `getc()`
- `getc_unlocked()`
- `gets()`
- `gets_unlocked()`
- `getchar()`
- `getchar_unlocked()`
- `scanf()`
- `scanf_unlocked()`
- `vscanf()`
- `vscanf_unlocked()`

`fread()` is the only interface allowed for reading record I/O or blocked I/O files. Except for blocked I/O files, a read operation directly after a write operation without an intervening call to `fflush()`, `fsetpos()`, `fseek()`, or `rewind()` fails. z/OS XL C/C++ treats the following as read operations:

- Calls to read functions that request 0 bytes
- Read requests that fail because of a system error
- Calls to the `ungetc()` function

z/OS XL C/C++ does not consider a read to be at EOF until you try to read past the last byte visible in the file. For example, in a file containing three bytes, the `feof()` function returns FALSE after three calls to `fgetc()`. Calling `fgetc()` one more time causes `feof()` to return TRUE.

You can set up a SIGIOERR handler to catch read or write system errors. See the debugging section in this book for more details.

Reading from binary files

z/OS XL C/C++ reads binary records in the order that they were written to the file. Any null padding is visible and treated as data. Record boundaries are meaningless.

Reading from text files

For non-ASA variable text files, the default for z/OS XL C/C++ is to ignore any empty physical records in the file. If a physical record contains a single blank, z/OS XL C/C++ reads in a logical record containing only a new-line. However, if the environment variable `_EDC_ZERO_RECLLEN` was set to Y, z/OS XL C/C++ reads an empty physical record as a logical record containing a new-line, and a physical record containing a single blank as a logical record containing a blank *and* a new-line. z/OS XL C/C++ differentiates between empty records and records containing single blanks, and does not ignore either of them. For more information about how z/OS XL C/C++ treats empty records in variable format, see [“Mapping C types to variable format”](#) on page 17.

For ASA variable text files, if a file was created without a control character as its first byte, the first byte defaults to the ' ' character. When the file is read back, the first character is read as a new-line.

On input, ASA characters are translated to the corresponding sequence of control characters. For more information about using ASA files, refer to [Chapter 6, “Using ASA text files,”](#) on page 25.

For undefined format text files, reading a file causes a new-line character to be inserted at the end of each record. On input, a record containing a single blank character is considered an empty record and is translated to a new-line character. Trailing blanks are preserved for each record.

For files opened in fixed text format, rightmost blanks are stripped off a record at input, and a new-line character is placed in the logical record. This means that a record consisting of a single new-line character is represented by a fixed-length record made entirely of blanks.

Reading from record I/O files

For files opened in record format, `fread()` is the only interface that supports reading. Each time you call `fread()` for a record I/O file, `fread()` reads one record. If you call `fread()` with a request for less than a complete record, the requested bytes are copied to your buffer, and the file position is set to the start of the next record. If the request is for more bytes than are in the record, one record is read and the position is set to the start of the next record. z/OS XL C/C++ does not strip any blank characters or interpret any data.

`fread()` returns the number of items read successfully, so if you pass a `size` argument equal to 1 and a `count` argument equal to the maximum expected length of the record, `fread()` returns the length, in bytes, of the record read. If you pass a `size` argument equal to the maximum expected length of the record, and a `count` argument equal to 1, `fread()` returns either 0 or 1, indicating whether a record of length `size` read. If a record is read successfully but is less than `size` bytes long, `fread()` returns 0.

A failed read operation may lead to undefined behavior until you reposition successfully.

Reading from blocked I/O files

For files opened in blocked format, `fread()` is the only interface that supports reading. Each time you call `fread()` for a blocked I/O file, `fread()` reads one block. If you call `fread()` with a request for less than a complete block, the requested bytes are copied to your buffer, and the file position is set to the start of the next block. If the request is for more bytes than are in the block, one block is read and the position is set to the start of the next block. z/OS XL C/C++ does not strip any blank characters or interpret any data.

`fread()` returns the number of items read successfully, so if you pass a `size` argument equal to 1 and a `count` argument equal to the maximum expected length of the block, `fread()` returns the length, in bytes, of the block read. If you pass a `size` argument equal to the maximum expected length of the block, and a `count` argument equal to 1, `fread()` returns either 0 or 1, indicating whether a block of length `size` read. If a block is read successfully but is less than `size` bytes long, `fread()` returns 0.

A failed read operation may lead to undefined behavior until you reposition successfully.

Writing to files

You can use the following library functions to write to a file:

- `fwrite()`
- `fwrite_unlocked()`
- `printf()`
- `printf_unlocked()`
- `fprintf()`
- `fprintf_unlocked()`
- `vprintf()`
- `vprintf_unlocked()`
- `vfprintf()`

- `vfprintf_unlocked()`
- `puts()`
- `puts_unlocked()`
- `fputc()`
- `fputc_unlocked()`
- `fputs()`
- `fputs_unlocked()`
- `putc()`
- `putc_unlocked()`
- `putchar()`
- `putchar_unlocked()`

`fwrite()` is the only interface allowed for writing to record I/O or blocked I/O files. See [z/OS C/C++ Runtime Library Reference](#) for more information on these library functions.

A write operation directly after a read operation without an intervening call to `fflush()`, `fsetpos()`, `fseek()`, or `rewind()` fails unless the read operation has reached EOF. The file pointer does not reach EOF until after you have tried to read *past* the last byte of the file.

z/OS XL C/C++ counts a call to a write function writing 0 bytes or a write request that fails because of a system error as a write operation.

If you are updating a file and a system failure occurs, z/OS XL C/C++ tries to set the file position to the end of the last record updated successfully. For a fully-buffered file, this is at the end of the last record in a block. For a line-buffered file, this may be any record in the current block. If you are writing new data at the time of a system failure, z/OS XL C/C++ puts the file position at the end of the last block of the file. In files opened for blocked output, you may lose data written by other writes to that block before the system failure. The contents of a file after a system write failure are indeterminate.

If one user opens a file for writing, and another later opens the same file for reading, the user who is reading the file can check for records that may have been written past the end of the file by the other user. If the file is a spanned variable text file, the reader can read part of a spanned record and reach the end of the file before reading in the last segment of the spanned record.

Writing to binary files

Data flows over record boundaries in binary files. Writes or updates past the end of a record go to the next record. When you are writing to files and not making any intervening calls to `fflush()`, blocks are written to the system as they are filled. If a fixed record is incomplete when you close the file, z/OS XL C/C++ completes it with nulls. You cannot change the length of existing records in a file by updating them.

If you are using variable binary files, note the following:

- On input and on update, records that have no length are ignored; you will not be notified. On output, zero-length records are not written. However, in spanned files, if the first segment of a record has been written to the system, and the user flushes or closes the file, a zero-length last segment may be written to the file.
- If you are writing new data in a data set that has variable-length records (RECFM=V, VB, VBM, etc.), z/OS XL C/C++ may split a record between two blocks to fill the first block out to the maximum block size. This means that when you read them, the record boundaries will not necessarily be the same.
- If your file is spanned, records are written up to length LRECL, spanning multiple blocks if necessary. You can create a spanned file by specifying a RECFM containing V and S on the `fopen()` call.

Writing to text files

z/OS XL C/C++ treats the control characters as follows when you are writing to a non-ASA text file:

- \a**
Alarm. Placed directly into the file; z/OS XL C/C++ does not interpret it.
- \b**
Backspace. Placed directly into the file; z/OS XL C/C++ does not interpret it.
- \f**
Form feed. Placed directly into the file; z/OS XL C/C++ does not interpret it.
- \n**
New-line. Defines a record boundary; z/OS XL C/C++ does not place it in the file.
- \r**
Carriage return. Defines a record boundary; z/OS XL C/C++ does not place it in the file. Treated like a new-line character.
- \t**
Horizontal tab character. Placed directly into the file; z/OS XL C/C++ does not interpret it.
- \v**
Vertical tab character. Placed directly into the file; z/OS XL C/C++ does not interpret it.
- \x0E**
DBCS shift-out character. Indicates the beginning of a DBCS string, if MB_CUR_MAX > 1. Placed into the file.
- \x0F**
DBCS shift-in character. Indicates the end of a DBCS string, if MB_CUR_MAX > 1. Placed into the file. See [z/OS XL C support for the double-byte character set in z/OS XL C/C++ Programming Guide](#) for more information about MB_CUR_MAX.

The way z/OS XL C/C++ treats text files depends on whether they are in fixed, variable, or undefined format, and whether they use ASA.

As with ASA files in other environments, the first character of each record is reserved for the ASA control character that represents a new-line, a carriage return, or a form feed. See [Chapter 6, “Using ASA text files,”](#) on page 25 for more information.

Table 12. C control to ASA characters

C Control Character Sequence	ASA Character	Description
\n	' '	skip one line
\n\n	'0'	skip two lines
\n\n\n	'.'	skip three lines
\f	'1'	new page
\r	'+'	overstrike

Writing to fixed-format text files

Records in fixed-format files are all the same length. You complete each record with a new-line or carriage return character. For fixed text files, the new-line character is not written to the file. Blank padding is inserted to the LRECL of each record of the block, and the block, when full, is written. For a more complete description of the way fixed-format files are handled, see [“Fixed-format records”](#) on page 12.

A logical record can be shortened to be an empty record (containing just a new-line) or extended to a record containing LRECL bytes of data plus a new-line. Because the physical record represents the new-line position by using padding blanks, the new-line position can be changed on an update as long as it is within the physical record.

Note: Using `ftell()` or `fgetpos()` values for positions that do not exist after you have shortened records results in undefined behavior.

When you are updating a file, writing new data into an existing record replaces the old data and, if the new data is longer or shorter than the old data, changes the size of the logical record by changing the

number of blank characters in the physical record. When you extend a record, thereby writing over the old new-line, a new-line character is implied after the last character of the update. Calling `fflush()` flushes the data out to the file and inserts blank padding between the last data character and the end of the record. Once you have called `fflush()`, you can call any of the read functions, which begin reading at the new-line. Once the new-line is read, reading continues at the beginning of the next record.

Writing to variable-format text files

In a file with variable-length records, each record may be a different length. The variable length formats permit both variable-length records and variable-length blocks. The first 4 bytes of each block are reserved for the Block Descriptor Word (BDW); the first 4 bytes of each record are reserved for the Record Descriptor Word (RDW).

For ASA and non-ASA, the `'\n'` (new-line) character implies a record boundary. On output, the new-line is not written to the physical file; instead, it is assumed to follow the data of the record.

If you have not set `_EDC_ZERO_RECLen`, z/OS XL C/C++ writes out a record containing a single blank character to represent a single new-line. On input, a record containing a single blank character is considered an empty record and is translated to a new-line character. Note that a single blank followed by a new-line is written out as a single blank, and is treated as just a new-line on input. When `_EDC_ZERO_RECLen` is set, writing a record containing only a new-line results in a zero-length variable record.

For more information about environment variables, refer to [Chapter 28, “Using environment variables,” on page 327](#). For more information about how z/OS XL C/C++ treats empty records in variable format, see [“Mapping C types to variable format” on page 17](#).

Attempting to shorten a record on update by specifying less data before the new-line causes the record to be padded with blanks to the original record size. For spanned records, updating a record to a shorter length results in the same blank padding to the original record length, over multiple blocks, if applicable.

Attempts to lengthen a record on update generally result in truncation. The exception to this rule is extending an empty record to a 1-byte record when the environment variable `_EDC_ZERO_RECLen` is not set. Because the physical representation for an empty record is a record containing one blank character, it is possible to extend the logical record to a single non-blank character followed by a new-line character. For standard streams, truncation in text files does not occur; data is wrapped automatically to the next record as if you had added a new-line.

When you are writing data to a non-blocked file without intervening flush or reposition requests, each record is written to the system when a new-line or carriage return character is written or when the file is closed.

When you are writing data to a blocked file without intervening flush or reposition requests, if the file is opened in full buffering mode, the block is written to the system on completion of the record that fills the block. If the blocked file is line buffered, each record is written to the system when it is completed. If you are using full buffering for a VB format file, a write may not fill a block completely. The data does not go to the system unless a block is full; you can complete the block with another write. If the subsequent write contains more data than is needed to fill the block, it flushes the current block to the system and starts writing your data to a new block.

When you are writing data to a spanned file without intervening flush or reposition requests, if the record spans multiple blocks, each block is written to the system once it is full and the user writes an additional byte of data.

For ASA variable text files, if a file was created without a control character as its first byte or record (after the RDW and BDW), the first byte defaults to the `' '` character. When the file is read back, the first character is read as a new-line.

Writing to undefined-format text files

In an undefined-format file, there is only one record per block. Each record may be a different length, up to a maximum length of `BLKSIZE`. Each record is completed with a new-line or carriage return character.

The new-line character is not written to the physical file; it is assumed to follow the data of the record. However, if a record contains only a new-line character, z/OS XL C/C++ writes a record containing a single blank to the file to represent an empty record. On input, the blank is read in as a new-line.

Once a record has been written, you cannot change its length. If you try to shorten a logical record by updating it with a shorter record, z/OS XL C/C++ completes the record with blank padding. If you try to lengthen a record by updating it with more data than it can hold, z/OS XL C/C++ truncates the new data. The only instance in which this does not happen is when you extend an empty record so that it contains a single byte. Any data beyond the single byte is truncated.

Truncation versus splitting

If you try to write more data to a record than z/OS XL C/C++ allows, and the file you are writing to is not one of the standard streams (the defaults, or those redirected by `freopen()` or command-level redirection), output is cut off at the record boundary and the remaining bytes are discarded. z/OS XL C/C++ does not count the discarded characters as characters that have been written out successfully.

In all truncation cases, the `SIGIOERR` signal is raised if the action for `SIGIOERR` is not `SIG_IGN`. The user error flag is set so that `ferror()` will return `TRUE`. For more information about `SIGIOERR`, `ferror()`, and other I/O-related debugging tools, see [Chapter 16, “Debugging I/O programs,” on page 161](#). z/OS XL C/C++ continues to discard new output until you complete the current record by writing a new-line or carriage return character, close the file, or change the file position.

If you are writing to one of the standard streams, attempting to write more data than a record can hold results in the data being split across multiple records.

Writing to record I/O files

`fwrite()` is the only interface allowed for writing to a file opened for record I/O. Only one record is written at a time. If you attempt to write more new data than a full record can hold or you try to update a record with more data than it currently has, z/OS XL C/C++ truncates your output at the record boundary. When z/OS XL C/C++ performs a truncation, it sets `errno` and raises `SIGIOERR`, if `SIGIOERR` is not set to `SIG_IGN`.

When you update a record, you can update less than the full record. The remaining data that you do not update is left untouched in the file.

When you are writing new records to a fixed-record I/O file, if you try to write a short record, z/OS XL C/C++ pads the record with nulls out to `LRECL`.

At the completion of an `fwrite()`, the file position is at the start of the next record. For new data, the block is flushed out to the system as soon as it is full.

Writing to blocked I/O files

`fwrite()` is the only interface allowed for writing to a file opened for blocked I/O. Only one block is written at a time. If you attempt to write more new data than a full block can hold or you try to update a block with more data than it currently has, z/OS XL C/C++ truncates your output at the block boundary. When z/OS XL C/C++ performs a truncation, it sets `errno` and raises `SIGIOERR`, if `SIGIOERR` is not set to `SIG_IGN`.

When you write less than `BLKSIZE` bytes, if the request is to create a new block, a short block will be created; if it is to update an existing block, only requested part of the block will be updated. z/OS XL C/C++ will not check the provided data. At the completion of an `fwrite()`, the file position is at the start of the next block, and the block is flushed out to the system. You might need to consider the following cases:

- Because all fixed-format records must be full, any block you write must be multiple of a record. Otherwise, z/OS XL C/C++ will fail the write request.
- When updating an FBS short block at the end of file, it could be updated to a full block or a longer short block from start of the short block.

- When writing or appending to an FBS short block at the end of file, z/OS XL C/C++ will use the request buffer to replace the previous block, which might extend or shrink the short block.
- You must make sure that there is no short block in the middle of an FBS data set.
- You must make sure that BDWs, RDWs, and SDWs in a block of variable record file are correct.

Flushing buffers

You can use the library function `fflush()` to flush streams to the system. For more information about `fflush()`, see *z/OS C/C++ Runtime Library Reference*.

The action taken by the `fflush()` library function depends on the buffering mode associated with the stream and the type of streams. If you call one z/OS XL C/C++ program from another z/OS XL C/C++ program by using the ISO C/C++ `system()` function, all open streams are flushed before control is passed to the callee, and again before control is returned to the caller. If you are running with `POSIX(ON)`, a call to the `POSIX system()` function does not flush any streams to the system.

Updating existing records

Calling `fflush()` while you are updating flushes the updates out to the system. If you call `fflush()` when you are in the middle of updating a record, z/OS XL C/C++ writes the partially updated record out to the system. A subsequent write continues to update the current record.

Reading updated records

If you have a file open for read at the same time that the file is open for write in the same application, you will be able to see the new data if you call `fflush()` to refresh the contents of the input buffer, as shown in [Figure 8 on page 62](#).

```
/* this example demonstrates how updated records are read */
#include <stdio.h>
int main(void)
{
    FILE * fp, * fp2;
    int rc, rc2, rc3, rc4;
    fp = fopen("a.b", "w+");

    fprintf(fp, "first record");

    fp2 = fopen("a.b", "r"); /* Simultaneous Reader */

    /* following gets EOF since fp has not completed first line
     * of output so nothing will be flushed to file yet */
    rc = fgetc(fp2);
    printf("return code is %i\n", rc);

    fputc('\n', fp); /* this will complete first line */
    fflush(fp); /* ensures data is flushed to file */

    rc2 = fgetc(fp2); /* this gets 'f' from first record */
    printf("value is now %c\n", rc2);

    rewind(fp);

    fprintf(fp, "some updates\n");
    rc3 = fgetc(fp2); /* gets 'i' ..doesn't know about update */
    printf("value is now %c\n", rc3);

    fflush(fp); /* ensure update makes it to file */

    fflush(fp2); /* this updates reader's buffer */

    rc4 = fgetc(fp2); /* gets 'm', 3rd char of updated record */
    printf("value is now %c\n", rc4);

    return(0);
}
```

Figure 8. Example of reading updated records

Writing new records

Writing new records is handled differently for:

- Binary streams
- Text streams
- Record I/O
- Blocked I/O

Binary streams

z/OS XL C/C++ treats line buffering and full buffering the same way for binary files.

If the file has a variable length or undefined record format, `fflush()` writes the current record out. This may result in short records. In blocked files, this means that the block is written to disk, and subsequent writes are to a new block. For fixed files, no incomplete records are flushed.

For single-volume disk files in FBS format, `fflush()` flushes complete records in an incomplete block out to the file. For all other types of FBS files, `fflush()` does not flush an incomplete block out to the file.

For files in FB format, `fflush()` always flushes out all complete records in the current block. For sequential DASD files, new completed records are added to the end of the flushed block if it is short. For non-DASD or non-sequential files, any new record will start a new block.

Text streams

- Line-Buffered Streams

`fflush()` has no effect on line-buffered text files, because z/OS XL C/C++ writes all records to the system as they are completed. All incomplete new records remain in the buffer.

- Fully Buffered Streams

Calling `fflush()` flushes all completed records in the buffer, that is, all records ending with a new-line or carriage return (or form feed character, if you are using ASA), to the system. z/OS XL C/C++ holds any incomplete record in the buffer until you complete the record or close the file.

For ASA text files, if a flush occurs while an ASA character that indicates more than one new-line is being updated, the remaining new-lines will be discarded and a read will continue at the first data character. For example, if `'\n\n\n'` is updated to be `'\n\n'` and a flush occurs, then a `'0'` will be written out in the ASA character position.

Record I/O

z/OS XL C/C++ treats line buffering and full buffering the same way for record I/O. For files in FB format, calling `fflush()` writes all records in the buffer to the system. For single-volume disk files in FBS format, `fflush()` will flush complete records in an incomplete block out to the file. For all other types of FBS files, `fflush()` will not flush an incomplete block out to the file. For all other formats, calling `fflush()` has no effect, because `fwrite()` has already written the records to disk.

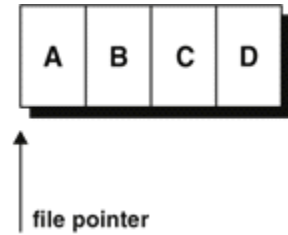
Blocked I/O

For all record formats and all buffering mode, calling `fflush()` has no effect, because `fwrite()` has already written the block to disk.

ungetc() considerations

`ungetc()` pushes characters back onto the input stream for binary and text files. `ungetc()` handles only single-byte characters. You can use it to push back as many as four characters onto the `ungetc()` buffer. For every character pushed back with `ungetc()`, `fflush()` backs up the file position by one character and clears all the pushed-back characters from the stream. Backing up the file position may end up going

across a record boundary. Remember that for text files, z/OS XL C/C++ counts the new-lines added to the records as single-byte characters when it calculates the file position.



For example, given the stream you can run the following code fragment:

```
fgetc(fp);      /* Returns A and puts the file position at */
                /* the beginning of the character B      */
ungetc('Z',fp); /* Logically inserts Z ahead of B        */
fflush(fp);     /* Moves the file position back by one to A, */
                /* removes Z from the logical stream      */
```

If you want `fflush()` to ignore `ungetc()` characters, you can set the `_EDC_COMPAT` environment variable. See [Chapter 28, “Using environment variables,”](#) on page 327 for more information.

Repositioning within files

You can use the following library functions to help you position within a file:

- `fseek()`
- `fseek_unlocked()`
- `fseeko()`
- `fseeko_unlocked()`
- `ftell()`
- `ftell_unlocked()`
- `ftello()`
- `ftello_unlocked()`
- `fgetpos()`
- `fgetpos_unlocked()`
- `fsetpos()`
- `fsetpos_unlocked()`
- `rewind()`
- `rewind_unlocked()`

With large file support enabled for AMODE 31 C/C++ applications, you can use the following library functions for 64-bit offsets:

- `fseeko()`
- `fseeko_unlocked()`
- `ftello()`
- `ftello_unlocked()`

For AMODE 64 C/C++ applications, large files are automatically supported in the LP64 programming model. All of the above functions (both lists) can be used with 64-bit offsets.

See [z/OS C/C++ Runtime Library Reference](#) for more information on these library functions.

Opening a file with `fopen()` and specifying the `NOSEEK` parameter disables all of these library functions except `rewind()`. A call to `rewind()` causes the file to be reopened, unless the file is a non-disk file

opened for write-only. In this case, `rewind()` sets `errno` and raises `SIGIOERR` (if `SIGIOERR` is not set to `SIG_IGN`, which is its default).

Calling any of these functions flushes all complete and updated records out to the system. If a repositioning operation fails, z/OS XL C/C++ attempts to restore the original file position and treats the operation as a call to `fflush()`, except that it does not account for the presence of `ungetc()` or `ungetwc()` characters, which are lost. After a successful repositioning operation, `feof()` always returns 0, even if the position is just after the last byte of data in the file.

The `fsetpos()` and `fgetpos()` library functions are generally more efficient than `ftell()` and `fseek()`. The `fgetpos()` function can encode the current position into a structure that provides enough room to hold the system position as well as position data specific to C or C++. The `ftell()` function must encode the position into a single word of storage, which it returns. This compaction forces `fseek()` to calculate certain position information specific to C or C++ at the time of repositioning. For variable-format binary files, you can choose to have `ftell()` return relative byte offsets. In previous releases, `ftell()` returned only encoded offsets, which contained the relative block number. Since you cannot calculate the block number from a relative byte offset in a variable-format file, `fseek()` may have to read through the file to get to the new position. `fsetpos()` has system position information available within the `fpos_t` structure and can generally reposition directly to the desired location.

You can use the `ftell()` and `fseek()` functions to set the current position within all types of files except for the following:

- Files on nonseekable devices (for example, printers)
- Partitioned data sets opened in `w` or `wb` mode.

Although repositioning within files opened for write mode is not available, you can use `fgetpos()` and `ftell()` to save the current position, and this position can later be used to reposition within the same file if opened in one of the modes where reposition is allowed.

For AMODE 31 C/C++ applications, the repositioning functions can be used with large format sequential data sets under the following conditions:

- The data set contains 256 TB - 256 bytes or less.
- The data set uses 2 GB blocks or less.

ungetc() considerations

For binary and text files, the library functions `fgetpos()` and `ftell()` take into account the number of characters you have pushed back onto the input stream with `ungetc()`, and adjust the file position accordingly. `ungetc()` backs up the file position by a single byte each time you call it. For text files, z/OS XL C/C++ counts the new-lines added to the records as single-byte characters when it calculates the file position.

If you make so many calls to `ungetc()` that the logical file position is before the beginning of the file, the next call to `ftell()` or `fgetpos()` fails.

When you are using `fseek()` with a whence value of `SEEK_CUR`, the starting point for the reposition also accounts for the presence of `ungetc()` characters and compensates as `ftell()` and `fgetpos()` do.

If you want `fgetpos()` and `fseek()` to ignore `ungetc()` characters, you can set the `_EDC_COMPAT` environment variable. See Chapter 28, “Using environment variables,” on page 327 for details. `ftell()` is not affected by the setting of `_EDC_COMPAT`.

How long fgetpos() and ftell() values last

As long as you do not re-create a file or shorten logical records, you can rely on the values returned by `ftell()` and `fgetpos()`, even across program boundaries and calls to `fclose()`. (Calling `fopen()` or `freopen()` with any of the `w` modes re-creates a file.) Using `ftell()` and `fgetpos()` values that point to information deleted or re-created results in undefined behavior. For more information about shortening records, see “Writing to variable-format text files” on page 60.

Using `fseek()` and `ftell()` in binary files

With binary files, `ftell()` returns two types of positions:

- Relative byte offsets
- Encoded offsets

Relative byte offsets

You get byte offsets by default when you are seeking or positioning in fixed-format binary files. You can also use byte offsets on a variable or undefined format file opened in binary mode with the `BYTESEEK` parameter specified on the `fopen()` or `freopen()` function call. You can specify `BYTESEEK` to be the default for `fopen()` calls by setting the environment variable `_EDC_BYTE_SEEK` to `Y`. See [Chapter 28, “Using environment variables,”](#) on page 327 for information on how to set environment variables.

You do not need to acquire an offset from `ftell()` to seek to a relative position; you may specify a relative offset to `fseek()` with a whence value of `SEEK_SET`. However, you cannot specify a negative offset to `fseek()` when you have specified `SEEK_SET`, because a negative offset would indicate a position before the beginning of the file. Also, you cannot specify a negative offset with whence values of `SEEK_CUR` or `SEEK_END` such that the resulting file position would be before the beginning of the file. If you specify such an offset, `fseek()` fails.

If your file is not opened read-only, you can specify a position that is beyond the current EOF. In such cases, a new end-of-file position is created; null characters are automatically added between the old EOF and the new EOF.

`fseek()` support of byte offsets in variable-format files generally requires reading all records from the whence value to the new position. The impact on performance is greatest if you open an existing file for append in `BYTESEEK` mode and then call `ftell()`. In this case, `ftell()` has to read from the beginning of the file to the current position to calculate the required byte offset. Support for byteseeeking is intended to ease portability from other platforms. If you need better performance, consider using `ftell()`-encoded offsets, see [“Encoded offsets”](#) on page 66.

For AMODE 31 C/C++ applications repositioning within a large format sequential data set that need `fseek()` and `ftell()` to access positions beyond 2 GB - 1 byte must use the large file version of `fseeko()` and `ftello()`.

Encoded offsets

If you do not specify the `BYTESEEK` parameter and you set the `_EDC_BYTE_SEEK` variable to `N`, any variable- or undefined-format binary file gets encoded offsets from `ftell()`. This keeps this release of z/OS XL C/C++ compatible with code generated by old releases of C/370.

Encoded offsets are values representing the block number and the relative byte within that block, all within one long `int`. Because z/OS XL C/C++ does not document its encoding scheme, you cannot rely on any encoded offset not returned by `ftell()`, except 0, which is the beginning of the file. This includes encoded offsets that you adjust yourself (for example, with addition or subtraction). When you call `fseek()` with the whence value `SEEK_SET`, you must use either 0 or an encoded offset returned from `ftell()`. For whence values of `SEEK_CUR` and `SEEK_END`, however, you specify relative byte offsets. If you want to seek to a certain relative byte offset, you can use `SEEK_SET` with an offset of 0 to rewind the file to the beginning, and then you can use `SEEK_CUR` to specify the desired relative byte offset.

In earlier releases, `ftell()` could determine position only for files with no more than 131,071 blocks. In the new design, this number increases depending on the block size. From a maximum block size of 32,760, every time this number decreases by half, the number of blocks that can be represented doubles. Using the large file version of `ftello()` for a large format sequential data set increases the maximum number of blocks that can be represented to 2 GB in AMODE 31 C/C++ applications.

If your file is not opened read-only, you can use `SEEK_CUR` or `SEEK_END` to specify a position that is beyond the current EOF. In such cases, a new end-of-file position is created; null characters are automatically added between the old EOF and the new EOF. This does not apply to PDS members, as they

cannot be extended. For `SEEK_SET`, because you are restricted to using offsets returned by `ftell()`, any offset that indicates a position outside the current file is invalid and causes `fseek()` to fail.

Using `fseek()` and `ftell()` in text files (ASA and Non-ASA)

In text files, `ftell()` produces only encoded offsets. It returns a long `int`, in which the block number and the byte offset within the block are encoded. You cannot rely on any encoded offset not returned by `ftell()` except 0. This includes encoded offsets that you adjust yourself (for example, with addition or subtraction).

When you call `fseek()` with the whence value `SEEK_SET`, you must use an encoded offset returned from `ftell()`. For whence values of `SEEK_CUR` and `SEEK_END`, however, you specify relative byte offsets. If you want to seek to a certain relative byte offset, you can use `SEEK_SET` with an offset of 0 to rewind the file to the beginning, and then you can use `SEEK_CUR` to specify the desired relative byte offset. z/OS XL C/C++ counts new-line characters and skips to the next record each time it reads one.

Unlike binary files you cannot specify offsets for `SEEK_CUR` and `SEEK_END` that set the file position past the end of the file. Any offset that indicates a position outside the current file is invalid and causes `fseek()` to fail.

In earlier releases, `ftell()` could determine position only for files with no more than 131071 blocks. In the new design, this number increases depending on the block size. From a maximum block size of 32760, every time this number decreases by half, the number of blocks that can be represented doubles. Using the large file version of `ftello()` for a large format sequential data set increases the maximum number of blocks that can be represented to 2 GB in AMODE 31 C/C++ applications.

Repositioning flushes all updates before changing position. An invalid call to `fseek()` is now always treated as a flush. It flushes all updated records or all complete new records in the block, and leaves the file position unchanged. If the flush fails, any characters in the `ungetc()` buffer are lost. If a block contains an incomplete new record, the block is saved and will be completed by another write or by closing the file.

Using `fseek()` and `ftell()` in record files

For files opened with `type=record`, `ftell()` returns relative record numbers. The behavior of `fseek()` and `ftell()` is similar to that when you use relative byte offsets for binary files, except that the unit is a record rather than a byte. For example,

```
fseek(fp, -2, SEEK_CUR);
```

seeks backward two records from the current position.

```
fseek(fp, 6, SEEK_SET);
```

seeks to relative record 6. You do not need to get an offset from `ftell()`.

You cannot seek past the end or before the beginning of a file.

The first record of a file is relative record 0.

For AMODE 31 C/C++ applications repositioning within a large format sequential data set that need `fseek()` and `ftell()` to access positions beyond 2 GB - 1 record must use the large file version of `fseeko()` and `ftello()`.

Using `fseek()` and `ftell()` in blocked files

For files opened with `type=blocked`, `ftell()` returns relative block numbers. The behavior of `fseek()` and `ftell()` is similar to that when you use relative byte offsets for binary files, except that the unit is a block rather than a byte. For example,

```
fseek(fp, -2, SEEK_CUR);
```

seeks backward two blocks from the current position.

```
fseek(fp,6,SEEK_SET);
```

seeks to relative block 6. You do not need to get an offset from `ftell()`.

You cannot seek past the end or before the beginning of a file.

The first block of a file is relative block 0.

For AMODE 31 C/C++ applications repositioning within a large format sequential data set that need `fseek()` and `ftell()` to access positions beyond 2 GB - 1 block must use the large file version of `fseeko()` and `ftello()`.

Porting old C code that uses `fseek()` or `ftell()`

The encoding scheme used by `ftell()` in non-BYTESEEK mode in the z/OS XL C/C++ RTL is different from that used in the z/OS C/C++ runtime library prior to C/370 Release 2.2 and Language Environment prior to release 1.3.

- If your code obtains `ftell()` values and passes them to `fseek()`, the change to the encoding scheme should not affect your application. On the other hand, your application may not work if you have saved encoded `ftell()` values in a file and your application reads in these encoded values to pass to `fseek()`. For non-record I/O and non-blocked I/O files, you can set the environment variable `_EDC_COMPAT` with the `ftell()` encoding set to tell z/OS XL C/C++ that you have old `ftell()` values. Files opened for record I/O or blocked I/O do not support old `ftell()` values saved across the program boundary.
- In previous versions, the `fseek()` support for the `ftell()` encoding scheme inadvertently supported seeking from `SEEK_SET` with a byte offset up to 32K. This is no longer supported. Users of this support must change to `BYTESEEK` mode. You can do this without changing your source code; just use the `_EDC_BYTE_SEEK` environment variable.

Closing files

Use the `fclose()` library function to close a file. z/OS XL C/C++ automatically closes files on normal program termination and attempts to do so under abnormal program termination or abend. See [z/OS C/C++ Runtime Library Reference](#) for more information on this library function.

For files opened in fixed binary mode, incomplete records will be padded with null characters when you close the file.

For files opened in variable binary mode, incomplete records are flushed to the system. In a spanned file, closing a file can cause a zero-length segment to be written. This segment will still be part of the non-zero-length record. For files opened in undefined binary mode, any incomplete output is flushed on close.

Closing files opened in text mode causes any incomplete new record to be completed with a new-line character. All records not yet flushed to the file are written out when the file is closed.

For files opened for record I/O, closing causes all records not yet flushed to the file to be written out.

Note: If an application has locked a (FILE *) object (with `flockfile()` or `ftrylockfile()`), it is responsible for relinquishing the locked (FILE *) object (with `funlockfile()`) before calling `fclose()`. Failure to relinquish a locked (FILE *) object may cause deadlock or looping.

When `fclose()` is used to close a stream associated with a z/OS data set, some failures may be unrecoverable, and will result in an ABEND. These ABENDs may include I/O ABENDs of the form x14 and x37. Control will not be returned to the caller of `fclose()` to report the error. To process these types of errors, applications need to use z/OS Language Environment condition handling to receive control (see [z/OS Language Environment Programming Guide](#)), or register a signal handler for SIGABND (see [Chapter 24, "Handling error conditions, exceptions, and signals,"](#) on page 273).

If an application fails during `fclose()` with a x37 abend, and the application would like to recover and perform any functions not related to file I/O, the following technique can be used. Refer to [Figure 9 on page 70](#) for an example.

1. Register a signal handler for SIGABND and SIGIOERR.
2. `fopen()` the file. The NOSEEK option cannot be specified.
3. Manipulate the file as needed by the application.
4. When the application is done with the file, `fflush()` the file, before any `fclose()` is issued. This will ensure, if an x37 is going to occur during `fflush()` or `fclose()` processing, that the x37 occurs in the `fflush()`, before the `fclose()` occurs.
5. An x37 abend occurs during `fflush()`.
6. The signal handler will receive control.
7. Once inside the signal handler, any functions not related to file I/O may be performed.

```

/* example of signal handler */

#include <stdio.h>
#include <stdlib.h>
#include <dynit.h>
#include <signal.h>
#include <setjmp.h>

void sighandler();
jmp_buf env;
FILE *f;

int main()
{
    int rc;
    int s=80;
    int w;
    char buff 80 ="data";
    __dyn_t ip;

    redo:
    dyninit(&ip);
    ip.__dsname="MY.DATASET";
    ip.__status=__DISP_OLD;
    ip.__ddname="NAMEDD";
    ip.__conddisp=__DISP_CATLG;
    rc=dynalloc(&ip);

    f=fopen("DD:NAMEDD","wb");
    if (f==0)
    { perror("open error");
      return 12;
    }

    signal(SIGABND,sighandler);
    signal(SIGIOERR,sighandler);

    while (1)
    {
        if (setjmp(env))
        {
            dyninit(&ip);
            ip.__ddname="NAMEDD";
            ip.__conddisp=__DISP_CATLG;
            rc= dynfree(&ip);
            goto retry;
        }
        w=fwrite(buff,1,s,f);

        fflush(f);
        fclose(f);

        retry:
        goto redo;
    }

    void sighandler() {
        fclose(f);
        longjmp(env,1);
    }
}

```

Figure 9. Example of signal handler

Note: When an abend condition occurs, a write-to-programmer message about the abend is issued and your DCB abend exit is given control, provided there is an active DCB abend exit routine address in the exit list contained in the DCB being processed. If STOW called the end-of-volume routines to get secondary space to write an end-of-file mark for a PDS, or if the DCB being processed is for an indexed sequential data set, the DCB abend exit routine is not given control if an abend condition occurs. If the situation described above is encountered, the Language Environment DCB abend exit will not receive control, and therefore the signal handler routine in an application will not receive control for the x37 abend.

Renaming and removing files

You can remove or rename a z/OS data set that has an uppercase filename by using the `remove()` or `rename()` library functions, respectively. `rename()` and `remove()` both accept data set names. `rename()` does not accept ddnames, but `remove()` does. You can use `remove()` or `rename()` on

individual members or entire PDSs or PDSEs. If you use `rename()` for a member, you can change only the name of the member, not the name of the entire data set. To rename both the member and the data set, make two calls to `rename()`, one for the member and one for the whole PDS or PDSE.

fldata() behavior

The format of the `fldata()` function is as follows:

```
int fldata(FILE *file, char *filename, fldata_t
*info);
```

The `fldata()` function is used to retrieve information about an open stream. The name of the file is returned in `filename` and other information is returned in the `fldata_t` structure, shown in [Figure 10 on page 72](#). Values specific to this category of I/O are shown in the comment beside the structure element. Additional notes follow the figure. For more information on the `fldata()` function, refer to [z/OS C/C++ Runtime Library Reference](#).

```

struct __fileData {
    unsigned int    __recfmF : 1, /* */
                  __recfmV : 1, /* */
                  __recfmU : 1, /* */
                  __recfmS : 1, /* */
                  __recfmBlk : 1, /* */
                  __recfmASA : 1, /* */
                  __recfmM : 1, /* */
                  __dsorgP0 : 1, /* */
                  __dsorgPDsmem : 1, /* */
                  __dsorgPDSdir : 1, /* */
                  __dsorgPS : 1, /* */
                  __dsorgConcat : 1, /* */
                  __dsorgMem : 1, /* N/A -- always off */
                  __dsorgHiper : 1, /* N/A -- always off */
                  __dsorgTemp : 1, /* */
                  __dsorgVSAM : 1, /* N/A -- always off */
                  __dsorgHFS : 1, /* N/A -- always off */
                  __openmode : 2, /* one of: */
                                /* __TEXT */
                                /* __BINARY */
                                /* __RECORD */
                  __modeflag : 4, /* combination of: */
                                /* __READ */
                                /* __WRITE */
                                /* __APPEND */
                                /* __UPDATE */
                  __dsorgPDSE : 1, /* N/A -- always off */
                  __vsamRLS : 3, /* N/A */
                  #if __EDC_TARGET >= 0x41080000
                  __vsamEA : 1, /* */
                  __recfmB : 1, /* */
                  __reserve2 : 3; /* */
                  #else
                  __reserve3 : 5; /* */
                  #endif
    __device_t      __device; /* one of: */
                                /* __DISK */
                                /* __TAPE */
                                /* __PRINTER */
                                /* __DUMMY */
                                /* __OTHER */
    unsigned long    __blksize, /* */
                  __maxreclen; /* */
    union {
        struct {
            unsigned short __vsam_type; /* N/A */
            unsigned long __vsam_keylen; /* N/A */
            unsigned long __vsam_RKP; /* N/A */
        } __vsam;
        struct {
            unsigned short __disk_vsam_type; /* */
            unsigned char __disk_access_method; /* */
            unsigned char __disk_noseek_to_seek; /* */
            long __disk_reserve[2]; /* */
        } __disk;
    } __device_specific; /* */
    char * __dsname; /* */
    unsigned int __reserve4; /* */
};
typedef struct __fileData fldata_t;

```

Figure 10. fldata() Structure

Notes:

1. If you have opened the file by its data set name, *filename* is fully qualified, including quotation marks. If you have opened the file by ddname, *filename* is dd:ddname, without any quotation marks. The ddname is uppercase. If you specified a member on the `fopen()` or `freopen()` function call, the member is returned as part of *filename*.
2. Any of the `__recfm` bits may be set on for OS files.
3. The `__dsorgP0` bit will be set on only if you are reading a directory or member of a partitioned data set, either regular or extended, regardless of whether the member is specified on a DD statement or on the `fopen()` or `freopen()` function call. The `__dsorgPS` bit will be set on for all other OS files.
4. The `__dsorgPDSE` bit will be set when processing an extended partitioned data set (PDSE).
5. The `__dsorgConcat` bit will be set on for a concatenation of sequential data sets, but not for a concatenation of partitioned data sets.

6. The `__dsorgTemp` bit will be set on only if the file was created using the `tmpfile()` function.
7. The `__blksize` value may include BDW and RDWs.
8. The `__maxreclen` value may include the ASA character.
9. The `__recfm` bits and the `__blksize` and `__maxreclen` values correspond to the attributes of the open stream. They do not necessarily reflect the attributes of the existing data set.
10. The `__dsname` field is filled in for **__DISK** files with the data set name. The member name is added if the file is a member of a partitioned data set, either regular or extended. The `__dsname` value is uppercase unless the `asis` option was specified on the `fopen()` or `freopen()` function call. The `__dsname` field is set to `NULL` for all other OS files.

Chapter 9. Performing z/OS UNIX file system I/O operations

You can create the following file types in the z/OS UNIX file system:

- Regular
- Link
- Directory
- Character special
- FIFO

The Single UNIX Specification defines another type of file called STREAMS. Even though the system interfaces are provided, it is impossible to have a valid STREAMS file descriptor. These interfaces will always return a return code of -1 with `errno` set to indicate an error such as, `EBADF`, `EINVAL`, or `ENOTTY`.

z/OS UNIX file system streams follow the binary model, regardless of whether they are opened for text, binary, or record I/O. You can simulate record I/O by using new-line characters as record boundaries.

For information on the z/OS UNIX file system and access to files within it from other than the C or C++ language, see *z/OS UNIX System Services User's Guide*. For an introduction to and description of the behavior of a POSIX-defined file system, see Zlotnick, Fred, *The POSIX.1 Standard: A Programmer's Guide*, Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991.

This topic describes C I/O stream functions as they can be used within C++ programs. If you want to use the C++ I/O stream classes instead, see Chapter 4, “Using the Standard C++ Library I/O Stream Classes,” on page 21. For more detailed information, see *Standard C++ Library Reference*. For information about using wide-character I/O with z/OS XL C/C++, see Chapter 7, “z/OS XL C support for the double-byte character set,” on page 29.

Creating files

You can use library functions to create the following types of z/OS UNIX file system files.

- Regular Files
- Link and Symbolic Link Files
- Directory Files
- Character Special Files
- FIFO Files

Regular files

Use any of the following C functions to create regular files in the z/OS UNIX file system:

- `creat()`
- `fopen()`
- `freopen()`
- `open()`

For a description of these and other I/O functions, see *z/OS C/C++ Runtime Library Reference*.

Link and symbolic link files

Use either of the following C functions to create z/OS UNIX file system link or symbolic link files:

- `link()`
- `symlink()`

Directory files

Use the `mkdir()` C function to create a z/OS UNIX file system directory file.

Character special files

Use the `mknod()` C function to create a character special file in the z/OS UNIX file system; you must have superuser authority to create a character special file.

Other functions used for character special files are:

- `ptsname()`
- `grantpt()`
- `unlockpt()`
- `tcgetsid()`
- `ttynname()`
- `isatty()`

FIFO files

Use the `mkfifo()` C function to create a FIFO file (named pipe) in the z/OS UNIX file system.

To create an unnamed pipe, use the `pipe()` C function.

Opening files

This section discusses the use of the `fopen()` or `freopen()` library functions to open z/OS UNIX file system I/O files. You can also access z/OS UNIX file system files using low-level I/O `open()` function. See [“Low-level z/OS UNIX I/O” on page 88](#) for information about low-level I/O, and [z/OS C/C++ Runtime Library Reference](#) for information about any of the functions listed above.

The name of a z/OS UNIX file system file can include characters chosen from the complete set of character values, except for null characters. If you want a portable filename, then choose characters from the POSIX .1 portable filename character set.

The complete *pathname* can begin with a slash and be followed by zero, one, or more filenames, each separated by a slash. If a directory is included within the pathname, it may have one or more trailing slashes. Multiple slashes following one another are interpreted as one slash.

If your program is running under POSIX(ON), all valid POSIX names are passed with the `asix` `fopen()` parameter to the POSIX `open()` function.

You can access either z/OS UNIX file system files or MVS™ data sets from programs. Programs accessing files or data sets can be executed with either the POSIX(OFF) or POSIX(ON) runtime options. There are basic file naming rules that apply for z/OS UNIX file system files and MVS data sets. However, there are also special z/OS XL C/C++ naming considerations that depend on how you execute your program.

The POSIX runtime option determines the type of z/OS XL C/C++ services and I/O available to your program. (See [z/OS XL C/C++ User's Guide](#) for a discussion of the z/OS UNIX programming environment and overview of binding z/OS UNIX XL C/C++ applications.)

Both the basic and special z/OS XL C/C++ file naming rules for z/OS UNIX file system files are described in the sections that follow. Examples are provided. All examples must be run with the POSIX(ON) option. For information about MVS data sets, see [Chapter 8, “Performing OS I/O operations,” on page 37](#).

If you specify `/dd:pathname` or `./dd:pathname`, a file named `dd:pathname` is opened in the file system root directory or your working directory, respectively. For example, if you code the following statement, the file `dd:parder` is opened in the root directory.

```
fopen("/dd:parder", "w+")
```

For z/OS UNIX file system files, leading and trailing white spaces are *significant*.

Opening a file by name

Which type of file (z/OS UNIX file system or MVS data set) you open may depend on whether the z/OS XL C/C++ application program is running under POSIX(ON).

For an application program that is to be run under POSIX(ON), you can include in your program statements similar to the following to open the file `parts.instock` for reading in the working directory:

```
FILE *stream;  
stream = fopen("parts.instock", "r");
```

To open the MVS data set `user-prefix.PARTS.INSTOCK` for reading, include statements similar to the following in your program:

```
FILE *stream;  
stream = fopen("//parts.instock", "r");
```

For an application program that is to be run as a traditional z/OS XL C/C++ application program, with POSIX(OFF), to open the MVS data set `user-prefix.PARTS.INSTOCK` for reading, include statements similar to the following in your program:

```
FILE *stream;  
stream = fopen("parts.instock", "r");
```

To open the file `parts.instock` in the working directory for reading, include statements similar to the following in your program:

```
FILE *stream;  
stream = fopen("./parts.instock", "r");
```

Opening a file by DDname

The DD statement enables you to write z/OS XL C/C++ source programs that are independent of the files and I/O devices they will use. You can modify the parameters of a file or process different files without recompiling your program.

When `//dd:ddname` is specified to `fopen()` or `freopen()`, the z/OS XL C/C++ library looks to find and resolve the data definition information for the filename to open. If the data definition information points to an MVS data set, MVS data set naming rules are followed. If a z/OS UNIX file system file is indicated using the PATH parameter on the data definition statement, a `ddname` is resolved to the associated pathname.

Note: Use of the z/OS XL C/C++ `fork()` library function from an application program under z/OS UNIX does not replicate the data definition information of the parent process in the child process. Use of any of the `exec()` library functions deallocates the data definition information for the application process.

For the declaration just shown for the file `parts.instock`, you should write a JCL DD statement similar to the following:

```
//PSTOCK DD PATH='/u/parts.instock',...
```

For more information on writing DD statements, you should refer to the job control language (JCL) manual [z/OS MVS JCL Reference](#).

To open the file by DD name under TSO/E, you must write an ALLOCATE command. For the declaration of a file parts.instock, you should write a TSO/E ALLOCATE command similar to the following:

```
ALLOCATE DDNAME(PSTOCK) PATH('/u/parts.instock')...
```

See [z/OS TSO/E Command Reference](#) for more information on TSO ALLOCATE.

fopen() and freopen() parameters

The following table lists the parameters that are available on the `fopen()` and `freopen()` functions, tells you which ones are useful for z/OS UNIX file system I/O, and lists the values that are valid for the applicable ones.

Table 13. Parameters for the `fopen()` and `freopen()` functions for z/OS UNIX file system I/O

Parameter	Allowed?	Applicable?	Notes
recfm=	Yes	No	z/OS UNIX file system I/O uses a continuous stream of data as its file format.
lrecl=	Yes	No	z/OS UNIX file system I/O uses a continuous stream of data as its file format.
blksize=	Yes	No	z/OS UNIX file system I/O uses a continuous stream of data as its file format.
space=	Yes	No	Not used for z/OS UNIX file system I/O.
type=	Yes	Yes	May be omitted. If you do specify it, type=record is the only valid value.
acc=	Yes	No	Not used for z/OS UNIX file system I/O.
password=	Yes	No	Not used for z/OS UNIX file system I/O.
asis	Yes	No	Not used for z/OS UNIX file system I/O.
bytesseek	Yes	No	Not used for z/OS UNIX file system I/O.
noseek	Yes	No	Not used for z/OS UNIX file system I/O.
OS	Yes	No	Not used for z/OS UNIX file system I/O.
abend=	Yes	No	Not used for z/OS UNIX file system I/O.

recfm=

Ignored for z/OS UNIX file system I/O.

lrecl= and blksize=

Ignored for z/OS UNIX file system I/O, except that `lrecl` affects the value returned in the `__maxreclen` field of `fldata()` as described below.

acc=

Ignored for z/OS UNIX file system I/O.

password

Ignored for z/OS UNIX file system I/O.

space=

Ignored for z/OS UNIX file system I/O.

type=

The only valid value for this parameter under the z/OS UNIX file system is `type=record`. If you specify this, your file follows the z/OS UNIX file system record I/O rules:

1. One record is defined to be the data up to the next new-line character.
2. When an `fread()` is done the data will be copied into the user buffer as if an `fgets(buf, size_item*num_items, stream)` were issued. Data is read into the user buffer *up to* the

number of bytes specified on the `fread()`, or until a new-line character or EOF is found. The new-line character is not included.

3. When an `fwrite()` is done the data will be written from the user buffer with a new-line character added by the RTL code. Data is written up to the number of bytes specified on the `fwrite()`; the new-line is added by the RTL and is not included in the return value from `fwrite()`.
4. If you have specified an `lrecl` and `type=record`, `fldata()` of this stream will return the `lrecl` you specified, in the `__maxreclen` field of the `__fileData` return structure of `stdio.h`. If you specified `type=record` but no `lrecl`, the `__maxreclen` field will contain 1024.

If `type=record` is not in effect, a call to `fldata()` of this stream will return 0 in the `__maxreclen` field of the `__fileData` return structure of `stdio.h`.

asis

Ignored for z/OS UNIX file system I/O.

byteseek

Ignored for z/OS UNIX file system I/O.

noseek

Ignored for z/OS UNIX file system I/O.

os

Ignored for z/OS UNIX file system I/O.

abend=

Ignored for z/OS UNIX file system I/O.

Reading from z/OS UNIX file system files

You can use the following library functions to read in information from z/OS UNIX file system files:

- `fread()`
- `fread_unlocked()`
- `fgets()`
- `fgets_unlocked()`
- `gets()`
- `gets_unlocked()`
- `fgetc()`
- `fgetc_unlocked()`
- `getc()`
- `getc_unlocked()`
- `getchar()`
- `getchar_unlocked()`
- `scanf()`
- `scanf_unlocked()`
- `fscanf()`
- `fscanf_unlocked()`
- `read()`
- `pread()`
- `vscanf()`
- `vscanf_unlocked()`
- `vfscanf()`
- `vfscanf_unlocked()`

`fread()` is the only interface allowed for reading record I/O files. See [z/OS C/C++ Runtime Library Reference](#) for more information on all of the above library functions.

For z/OS UNIX low-level I/O, you can use the `read()` and `readv()` function. See [“Low-level z/OS UNIX I/O”](#) on page 88.

Opening and reading from z/OS UNIX file system directory files

To open a z/OS UNIX file system directory, you can use the `opendir()` function.

You can use the following library functions to read from, and position within, z/OS UNIX file system directories:

- `readdir()`
- `seekdir()`
- `tellldir()`

To close a directory, use the `closedir()` function.

Writing to z/OS UNIX file system files

You can use the following library functions to write to z/OS UNIX file system files:

- `fwrite()`
- `fwrite_unlocked()`
- `printf()`
- `printf_unlocked()`
- `fprintf()`
- `fprintf_unlocked()`
- `vprintf()`
- `vprintf_unlocked()`
- `fprintf()`
- `fprintf_unlocked()`
- `puts()`
- `puts_unlocked()`
- `fputs()`
- `fputs_unlocked()`
- `fputc()`
- `fputc_unlocked()`
- `putc()`
- `putc_unlocked()`
- `putchar()`
- `putchar_unlocked()`
- `write()`
- `pwrite()`

`fwrite()` is the only interface allowed for writing to record I/O files. See [z/OS C/C++ Runtime Library Reference](#) for more information on all of the above library functions. For z/OS UNIX low-level I/O, you can use the `write()` and `writew()` function.

Flushing records

You can use the library function `fflush()` to flush streams to the system. For more information about `fflush()`, see [z/OS C/C++ Runtime Library Reference](#).

The action taken by the `fflush()` library function depends on the buffering mode associated with the stream and the type of streams. If you call one z/OS XL C/C++ program from another z/OS XL C/C++ program by using the ISO C/C++ `system()` function, all open streams are flushed before control is passed to the callee, and again before control is returned to the caller. A call to the POSIX `system()` function does not flush any streams.

For z/OS UNIX file system files, the `fflush()` function copies the data from the runtime buffer to the file system. The `fsync()` function copies the data from the file system buffer to the storage device.

Setting positions within files

You can use the following library functions to help you reposition within a regular file:

- `fseek()`
- `fseek_unlocked()`
- `fseeko()`
- `fseeko_unlocked()`
- `ftell()`
- `ftell_unlocked()`
- `ftello()`
- `ftello_unlocked()`
- `fgetpos()`
- `fgetpos_unlocked()`
- `fsetpos()`
- `fsetpos_unlocked()`
- `lseek()`
- `rewind()`
- `rewind_unlocked()`

With large file support enabled for AMODE 31 C/C++ applications, you can use the following library functions for 64-bit offsets:

- `fseeko()`
- `fseeko_unlocked()`
- `ftello()`
- `ftello_unlocked()`
- `lseek()`

For AMODE 64 C/C++ applications, large files are automatically supported in the LP64 programming model. All of the above functions (both lists) can be used with 64-bit offsets.

See [z/OS C/C++ Runtime Library Reference](#) for more information on these library functions.

Closing files

You can use `fclose()`, `freopen()`, or `close()` to close a file. z/OS XL C/C++ automatically closes files on normal program termination, and attempts to do so under abnormal program termination or abend.

See [z/OS C/C++ Runtime Library Reference](#) for more information on these library functions. For z/OS UNIX

low-level I/O, you can use the `close()` function. When you use any `exec()` or `fork()` function, files defined as "marked to be closed" are closed before control is returned.

Deleting files

Use the `unlink()` or `remove()` z/OS XL C/C++ function to delete the following types of z/OS UNIX file system files:

- Regular
- Character special
- FIFO
- Link files

Use the `rmdir()` z/OS XL C/C++ function to delete a z/OS UNIX file system directory file. See [z/OS C/C++ Runtime Library Reference](#) for more information about these functions.

Pipe I/O

POSIX.1 pipes represent an I/O channel that processes can use to communicate with other processes. Pipes are conceptually like z/OS UNIX file system files. One process can write data into a pipe, and another process can read data from the pipe.

z/OS UNIX XL C/C++ supports two types of POSIX.1-defined pipes: unnamed pipes and named pipes (FIFO files).

An *unnamed pipe* is accessible only by the process that created the pipe and its child processes. An unnamed pipe does not have to be opened before it can be used. It is a temporary file that lasts only until the last file descriptor that references it is closed. You can create an unnamed pipe by calling the `pipe()` function.

A *named pipe* can be used by independent processes and must be explicitly opened and closed. Named pipes are also referred to as first-in, first-out (FIFO) files, or FIFOs. You can create a named pipe by calling the `mkfifo()` function. If you want to stream I/O after a `pipe()` function, call the `fdopen()` function to build a stream on one of the file descriptors returned by `pipe()`. If you want to stream I/O on a FIFO file, open the file with `fdopen()` together with one of `fopen()`, `freopen()`, or `open()`. When the stream is built, you can then use Standard C I/O functions, such as `fgets()` or `printf()`, to carry out input and output.

Using unnamed pipes

If your z/OS UNIX XL C/C++ application program forks processes that need to communicate among themselves for work to be done, you can take advantage of POSIX.1-defined unnamed pipes. If your application program's processes need to communicate with other processes that it did not fork, you should use the POSIX.1-defined named pipe (FIFO special file) support. See [“Using named pipes” on page 84](#) for more information.

When you code the `pipe()` function to create a pipe, you pass a pointer to a two-element integer array where `pipe()` puts the file descriptors it creates. One descriptor is for the input end of the pipe, and the other is for the output end of the pipe. You can code your application so that one process writes data to the input end of the pipe and another process reads from the output end on a first-in-first-out basis. You can also build a stream on the pipe by using `fdopen()`, and use buffered I/O functions. The result is that you can communicate data between a parent process and any of its child processes.

The opened pipe is assigned the two lowest-numbered file descriptors available.

z/OS UNIX provides no security checks for unnamed pipes, because such a pipe is accessible only by the parent process that creates the pipe and any of the parent process's descendent processes. When the parent process ends, an unnamed pipe created by the process can still be used, if needed, by any existing descendant process that has an open file descriptor for the pipe.

Consider the sample program (CCNGHF1) in [Figure 11 on page 84](#). In this example, where you open a pipe, do a write operation, and later do a read operation from the pipe. For more information on the `pipe()` function and the file I/O functions, see [z/OS C/C++ Runtime Library Reference](#).

```
/* this example shows how unnamed pipes may be used */

#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main() {
    int ret_val;
    int pfd[2];
    char buff[32];
    char string1[]="String for pipe I/O";

    ret_val = pipe(pfd);                /* Create pipe */
    if (ret_val != 0) {                 /* Test for success */
        printf("Unable to create a pipe; errno=%d\n",errno);
        exit(1);                       /* Print error message and exit */
    }
    if (fork() == 0) {
        /* child program */
        close(pfd[0]); /* close the read end */
        ret_val = write(pfd[1],string1,strlen(string1)); /*Write to pipe*/
        if (ret_val != strlen(string1)) {
            printf("Write did not return expected value\n");
            exit(2);                   /* Print error message and exit */
        }
    }
    else {
        /* parent program */
        close(pfd[1]); /* close the write end of pipe */
        ret_val = read(pfd[0],buff,strlen(string1)); /* Read from pipe */
        if (ret_val != strlen(string1)) {
            printf("Read did not return expected value\n");
            exit(3);                   /* Print error message and exit */
        }
        printf("parent read %s from the child program\n",buff);
    }
    exit(0);
}
```

Figure 11. Unnamed pipes example

Using named pipes

If the z/OS UNIX XL C/C++ application program you are developing requires its active processes to communicate with other processes that are active but may not be from the same program, code your application program to create a *named pipe* (FIFO file). Named pipes allow transfer of data between processes in a FIFO manner and synchronization of process execution. Use of a named pipe allows processes to communicate even though they do not know what processes are on the other end of the pipe. Named pipes differ from standard unnamed pipes, created using the `pipe()` function, in that they involve the creation of a real file that is available for I/O operations to properly authorized processes.

Within the application program, you create a named pipe by coding a `mkfifo()` or `mknod()` function. You give the FIFO a name and an access mode when you create it. If the access mode allows all users read and write access to the named pipe, any process that knows its name can use it to send or receive data.

Processes can use the `open()` function to access named pipes and then use the regular I/O functions for files, such as `read()`, `write()`, and `close()`, when manipulating named pipes. Buffered I/O functions can also be used to access and manipulate named pipes. For more information on the `mkfifo()` and `mknod()` functions and the file I/O functions, see [z/OS C/C++ Runtime Library Reference](#).

Restriction: If `fopen()` is used to open named pipes in a multi-threaded environment, a deadlock will occur. This deadlock is caused by a named pipe waiting for the other end of the pipe to be opened, while still holding the `fopen()` multi-thread mutex. To prevent this deadlock, use `open()` to open the named pipe, instead of `fopen()`.

z/OS UNIX does security checks on named pipes.

The following steps outline how to use a named pipe from z/OS UNIX XL C/C++ application programs:

1. Create a named pipe using the `mkfifo()` function. Only one of the processes that use the named pipe needs to do this.
2. Access the named pipe using the appropriate I/O method.
3. Communicate through the pipe with another process using file I/O functions:
 - a. Write data to the named pipe.
 - b. Read data from the named pipe.
4. Close the named pipe.
5. If the process created the named pipe and the named pipe is no longer needed, remove that named pipe using the `unlink()` function.

A process running the following simple example program creates a new named pipe with the file pathname pointed to by the `path` value coded in the `mkfifo()` function. The access mode of the new named pipe is initialized from the mode value coded in the `mkfifo()` function. The file permission bits of the mode argument are modified by the process file creation mask.

As an example, a process running the program code (CCNGHF2) in [Figure 12 on page 86](#) creates a child process and then creates a named pipe called `fifo.test`. The child process then writes a data string to the pipe file. The parent process reads from the pipe file and verifies that the data string it reads is the expected one.

Note: The two processes are related and have agreed to communicate through the named pipe. They need not be related, however. Other authorized users can run the same program and participate in (or interfere with) the process communication.

```

/* this example shows how named pipes may be used */
#define _OPEN_SYS
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>
/*
 * Sample use of mkfifo()
 */
main()

{
    /* start of program */

    int flags, ret_value, c_status;
    pid_t pid;
    size_t n_elements;
    char char_ptr[32];
    char str[] = "string for fifo ";
    char fifoname[] = "temp.fifo";
    FILE *rd_stream,*wr_stream;

    if ((mkfifo(fifoname,S_IRWXU)) != 0) {
        printf("Unable to create a fifo; errno=%d\n",errno);
        exit(1);
        /* Print error message and return */
    }

    if ((pid = fork()) < 0) {
        perror("fork failed");
        exit(2);
    }

    if (pid == (pid_t)0) { /* CHILD process */
        /* issue fopen for write end of the fifo */
        wr_stream = fopen(fifoname,"w");
        if (wr_stream == (FILE *) NULL) {
            printf("In child process\n");
            printf("fopen returned a NULL, expected valid stream\n");
            exit(100);
        }

        /* perform a write */
        n_elements = fwrite(str,1,strlen(str),wr_stream);
        if (n_elements != (size_t) strlen(str)) {
            printf("Fwrite returned %d, expected %d\n",
                (int)n_elements,strlen(str));
            exit(101);
        }
        exit(0);
        /* return success to parent */
    }

    else { /* PARENT process */
        /* issue fopen for read */
        rd_stream = fopen(fifoname,"r");
        if (rd_stream == (FILE *) NULL) {
            printf("In parent process\n");
            printf("fopen returned a NULL, expected valid pointer\n");
            exit(2);
        }
    }
}

```

Named pipes example (Part 1 of 2)

Figure 12. Named pipes example

```

/* get current flag settings of file */
if ((flags = fcntl(fileno(rd_stream),F_GETFL)) == -1) {
    printf("fcntl returned -1 for %s\n",fifoname);
    exit(3);
}

/* clear O_NONBLOCK and reset file flags */
flags &= (O_NONBLOCK);
if ((fcntl(fileno(rd_stream),F_SETFL,flags)) == -1) {
    printf("\nfcntl returned -1 for %s",fifoname);
    exit(4);
}

/* try to read the string */
ret_value = fread(char_ptr,sizeof(char),strlen(str),rd_stream);
if (ret_value != strlen(str)) {
    printf("\nFread did not read %d elements as expected ",
        strlen(str));
    printf("\nret_value is %d ",ret_value);
    exit(6);
}

if (strcmp(char_ptr,str,strlen(str))) {
    printf("\ncontents of char_ptr are %s ",
        char_ptr);
    printf("\ncontents of str are %s ",
        str);
    printf("\nThese should be equal");
    exit(7);
}

ret_value = fclose(rd_stream);
if (ret_value != 0) {
    printf("\nFclose failed for %s",fifoname);
    printf("\nerrno is %d",errno);
    exit(8);
}
ret_value = remove(fifoname);
if (ret_value != 0) {
    printf("\nremove failed for %s",fifoname);
    printf("\nerrno is %d",errno);
    exit(9);
}

pid = wait(c_status);
if ((WIFEXITED(c_status) !=0) && (WEXITSTATUS(c_status) !=0)) {
    printf("\nchild exited with code %d",WEXITSTATUS(c_status));
    exit(10);
}
} /* end of else clause */
printf("About to issue exit(0), \
    processing completed successfully\n");
exit(0);
}

```

Named pipes example (Part 2 of 2)

Character special file I/O

A named pipe (FIFO file) is a type of character special file. Therefore, it obeys the I/O rules for character special files rather than the rules for regular files:

- It cannot be opened in read/write mode. A process must open a named pipe in either write-only or read-only mode.
- It must be opened in read mode by a process before it can be opened in write mode by another process. Otherwise, the file is blocked from use for I/O by processes. Blocked processes can cause an application program to hang.

A single process intending to access a named pipe can use an `open()` function with `O_NONBLOCK` to open the read end of the named pipe. It can then open the named pipe in write mode.

Note: The `fopen()` function cannot be used to accomplish this.

Low-level z/OS UNIX I/O

Low-level z/OS UNIX I/O is the POSIX.1-defined I/O method. All input and output is processed using the defined `read()`, `readv()`, `write()`, and `writenv()` functions.

For application programmers used to a UNIX environment, z/OS UNIX behaves in familiar and predictable ways. Standard UNIX programming practices for shared resources, along with designing applications to respect locks put on files by multiple threads running in a process, will ensure that data is handled predictably.

For a discussion of POSIX.1-defined low-level I/O and some of the practical considerations to take into account when designing an application, see *The POSIX.1 Standard: A Programmer's Guide*, by Fred Zlotnick (Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991).

Example of z/OS UNIX file system I/O functions

This section contains examples that demonstrate different uses of z/OS UNIX I/O file system functions

Figure 13 on page 88 is example code (CCNGHF3) that demonstrates the use of z/OS UNIX stream input/output by writing streams to a file, reading the input lines, and replacing a line.

```
/* this example uses HFS stream I/O */

#define _OPEN_SYS
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#undef _OPEN_SYS
FILE *stream;

char string1[] = "A line of text."; /* NOTE: There are actually 16 */
char string2[] = "Find this line."; /* characters in each line of */
char string3[] = "Another stream."; /* text. The 16th is a null */
char string4[16]; /* terminator on each string. */
long position, strpos; /* Since the null character */
int i, result, fd; /* is not being written to */
int rc; /* the file, 15 is used as */
/* the data stream length. */

ssize_t x;
char buffer[16];
```

Example of z/OS UNIX stream input and output functions (Part 1 of 3)

Figure 13. Example of z/OS UNIX stream input and output functions

```

int main(void)
{
    /* Write continuous streams to file */

    if ((stream = fopen("./myfile.data","wb"))==NULL) {
        perror("Error opening file");
        exit(0);
    }

    for(i=0; i<12;i++) {
        int len1 = strlen(string1);
        rc = fwrite(string1, 1, len1, stream);
        if (rc != len1) {
            perror("fwrite failed");
            printf("i = %d\n", i);
            exit(99);
        }
    }
    rc = fwrite(string2,1,sizeof(string2)-1,stream);

    if (rc != sizeof(string2)-1) {
        perror("fwrite failed");
        exit(99);
    }

    for(i=0;i<12;i++) {
        rc = fwrite(string1,1,sizeof(string1)-1,stream);

        if (rc != sizeof(string1)-1) {
            perror("fwrite failed");
            printf("i = %d\n", i);
            exit(99);
        }
    }
    fclose(stream);
    /* Read data stream and search for location of string2.      */
    /* EOF is not set until an attempt is made to read past the */
    /* end-of-file, thus the fread is at the end of the while loop */

    stream = fopen("./myfile.data", "rb");

    if ((position = ftell(stream)) == -1L)
        perror("Error saving file position.");
    rc = fread(string4, 1, sizeof(string2)-1, stream);
}

```

Example of z/OS UNIX stream input and output functions (Part 2 of 3)

```

while(!feof(stream)) {
    if (rc != sizeof(string2)-1) {
        perror("fread failed");
        exit(99);
    }

    if (strstr(string4,string2) != NULL) /* If string2 is found */
        stpos = position ;              /* then save position. */

    if ((position=ftell(stream)) == -1L)
        perror("Error saving file position.");
    rc = fread(string4, 1, sizeof(string2)-1, stream);
}

fclose(stream);
/* Replace line containing string2 with string3 */

fd = open("test.data",O_RDWR);

if (fd < 0){
    perror("open failed\n");
}

x = write(fd,"a record",8);

if (x < 8){
    perror("write failed\n");
}

rc = lseek(fd,0,SEEK_SET);
x = read(fd,buffer,8);

if (x < 8){
    perror("read failed\n");
}
printf("data read is %.8s\n",buffer);

close(fd);
}

```

Example of z/OS UNIX stream input and output functions (Part 3 of 3)

To use 64-bit offset and file sizes, you must make the following changes in your code:

1. Change any variables used for offsets in `fseek()` or `ftell()` that are `int` or `long` to the `off_t` data type.
2. Define the `_LARGE_FILES 1` feature test macro.
3. Replace `fseek()/ftell()` with `fseeko()/ftello()`. See [z/OS C/C++ Runtime Library Reference](#) for descriptions of these functions.
4. Compile with the `LANGVL(LONGLONG)` compiler option.

Notes:

1. These changes are compatible with your older files.
2. Large Files support (64-bit offset and file sizes) is automatic in the LP64 programming model that is used in 64-bit. The `long` data type is widened to 64-bits. This enables `fseek()` and `ftell()` to work with the larger offsets with no code change. The `fseeko()` and `ftello()` functions also work with 64-bit offsets since `off_t` is typedef'd as a `long int`.

The example program (CCNGHF4) in [Figure 14 on page 91](#) provides the same function as CCNGHF3, but it uses 64-bit offsets. The changed lines are marked in a **bold** font.

```

/* this example uses HFS stream I/O and 64-bit offsets*/

#define _OPEN_SYS
#define _LARGE_FILES 1
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#undef _OPEN_SYS
FILE *stream;

char string1[] = "A line of text."; /* NOTE: There are actually 16 */
char string2[] = "Find this line."; /* characters in each line of */
char string3[] = "Another stream."; /* text. The 16th is a null */
char string4[16]; /* terminator on each string. */
off_t position, strpos; /* Since the null character */
int i, result, fd; /* is not being written to */
int rc; /* the file, 15 is used as */
/* the data stream length. */

ssize_t x;
char buffer[16];

int main(void)
{
    /* Write continuous streams to file */

    if ((stream = fopen("./myfile.data", "wb")) == NULL) {
        perror("Error opening file");
        exit(0);
    }

    for(i=0; i<12; i++) {
        int len1 = strlen(string1);
        rc = fwrite(string1, 1, len1, stream);
        if (rc != len1) {
            perror("fwrite failed");
            printf("i = %d\n", i);
            exit(99);
        }
    }
    rc = fwrite(string2, 1, sizeof(string2)-1, stream);

    if (rc != sizeof(string2)-1) {
        perror("fwrite failed");
        exit(99);
    }

    for(i=0; i<12; i++) {
        rc = fwrite(string1, 1, sizeof(string1)-1, stream);

        if (rc != sizeof(string1)-1) {
            perror("fwrite failed");
            printf("i = %d\n", i);
            exit(99);
        }
    }
}

```

Example of HFS stream input and output functions (Part 1 of 2)

Figure 14. Example of HFS stream input and output functions

```

fclose(stream);
/* Read data stream and search for location of string2.      */
/* EOF is not set until an attempt is made to read past the */
/* end-of-file, thus the fread is at the end of the while loop */

stream = fopen("./myfile.data", "rb");

if ((position=ftello(stream)) == -1LL)
    perror("Error saving file position.");

rc = fread(string4, 1, sizeof(string2)-1, stream);

while(!feof(stream)) {
    if (rc != sizeof(string2)-1) {
        perror("fread failed");
        exit(99);
    }

    if (strstr(string4,string2) != NULL) /* If string2 is found */
        strpos = position ;             /* then save position. */

    if ((position=ftello(stream)) == -1LL)
        perror("Error saving file position.");

    rc = fread(string4, 1, sizeof(string2)-1, stream);
}

fclose(stream);
/* Replace line containing string2 with string3 */

fd = open("test.data",O_RDWR);

if (fd < 0){
    perror("open failed\n");
}

x = write(fd,"a record",8);

if (x < 8){
    perror("write failed\n");
}
strpos = lseek(fd,0LL,SEEK_SET);      /* Note off_t is 64bits with _LARGE_FILES */
/*      set and the off_t variable      */
/*      needs a 64bit constant of 0LL */

x = read(fd,buffer,8);

if (x < 8){
    perror("read failed\n");
}
printf("data read is %.8s\n",buffer);

close(fd);
}

```

Example of HFS stream input and output functions (Part 2 of 2)

fldata() behavior

The format of the fldata() function is as follows:

```

int fldata(FILE *file, char *filename,
fldata_t
*info);

```

The fldata() function is used to retrieve information about an open stream. The name of the file is returned in *filename* and other information is returned in the fldata_t structure, shown in [Figure 15 on page 93](#). Values specific to this category of I/O are shown in the comment beside the structure element. Additional

notes pertaining to this category of I/O follow the figure. For more information on the `fldata()` function, refer to *z/OS C/C++ Runtime Library Reference*.

```
struct __fileData {
    unsigned int  __recfmF      : 1, /* always off          */
                  __recfmV      : 1, /* always off          */
                  __recfmU      : 1, /* always on           */
                  __recfmS      : 1, /* always off          */
                  __recfmBlk     : 1, /* always off          */
                  __recfmASA     : 1, /* always off          */
                  __recfmM      : 1, /* always off          */
                  __dsorgP0      : 1, /* N/A -- always off  */
                  __dsorgPDSmem : 1, /* N/A -- always off  */
                  __dsorgPDSdir : 1, /* N/A -- always off  */
                  __dsorgPS      : 1, /* N/A -- always off  */
                  __dsorgConcat : 1, /* N/A -- always off  */
                  __dsorgMem     : 1, /* N/A -- always off  */
                  __dsorgHiper   : 1, /* N/A -- always off  */
                  __dsorgTemp    : 1, /* N/A -- always off  */
                  __dsorgVSAM    : 1, /* N/A -- always off  */
                  __dsorgHFS     : 1, /* always on           */
                  __openmode     : 2, /* one of:             */
                                /* __BINARY             */
                                /* __RECORD             */
    __modeflag    : 4, /* combination of:     */
                                /* __READ               */
                                /* __WRITE              */
                                /* __APPEND             */
                                /* __UPDATE             */
                  __dsorgPDSE    : 1, /* N/A -- always off  */
                  __reserve2     : 8; /*                     */
    __device_t    __device;     /* __HFS               */
    unsigned long __blksize;     /* 0                   */
    unsigned long __maxreclen;   /*                     */
    unsigned short __vsamtype;    /* N/A                 */
    unsigned long  __vsamkeylen;  /* N/A                 */
    unsigned long  __vsamRKP;     /* N/A                 */
    char *         __dsname;      /*                     */
    unsigned int   __reserve4;    /*                     */
};
typedef struct __fileData fldata_t;
```

Figure 15. `fldata()` structure

Notes:

1. The *filename* is the same as specified on the `fopen()` or `freopen()` function call.
2. The `__maxreclen` value is 0 for regular I/O (binary). For record I/O the value is `lrecl` or the default of 1024 when `lrecl` is not specified.
3. The `__dsname` value is the real POSIX pathname.

File tagging and conversion

In general, the file system knows the contents of a file only as a set of bytes. Applications which create and process bytes in a file know whether these bytes represent binary data, text (character) data, or a mixture of both. File tags are file metadata fields which describe the contents of a file. Enhanced ASCII includes the following file tag fields:

txtflag

A flag indicating if a file consists solely of character data encoded by a single coded character set ID (CCSID).

file ccsid

A 16 bit field specifying the CCSID of characters in the file.

Applications can explicitly tag files using `new open()` or `fcntl()` options, or applications can allow the logical file system (LFS) to tag new files on first write, `fopen()`. A new environment variable, `_BPXX_CCSID`, is used to assign a program CCSID to an application, which LFS will use to tag new files on first write. LFS also uses the program CCSID derived from `_BPXX_CCSID` to set up auto-conversion of pure text datastreams. LFS attempts to set up auto-conversion when:

- Auto-conversion is enabled for an application by the `_BPXK_AUTOCVT` environment variable
- The file `txtflag` flag is set indicating a pure text file
- The file and program CCSIDs do not match.

Automatic file conversion and file tagging include the following facilities:

- `_OPEN_SYS_FILE_EXT` feature test macro. For more information, see [z/OS C/C++ Runtime Library Reference](#).
- `_BPXK_AUTOCVT` and `_BPXK_CCIDS` environment variables. For more information, see [Chapter 28, “Using environment variables,”](#) on page 327.
- z/OS Language Environment FILETAG runtime option. For more information, see [z/OS Language Environment Programming Reference](#).
- `__chattr()` and `__fchattr()` functions; `F_SETTAG` and `F_CONTROL_CVT` arguments for the `fcntl()` function; options for the `fopen()`, `popen()`, `stat()`, `fstat()`, and `lstat()` functions. For more information, see [z/OS C/C++ Runtime Library Reference](#).

Access control lists (ACLs)

Access control lists (ACLs) enable you to control access to files and directories by individual user (UID) and group (GID). ACLs are used in conjunction with permission bits. You can create, modify, and delete ACLs using the following functions:

- `acl_create_entry()`
- `acl_delete_entry()`
- `acl_delete_fd()`
- `acl_delete_file()`
- `acl_first_entry()`
- `acl_free()`
- `acl_from_text()`
- `acl_get_entry()`
- `acl_get_fd()`
- `acl_get_file()`
- `acl_init()`
- `acl_set_fd()`
- `acl_set_file()`
- `acl_sort()`
- `acl_to_text()`
- `acl_update_entry()`
- `acl_valid()`

For descriptions of these functions see [z/OS C/C++ Runtime Library Reference](#). For more information on using ACLs to protect file system resources see [z/OS UNIX System Services Planning](#) and [z/OS Security Server RACF Security Administrator's Guide](#).

Chapter 10. Performing VSAM I/O operations

This chapter outlines the use of Virtual Storage Access Method (VSAM) data sets in z/OS XL C/C++. Three I/O processing modes for VSAM data sets are available in z/OS XL C/C++:

- Record
- Text Stream
- Binary Stream

Because VSAM is a record-based access method, record mode is the logical processing mode and is specified by coding the `type=record` keyword parameter on the `fopen()` function call. z/OS XL C/C++ also provides limited support for VSAM text streams and binary streams. Because of the record-based nature of VSAM, this chapter is organized differently from the other chapters in this section. The focus of this chapter is on record I/O, and only those aspects of text and binary I/O that are specific to VSAM are also discussed.

For more information about the facilities of VSAM, see the list of “DFSMS” on page 774 publications.

See Chapter 7, “z/OS XL C support for the double-byte character set,” on page 29 for information about using wide-character I/O with z/OS XL C/C++.

Notes:

1. This chapter describes C I/O as it can be used within C++ programs.
2. The C++ I/O stream libraries cannot be used for VSAM I/O because these do not support the record processing mode (where `type=record` is specified).
3. Starting in z/OS V1R10, the C/C++ runtime library provides support for VSAM data sets in the extended addressing space on extended address volumes (EAVs).

VSAM types (data set organization)

There are three types of VSAM data sets supported by z/OS XL C/C++, all of which are held on direct-access storage devices.

- Key-Sequenced Data Set (KSDS) is used when a record is accessed through a key field within the record (for example, an employee directory file where the employee number can be used to access the record). KSDS also supports sequential access. Each record in a KSDS must have a unique key value.
- Entry-Sequenced Data Set (ESDS) is used for data that is primarily accessed in the order it was created (or the reverse order). It supports direct access by Relative Byte Address (RBA), and sequential access.
- Relative Record Data Set (RRDS) is used for data in which each item has a particular number, and the relevant record is accessed by that number (for example, a telephone system with a record associated with each number). It supports direct access by Relative Record Number (RRN), and sequential access.

In addition to the primary VSAM access described above, for KSDS and ESDS, there is also direct access by one or more additional key fields within each record. These additional keys can be unique or non-unique; they are called an alternate index (AIX®).

Notes:

1. VSAM Linear Data Sets are not supported in z/OS XL C/C++ I/O.
2. z/OS XL C/C++ supports extended addressable KSDS, ESDS, and RRDS data sets. Extended addressable KSDS data sets can be accessed through an alternate index, but extended addressable ESDS data sets cannot.

Access method services

Access Method Services are generally known by the name IDCAMS on MVS. For more information, see [*z/OS DFSMS Access Method Services Commands*](#).

Before a VSAM data set is used for the first time, its structure is defined to the system by the Access Method Services `DEFINE CLUSTER` command. This command defines the type of VSAM data set, its structure, and the space it requires.

Before a VSAM alternate index is used for the first time, its structure is defined to the system by the Access Method Services `DEFINE ALTERNATEINDEX` command. To enable access to the base cluster records through the alternate index, use the `DEFINE PATH` command. Finally, to build the alternate index, use the `BLDINDEX` command.

When you have built the alternate index, you call `fopen()` and specify the `PATH` in order to access the base cluster through the alternate index. Do not use `fopen()` to access the alternate index itself.

Note: You cannot use the `BLDINDEX` command on an empty base cluster.

Choosing VSAM data set types

When you plan your program, you must first decide the type of data set to use. [Figure 16 on page 97](#) shows you the possibilities available with the types of VSAM data sets.

The diagrams show how the information contained in the family tree below could be held in VSAM data sets of different types.

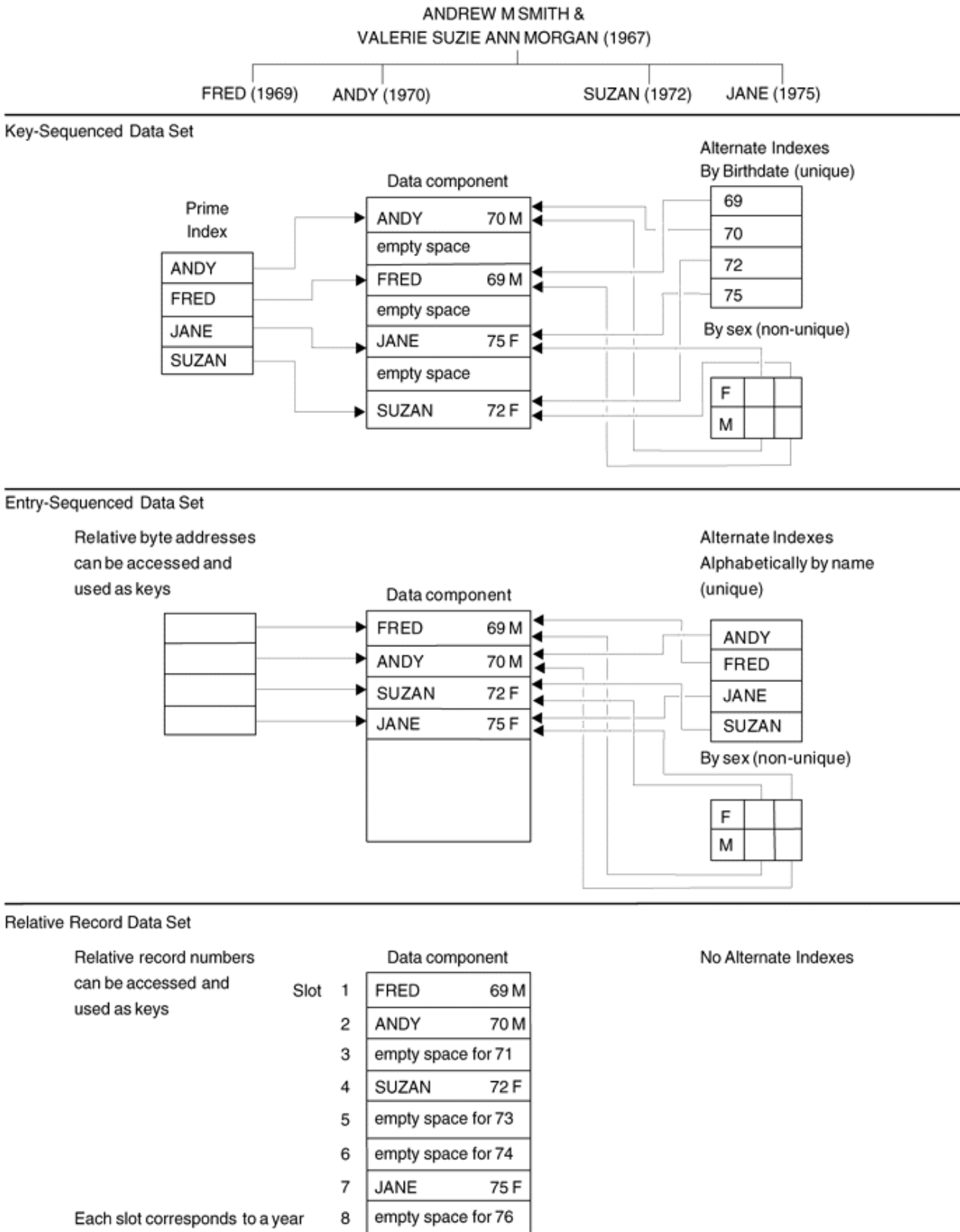


Figure 16. Types and advantages of VSAM data sets

When choosing the VSAM data set type, you should base your choice on the most common sequence in which you require data. You should follow a procedure similar to the one suggested below to help ensure a combination of data sets and indexes that provide the function you require.

1. Determine the type of data and its primary access.
 - sequentially — favors ESDS
 - by key — favors KSDS
 - by number — favors RRDS
2. Determine whether you require access through an alternate index path. These are only supported on KSDS and ESDS. If you do, determine whether the alternate index is to have unique or nonunique keys. You should keep in mind that making an assumption that all future records will have unique keys may not be practical, and an attempt to insert a record with a nonunique key in an index that has been created for unique keys causes an error.
3. When you have determined the data sets and paths that you require, ensure that the operations you have in mind are supported.

Keys, RBAs and RRNs

All VSAM data sets have keys associated with their records. For KSDS, KSDS AIX, and ESDS AIX, the key is a defined field within the logical record. For ESDS, the key is the *relative byte address* (RBA) of the record. For RRDS, the key is a *relative record number* (RRN).

Keys for indexed VSAM data sets

For KSDS, KSDS AIX, and ESDS AIX, keys are part of the logical records recorded on the data set. For KSDS, the length and location of the keys are defined by the `DEFINE CLUSTER` command of Access Method Services. For KSDS AIX and ESDS AIX, the keys are defined by the `DEFINE ALTERNATEINDEX` command.

Relative byte addresses

Relative byte addresses (RBAs) enable you to access ESDS files directly. The RBAs are either 4 or 8 byte fields, depending on the usage, and their values are computed by VSAM. The 4 byte RBA can only be used when accessing within the first 4GB of a VSAM data set. The 8 byte RBA can be used to access beyond 4GB in an extended addressable VSAM data set.

Notes:

1. KSDS can also use RBAs. However, because the RBA of a KSDS record can change if an insert, delete or update operation is performed elsewhere in the file, it is not recommended.
2. You can call `flocate()` with RBA values in an RRDS cluster, but `flocate()` with RBA values does not work across control intervals. Therefore, using RBAs with RRDS clusters is not recommended. The RRDS access method does not support RBAs. z/OS XL C/C++ supports the use of RBAs in an RRDS cluster by translating the RBA value to an RRN. It does this by dividing the RBA value by the `LRECL`.
3. Alternate indexes do not allow positioning by RBA.

The RBA value is stored in the C structure `__amrc`, which is defined in the C `<stdio.h>` header file. The `__amrc->__RBA` field is defined as an unsigned int, and therefore will contain only a 4-byte RBA value. The `__amrc->__XRBA` field is 8 bytes (unsigned long long in AMODE 31 applications, and unsigned long in AMODE 64 applications), and therefore can hold the RBA for all locations within an extended addressable VSAM data set.

You can access the field `__amrc->__RBA`, as shown in [<xref refid="gvs1">](#). This example code (CCNGVS1) can be converted to use `__amrc->__XRBA` with just a few modifications. For more information about the `__amrc` structure, refer to [Chapter 16, “Debugging I/O programs,” on page 161](#).

```

/* this example shows how to access the __amrc->__RBA field */
/* it assumes that an ESDS has already been defined, and has been */
/* assigned the ddname ESDSCLUS */

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

main() {
    FILE *ESDSfile;
    unsigned int myRBA;
    char recbuff[100]="This is record one.";
    int w_retc;
    int l_retc;
    int r_retc;

    printf("calling fopen(\"dd:esdsclus\", \"rb+,type=record\");\n");
    ESDSfile = fopen("dd:esdsclus", "rb+,type=record");
    printf("fopen() returned 0X%.8x\n", ESDSfile);
    if (ESDSfile==NULL) exit;

    w_retc = fwrite(recbuff, 1, sizeof(recbuff), ESDSfile);
    printf("fwrite() returned %d\n", w_retc);
    if (w_retc != sizeof(recbuff)) exit;
    myRBA = __amrc->__RBA;

    l_retc = flocate(ESDSfile, &myRBA, sizeof(myRBA), __RBA_EQ);
    printf("flocate() returned %d\n", l_retc);
    if (l_retc !=0) exit;

    r_retc = fread(recbuff, 1, sizeof(recbuff), ESDSfile);
    printf("fread() returned %d\n", r_retc);
    if (l_retc !=0) exit;

    return(0);
}

```

Figure 17. VSAM example

Relative record numbers

Records in an RRDS are identified by a relative record number that starts at 1 and is incremented by 1 for each succeeding record position. Only RRDS files support accessing a record by its relative record number.

Summary of VSAM I/O operations

Table 14 on page 99 summarizes VSAM data set characteristics and the allowable I/O operations on them.

Table 14. Summary of VSAM data set characteristics and allowable I/O operations

Characteristic or I/O Operation	KSDS	ESDS	RRDS
Record Length	Variable. Length can be changed by update.	Variable. Length cannot be changed by update.	Fixed.
Alternate index Note: z/OS XL C/C++ does not support extended addressable ESDS alternate indexes.	Allows access using unique or non-unique keys.	Allows access using unique or non-unique keys.	Not supported by VSAM.
Record Read (Sequential)	The order is determined by the VSAM key	By entry sequence. Reads proceed in key sequence for the key of reference.	By relative record number.
Record Write (Direct)	Position determined by the value in the field designated as the key.	Record written at the end of the file.	By relative record number.

Table 14. Summary of VSAM data set characteristics and allowable I/O operations (continued)

Characteristic or I/O Operation	KSDS	ESDS	RRDS
Positioning for Record Read	By key or by RBA value. Positioning by RBA value is not recommended because changes to the file change the RBA.	By RBA value. Alternate index allows use by key.	By relative record number.
Delete (Record)	If not already in correct position, reposition the file pointer; read the record using <code>fread()</code> ; delete the record using <code>fdelrec()</code> . <code>fread()</code> must immediately precede <code>fdelrec()</code> .	Not supported by VSAM.	If not already in correct position, position the file pointer; read the record using <code>fread()</code> ; delete the record using <code>fdelrec()</code> . <code>fread()</code> must immediately precede <code>fdelrec()</code> .
Update (Record)	If not already in correct position, reposition the file pointer; read the record using <code>fread()</code> ; update the record using <code>fupdate()</code> . <code>fread()</code> must immediately precede <code>fupdate()</code> .	If not already in correct position, reposition the file pointer; read the record using <code>fread()</code> ; update the record using <code>fupdate()</code> . <code>fread()</code> must immediately precede <code>fupdate()</code> .	If not already in correct position, reposition the file pointer; read the record using <code>fread()</code> ; update the record using <code>fupdate()</code> . <code>fread()</code> must immediately precede <code>fupdate()</code> .
Empty the file	Define the file as reusable using the <code>DEFINE CLUSTER</code> command, and then open the data set in write (" <code>wb,type=record</code> " or " <code>wb+,type=record</code> ") mode. Not supported for alternate indexes.	Define the file as reusable using the <code>DEFINE CLUSTER</code> command, and then open the data set in write (" <code>wb,type=record</code> " or " <code>wb+,type=record</code> ") mode. Not supported for alternate indexes.	Define the file as reusable using the <code>DEFINE CLUSTER</code> command, and then open the data set in write (" <code>wb,type=record</code> " or " <code>wb+,type=record</code> ") mode.
Stream Read	Supported by z/OS XL C/C++.	Supported by z/OS XL C/C++.	Supported by z/OS XL C/C++.
Stream Write/Update	Not supported by z/OS XL C/C++.	Supported by z/OS XL C/C++.	Supported by z/OS XL C/C++.
Stream Repositioning	Supported by z/OS XL C/C++.	Supported by z/OS XL C/C++.	Supported by z/OS XL C/C++.

Opening VSAM data sets

To open a VSAM data set, use the Standard C library functions `fopen()` and `freopen()` just as you would for opening non-VSAM data sets. The `fopen()` and `freopen()` functions are described in [z/OS C/C++ Runtime Library Reference](#).

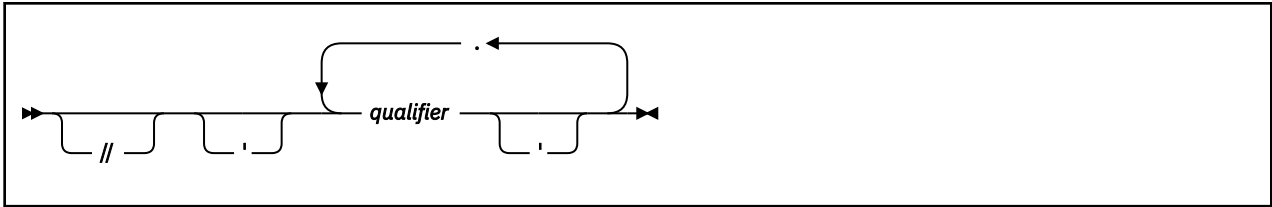
This section describes considerations for using `fopen()` and `freopen()` with VSAM files. Remember that a VSAM file must exist and be defined as a VSAM cluster before you call `fopen()`.

Using `fopen()` or `freopen()`

This section covers using file names for MVS data sets, specifying `fopen()` and `freopen()` keywords, and buffering.

File names for MVS data sets: Using a data set name

The following diagram shows the syntax for the *filename* argument on your `fopen()` or `freopen()` call:



The following is a sample construct:

```
'qualifier1.qualifier2'
```

'

Single quotation marks indicate that you are passing a *fully-qualified* data set name, that is, one which includes the high-level qualifier. If you pass a data set name without single quotation marks, the z/OS XL C/C++ compiler prefixes the high-level qualifier (usually the user ID) to the name. See [Chapter 8, “Performing OS I/O operations,”](#) on page 37 for information on fully qualified data set names.

//

Specifying these slashes indicates that the file names refer to MVS data sets.

qualifier

Each qualifier is a 1- to 8-character name. These characters may be alphanumeric, national (\$, #, @), the hyphen, or the character \xC0. The first character should be either alphabetic or national. Do not use hyphens in names for RACF-protected data sets.

You can join qualifiers with periods. The maximum length of a data set name is generally 44 characters, including periods.

To open a data set by its name, you can code something like the following in your C or C++ program:

```
infile=fopen("//'VSAM.CLUSTER1'", "ab+, type=record");
```

File names for MVS data sets: Using a DDname

To access a cluster or path by ddname, you can write the required DD statement and call `fopen()` as shown in the following example.

If your data set is VSAM.CLUSTER1, your C or C++ program refers to this data set by the ddname CFILE, and you want exclusive control of the data set for update, you can write the DD statement:

```
//CFILE DD DSNAME=VSAM.CLUSTER1,DISP=OLD
```

and code the following in your C or C++ source program:

```
#include <stdio.h>

FILE *infile;
main()
{
    infile=fopen("DD:CFILE", "ab+, type=record");
    :
}
```

To share your data set, use DISP=SHR on the DD statement. DISP=SHR is the default for `fopen()` calls that use a data set name and specify any of the `r`, `rb`, `rb+`, and `r+b` open modes.

Note: z/OS XL C/C++ does not check the value of shareoptions at `fopen()` time, and does not provide support for read-integrity and write-integrity, as required to share files under shareoptions 3 and 4.

For more information on shareoptions, see the information on DEFINE CLUSTER in the books listed in [“DFSMS”](#) on page 774.

Specifying fopen() and freopen() keywords

The *mode* argument is a character string specifying the type of access requested for the file. The *mode* argument contains one positional parameter (access mode) followed by keyword parameters. A description of these parameters, along with an explanation of how they apply to VSAM data sets is given in the following sections.

Specifying access mode

The access mode is specified by the positional parameter of the `fopen()` function call. The possible record I/O and binary modes you can specify are:

rb

Open for reading. If the file is empty, `fopen()` fails.

wb

Open for writing. If the cluster is defined as reusable, the existing contents of the cluster are destroyed. If the cluster is defined as not reusable (clusters with paths are, by definition, not reusable), `fopen()` fails. However, if the cluster has been defined but not loaded, this mode can be used to do the initial load of both reusable and non reusable clusters.

Note: If a "key out of sequence" condition is encountered, the data set will automatically be reopened with a mode string "ab+" and will no longer be in create mode.

ab

Open for writing.

rb+ or r+b

Open for reading, writing, and/or updating.

wb+ or w+b

Open for reading, writing, and/or updating. If the cluster is defined as reusable, the existing contents of the cluster are destroyed. If the cluster is defined as not reusable (clusters with paths are, by definition, not reusable), the `fopen()` fails. However, if the cluster has been defined but not loaded, this mode can be used to do the initial load of both reusable and non reusable clusters.

ab+ or a+b

Open for reading, writing, and/or updating.

For text files, you can specify the following modes: `r`, `w`, `a`, `r+`, `w+`, and `a+`.

Note: For KSDS, KSDS AIX and ESDS AIX in text and binary I/O, the only valid modes are `r` and `rb`, respectively.

fopen() and freopen() keywords

The following table lists the keywords that are available on the `fopen()` and `freopen()` functions, tells you which ones are useful for VSAM I/O, and lists the values that are valid for the applicable ones.

Table 15. Keywords for the `fopen()` and `freopen()` functions for VSAM data sets

Keyword	Allowed?	Applicable?	Notes
recfm=	Yes	No	Ignored.
lrecl=	Yes	No	Ignored.
blksize=	Yes	No	Ignored.
space=	Yes	No	Ignored.
type=	Yes	Yes	May be omitted. If you do specify it, <code>type=record</code> is the only valid value.
acc=	Yes	Yes	Specifies the access direction for VSAM data sets. Valid values are BWD and FWD.
password=	Yes	Yes	Specifies the password for a VSAM data set.

Table 15. Keywords for the `fopen()` and `freopen()` functions for VSAM data sets (continued)

Keyword	Allowed?	Applicable?	Notes
asis	Yes	No	Enables the use of mixed-case file names. Not supported for VSAM.
byteseek	Yes	Yes	Used for binary stream files to specify that the seeking functions should use relative byte offsets instead of encoded offsets. This is the default setting.
noseek	Yes	No	Ignored.
OS	Yes	No	Ignored.
rls=	Yes	Yes	Indicates the VSAM RLS/TVS access mode in which a VSAM file is to be opened.

Keyword descriptions

recfm=

Any values passed into `fopen()` are ignored.

lrecl= and blksize=

These keywords are set to the maximum record size of the cluster as initialized in the cluster definition. Any values passed into `fopen()` are ignored.

space=

This keyword is not supported under VSAM.

type=

If you use the `type=` keyword, the only valid value for VSAM data sets is `type=record`. This opens a file for record I/O.

acc=

For VSAM files opened with the keyword `type=record`, you can specify the direction by using the `acc=access_type` keyword on the `fopen()` function call. For text and binary files, the access direction is always forward. Attempts to open a VSAM data set with `acc=BWD` for either binary or text stream I/O will fail. The `access_type` can be one of the following:

FWD

The `acc=FWD` keyword specifies that the file be processed in a forward direction. When the file is opened, it will be positioned at the beginning of the first physical record, and any subsequent read operations sets the file position indicator to the beginning of the next record. The default value for the access keyword is `acc=FWD`.

BWD

The `acc=BWD` keyword specifies that the file be processed in a backward direction. When the file is opened, it is positioned at the beginning of the last physical record and any subsequent read operation sets the file position indicator to the beginning of the preceding record.

You can change the direction of sequential processing (from forward to backward or from backward to forward) by using the `flocate()` library function. For more information about `flocate()`, see [“Repositioning within record I/O files” on page 108](#).

Note: When opening paths, records with duplicate alternate index keys are processed in order of arrival time (oldest to newest) regardless of the current processing direction.

password=

VSAM facilities provide password protection for your data sets. You access a data set that has password protection by specifying the password on the `password` keyword parameter of the `fopen()` function call; the password resides in the VSAM catalog entry for the named file. There can be more than one password in the VSAM catalog entry; data sets can have different passwords for different levels of authorization such as reading, writing, updating, inserting, or deleting. For a complete description of password protection on VSAM files, see the list of publications on [“DFSMS” on page 774](#).

The password keyword has the following form, where *x* is a 1- to 8-character password, and *n* is the exact number of characters in the password. The password can contain special characters such as blanks and commas.

```
password=nx
```

If a required password is not supplied, or if an incorrect password is given, `fopen()` fails.

asis

This keyword is not supported for VSAM.

byteseek

When you specify this keyword and open a file in binary stream mode, `fseek()` and `ftell()` use relative byte offsets from the beginning of the file. This is the default setting.

noseek

This keyword is ignored for VSAM data sets.

OS

This keyword is ignored for VSAM data sets.

rls=

Indicates the VSAM RLS/TVS access mode in which a VSAM file is to be opened. This keyword is ignored for non-VSAM files. The following values are valid:

- *nri* — No Read Integrity
- *cr* — Consistent Read
- *cre* — Consistent Read Explicit

Note: When the RLS keyword is specified, DISP is changed to default to SHR when dynamic allocation of the data set is performed. In the rare case when a batch job must use RLS without sharing the data set with other tasks, DISP should be OLD. To set DISP to OLD, the application must specify DISP=OLD in the DD statement and start the application using JCL. You cannot specify DISP in the `fopen()` mode argument.

Buffering

Full buffering is the default. You can specify line buffering, but z/OS XL C/C++ treats line buffering as full buffering for VSAM data sets. Unbuffered I/O is not supported under VSAM; if you specify it, your `setvbuf()` call fails.

To find out how to optimize VSAM performance by controlling the number of VSAM buffers used for your data set, refer to [z/OS DFSMS Access Method Services Commands](#).

Record I/O in VSAM

This section describes how to use record I/O in VSAM. The following topics are covered:

- [“RRDS record structure” on page 105](#)
- [“Reading record I/O files” on page 105](#)
- [“Writing to record I/O files” on page 106](#)
- [“Updating record I/O files” on page 107](#)
- [“Deleting records” on page 108](#)
- [“Repositioning within record I/O files” on page 108](#)
- [“Flushing buffers” on page 110](#)
- [“Summary of VSAM record I/O operations” on page 110](#)
- [“Reading from text and binary I/O files” on page 114](#)
- [“Writing to and updating text and binary I/O files” on page 114](#)
- [“Deleting records in text and binary I/O files” on page 114](#)

- [“Repositioning within text and binary I/O files” on page 114](#)
- [“Flushing buffers” on page 116](#)
- [“Summary of VSAM text I/O operations” on page 116](#)
- [“Summary of VSAM binary I/O operations” on page 117](#)

RRDS record structure

For RRDS files opened in record mode, z/OS XL C/C++ defines the following key structure in the C header file `<stdio.h>`:

```
typedef struct {
#ifdef _LP64
    unsigned int __fill, /* version: either 0 or 1 */
                __recnum; /* the key, starting at 1 */
#else
    unsigned long __fill, /* version: either 0 or 1 */
                 __recnum; /* the key, starting at 1 */
#endif /* not _LP64 */
} __rrds_key_type;
```

In your source program, you can define an RRDS record structure as either:

```
struct {
    __rrds_key_type rrds_key; /* __fill value always 0 */
    char data[MY_REC_SIZE];
} rrds_rec_0;
```

or:

```
struct {
    __rrds_key_type rrds_key; /* __fill value always 1 */
    char *data;
} rrds_rec_1;
```

The z/OS XL C/C++ library recognizes which type of record structures you have used by the value of `rrds_key.__fill`. Zero indicates that the data is contiguous with `rrds_key` and 1 indicates that a pointer to the data follows `rrds_key`.

Reading record I/O files

To read from a VSAM data set opened with `type=record`, use the Standard C `fread()` library function. If you set the size argument to 1 and the count argument to the maximum record size, `fread()` returns the number of bytes read successfully. For more information on `fread()`, see [z/OS C/C++ Runtime Library Reference](#).

`fread()` reads one record from the system from the current file position. Thus, if you want to read a certain record, you can call `flocate()` to position the file pointer to point to it; the subsequent call to `fread()` reads in that record.

If you use an `fread()` call to request more bytes than the record about to be read contains, `fread()` reads the entire record and returns the number of bytes read. If you use `fread()` to request fewer bytes than the record about to read contains, `fread()` reads the number of bytes that you specified and returns your request.

z/OS XL C/C++ VSAM Record I/O does not allow a read operation to immediately follow a write operation without an intervening reposition. z/OS XL C/C++ treats the following as read operations:

- Calls to read functions that request 0 bytes
- Read requests that fail because of a system error
- Calls to the `ungetc()` function

Calling `fread()` several times in succession, with no other operations on this file in between, reads several records in sequence (sequential processing), which can be forward or backward, depending on the access direction, as described in the following.

- **KSDS, KSDS AIX and ESDS AIX**

The records are retrieved according to the sequence of the key of reference, or in reverse key sequence.

Note: Records with duplicate alternate index keys are processed in order of arrival time (oldest to newest) regardless of the current processing direction

- **ESDS**

The records are retrieved according to the sequence they were written to the file (entry sequence), or in reverse entry sequence.

- **RRDS**

The records are retrieved according to relative record number sequence or reverse relative record number sequence.

When records are being read, RRNs without an associated record are ignored. For example, if a file has relative records of 1, 2, and 5, the nonexistent records 3 and 4 are ignored.

By default, in record mode, `fread()` must be called with a pointer to an RRDS record structure. The field `__rrds_key_type.__fill` must be set to either 0 or 1 indicating the type of the structure, and the count argument must include the length of the `__rrds_key_type`. `fread()` returns the RRN number in the `__recnum` field, and includes the length of the `__rrds_key_type` in the return value. You can override these operations by setting the `_EDC_RRDS_HIDE_KEY` environment variable to Y. Once this variable is set, `fread()` is called with a data buffer and not an RRDS data structure. The return value of `fread()` is now only the length of the data read. In this case, `fread()` cannot return the RRN. For information on setting environment variables, see [Chapter 28, “Using environment variables,”](#) on page 327.

Writing to record I/O files

To write new records to a VSAM data set opened with `type=record`, use the Standard C `fwrite()` library function. If you set `size` to 1 and `count` to the desired record size, `fwrite()` returns the number of bytes written successfully. For more information on `fwrite()` and the `type=record` parameter, see [z/OS C/C++ Runtime Library Reference](#).

In general, C I/O does not allow a write operation to follow a read operation without an intervening reposition or `fflush()`. z/OS XL C/C++ counts a call to a write function writing 0 bytes or a write request that fails because of a system error as a write operation. However, z/OS XL C/C++ VSAM record I/O allows a write to directly follow a read. This feature has been provided for compatibility with earlier releases.

The process of writing to a data set for the first time is known as *initial loading*. Using the `fwrite()` function, you can write to a new VSAM file in *initial load* mode just as you would to a file not in *initial load* mode. Writing to a KSDS PATH or an ESDS PATH in *initial load* mode is not supported.

If your `fwrite()` call does not try to write more bytes than the maximum record size, `fwrite()` writes a record of the length you asked for and returns your request. If your `fwrite()` call asks for more than the maximum record size, `fwrite()` writes the maximum record size, sets `errno`, and returns the maximum record size. In either case, the next call to `fwrite()` writes to the following record.

Note: If an `fwrite()` fails, you must reposition the file before you try to read or write again.

- **KSDS, KSDS AIX**

Records are written to the cluster according to the value stored in the field designated as the prime key.

You can load a KSDS in any key order but it is most efficient to perform the `fwrite()` operations in key sequence.

- **ESDS, ESDS AIX**

Records are written to the end of the file.

- **RRDS**

Records are written according to the value stored in the relative record number field.

`fwrite()` is called with the RRDS record structure.

By default, in record mode, `fwrite()` and `fupdate()` must be called with a pointer to an RRDS record structure. The `__rrds_key_type` fields `__fill` and `__recnum` must be set. `__fill` is set to 0 or 1 to indicate the type of the structure. The `__recnum` field specifies the RRN to write, and is required for `fwrite()` but not `fupdate()`. The count argument must include the length of the `__rrds_key_type`. `fwrite()` and `fupdate()` include the length of the `__rrds_key_type` in the return value.

Updating record I/O files

The `fupdate()` function, a z/OS XL C/C++ extension to the SAA C library, is used to update records in a VSAM file. For more information on this function, see [z/OS C/C++ Runtime Library Reference](#).

• KSDS, ESDS, and RRDS

To update a record in a VSAM file, you must perform the following operations:

1. Open the VSAM file in update mode (`rb+/r+b`, `wb+/w+b`, or `ab+/a+b` specified as the required positional parameter of the `fopen()` function call and `type=record`).
2. If the file is not already positioned at the record you want to update, reposition to that record.
3. Read in the record using `fread()`.

Once the record you want to update has been read in, you must ensure that no reading, writing, or repositioning operations are performed before `fupdate()`.

4. Make the necessary changes to the copy of the record in your buffer area.
5. Update the record from your local buffer area using the `fupdate()` function.

If an `fupdate()` fails, you must reposition using `flocate()` before trying to read or write.

Notes:

1. If a file is opened in update mode, a read operation can result in the locking of control intervals, depending on shareoptions specification of the VSAM file. If after reading a record, you decide not to update it, you may need to unlock a control interval by performing a file positioning operation to the same record, such as an `flocate()` using the same key.
2. If `fupdate()` wrote out a record the file position is the start of the next record. If the `fupdate()` call did not write out a record, the file position remains the same.

• KSDS and KSDS PATH

You can change the length of the record being updated. If your request does not exceed the maximum record size of the file, `fupdate()` writes a record of the length requested and returns the request. If your request exceeds the maximum record size of the file, `fupdate()` writes a record that is the maximum record size, sets `errno`, and returns the maximum record size.

You cannot change the prime key field of the record, and in KSDS AIX, you cannot change the key of reference of the record.

• ESDS

You cannot change the length of the record being updated. If the size of the record being updated is less than the current record size, `fupdate()` updates the amount you specify and does not alter the data remaining in the record. If your request exceeds the length of the record that was read, `fupdate()` writes a record that is the length of the record that was read, sets `errno`, and returns the length of the record that was read.

• ESDS PATH

You cannot change the length of the record being updated or the key of reference of the record. If the size of the record being updated is less than the current record size, `fupdate()` updates the amount you specify and does not alter the data remaining in the record. If your request exceeds the length of the record that was read, `fupdate()` writes a record that is the length of the record that was read, sets `errno`, and returns the length of the record that was read.

- **RRDS**

RRDS files have fixed record length. If you update the record with less than the record size, only those characters specified are updated, and the remaining data is not altered. If your request exceeds the record size of the file, `fupdate()` writes a record that is the record size, sets `errno`, and returns the length of the record that was read.

Deleting records

To delete records, use the library function `fdelrec()`, a z/OS XL C/C++ extension to the SAA C library. For more information on this function, see [z/OS C/C++ Runtime Library Reference](#).

- **KSDS, KSDS PATH, and RRDS**

To delete records, you must perform the following operations:

1. Open the VSAM file in update mode (`rb+ / r+b`, `ab+ / a+b`, or `wb+ / w+b` specified as the required positional parameter of the `fopen()` function call and `type=record`).
2. If the file is not already positioned at the record you want to delete, reposition to that record.
3. Read the record using the `fread()` function.

Once the record you want to delete has been read in, you must ensure that no reading, writing, or repositioning operations are performed before `fdelrec()`.

4. Delete the record using the `fdelrec()` function.

Note: If the data set was opened with an access mode of `rb+` or `r+b`, a read operation can result in the locking of control intervals, depending on `shareoptions` specification of the VSAM file. If after reading a record, you decide not to delete it, you may need to unlock a control interval by performing a file-positioning operation to the same record, such as an `flocate()` using the same key.

- **ESDS and ESDS PATH**

VSAM does not support deletion of records in ESDS files.

Repositioning within record I/O files

You can use the following functions to locate a record within a VSAM data set:

- `flocate()`
- `ftell()`, `ftello()` and `fseek()`, `fseeko()`
- `fgetpos()` and `fsetpos()`
- `rewind()`

For complete details on these library functions, see [z/OS C/C++ Runtime Library Reference](#).

flocate()

The `flocate()` C library function can be used to locate a specific record within a VSAM data set given the key, relative byte address, or the relative record number. The `flocate()` function also sets the access direction.

VSAM extended addressability support includes an 8 byte RBA for use with positioning functions such as `flocate()`. `flocate()` supports RBA lengths of 4 and 8 bytes. Existing applications that use `flocate()` with a 4 byte RBA will continue unaffected, but must use a key length of 8 to locate an RBA beyond 4GB.

The following `flocate()` parameters set the access direction to forward:

- `__KEY_FIRST` (the `key` and `key_len` parameters are ignored)
- `__KEY_EQ`
- `__KEY_GE`
- `__RBA_EQ`

The following `flocate()` parameters all set the access direction to backward and are only valid for record I/O:

- `__KEY_LAST` (the `key` and `key_len` parameters are ignored)
- `__KEY_EQ_BWD`
- `__RBA_EQ_BWD`

Note: The `__RBA_EQ` and `__RBA_EQ_BWD` parameters are not valid for paths and are not recommended for KSDS and RRDS data sets.

You can use the `rewind()` library function instead of calling `flocate()` with `__KEY_FIRST`.

- **KSDS, KSDS AIX, and ESDS AIX**

The `key` parameter of `flocate()` for the options `__KEY_EQ`, `__KEY_GE`, and `__KEY_EQ_BWD` is a pointer to the key of reference of the data set. The `key_len` parameter is the key length as defined for the data set for a full key search, or less than the defined key length for a generic key search (a partial key match).

For KSDSs, `__RBA_EQ` and `__RBA_EQ_BWD` are supported, but are not recommended.

For `__KEY_EQ_BWD` the `key_len` parameter must be equal to the key length as defined for the data set for a full key search.

Alternate indexes do not allow positioning by RBA.

- **ESDS**

The `key` parameter of `flocate()` is a pointer to the specified RBA value. The `key_len` parameter is either 4 or 8 depending on the size of the RBA.

- **RRDS**

For `__KEY_EQ`, `__KEY_GE`, and `__KEY_EQ_BWD`, the `key` parameter of `flocate()` is a pointer to an unsigned long integer containing the specified relative record number. The `key_len` parameter is `sizeof(unsigned long)`. For `__RBA_EQ` and `__RBA_EQ_BWD`, the `key` parameter of `flocate()` is a pointer to the specified RBA. However, seeking to RBA values is not recommended, because it is not supported across control intervals. The `key_len` parameter is either 4 or 8 depending on the size of the RBA.

fgetpos() and fsetpos()

`fgetpos()` is used to store the current file position and access direction. `fsetpos()` is used to relocate to a file position stored by `fgetpos()` and restore the saved access direction.

- **KSDS**

`fgetpos()` stores the RBA value. This RBA value may be invalidated by subsequent insertions, deletions, or updates.

- **KSDS AIX and ESDS AIX**

`fgetpos()` and `fsetpos()` are not supported for PATHs.

- **ESDS and RRDS**

There are no special considerations.

ftell() and fseek()

`ftell()` is used to store the current file position. `fseek()` is used to relocate to one of the following:

- A file position stored by `ftell()`
- A calculated record number (`SEEK_SET`)
- A position relative to the current position (`SEEK_CUR`)
- A position relative to the end of the file (`SEEK_END`).

`ftell()` and `fseek()` offsets in record mode I/O are relative record offsets. For example, the following call moves the file position to the start of the previous record:

```
fseek(fp, -1L, SEEK_CUR);
```

You cannot use `fseek()` to reposition to a file position before the beginning of the file or to a position beyond the end of the file.

Note: In general, the performance of this method is inferior to `flocate()`.

The access direction is unchanged by the repositioning.

- **KSDS and RRDS**

There are no special considerations.

- **KSDS AIX and ESDS AIX**

`ftell()` and `fseek()` are not supported.

- **ESDS**

`ftell()` is not supported.

- **RRDS**

`fseek()` seeks to a relative position in the file, and not to an RRN value. For example, in a file consisting of RRNs 1, 3, 5 and 7, `fseek(fp, 3L, SEEK_SET);` followed by an `fread()` would read in RRN 7, which is at offset 3 in the file.

rewind()

The `rewind()` function repositions the file position to the beginning of the file, and clears the error setting for the file. `rewind()` does not reset the file access direction. For example, a call to `flocate()` with `__KEY_LAST` sets the file pointer to the end of the file and sets the access direction to backwards. A subsequent call to `rewind()` sets the file pointer to the beginning of the file, but the access direction remains backwards.

Flushing buffers

You can use the C library function `fflush()` to flush buffers. However, `fflush()` writes nothing to the system, because all records have already been written there by `fwrite()`. `fflush()` after a read operation does not refresh the contents of the buffer. For more information on `fflush()`, see [z/OS C/C++ Runtime Library Reference](#).

Summary of VSAM record I/O operations

Table 16. Summary of VSAM record I/O operations

	KSDS	ESDS	RRDS	PATH
<code>fopen()</code> , <code>freopen()</code>	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+
<code>fwrite()</code> , <code>fwrite_unlocked()</code>	rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	rb+, ab, ab+
<code>fread()</code> , <code>fread_unlocked()</code>	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+
<code>ftell()</code> , <code>ftell_unlocked()</code>	rb, rb+, ab, ab+, wb, wb+ (see note 1)		rb, rb+, ab, ab+, wb, wb+	
<code>ftello()</code> , <code>ftello_unlocked()</code>	rb, rb+, ab, ab+, wb, wb+ (see note 1)		rb, rb+, ab, ab+, wb, wb+	
<code>fseek()</code> , <code>fseek_unlocked()</code>	rb, rb+, ab, ab+, wb, wb+ (see note 1)	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	

Table 16. Summary of VSAM record I/O operations (continued)

	KSDS	ESDS	RRDS	PATH
fseeko(), fseeko_unlocked()	rb, rb+, ab, ab+, wb, wb+ (see note 1)	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
fgetpos(), fgetpos_unlocked()	rb, rb+, ab, ab+, wb, wb+ (see note 2)	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
fsetpos(), fsetpos_unlocked()	rb, rb+, ab, ab+, wb, wb+ (see note 2)	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
flocate(), flocate_unlocked()	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+
rewind(), rewind_unlocked()	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
fflush(), fflush_unlocked()	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
fdelrec(), fdelrec_unlocked()	rb+, ab+, wb+		rb+, ab+, wb+	rb+, ab+ (not ESDS)
fupdate(), fupdate_unlocked()	rb+, ab+, wb+	rb+, ab+, wb+	rb+, ab+, wb+	rb+, ab+
ferror(), ferror_unlocked()	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
feof(), feof_unlocked()	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
clearerr(), clearerr_unlocked()	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
fclose()	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
fldata(), fldata_unlocked()	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+

Notes:

1. The saved position is based on the relative position of the record within the data set. Subsequent insertions or deletions may invalidate the saved position.
2. The saved position is based on the RBA of the record. Subsequent insertions, deletions or updates may invalidate the saved position.

VSAM record level sharing and transactional VSAM

VSAM Record Level Sharing (RLS) and Transactional VSAM (VSAM RLS/TVS) provide for the sharing of VSAM data at the record level, using the locking and caching functions of the coupling facility hardware. For more information on Record Level Sharing, see *z/OS DFSMS Introduction*.

The C/C++ runtime library provides the following support for VSAM RLS/TVS:

- Specification of RLS/TVS-related keywords in the mode string of `fopen()` and `freopen()`.
- Specification of RLS/TVS-related text unit key values in the `__dyn_t` structure, which is used as input to the `dynalloc()` function.
- Provides the application with VSAM return and reason codes for VSAM I/O errors.
- Performs implicit positioning for files opened for RLS/TVS access.

VSAM RLS/TVS has three read integrity file access modes. These modes tell VSAM the level of locking to perform when records are accessed within a file that has **not been opened in update mode**. The access modes are:

nri

No Read Integrity indicates that requests performed by the application are not to be serialized with updates or erases of the records by other calling programs. VSAM accesses the records without obtaining a lock on the record.

cr

Consistent Read indicates that requests performed by the application are to be serialized with updates or erases of the records by other calling programs. VSAM obtains a share lock when accessing the record. This lock is released once the record has been returned to the caller.

cre

Consistent Read Explicit indicates that requests performed by the application are to be serialized with updates or erases of the records by other requestors. VSAM obtains a share lock when accessing the record. This lock is held until the application commits its changes. This ensures that records read by the application are not changed by other requestors until the application commits or aborts its changes. Consistent Read Explicit is for use only by commit protocol applications.

VSAM RLS locks records to support record integrity. An application may wait for an exclusive record lock if another user has the record locked. The application is also subject to new locking errors such as deadlock or timeout errors.

If the file has been **opened in update mode**, and RLS=CR or RLS=CRE is specified, VSAM also serializes access to the records within the file. However, the type of serialization differs from **non-update mode** in the following ways:

- A reposition within the file causes VSAM to obtain a share lock for the record.
- A read of a record causes VSAM to obtain an exclusive lock for the record. The lock is held until the record is updated in the file, or another record is read. If RLS=CRE is specified (for commit protocol applications), the lock is held until the application commits or aborts its changes.

Notes:

1. When a file is opened, it is implicitly positioned to the first record to be accessed.
2. You can also specify the RLS/TVS keyword on the JCL DD statement. When specified on both the JCL DD statement and in the mode string on `fopen()` or `freopen()`, the read integrity options specified in the mode string override those specified on the JCL DD statement.
3. VSAM RLS/TVS access is supported for the 3 types of VSAM files that the C/C++ runtime library supports: Key-Sequenced (KSDS), Entry-Sequenced (ESDS), and Relative Record (RRDS) data sets.
4. VSAM RLS/TVS functions require the use of a Coupling Facility. For more information on using the Coupling Facility, see [z/OS DFSMS Introduction](#), and [z/OS Parallel Sysplex® Overview](#).
5. In an environment where one thread opens and another thread issues record management requests, VSAM RLS/TVS requires that record management requests be issued from a thread whose Task Control Block (TCB) is subordinate to the TCB of the thread which opened the file.
6. VSAM RLS/TVS does not support the following:
 - Key range data sets
 - Direct open of an AIX cluster as a KSDS
 - Access to individual components of a cluster
 - OS Checkpoint and Restart

Error reporting

Errors are reported through the `__amrc` structure and the SIGIOERR signal. The following are additional considerations for error reporting in a VSAM RLS application:

- VSAM RLS/TVS uses the SMSVSAM server address space. When a file open fails because the server is not available, the C runtime library places the error return code and error value in the `__amrc` structure, and returns a null file descriptor. Record management requests return specific error return/reason codes, if the SMSVSAM server is not available. The server address space is automatically restarted. To recover from this type of error, an application should first close the file to clean up the file status, and then open the file prior to attempting record management requests. The close for the file returns a return code of 4, and an error code of 170(X'AA'). This is the expected result. It is not an error.

- Opening a recoverable file for output is not supported. If you attempt to do so, the open will fail with error return code 255 in the `__amrc` structure.
- Some of the VSAM errors, that are reported in the `__amrc` structure, are situations from which an application can recover. These are problems that can occur unpredictably in a sharing environment. Usually, the application can recover by simply accessing another record. Examples of such errors are the following:
 - RC 8, 21(X'15'): Request cancelled as part of deadlock resolution.
 - RC 8, 22(X'16'): Request cancelled as part of timeout resolution.
 - RC 8, 24(X'18'): Request cancelled because transaction backout is pending on the requested record.
 - RC 8, 29(X'14'): Intra-luwid contention between threads under a given TCB.

The application can intercept errors by registering a condition handler for the SIGIOERR condition. Within the condition handler, the application can examine the information in the `__amrc` structure and determine how to recover from each specific situation.

Refer to [z/OS DFSMS Macro Instructions for Data Sets](#) for a complete list of return and reason codes.

VSAM extended addressability

DFSMS supports VSAM data sets greater than 4GB in size through extended addressability (XADDR) support. XADDR support is an extension to DFSMS extended-format data set support. VSAM XADDR supports key sequenced data sets (KSDS), entry sequenced data sets (ESDS), and relative-record data set (RRDS).

Restriction: z/OS XL C/C++ does not support XADDR for ESDS alternate indexes.

VSAM XADDR support includes an 8 byte relative byte address for use with positioning functions such as `flocate()`. `flocate()` supports key lengths of 4 and 8 bytes. Existing applications that use `flocate()` to locate with a 4 byte relative byte address will continue unaffected, but must use a key length of 8 to locate a record within XADDR addresses.

The RBA field in the `__amrc` structure is set to -1 when applications access beyond the addresses that can be represented by the 4 byte value, effectively appearing to be EOF to any 4 byte RBA positioning (`flocate()`) calls. The `__XRBA` field will always be updated with the address, and must be used in these cases.

For AMODE 31 applications repositioning within a VSAM data set, users of `ftell()` and `fseek()` that need to access XADDR addresses, must use the large file version of `ftello()` and `fseeko()`.

XADDR support for AMODE 31 applications is listed in the following table:

Table 17. AMODE31 application XADDR support

Function	XADDR support
<code>fgetpos()</code> , <code>fgetpos_unlocked()</code>	Yes
<code>fsetpos()</code> , <code>fsetpos_unlocked()</code>	Yes
<code>ftell()</code> , <code>ftell_unlocked()</code>	No
<code>fseek()</code> , <code>fseek_unlocked()</code>	No
<code>ftello()</code> or <code>ftello_unlocked()</code> non-large files version	No
<code>fseeko()</code> or <code>fseeko_unlocked()</code> non-large files version	No
<code>ftello()</code> or <code>ftello_unlocked()</code> large files version	Yes
<code>fseeko()</code> or <code>fseeko_unlocked()</code> large files version	Yes
<code>flocate()</code> or <code>flocate_unlocked()</code>	Yes
<code>fldata()</code> or <code>fldata_unlocked()</code>	Yes

Note: AMODE 64 applications also have the above restrictions on XADDR support.

Text and binary I/O in VSAM

Because VSAM is primarily record-based, this section only discusses those aspects of text and binary I/O that are specific to VSAM. For general information on text and binary I/O, refer to the respective sections in Chapter 8, “Performing OS I/O operations,” on page 37.

Reading from text and binary I/O files

- **RRDS**

All the read functions support reading from text and binary RRDS files. `fread()` is called with a character buffer instead of an RRDS record structure.

Writing to and updating text and binary I/O files

- **KSDS, KSDS AIX, and ESDS AIX**

z/OS XL C/C++ VSAM support for streams does not provide for writing and updating these types of data sets opened for text or binary stream I/O.

- **ESDS**

Writes are supported for ESDSs opened as binary or text streams. Updating data in an ESDS stream cannot change the length of the record in the external file. Therefore, in a binary stream:

- updates for less than the existing record length leave existing data beyond the updated length unchanged;
- updates for longer than the existing record length flow over the record boundary and update the start of the next record.

In text streams:

- updates that specify records shorter than the original record pad the updated record to the existing record length with blanks;
- updates for longer than the existing record length result in truncation, unless the original record contained only a new-line character, in which case it may be updated to contain one byte of data plus a new-line character.

- **RRDS**

`fwrite()` is called with a character buffer instead of an RRDS record structure.

Records are treated as contiguous. Once the current record is filled, the next record in the file is written to. For example, if the file consisted of only record 1, record 5, and record 28, a write would complete record 1 and then go directly to record 5.

Writing past the last record in the file is allowed, up to the maximum size of the RRDS data set. For example, if the last record in the file is record 28, the next record to be written is record 29.

Insertion of records is not supported. For example, in a file of records 1, 5, and 28, you cannot insert record 3 into the file.

Deleting records in text and binary I/O files

`fdelrec()` is not supported for text and binary I/O in VSAM.

Repositioning within text and binary I/O files

You can use the following functions to locate a record within a VSAM data set:

- `flocate()`

- `ftell()`, `ftello()`, `fseek()`, and `fseeko()`
- `fgetpos()` and `fsetpos()`
- `rewind()`

For complete details on these library functions, see [z/OS C/C++ Runtime Library Reference](#).

flocate()

The `flocate()` C library function can be used to reposition to the beginning of a specific record within a VSAM data set given the key, relative byte address, or the relative record number. For more information on this function, see [z/OS C/C++ Runtime Library Reference](#).

VSAM extended addressability support includes an 8 byte RBA for use with positioning functions such as `flocate()`. `flocate()` supports RBA lengths of 4 and 8 bytes. Existing applications that use `flocate()` with a 4 byte RBA will continue unaffected, but must use a key length of 8 to locate an RBA beyond 4GB.

The following `flocate()` parameters set the direction access to forward:

- `__KEY_FIRST` (the `key` and `key_len` parameters are ignored)
- `__KEY_EQ`
- `__KEY_GE`
- `__RBA_EQ`

The following `flocate()` parameters all set the access direction to backward and are not valid for text and binary I/O, because backwards access is not supported:

- `__KEY_LAST` (the `key` and `key_len` parameters are ignored)
- `__KEY_EQ_BWD`
- `__RBA_EQ_BWD`

You can use the `rewind()` library function instead of calling `flocate()` with `__KEY_FIRST`.

• **KSDS, KSDS AIX, and ESDS AIX**

The `key` parameter of `flocate()` for the options `__KEY_EQ` and `__KEY_GE` is a pointer to the key of reference of the data set. The `key_len` parameter is the key length as defined for the data set for a full key search, or less than the defined key length for a generic key search (a partial key match).

Alternate indexes do not allow positioning by RBA.

Note: The `__RBA_EQ` parameter is not valid for paths and is not recommended.

• **ESDS**

The `key` parameter of `flocate()` is a pointer to the specified RBA value. The `key_len` parameter is either 4 or 8 depending on the size of the RBA.

• **RRDS**

For `__KEY_EQ` and `__KEY_GE`, the `key` parameter of `flocate()` is a pointer to an unsigned long integer containing the specified relative record number. The `key_len` parameter is `sizeof(unsigned long)`. For `__RBA_EQ`, the `key` parameter of `flocate()` is a pointer to the specified RBA. However, seeking to RBA values is not recommended, because it is not supported across control intervals. The `key_len` parameter is either 4 or 8 depending on the size of the RBA.

fgetpos() and fsetpos()

`fgetpos()` saves the access direction, an RBA value, and the file position, and `fsetpos()` restores the saved access direction. `fgetpos()` accounts for the presence of characters in the `ungetc()` buffer unless you have set the `_EDC_COMPAT` variable. See Chapter 28, “Using environment variables,” on page 327 for information about `_EDC_COMPAT`. If `ungetc()` characters back the file position up to before the start of the file, calls to `fgetpos()` fail.

- **KSDS**

`fgetpos()` stores the RBA value. This RBA value may be invalidated by subsequent insertions, deletions or updates.

- **KSDS PATH and ESDS PATH**

`fgetpos()` and `fsetpos()` are not supported for PATHs.

- **ESDS and RRDS**

There are no special considerations.

ftell() and fseek()

Using `fseek()` to seek beyond the current end of file in a writable ESDS or RRDS binary file results in the file being extended with nulls to the new position. An incomplete last record is completed with nulls, records of length `lrec1` are added as required, and the current record is filled with the remaining number of nulls and left in the current buffer. This is supported for relative byte offset from `SEEK_SET`, `SEEK_CUR` and `SEEK_END`.

For AMODE 31 applications repositioning within a VSAM data set, users of `ftell()` and `fseek()` that need to access positions beyond 4GB, must use the large file version of `ftello()` and `fseeko()`.

Table 18 on page 116 provides a summary of the `fseek()` and `ftell()` parameters in binary and text.

Table 18. Summary of `fseek()` and `ftell()` parameters in text and binary

Type	Mode	<code>ftell()</code> return values	<code>fseek()</code> <code>SEEK_SET</code>	<code>SEEK_CUR</code>	<code>SEEK_END</code>
KSDS	Binary	relative byte offset	relative byte offset	relative byte offset	relative byte offset
	Text	not supported	zero only	relative byte offset	relative byte offset
ESDS	Binary	relative byte offset	relative byte offset	relative byte offset	relative byte offset
	Text	not supported	zero only	relative byte offset	relative byte offset
RRDS	Binary	encoded byte offset	encoded byte offset	relative byte offset	relative byte offset
	Text	encoded byte offset	encoded byte offset	relative byte offset	relative byte offset
PATH	Binary	not supported	not supported	not supported	not supported
	Text	not supported	not supported	not supported	not supported

Flushing buffers

You can use the C library function `fflush()` to flush data.

For text files, calling `fflush()` to flush an update to a record causes the new data to be written to the file.

If you call `fflush()` while you are updating, the updates are flushed out to VSAM.

For more information on `fflush()`, see [z/OS C/C++ Runtime Library Reference](#).

Summary of VSAM text I/O operations

Table 19 on page 117 summarizes the VSAM text I/O operations.

Table 19. Summary of VSAM text I/O operations

	KSDS	ESDS	RRDS	PATH
<code>fopen()</code> , <code>freopen()</code>	r	r, r+, a, a+, w, w+ (empty cluster or reuse specified for w & w+)	r, r+, a, a+, w, w+ (empty cluster or reuse specified for w & w+)	r
<code>fwrite()</code> , <code>fwrite_unlocked()</code>		r+, a, a+, w, w+	r+, a, a+, w, w+	
<code>fprintf()</code> , <code>fprintf_unlocked()</code>		r+, a, a+, w, w+	r+, a, a+, w, w+	
<code>fputs()</code> , <code>fputs_unlocked()</code>		r+, a, a+, w, w+	r+, a, a+, w, w+	
<code>fputc()</code> , <code>fputc_unlocked()</code>		r+, a, a+, w, w+	r+, a, a+, w, w+	
<code>putc()</code> , <code>putc_unlocked()</code>		r+, a, a+, w, w+	r+, a, a+, w, w+	
<code>vfprintf()</code> , <code>vfprintf_unlocked()</code>		r+, a, a+, w, w+	r+, a, a+, w, w+	
<code>vprintf()</code> , <code>vprintf_unlocked()</code>		r+, a, a+, w, w+	r+, a, a+, w, w+	
<code>fread()</code> , <code>fread_unlocked()</code>	r	r, r+, a+, w+	r, r+, a+, w+	r
<code>fscanf()</code> , <code>fscanf_unlocked()</code>	r	r, r+, a+, w+	r, r+, a+, w+	r
<code>vfscanf()</code> , <code>vfscanf_unlocked()</code>	r	r, r+, a+, w+	r, r+, a+, w+	r
<code>fgets()</code> , <code>fgets_unlocked()</code>	r	r, r+, a+, w+	r, r+, a+, w+	r
<code>fgetc()</code> , <code>fgetc_unlocked()</code>	r	r, r+, a+, w+	r, r+, a+, w+	r
<code>getc()</code> , <code>getc_unlocked()</code>	r	r, r+, a+, w+	r, r+, a+, w+	r
<code>ungetc()</code> , <code>ungetc_unlocked()</code>	r	r, r+, a+, w+	r, r+, a+, w+	r
<code>ftell()</code> , <code>ftell_unlocked()</code>			r, r+, a, a+, w, w+	
<code>ftello()</code> , <code>ftello_unlocked()</code>			r, r+, a, a+, w, w+	
<code>fseek()</code> , <code>fseek_unlocked()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	
<code>fseeko()</code> , <code>fseeko_unlocked()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	
<code>fgetpos()</code> , <code>fgetpos_unlocked()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	
<code>fsetpos()</code> , <code>fsetpos_unlocked()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	
<code>flocate()</code> , <code>flocate_unlocked()</code>	r	r, r+, a+, w+	r, r+, a+, w+	r
<code>rewind()</code> , <code>rewind_unlocked()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
<code>fflush()</code> , <code>fflush_unlocked()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
<code>ferror()</code> , <code>ferror_unlocked()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
<code>fdelrec()</code> , <code>fdelrec_unlocked()</code>				
<code>fupdate()</code> , <code>fupdate_unlocked()</code>				
<code>feof()</code> , <code>feof_unlocked()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
<code>clearerr()</code> , <code>clearerr_unlocked()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
<code>fclose()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
<code>fldata()</code> , <code>fldata_unlocked()</code>	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r

Summary of VSAM binary I/O operations

Table 20 on page 117 summarizes the VSAM binary I/O operations.

Table 20. Summary of VSAM binary I/O operations

	KSDS	ESDS	RRDS	PATH
<code>fopen()</code> , <code>freopen()</code>	rb	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb

Table 20. Summary of VSAM binary I/O operations (continued)

	KSDS	ESDS	RRDS	PATH
<code>fwrite()</code> , <code>fwrite_unlocked()</code>		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
<code>fprintf()</code> , <code>fprintf_unlocked()</code>		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
<code>fputs()</code> , <code>fputs_unlocked()</code>		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
<code>fputc()</code> , <code>fputc_unlocked()</code>		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
<code>putc()</code> , <code>putc_unlocked()</code>		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
<code>vfprintf()</code> , <code>vfprintf_unlocked()</code>		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
<code>vprintf()</code> , <code>vprintf_unlocked()</code>		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
<code>fread()</code> , <code>fread_unlocked()</code>	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
<code>fscanf()</code> , <code>fscanf_unlocked()</code>	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
<code>vfscanf()</code> , <code>vfscanf_unlocked()</code>	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
<code>fgets()</code> , <code>fgets_unlocked()</code>	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
<code>fgetc()</code> , <code>fgetc_unlocked()</code>	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
<code>getc()</code>	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
<code>getc_unlocked()</code>	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
<code>ungetc()</code> , <code>ungetc_unlocked()</code>	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
<code>ftell()</code> , <code>ftell_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
<code>ftello()</code> , <code>ftello_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
<code>fseek()</code> , <code>fseek_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
<code>fseeko()</code> , <code>fseeko_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
<code>fgetpos()</code> , <code>fgetpos_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
<code>fsetpos()</code> , <code>fsetpos_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
<code>flocate()</code> , <code>flocate_unlocked()</code>	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
<code>rewind()</code> , <code>rewind_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
<code>fflush()</code> , <code>fflush_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
<code>ferror()</code> , <code>ferror_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
<code>fdelrec()</code> , <code>fdelrec_unlocked()</code>				
<code>fupdate()</code> , <code>fupdate_unlocked()</code>				
<code>feof()</code> , <code>feof_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
<code>clearerr()</code> , <code>clearerr_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
<code>fclose()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
<code>fldata()</code> , <code>fldata_unlocked()</code>	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb

Closing VSAM data sets

To close a VSAM data set, use the Standard C `fclose()` library function as you would for closing non-VSAM files. See [z/OS C/C++ Runtime Library Reference](#) for more details on the `fclose()` library function.

For ESDS binary files, if `fclose()` is called and there is a new record in the buffer that is less than the maximum record size, this record is written to the file at its current size. A new RRDS binary record that is incomplete when the file is closed is filled with null characters to the record size.

A new ESDS or RRDS text record that is incomplete when the file is closed is completed with a new-line.

VSAM return codes

When failing return codes are received from z/OS XL C/C++ VSAM I/O functions, you can access the `__amrc` structure to help you diagnose errors. The `__amrc_type` structure is defined in the header file `stdio.h` (when the compiler option `LANGlvl (LIBEXT)` is used).

Note: The `__amrc` struct is global and can be reset by another I/O operation (such as `printf()`).

The following fields of the structure are important to VSAM users:

`__amrc.__code.__feedback.__rc`

Stores the VSAM R15.

`__amrc.__code.__feedback.__fdbk`

Stores the VSAM error code or reason code.

`__amrc.__RBA`

Stores the RBA after some operations. The `__amrc.__RBA` field is defined as an unsigned int, and therefore will only contain a 4-byte RBA value. This field will be set to -1 when the RBA is beyond 4GB in an extended addressable VSAM data set. In this case, the `__XRBA` field should be used.

`__amrc.__XRBA`

The 8 byte relative byte address returned by VSAM after an ESDS or KSDS record is written out. For an RRDS, it is the calculated value from the record number. It may be used in subsequent calls to `flocate()`.

`__amrc.__last_op`

Stores a code for the last operation. The codes are defined in the header file `stdio.h`.

`__amrc.__rplfdbwd`

Stores the feedback code from the IFGRPL control block.

For definitions of these return codes and feedback codes, refer to the publications listed in [“DFSMS” on page 774](#).

You can set up a SIGIOERR handler to catch read or write system errors. See [Chapter 16, “Debugging I/O programs,” on page 161](#) for more information.

VSAM examples

This section provides several examples of using I/O under VSAM.

KSDS example

The example in [Figure 18 on page 120](#) shows a sample program (CCNGVS2) with two functions from an employee record entry system with a mainline driver to process selected options (display, display next, update, delete, create). The update routine is an example of KSDS clusters, and the display routine is an example of both KSDS clusters and alternate indexes.

For these examples, the clusters and alternate indexes should be defined as follows:

- The KSDS cluster has a record size of 150 with a key length of 4 with offset 0.
- The unique KSDS AIX has a key length of 20 with an offset of 10.
- The non-unique KSDS AIX has a key length of 40 with an offset of 30.

The update routine is passed the following:

- `data_ptr`, which points to the information that is to be updated
- `orig_data_ptr`, which points to the information that was originally displayed using the display option
- A file pointer to the KSDS cluster

The display routine is passed the following:

- `data_ptr`, which points to the information that was entered on the screen for the search query

- `orig_data_ptr`, which is returned with the information for the record to be displayed if it exists
- File pointers for the primary cluster, unique alternate index and non-unique alternate index

By definition, the primary key is unique and therefore the employee number was chosen for this key. The `user_id` is also a unique key; therefore, it was chosen as the unique alternate index key. The name field may not be unique; therefore, it was chosen as the non-unique alternate index key.

```

/* this example demonstrates the use of a KSDS file */
/* part 1 of 2-other file is CCNGVS3 */

#include <stdio.h>
#include <string.h>

/* global definitions */

struct data_struct {
    char    emp_number[4];
    char    user_id[8];
    char    name[20];
    char    pers_info[37];
};

#define REC_SIZE          69
#define CLUS_KEY_SIZE     4
#define AIX_UNIQUE_KEY_SIZE  8
#define AIX_NONUNIQUE_KEY_SIZE 20

static void print_amrc() {
    __amrc_type currErr = *__amrc; /* copy contents of __amrc */
                                   /* structure so that values */
                                   /* don't get jumbled by printf */
    printf("R15 value    = %d\n", currErr.__code.__feedback.__rc);
    printf("Reason code = %d\n", currErr.__code.__feedback.__fdbk);
    printf("RBA        = %d\n", currErr.__RBA);
    printf("Last op     = %d\n", currErr.__last_op);
    return;
}

/* update_emp_rec() function definition */

int update_emp_rec (struct data_struct *data_ptr,
                   struct data_struct *orig_data_ptr,
                   FILE *fp)
{
    int      rc;
    char     buffer[REC_SIZE+1];

```

KSDS example (Part 1 of 5)

Figure 18. KSDS example


```

/* Check to see if update will change primary key (emp_number) */
    if (memcmp(data_ptr->emp_number,orig_data_ptr->emp_number,4) !=
0) {
    /* Check to see if changed primary key exists */
    rc = flocate(fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,__KEY_EQ);
    if (rc == 0) {
        print_amrc();
        printf("Error: new employee number already exists\n");
        return 10;
    }

    clearerr(fp);

    /* Write out new record */
    rc = fwrite(data_ptr,1,REC_SIZE,fp);
    if (rc != REC_SIZE || ferror(fp)) {
        print_amrc();
        printf("Error: write with new employee number failed\n");
        return 20;
    }

    /* Locate to old employee record so it can be deleted */
    rc = flocate(fp,&(orig_data_ptr->emp_number),CLUS_KEY_SIZE,
__KEY_EQ);
    if (rc != 0) {
        print_amrc();
        printf("Error: flocate to original employee number failed\n");
        return 30;
    }

    rc = fread(buffer,1,REC_SIZE,fp);
    if (rc != REC_SIZE || ferror(fp)) {
        print_amrc();
        printf("Error: reading old employee record failed\n");
        return 40;
    }

    rc = fdelrec(fp);
    if (rc != 0) {
        print_amrc();
        printf("Error: deleting old employee record failed\n");
        return 50;
    }
} /* end of checking for change in primary key */
else { /* Locate to current employee record */
    rc = flocate(fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,__KEY_EQ);
    if (rc == 0) {
        /* record exists, so update it */
        rc = fread(buffer,1,REC_SIZE,fp);
        if (rc != REC_SIZE || ferror(fp)) {
            print_amrc();
            printf("Error: reading old employee record failed\n");
            return 60;
        }
        rc = fupdate(data_ptr,REC_SIZE,fp);
        if (rc == 0) {
            print_amrc();
            printf("Error: updating new employee record failed\n");
            return 70;
        }
    }
}
}

```

KSDS example (Part 2 of 5)

```

        else { /* record doesn't exist so write out new record */
            clearerr(fp);
            printf("Warning: record previously displayed no longer\n");
            printf("          : exists, new record being created\n");
            rc = fwrite(data_ptr,1,REC_SIZE,fp);
            if (rc != REC_SIZE || ferror(fp)) {
                print_amrc();
                printf("Error: write with new employee number failed\n");
                return 80;
            }
        }
    }
    return 0;
}

/* display_emp_rec() function definition */
int display_emp_rec (struct data_struct *data_ptr,
                    struct data_struct *orig_data_ptr,
                    FILE *clus_fp, FILE *aix_unique_fp,
                    FILE *aix_non_unique_fp)
{
    int    rc = 0;
    char    buffer[REC_SIZE+1];

    /* Primary Key Search */
    if (memcmp(data_ptr->emp_number, "\\0\\0\\0\\0", 4) != 0) {
        rc = flocate(clus_fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,
                    _KEY_EQ);
        if (rc != 0) {
            printf("Error: flocate with primary key failed\n");
            return 10;
        }

        /* Read record for display */
        rc = fread(orig_data_ptr,1,REC_SIZE,clus_fp);
        if (rc != REC_SIZE || ferror(clus_fp)) {
            printf("Error: reading employee record failed\n");
            return 15;
        }
    }

    /* Unique Alternate Index Search */
    else if (data_ptr->user_id[0] != '\\0') {
        rc = flocate(aix_unique_fp,data_ptr->user_id,AIX_UNIQUE_KEY_SIZE,
                    _KEY_EQ);
        if (rc != 0) {
            printf("Error: flocate with user id failed\n");
            return 20;
        }

        /* Read record for display */
        rc = fread(orig_data_ptr,1,REC_SIZE,aix_unique_fp);
        if (rc != REC_SIZE || ferror(aix_unique_fp)) {
            printf("Error: reading employee record failed\n");
            return 25;
        }
    }
}

```

KSDS example (Part 3 of 5)

```

/* Non-unique Alternate Index Search */
else if (data_ptr->name[0] != '\0') {
    rc = flocate(aix_non_unique_fp, data_ptr->name,
                AIX_NONUNIQUE_KEY_SIZE, __KEY_GE);
    if (rc != 0) {
        printf("Error: flocate with name failed\n");
        return 30;
    }

    /* Read record for display */
    rc = fread(orig_data_ptr, 1, REC_SIZE, aix_non_unique_fp);
    if (rc != REC_SIZE || ferror(aix_non_unique_fp)) {
        printf("Error: reading employee record failed\n");
        return 35;
    }
}
else {
    printf("Error: invalid search argument; valid search arguments\n"
           "      : are either employee number, user id, or name\n");
    return 40;
}
/* display record data */
printf("Employee Number: %.4s\n", orig_data_ptr->emp_number);
printf("Employee Userid: %.8s\n", orig_data_ptr->user_id);
printf("Employee Name:   %.20s\n", orig_data_ptr->name);
printf("Employee Info:   %.37s\n", orig_data_ptr->pers_info);
return 0;
}
/* main() function definition */
int main() {
    FILE*      clus_fp;
    FILE*      aix_ufp;
    FILE*      aix_nufp;
    int         i;
    struct data_struct buf1, buf2;

    char data[3][REC_SIZE+1] = {
        " 1LARRY   LARRY           HI, I'M LARRY,      ",
        " 2DARRYL1 DARRYL          AND THIS IS MY BROTHER DARRYL, ",
        " 3DARRYL2 DARRYL           "
    };

    /* open file three ways */
    clus_fp = fopen("dd:cluster", "rb+,type=record");
    if (clus_fp == NULL) {
        print_amrc();
        printf("Error: fopen(\"dd:cluster\"...) failed\n");
        return 5;
    }

    /* assume base cluster was loaded with at least one dummy record */
    /* so aix could be defined */
    aix_ufp = fopen("dd:aixuniq", "rb,type=record");
    if (aix_ufp == NULL) {
        print_amrc();
        printf("Error: fopen(\"dd:aixuniq\"...) failed\n");
        return 10;
    }

    /* assume base cluster was loaded with at least one dummy record */
    /* so aix could be defined */
    aix_nufp = fopen("dd:aixnuniq", "rb,type=record");

```

KSDS example (Part 4 of 5)

```

if (aix_nufp == NULL) {
    print_amrc();
    printf("Error: fopen(\"dd:aixnuniq\"...) failed\n");
    return 15;
}

/* load sample records */
for (i = 0; i < 3; ++i) {
    if (fwrite(data[i],1,REC_SIZE,clus_fp) != REC_SIZE) {
        print_amrc();
        printf("Error: fwrite(data[%d]...) failed\n", i);
        return 66+i;
    }
}

/* display sample record by primary key */
memcpy(buf1.emp_number, "1", 4);
if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
    return 69;

/* display sample record by nonunique aix key */
memset(buf1.emp_number, '\0', 4);
buf1.user_id[0] = '\0';
memcpy(buf1.name, "DARRYL", 20);
if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
    return 70;

/* display sample record by unique aix key */
memcpy(buf1.user_id, "DARRYL2", 8);
if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
    return 71;

/* update record just read with new personal info */
memcpy(&buf1, &buf2, REC_SIZE);
memcpy(buf1.pers_info, "AND THIS IS MY OTHER BROTHER DARRYL.", 37);
if (update_emp_rec(&buf1, &buf2, clus_fp) != 0) return 72;

/* display sample record by unique aix key */
if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
    return 73;

return 0;
}

```

KSDS example (Part 5 of 5)

The JCL in the sample code (CCNGVS3) in [Figure 19 on page 125](#) can be used to test the example code in [Figure 18 on page 120](#).

```

/* this example illustrates the use of a KSDS file
/* part 2 of 2-other file is CCNGVS2
/*-----
/* Delete cluster, and AIX and PATH
/*-----
//DELETEDC EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
DELETE -
    userid.KSDS.CLUSTER -
    CLUSTER -
    PURGE -
    ERASE
/*-----
/* Define KSDS
/*-----
//DEFINE EXEC PGM=IDCAMS
//VOLUME DD UNIT=SYSDA,DISP=SHR,VOL=SER=(XXXXXX)
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
DEFINE CLUSTER -
    (NAME(userid.KSDS.CLUSTER) -
    FILE(VOLUME) -
    VOL(XXXXXX) -
    TRK(4 4) -
    RECSZ(69 100) -
    INDEXED -
    NOREUSE -
    KEYS(4 0) -
    OWNER(userid) ) -
DATA -
    (NAME(userid.KSDS.DA)) -
INDEX -
    (NAME(userid.KSDS.IX))
/*

```

KSDS example (Part 1 of 3)

Figure 19. KSDS example

```

/*-----
/* Repro data into KSDS
/*-----
//REPRO    EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
      REPRO INDATASET(userid.DUMMY.DATA) -
            OUTDATASET(userid.KSDS.CLUSTER)
/*
/*-----
/* Define unique AIX, define and build PATH
/*-----
//DEFAIX   EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
      DEFINE AIX -
            (NAME(userid.KSDS.UAIX) -
             RECORDS(25) -
             KEYS(8,4) -
             VOL(XXXXXX) -
             UNIQUEKEY -
             RELATE(userid.KSDS.CLUSTER)) -
      DATA -
            (NAME(userid.KSDS.UAIXDA)) -
      INDEX -
            (NAME(userid.KSDS.UAIXIX))
      DEFINE PATH -
            (NAME(userid.KSDS.UPATH) -
             PATHENTRY(userid.KSDS.UAIX))
      BLDINDEX -
            INDATASET(userid.KSDS.CLUSTER) -
            OUTDATASET(userid.KSDS.UAIX)
/*

```

KSDS example (Part 2 of 3)

```

/*-----
/* Define nonunique AIX, define and build PATH
/*-----
//DEFAIX EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
    DEFINE AIX -
        (NAME(userid.KSDS.NUAIX) -
         RECORDS(25) -
         KEYS(20, 12) -
         VOL(XXXXXX) -
         NONUNIQUEKEY -
         RELATE(userid.KSDS.CLUSTER)) -
    DATA -
        (NAME(userid.KSDS.NUAIXDA)) -
    INDEX -
        (NAME(userid.KSDS.NUAIXIX))
    DEFINE PATH -
        (NAME(userid.KSDS.NUPATH) -
         PATHENTRY(userid.KSDS.NUAIX))
    BLDINDEX -
        INDATASET(userid.KSDS.CLUSTER) -
        OUTDATASET(userid.KSDS.NUAIX)
/*
/*-----
/* Run the testcase
/*-----
//GO EXEC PGM=CCNGVS2,REGION=5M
//STEPLIB DD DSN=userid.TEST.LOAD,DISP=SHR
// DD DSN=CEE.SCEERUN,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTEM DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//PLIDUMP DD SYSOUT=*
//SYSABEND DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//CLUSTER DD DSN=userid.KSDS.CLUSTER,DISP=SHR
//AIXUNIQ DD DSN=userid.KSDS.UPATH,DISP=SHR
//AIXNUNIQ DD DSN=userid.KSDS.NUPATH,DISP=SHR
/*-----
/* Print out the cluster
/*-----
//PRINTF EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
    PRINT -
        INDATASET(userid.KSDS.CLUSTER) CHAR
/*

```

KSDS example (Part 3 of 3)

RRDS example

The sample program (CCNGVS4) in [Figure 20 on page 128](#) illustrates the use of an RRDS file. It performs the following operations:

1. Opens an RRDS file in record mode (the cluster must be defined)
2. Writes three records (RRN 2, RRN 10, and RRN 32)
3. Sets the file position to the first record
4. Reads the first record in the file
5. Deletes it
6. Locates the last record in the file and sets the access direction to backwards
7. Reads the record
8. Updates the record

9. Sets the _EDC_RRDS_HIDE_KEY environment variable
10. Reads the next record in sequence (RRN 10) into a character string

```

/* this example illustrates the use
of an RRDS file */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <env.h>

struct rrds_struct {
    __rrds_key_type    rrds_key;
    char               *rrds_buf;
};

typedef struct rrds_struct RRDS_STRUCT;

main() {
    FILE                *fileptr;
    RRDS_STRUCT         RRDSstruct;
    RRDS_STRUCT         *rrds_rec = &RRDSstruct;
    char                buffer1[80] =
        "THIS IS THE FIRST RECORD IN THE FILE. I"
        "T WILL BE WRITTEN AT RRN POSITION 2.  ";
    char                buffer2[80] =
        "THIS IS THE SECOND RECORD IN THE FILE. I"
        "T WILL BE WRITTEN AT RRN POSITION 10.  ";
    char                buffer3[80] =
        "THIS IS THE THIRD RECORD IN THE FILE. I"
        "T WILL BE WRITTEN AT RRN POSITION 32.  ";
    char                outputbuf[80];
    unsigned long        flocate_key = 0;

    /*-----*/
    /*| select RRDS record structure 2 by setting __fill to 1          */
    /*|                                                                */
    /*| 1. open an RRDS file record mode (the cluster must be defined) */
    /*| 2. write three records (RRN 2, RRN 10, RRN 32)                 */
    /*|-----*/
    rrds_rec->rrds_key.__fill = 1;

    fileptr = fopen("DD:RRDSFILE", "wb+,type=record");
    if (fileptr == NULL) {
        perror("fopen");
        exit(99);
    }
    rrds_rec->rrds_key.__recnum = 2;
    rrds_rec->rrds_buf = buffer1;
    fwrite(rrds_rec,1,88, fileptr);

    rrds_rec->rrds_key.__recnum = 10;
    rrds_rec->rrds_buf = buffer2;
    fwrite(rrds_rec,1,88, fileptr);

    rrds_rec->rrds_key.__recnum = 32;
    rrds_rec->rrds_buf = buffer3;
    fwrite(rrds_rec,1,88, fileptr);

```

RRDS example (Part 1 of 2)

Figure 20. RRDS example


```

/*-----*/
/*| 3. set file position to the first record                               */
/*| 4. read the first record in the file                                */
/*| 5. delete it                                                         */
/*-----*/
    flocate(fileptr, &flocate_key,; sizeof(unsigned long), __KEY_FIRST);

    memset(outputbuf,0x00,80);
    rrds_rec->rrds_buf = outputbuf;

    fread(rrds_rec,1, 88, fileptr);
    printf("The first record in the file (this will be deleted):\n");
    printf("RRN %d: %s\n\n",rrds_rec->rrds_key.__recnum,outputbuf);

    fdelrec(fileptr);

/*-----*/
/*| 6. locate last record in file and set access direction backwards*/
/*| 7. read the record                                                  */
/*| 8. update the record                                                */
/*-----*/
    flocate(fileptr, &flocate_key,; sizeof(unsigned long), __KEY_LAST);

    memset(outputbuf,0x00,80);
    rrds_rec->rrds_buf = outputbuf;

    fread(rrds_rec,1, 88, fileptr);
    printf("The last record in the file (this one will be updated):\n");
    printf("RRN %d: %s\n\n",rrds_rec->rrds_key.__recnum,outputbuf);

    memset(outputbuf,0x00,80);
    memcpy(outputbuf,"THIS IS THE UPDATED STRING... ",30);
    fupdate(rrds_rec,88,fileptr);

/*-----*/
/*| 9. set _EDC_RRDS_HIDE_KEY environment variable                     */
/*|10. read the next record in sequence (ie. RRN 10) into a            */
/*|    + character string                                              */
/*-----*/

    setenv("_EDC_RRDS_HIDE_KEY","Y",1);
    memset(outputbuf,0x00,80);
    fread(outputbuf, 1, 80, fileptr);
    printf("The middle record in the file (read into char string):\n");
    printf("%80s\n\n",outputbuf);

    fclose(fileptr);
}

```

RRDS example (Part 2 of 2)

fldata() behavior

The format of the `fldata()` function is as follows:

```
int fldata(FILE *file, char *filename, fldata_t *info);
```

The `fldata()` function is used to retrieve information about an open stream. The name of the file is returned in *filename* and other information is returned in the `fldata_t` structure, shown in Figure 21 on page 130. Values specific to this category of I/O are shown in the comment beside the structure element. Other notes about this category of I/O follow the figure. For more information on the `fldata()` function, see [z/OS C/C++ Runtime Library Reference](#).

```

struct __fileData {
    unsigned int    __recfmF : 1, /* */
                  __recfmV : 1, /* */
                  __recfmU : 1, /* */
                  __recfmS : 1, /* always off */
                  __recfmBlk : 1, /* always off */
                  __recfmASA : 1, /* always off */
                  __recfmM : 1, /* always off */
                  __dsorgPO : 1, /* N/A -- always off */
                  __dsorgPDSmem : 1, /* N/A -- always off */
                  __dsorgPDSdir : 1, /* N/A -- always off */
                  __dsorgPS : 1, /* N/A -- always off */
                  __dsorgConcat : 1, /* N/A -- always off */
                  __dsorgMem : 1, /* N/A -- always off */
                  __dsorgHiper : 1, /* N/A -- always off */
                  __dsorgTemp : 1, /* N/A -- always off */
                  __dsorgVSAM : 1, /* always on */

    #if __TARGET_LIB__ >= __EDC_LE /* */
        __dsorgHFS : 1, /* */
    #else
        __reserve1 : 1, /* */
    #endif

    __openmode : 2, /* one of: */
                /* --TEXT */
                /* --BINARY */
                /* --RECORD */
    __modeflag : 4, /* combination of: */
                /* --READ */
                /* --WRITE */
                /* --APPEND */
                /* --UPDATE */
    __dsorgPDSE : 1, /* N/A -- always off */
    __vsamRLS : 3, /* One of: */
                /* --NORLS */
                /* --RLS */

    #if __EDC_TARGET >= 0x41080000 /* */
        __vsamEA : 1, /* */
        __reserve2 : 4; /* */
    #else
        __reserve3 : 5; /* */
    #endif

    __device_t    __device; /* */
    unsigned long __blksize; /* */
    unsigned long __maxreclen; /* */
    union {
        struct {
            unsigned short __vsam_type; /* */
            unsigned long __vsam_keylen; /* */
            unsigned long __vsam_RKP; /* */
        } __vsam; /* */
    #if __EDC_TARGET >= 0x41080000 /* */
        struct {
            unsigned char __disk_access_method; /* */
            unsigned char __disk_noseek_to_seek; /* */
            long __disk_reserve[2]; /* */
        } __disk; /* */
    #endif
    } __device_specific;

    char * __dsname; /* */
    unsigned int __reserve4; /* */
};
typedef struct __fileData fldata_t;

```

Figure 21. *fldata()* structure

Notes:

1. If you have opened the file by its data set name, the *filename* is fully qualified, including quotation marks. If you have opened the file by ddname, *filename* is dd:ddname, without any quotation marks. The ddname is uppercase.
2. The `__dsname` field is filled in with the data set name. The `__dsname` value is uppercase unless the `asis` option was specified on the `fopen()` or `freopen()` function call.

Chapter 11. Performing terminal I/O operations

This chapter describes how to use input and output interactively with a terminal (using TSO or z/OS UNIX). Terminal I/O supports text, binary, and record I/O, in undefined, variable and fixed-length formats, except that ASA format is not valid for any text terminal files.

Note: You cannot use the z/OS XL C/C++ I/O functions for terminal I/O under either IMS or CICS. Terminal I/O under CICS is supported through the CICS command level interface.

This chapter describes C I/O stream functions as they can be used within C++ programs. If you want to use the C++ I/O stream classes instead, see [Using the Standard C++ Library I/O Stream Classes in z/OS XL C/C++ Programming Guide](#) for general information. For more detailed information, see [Standard C++ Library Reference](#), which discusses the Standard C++ I/O stream classes.

Opening files

You can use the library functions `fopen()` or `freopen()` to open a file.

Using `fopen()` and `freopen()`

This section covers:

- Opening a file by data set name
- Opening a file by DD name
- `fopen()` and `freopen()` keywords
- Opening a terminal file under a shell

Opening a file by data set name

Files are opened with a call to `fopen()` or `freopen()` in the format `fopen("filename", "mode")`. The first character of the filename must be an asterisk (*).

z/OS UNIX Considerations

If you have specified `POSIX(ON)`, `fopen("*file.data", "r");` does not open a terminal file. Instead, it opens a file called `*file.data` in the UNIX file system. To open a terminal file under POSIX, you must specify two slashes before the asterisk, as follows:

```
fopen("/*file.data", "r");
```

Terminal files cannot be opened in update mode.

Terminal files opened in append mode are treated as if they were opened in write mode.

Opening a file by DDname

The data set name that is associated with the DD statement must be an asterisk(*). For example:

```
TSO ALLOC f(ddname) DA(*)  
fopen("dd:ddname", "mode");
```

`fopen()` and `freopen()` keywords

The following table lists the keywords that are available on the `fopen()` and `freopen()` functions, tells you which ones are useful for terminal I/O, and lists the values that are valid for the applicable ones.

Table 21. Keywords for the `fopen()` and `freopen()` functions for terminal I/O

Parameter	Allowed?	Applicable?	Notes
<code>recfm=</code>	Yes	Yes	F, V, U and additional keywords A, B, S, M are the valid values. A, B, S, and M are ignored.
<code>lrecl=</code>	Yes	Yes	See below.
<code>blksize=</code>	Yes	Yes	See below.
<code>space=</code>	Yes	No	Has no effect for terminal I/O.
<code>type=</code>	Yes	Yes	May be omitted. If you do specify it, <code>type=record</code> is the only valid value.
<code>acc=</code>	No	No	Not used for terminal I/O.
<code>password=</code>	No	No	Not used for terminal I/O.
<code>asis</code>	Yes	No	Has no effect for terminal I/O.
<code>bytesseek</code>	Yes	No	Has no effect for terminal I/O.
<code>noseek</code>	Yes	No	Has no effect for terminal I/O.
<code>OS</code>	Yes	No	Not used for terminal I/O.

recfm=

z/OS XL C/C++ allows you to specify any of the 27 possible RECFM types (listed in “Fixed-format records” on page 12, “Variable-format records” on page 15, and “Undefined-format records” on page 18). The default is `recfm=U`. Any specification of ASA for the record format is ignored.

lrecl= and blksize=

The `lrecl` and `blksize` parameters allow you to set the record size and block size, respectively. The maximum limits on `lrecl` values are as follows:

32771

For input z/OS variable terminals (data length of 32767)

32767

For input z/OS fixed and undefined terminals

32770

For output z/OS variable terminals (data length of 32766)

32766

For output z/OS fixed and undefined terminals

In fixed and undefined terminal files, `blksize` is always the size of `lrecl`. In variable terminal files, `blksize` is always the size of `lrecl` plus 4 bytes. It is not necessary to specify values for `lrecl` and `blksize`. If neither is specified, the default values are used. The default `lrecl` sizes (not including the extra 4 bytes in the `lrecl` of variable length types) are as follows:

- Screen width for output terminals
- 1000 for input z/OS text terminals
- 254 for all other input terminals

space=

This parameter is accepted as an option for terminal I/O, but it is ignored. It does not generate an error.

type=

`type=record` specifies that the file is to be opened for sequential record I/O. The file must be opened as a binary file.

acc=

This parameter is not valid for terminal I/O. If you specify it, your `fopen()` call fails.

password=

This parameter is not valid for terminal I/O. If you specify it, your `fopen()` call fails.

asis

This parameter is accepted as an option for terminal I/O, but it is ignored. It does not generate an error.

byteseek

This parameter is accepted as an option for terminal I/O, but it is ignored. It does not generate an error.

noseek

This parameter is accepted as an option for terminal I/O, but it is ignored. It does not generate an error.

OS

This parameter is not valid for terminal I/O. If you specify it, your `fopen()` call fails.

When you perform input and output in an interactive mode with the terminal, all standard streams and all files with `*` as the first character of their names are associated with the terminal. Output goes to the screen; input comes from the keyboard.

An input EOF can be generated by a `/*` if you open a stream in text mode. If you open the stream in binary or record mode, you can generate an EOF by entering a null string.

ASA characters are not interpreted in terminal I/O.

Opening a terminal file under a shell

Files are opened with a call to `fopen()` in the format `fopen("/dev/tty", "mode")`.

Buffering

z/OS XL C/C++ uses buffers to map byte-level I/O (data stored in records and blocks) to system-level C I/O.

In terminal I/O, line buffering is always in effect.

The `setvbuf()` and `setbuf()` functions can be used to control buffering before any read or write operation to the file. If you want to reset the buffering mode, you must call `setvbuf()` or `setbuf()` before any other operation occurs on a file, because you cannot change the buffering mode after an I/O operation to the file.

Reading from files

You can use the following library functions to read in information from terminal files; see [z/OS C/C++ Runtime Library Reference](#) for more information on these library functions.

- `fread()`
- `fread_unlocked()`
- `fgets()`
- `fgets_unlocked()`
- `gets()`
- `gets_unlocked()`
- `fgetc()`
- `fgetc_unlocked()`
- `getc()`
- `getc_unlocked()`
- `getchar()`
- `getchar_unlocked()`

- `scanf()`
- `scanf_unlocked()`
- `fscanf()`
- `fscanf_unlocked()`
- `vscanf()`
- `vscanf_unlocked()`
- `vfscanf()`
- `vfscanf_unlocked()`

You can set up a SIGIOERR handler to catch read or write system errors. See [Chapter 16, “Debugging I/O programs,” on page 161](#) for more information.

A call to the `rewind()` function clears unread input data in the terminal buffer so that on the next read request, the system waits for more user input.

With z/OS Language Environment, an empty record is considered EOF in binary mode or record mode. This remains in effect until a `rewind()` or `clearerr()` is issued. When the `rewind()` is issued, the buffer is cleared and reading can continue.

Under TSO, the virtual line size of the terminal is used to determine the line length.

When reading from the terminal and the RECFM has been set to be F (for example, by an ALLOCATE under TSO) in binary or record mode, the input is padded with blanks to the record length.

On input, all terminal files opened for output flush their output, no matter what type of file they are and whether a record is complete or not. This includes fixed terminal files that would normally withhold output until a record is completed, as well as text records that normally wait until a new-line or carriage return. In all cases, the data is placed into one line with a blank added to separate output from different terminal files. Fixed terminal files do not pad the output with blanks when flushing this way.

Note: This flush is not the same as a call to `fflush()`, because fixed terminal files do not have incomplete records and text terminal files do not output until the new-line or carriage return. This flush occurs only when actual input is required from the terminal. When data is still in the buffer, that data is read without flushing output terminal files.

Reading from binary files

This discussion includes reading from fixed binary files and from variable or undefined binary files.

Reading from fixed binary files

- Any input that is smaller than the record length is padded with blanks to the record length. The default record length is 254 bytes.
- The carriage return or new-line is not included as part of the data.
- An input line longer than the record length is returned to the calling program on subsequent system reads.

For example, suppose a program requests 30 bytes of user input from an input fixed binary terminal with record length 25. The full 30 bytes of user input returns to satisfy the request, so that you do not need to enter a second line of input.

- An empty input line indicates EOF.

Reading from variable or undefined binary files

These files behave like fixed-length binary files, except that no padding is performed if the input is smaller than the record length.

Reading from text files

This discussion includes reading from fixed text files and from variable or undefined text files.

Reading from fixed text files

- The carriage return indicates the end of the record.
- A new-line character is added as part of the data to indicate the end of an input line.
- If the input is larger than the record length, it is truncated to the record length. The truncation causes SIGIOERR to be raised, if the default action for SIGIOERR is not SIG_IGN.
- When an input line is smaller than the record length, it is not padded with blanks.
- The character sequence /* indicates that the end of the file has been reached.

Reading from variable or undefined text files

These files behave like fixed-length text files.

Reading from record I/O files

This discussion includes reading from fixed record I/O files and from variable or undefined record I/O files.

Reading from fixed record I/O files

- Records smaller than the record length are padded with blanks up to the record length. The default record length is 254 bytes.
- Input record terminal records have an implicit logical record boundary at the record length if the input size exceeds the record length.

If you enter input data larger than the record length, each subsequent block of record-length bytes from the user input satisfies successive read requests.

- The carriage return or new-line is not included as part of the data.
- An empty line indicates an EOF.

Reading from variable or undefined record I/O files

These files behave like fixed-length record files, except that no padding is performed.

Writing to files

You can use the following library functions to write to a terminal file; see [*z/OS C/C++ Runtime Library Reference*](#) for more information on these library functions.

- `fwrite()`
- `fwrite_unlocked()`
- `printf()`
- `printf_unlocked()`
- `fprintf()`
- `fprintf_unlocked()`
- `vprintf()`
- `vprintf_unlocked()`
- `fprintf()`
- `fprintf_unlocked()`
- `puts()`

- `puts_unlocked()`
- `fputs()`
- `fputs_unlocked()`
- `fputc()`
- `fputc_unlocked()`
- `putc()`
- `putc_unlocked()`
- `putchar()`
- `putchar_unlocked()`

If no record length is specified for the output terminal file, it defaults to the virtual line size of the terminal.

On output, records are written one line at a time up to the record length. For all output terminal files, records are not truncated. If you are printing a long string, it wraps around to another line.

Writing to binary files

This discussion includes writing to fixed binary files and to variable or undefined binary files.

Writing to fixed binary files

- Output data is sent to the terminal when the last character of a record is written.
- When closing an output terminal, any unwritten data is padded to the record length with blanks before it is flushed.

Writing to variable or undefined binary files

These files behave the same as fixed-length binary files, except that no padding occurs for output that is smaller than the record length.

Writing to text files

The following control characters are supported:

- \a**
Alarm. Causes the terminal to generate an audible beep.
- \b**
Backspace. Backs up the output position by one byte. If you are at the start of the record, you cannot back up to previous record, and backspace is ignored.
- \f**
Form feed. Sends any unwritten data to the terminal and clears the screen if the environment variable `_EDC_CLEAR_SCREEN` is set. If the variable is not set, the `\f` character is written to the screen.
- \n**
New-line. Sends the preceding unwritten character to the terminal. If no preceding data exists, it sends a single blank character.
- \t**
Horizontal tab. Pads the output record with blanks up to the next tab stop (set at eight characters).
- \v**
Vertical tab. Placed in the output as is.
- \r**
Carriage return. Treated as a new-line, sends preceding unwritten data to the terminal.

Writing to fixed text files

- Lines that are longer than the record length are not truncated. They are split across multiple lines, each LRECL bytes long. Subsequent writes begin on a new line.
- Output data is sent to the terminal when one character more than the record length is written, or when a `\t`, `\n`, or `\f` character is written. In the case of `\f`, output is displayed only if the `_EDC_CLEAR_SCREEN` environment variable is set.
- No padding occurs on output when a record is smaller than the record length.

Writing to variable or undefined text files

These terminal files behave like fixed-length terminal files.

Writing to record I/O files

This discussion includes writing to fixed record I/O files and to variable or undefined record I/O files.

Writing to fixed record I/O files

- Any output record that is smaller than the record length is padded to the record length with blanks, and trailing blanks are displayed.
- If a record is longer than the record length, all data is written to the terminal, wrapping at the record length.
- Output data is sent to the terminal with every record write.

Writing to variable or undefined record I/O files

These files behave like fixed-length record files except that no padding occurs when the output record is smaller than the record length.

Flushing records

The action taken by the `fflush()` library function depends on the file mode. The `fflush()` function only flushes buffers in binary files with Variable or Undefined record format.

If you call one z/OS XL C/C++ program from another z/OS XL C/C++ program by using the ISO C/C++ `system()` function, all open streams are flushed before control is passed to the callee, and again before control is returned to the caller. If you are running with POSIX(ON), a call to the POSIX `system()` function does not flush any streams to the system.

Text streams

- Writing a new record:

Because a new-line character has not been encountered to indicate the end-of-line, `fflush()` takes no action. The record is written as a new record when one of the following takes place:

- A new-line character is written.
- The file is closed.

- Reading a record:

`fflush()` clears a previous `ungetc()` character.

Binary streams

- Writing a new record:

If the file is variable or undefined length in record format, `fflush()` causes the current record to be written out, which in turn causes a new record to be created for subsequent writes. If the file is of fixed record length, no action is taken.

- Reading a record:

`fflush()` clears a previous `ungetc()` character.

Record I/O

- Writing a new record: `fflush()` takes no action.
- Reading a record: `fflush()` takes no action.

Repositioning within files

In terminal I/O, `rewind()` is the only positioning library function available. Using the library functions `fseek()`, `fgetpos()`, `fsetpos()`, and `ftell()` generates an error. See [z/OS C/C++ Runtime Library Reference](#) for more information on these library functions.

When an input terminal reaches an EOF, the `rewind()` function:

1. Clears the EOF condition.
2. Enables the terminal to read again.

You can also use `rewind()` when reading from the terminal to flush out your record buffer for that stream.

Closing files

Use the `fclose()` library function to close a file. z/OS XL C/C++ automatically closes files on normal program termination and attempts to do so under abnormal program termination or abend. When closing a fixed binary terminal, z/OS XL C/C++ pads the last record with blanks if it is incomplete.

See [z/OS C/C++ Runtime Library Reference](#) for more information on this library function.

fldata() behavior

The format of the `fldata()` function is as follows:

```
int fldata(FILE *file, char *filename, fldata_t *info);
```

The `fldata()` function is used to retrieve information about an open stream. The name of the file is returned in `filename` and other information is returned in the `fldata_t` structure, shown in [tFigure 22 on page 139](#). Values specific to this category of I/O are shown in the comment beside the structure element. Additional notes pertaining to this category of I/O follow the figure.

For more information on the `fldata()` function, refer to [z/OS C/C++ Runtime Library Reference](#).

```

struct __fileData {
    unsigned int    __recfmF   : 1, /* */
                  __recfmV   : 1, /* */
                  __recfmU   : 1, /* */
                  __recfmS   : 1, /* always off */
                  __recfmBlk : 1, /* always off */
                  __recfmASA : 1, /* always off */
                  __recfmM   : 1, /* always off */
                  __dsorgPO  : 1, /* N/A -- always off */
                  __dsorgPDsmem : 1, /* N/A -- always off */
                  __dsorgPDsdir : 1, /* N/A -- always off */
                  __dsorgPS  : 1, /* N/A -- always off */
                  __dsorgConcat : 1, /* N/A -- always off */
                  __dsorgMem  : 1, /* N/A -- always off */
                  __dsorgHiper : 1, /* N/A -- always off */
                  __dsorgTemp : 1, /* N/A -- always off */
                  __dsorgVSAM : 1, /* N/A -- always off */
                  __dsorgHFS  : 1, /* N/A -- always off */
                  __openmode : 2, /* one of: */
                              /* __TEXT */
                              /* __BINARY */
                              /* __RECORD */
                  __modeflag : 4, /* combination of: */
                              /* __READ */
                              /* __WRITE */
                              /* __APPEND */
                  __dsorgPDSE : 1, /* N/A -- always off */
                  __reserve2  : 8; /* */
    __device_t      __device; /* __TERMINAL */
    unsigned long   __blksize; /* */
                  __maxreclen; /* */
    unsigned short  __vsamtype; /* N/A */
    unsigned long   __vsamkeylen; /* N/A */
    unsigned long   __vsamRKP; /* N/A */
    char *          __dsname; /* N/A -- always NULL */
    unsigned int    __reserve4; /* */
};
typedef struct __fileData fldata_t;

```

Figure 22. *fldata()* structure

Notes:

1. The *filename* value is dd:ddname if the file is opened by ddname; otherwise, the value is *. The ddname is uppercase.
2. Either __recfmF, __recfmV, or __recfmU will be set according to the recfm parameter specified on the *fopen()* or *freopen()* function call.

Chapter 12. Performing memory file and hiperspace I/O operations

This chapter describes how to perform memory file and hiperspace I/O operations. z/OS XL C/C++ supports files known as *memory files*. Memory files are temporary work files that are stored in main memory rather than in external storage. There are two types of memory files:

- Regular memory files, which exist in your virtual storage
- Hiperspace memory files, which use special storage areas called *hiperspaces*.

Memory files can be written to, read from, and repositioned within like any other type of file. Memory files exist for the life of your root program, unless you explicitly delete them by using the `remove()` or `clrmemf()` functions. The root program is the first `main()` to be invoked. Any `main()` program called by a `system()` call is known as a *child program*. When the root program terminates, z/OS XL C/C++ removes memory files automatically. Memory files may give you better performance than other types of files.

Note: There may not be a one-to-one correspondence between the bytes in a memory file and the bytes in some other external representation of the file, such as a disk file. Applications that mix open modes on a file (for example, writing a file as text file and reading it back as binary) may not port readily from external I/O to memory file I/O.

This chapter describes C I/O streams as they can be used within C++ programs. If you want to use the C++ I/O stream classes instead, see Chapter 4, “Using the Standard C++ Library I/O Stream Classes,” on page 21 for general information. For more detailed information, see [Standard C++ Library Reference](#), which discusses the Standard C++ I/O stream classes.

Using hiperspace operations

Restriction: Hiperspace memory files are not supported in AMODE 64 applications. Attempts to open a memory file with `type=memory(hiperspace)` will be converted to a regular memory file.

On z/OS systems, large memory files can be placed in hiperspaces to reduce memory requirements within your address space.

If your installation supports hiperspaces, and you are not using CICS, you can use hiperspace memory files (see the appropriate book as listed in [z/OS Information Roadmap](#) for more information on hiperspaces). Whereas a regular memory file stores all the file data in your address space, a hiperspace memory file uses one buffer in your address space, and keeps the rest of the data in the hiperspace. Therefore, a hiperspace memory file requires only a certain amount of storage in your address space, regardless of how large the file is. If you use `setvbuf()`, z/OS XL C/C++ may or may not accept your buffer for its internal use. For a hiperspace memory file, if the size of the buffer specified to `setvbuf()` is greater than 4K, then only the first 4K of the user buffer will be used.

Hiperspace memory files may not be shared by multiple threads. A hiperspace memory file that is created on one thread can only be read/written/closed by the same thread.

Opening files

Use the Standard C `fopen()` or `freopen()` library functions to open a memory file.

Using `fopen()` or `freopen()`

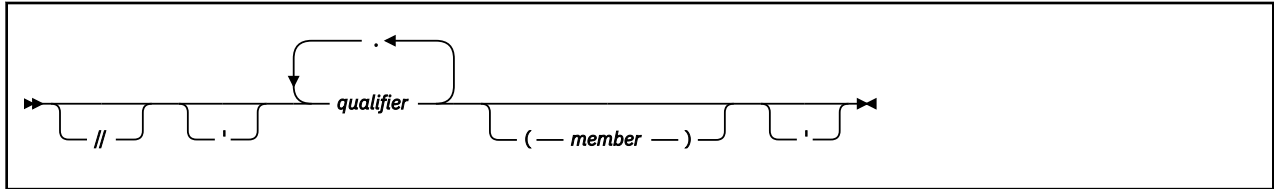
This section describes considerations for using `fopen()` and `freopen()` with memory files. Memory files are always treated as binary streams of bytes, regardless of the parameters you specify on the function call that opens them.

File-naming considerations

When you open a file using `fopen()` or `freopen()`, you must specify the filename (a data set name) or the ddname.

Using a data set name

Files are opened with a call to `fopen()` or `freopen()` in the format `fopen("filename", "mode")`. The following diagram shows the syntax for the *filename* argument on your `fopen()` or `freopen()` call:



The following is a sample construct:

```
'qualifier1.qualifier2(member)'
```

//

Ignored for memory files.

qualifier

There is no restriction on the length of each qualifier. All characters are considered valid. The total number of characters for all of the qualifiers, including periods and a TSO prefix, cannot exceed 44 characters when running POSIX(OFF). Under POSIX(ON), the TSO prefix is not added, and the total number of characters is not limited, except that the full file name, including the member, cannot exceed the limit for a POSIX pathname, currently 1024 characters.

(member)

If you specify a *member*, the data set you are opening is considered to be a simulated PDS or a PDSE. For more information about PDSs and PDSEs, see [“Simulating partitioned data sets” on page 145](#). For members, the member name (including trailing blanks) can be up to 8 characters long. A member name cannot begin with leading blanks.

When you enclose a name in single quotation marks, the name is *fully qualified*. The file opened is the one specified by the name inside the quotation marks. If the name is not fully qualified, z/OS XL C/C++ does one of the following:

- If your system does not use RACF, z/OS XL C/C++ does not add a high-level qualifier to the name you specified.
- If you are running under TSO (batch or interactive), z/OS XL C/C++ appends the TSO user prefix to the front of the name. For example, the statement `fopen("a.b", "w")`; opens a data set *tsopref.A.B*, where *tsopref* is the user prefix. You can set the user prefix by using the TSO PROFILE command with the PREFIX parameter.

Note: The TSO prefix is not added when running POSIX(ON).

- If you are running under MVS batch or IMS (batch or online), z/OS XL C/C++ appends the RACF user ID to the front of the name.

Using a DDname

You can specify names that begin with `dd:`, but z/OS XL C/C++ treats the `dd:` as part of the file name.

z/OS UNIX Considerations

Using the `fork()` library function from z/OS UNIX application programs causes the memory file to be copied into the child process. The memory file data in the child is identical to that of the parent at the time of the `fork()`. The memory file can be used in either the child or the parent, but the data is not visible in the other process.

fopen() and freopen() keywords

Table 22 on page 143 lists the keywords that are available on the `fopen()` and `freopen()` functions, tells you which ones are useful for memory file I/O, and lists the values that are valid for the applicable ones.

Table 22. Keywords for the `fopen()` and `freopen()` functions for memory file I/O

Keyword	Allowed?	Applicable?	Notes
<code>recfm=</code>	Yes	No	This parameter is ignored for memory file and hiperspace I/O. If you specify a <code>RECFM</code> , it must have correct syntax. Otherwise the <code>fopen()</code> call fails.
<code>lrecl=</code>	Yes	No	This parameter is ignored for memory file and hiperspace I/O. If you specify an <code>LRECL</code> , it must have correct syntax. Otherwise <code>fopen()</code> call fails.
<code>blksize=</code>	Yes	No	This parameter is ignored for memory file and hiperspace I/O. If you specify a <code>BLKSIZE</code> , it must have correct syntax. Otherwise <code>fopen()</code> call fails.
<code>acc=</code>	Yes	No	This parameter is ignored for memory file and hiperspace I/O. If you specify an <code>ACC</code> , it must have correct syntax. Otherwise <code>fopen()</code> fails.
<code>password=</code>	No	No	Ignored for memory files.
<code>space=</code>	Yes	No	This parameter is ignored for memory file and hiperspace I/O. If you specify a <code>SPACE</code> , it must have correct syntax. Otherwise, <code>fopen()</code> call fails.
<code>type=</code>	Yes	Yes	Valid values are <code>memory</code> and <code>memory(hiperspace)</code> . See the parameter list below.
<code>asis</code>	Yes	Yes	Enables the use of mixed-case file names.
<code>byteseek</code>	Yes	No	Ignored for memory files, as they use byteseeking by default.
<code>noseek</code>	Yes	No	This parameter is ignored for memory file and hiperspace I/O.
<code>OS</code>	No	No	This parameter is not valid for memory file and hiperspace I/O. If you specify <code>OS</code> , your <code>fopen()</code> call fails.

recfm=

z/OS XL C/C++ parses your specification for these values. If they do not have the correct syntax, your function call fails. If they do, z/OS XL C/C++ ignores their values and continues.

lrecl= and blksize=

z/OS XL C/C++ parses your specification for these values. If they do not have the correct syntax, your function call fails. If they do, z/OS XL C/C++ ignores their values and continues.

acc=

z/OS XL C/C++ parses your specification for these values. If they do not have the correct syntax, your function call fails. If they do, z/OS XL C/C++ ignores their values and continues.

password=

This parameter is not valid for memory file and hiperspace I/O. If you specify `PASSWORD`, your `fopen()` call fails.

space=

z/OS XL C/C++ parses your specification for these values. If they do not have the correct syntax, your function call fails. If they do, z/OS XL C/C++ ignores their values and continues.

type=

To create a memory file, you must specify `type=memory`. You cannot specify `type=record` or `type=blocked`; if you do, `fopen()` or `freopen()` fails. To create a hiperspace memory file, you must specify `type=memory(hiperspace)`.

asis

If you use this parameter, you can specify mixed-case filenames, such as `JaMeS dAtA` or `pErCy.FILE`. If you are running with `POSIX(ON)`, `asis` is the default.

byteseek

This parameter is ignored for memory file and hiperspace I/O.

noseek

This parameter is ignored for memory file and hiperspace I/O.

OS

This parameter is not allowed for memory file and hiperspace I/O. If you specify `OS`, your `fopen()` call fails.

Once a memory file has been created, it can be accessed by the module that created it as well as by any function or module that is subsequently invoked (including modules that are called using the `system()` library function), and by any modules in the current chain of `system()` calls, if you are running with `POSIX(OFF)`. If you are running with `POSIX(ON)`, the `system()` function is the POSIX one, not the ISO C/C++ one, and it does not propagate memory files to a child program. Once the file has been created, you can open it with the same name, without specifying the `type=memory` parameter. You cannot specify `type=record` or `type=blocked` for a memory file.

This is how z/OS XL C/C++ searches for memory files:

1. `fopen("my.file", "w...., type=memory")`; z/OS XL C/C++ checks the open files to see if a file with that name is already open. If not, it creates a memory file.
2. `fopen("my.file", "w.....")`; z/OS XL C/C++ checks the open files to see if a file with that name is already open. If not, it then checks to see whether a memory file exists with that name. If so, it opens the memory file; if not, it creates a disk file.
3. `fopen("my.file", "a....., type=memory")`; z/OS XL C/C++ checks the open files to see if a file with that name is already open. If not, it searches the existing memory files to see whether a memory file exists with that name. If so, z/OS XL C/C++ opens it; if not, it creates a new memory file.
4. `fopen("my.file", "a....")`; z/OS XL C/C++ checks the open files to see if a file with that name is already open. If not, z/OS XL C/C++ searches existing files (both disk and memory) according to file mode, and opens the first file that has that name. If there is no such file, z/OS XL C/C++ creates a disk file.
5. `fopen("my.file", "r...., type=memory")`; z/OS XL C/C++ searches the memory files to see if a file with that name exists. If one does, z/OS XL C/C++ opens it. Otherwise, the `fopen()` call fails.
6. `fopen("my.file", "r....")`; z/OS XL C/C++ searches first through memory files. If it does not find the specified one, it then tries to open a disk file.

If you specify a memory file name that has an asterisk (*) as the first character, a name is created for that file. (You can acquire this name by using `fldata()`.) For example, you can specify `fopen("*", "type=memory")`; . Opening a memory file this way is faster than using the `tmpnam()` function.

z/OS UNIX System Services Considerations: If you have specified `POSIX(ON)`, `fopen("*file.data", "w, type=memory")` does not generate a name for the memory file. Instead, it opens a memory file called `*file.data`. To generate a memory file name when `POSIX(ON)` has been specified, you must use two slashes before the asterisk, shown as follows:

```
fopen("//*file.data", "w, type=memory")
```

You cannot have any blanks or periods in the member name of a memory file. Otherwise, all valid data set names are accepted for memory files. Note that if invalid disk file names are used for memory files, difficulties could occur when you try to port memory file applications to disk-file applications.

Memory files are always opened in fixed binary mode regardless of the open mode. There is no blank padding, and control characters such as the new line are written directly into the file (even if the `fopen()` specifies text mode).

Opening hiperspace files

To create a memory file in hiperspace, specify `type=memory(hiperspace)` on the `fopen()` call that creates the file. If hiperspace is not available, you get a regular memory file. Under systems that do not support hiperspaces, as well as when you are running with `POSIX(ON)` and `TRAP(OFF)`, a specification of `type=memory(hiperspace)` is treated as `type=memory`. Use of `TRAP(OFF)` is not recommended.

You must decide whether a file is to be a hiperspace memory file before you create it. You cannot change a memory file to a hiperspace memory file by specifying `type=memory(hiperspace)` on a subsequent call to `fopen()` or `freopen()`. If the hiperspace to store the file cannot be created, the `fopen()` or `freopen()` call fails.

Once you have created a hiperspace memory file, you do not have to specify `type=memory(hiperspace)` on subsequent function calls that open the file.

If you open a hiperspace memory file for read at the same time that it is opened for write, you can attempt to read extensions made by the writer, even after the EOF flag has been set on by a previous read. If such a read succeeds, the EOF flag is set off until the new EOF is reached. If you have opened a file once for write and one or more times for read, a reader can now read past the original EOF.

Simulating partitioned data sets

You can create memory files that are conceptually grouped as a partitioned data set (PDS). Grouping the files in this way offers the following advantages:

- You can remove all the members of a PDS by stating the data set name.
- You can rename the qualifiers of a PDS without renaming each member individually.

When you establish that a memory file has members, you can rename and remove all the members by specifying the file name and no members, just as with a PDS or PDSE. None of the members can be open for you to perform this action. Once a memory file is created with or without a member, another memory file with the same name (with or without a member) cannot be created as well. For example, if you open memory file `a.b` and write to it, z/OS XL C/C++ does not allow a memory file named `a.b(c)` until you close and remove `a.b`. Also, if you create a memory file named `a.b(mbr1)`, you cannot open a file named `a.b` until you close and remove `a.b(mbr1)`.

Sample program CCNGMF1 (Figure 23 on page 146) demonstrates the removal of all the members of the data set `a.b`. After the call to `remove()`, neither `a.b(mbr1)` nor `a.b(mbr2)` exists.

```

/* this example shows how to remove members of a PDS */
#include <stdio.h>

int main(void)
{
    FILE * fp1, * fp2;
    fp1=fopen("a.b(mbr1)", "w,type=memory");
    fp2=fopen("a.b(mbr2)", "w,type=memory");
    fwrite("hello, world\n", 1, 13, fp1);
    fwrite("hello, world\n", 1, 13, fp2);
    fclose(fp1);
    fclose(fp2);
    remove("a.b");
    fp1=fopen("a.b(mbr1)", "r,type=memory");
    if (fp1 == NULL) {
        perror("fopen()");
        printf("fopen(\"a.b(mbr1)\") failed as expected: "
            "the file has been removed\n");
    }
    else {
        printf("fopen() should have failed\n");
    }
    return(0);
}

```

Figure 23. Removing members of a PDS

Sample program CCNGMF2 ([Figure 24 on page 146](#)) demonstrates the renaming of a PDS from a .b to c .d.

```

/* this example shows how to rename a PDS */
#include <stdio.h>

int main(void)
{
    FILE * fp1, * fp2;

    fp1=fopen("a.b(mbr1)", "w,type=memory");
    fp2=fopen("a.b(mbr2)", "w,type=memory");
    fclose(fp1);
    fclose(fp2);
    rename("a.b", "c.d");

    /* after renaming, you cannot access members of PDS a.b */
    fp1=fopen("a.b(mbr1)", "r,type=memory");
    if (fp1 == NULL) {
        perror("fopen()");
        printf("fopen(\"a.b(mbr1)\") failed as expected: "
            "the file has been renamed\n");
    }
    else {
        printf("fopen() should have failed\n");
    }

    fp2=fopen("c.d(mbr2)", "r,type=memory");
    if (fp2 != NULL) {
        printf("fopen(\"c.c(mbr1)\") worked as expected: "
            "the file has been renamed\n");
    }
    else {
        perror("fopen()");
        printf("fopen() should have worked\n");
    }
    return(0);
}

```

Figure 24. Renaming members of a PDS

Note: If you are using simulated PDSs, you can change either the name of the PDS, or the member name. You cannot rename a .b(mbr1) to either c .d(mbr2) or c .d, but you can rename a .b(mbr1) to a .b(mbr2), and a .b to c .d.

Memory files that are open as a sequential data set cannot be opened again with a member name specified. Also, if a data set is already open with a member name, the sequential data set version with only the data set name cannot be opened. These operations result in `fopen()` returning `NULL`. For example, `fopen()` returns `NULL` in the second line of the following:

```
fp = fopen("a.b", "w, type=memory");  
fp1 = fopen("a.b(m1)", "w, type=memory");
```

You cannot use the `rename()` or `remove()` functions on open files.

Buffering

Regular memory files are not buffered. Any parameters passed to `setvbuf()` are ignored. Each character that you write is written directly to the memory file.

Hiperspace memory files are fully buffered. The size of the I/O buffer in your own address space is 4KB.

If you call `setvbuf()` for a hiperspace memory file:

- If the size value is greater than or equal to 4K, it will be set to 4K, and this buffer size will be used. Otherwise, the size value is ignored, and z/OS XL C/C++ will allocate a buffer.
- If a pointer to a buffer is passed, the buffer must be aligned on a 4K boundary. Otherwise, z/OS XL C/C++ will allocate a buffer.

Reading from files

You can use the following library functions to read information from memory files; see [z/OS C/C++ Runtime Library Reference](#) for more information on these library functions.

- `fread()`
- `fread_unlocked()`
- `fgets()`
- `fgets_unlocked()`
- `gets()`
- `gets_unlocked()`
- `fgetc()`
- `fgetc_unlocked()`
- `getc()`
- `getc_unlocked()`
- `getchar()`
- `getchar_unlocked()`
- `scanf()`
- `scanf_unlocked()`
- `fscanf()`
- `fscanf_unlocked()`
- `vscanf()`
- `vscanf_unlocked()`
- `vfscanf()`
- `vfscanf_unlocked()`

The `gets()`, `getchar()`, `scanf()`, and `vscanf()` functions read from `stdin`, which can be redirected to a memory or hiperspace memory file.

You can open an existing file for read one or more times, even if it is already open for write. You cannot open a file for write if it is already open (for either read or write). If you want to update or truncate a file or append to a file that is already open for reading, you must first close all the other streams that refer to that file.

For memory files, a read operation directly after a write operation without an intervening call to `fflush()`, `fsetpos()`, `fseek()`, or `rewind()` fails. z/OS XL C/C++ treats the following as read operations:

- Calls to read functions that request 0 bytes
- Read requests that fail because of a system error
- Calls to the `ungetc()` function

You can set up a `SIGIOERR` handler to catch read or write system errors that happen when you are using hiperspace memory files. See [Chapter 16, “Debugging I/O programs,” on page 161](#) for more information.

Writing to files

You can use the following library functions to write to a file. See [z/OS C/C++ Runtime Library Reference](#) for more information on these library functions.

- `fwrite()`
- `fwrite_unlocked()`
- `printf()`
- `printf_unlocked()`
- `fprintf()`
- `fprintf_unlocked()`
- `vprintf()`
- `vprintf_unlocked()`
- `fprintf()`
- `fprintf_unlocked()`
- `puts()`
- `puts_unlocked()`
- `fputs()`
- `fputs_unlocked()`
- `fputc()`
- `fputc_unlocked()`
- `putc()`
- `putc_unlocked()`
- `putchar()`
- `putchar_unlocked()`

The `printf()`, `puts()`, `putchar()`, and `vprintf()` functions write to `stdout`, which can be redirected to a memory or hiperspace memory file.

In hiperspace memory files, each library function causes your data to be moved into the buffer in your address space. The buffer is written to hiperspace each time it is filled, or each time you call the `fflush()` library function.

z/OS XL C/C++ counts a call to a write function writing 0 bytes or a write request that fails because of a system error as a write operation. For regular memory files, the only possible system error that can occur is an error in acquiring storage.

Flushing records

`fflush()` does not move data from an internal buffer to a memory file, because the data is written to the memory file as it is generated. However, `fflush()` does make the data visible to readers who have a regular or hiperspace memory file open for reading while a user has it open for writing.

Hiperspace memory files are fully buffered. The `fflush()` function writes data from the internal buffer to the hiperspace.

Any repositioning operation writes data to the hiperspace.

The `fclose()` function also invokes `fflush()` when it detects an incomplete buffer for a file that is open for writing or appending.

ungetc() considerations

`ungetc()` pushes characters back onto the input stream for memory files. `ungetc()` handles only single-byte characters. You can use it to push back as many as four characters onto the `ungetc()` buffer. For every character pushed back with `ungetc()`, `fflush()` backs up the file position by one character and clears all the pushed-back characters from the stream. Backing up the file position may end up going across a record boundary.

If you want `fflush()` to ignore `ungetc()` characters, you can set the `_EDC_COMPAT` environment variable. See Chapter 28, “Using environment variables,” on page 327 for more information.

Repositioning within files

You can use the following library functions to help you position within a memory or hiperspace memory file. See *z/OS C/C++ Runtime Library Reference* for more information on these library functions.

- `fgetpos()`
- `fgetpos_unlocked()`
- `fsetpos()`
- `fsetpos_unlocked()`
- `fseek()`
- `fseek_unlocked()`
- `ftell()`
- `ftell_unlocked()`
- `rewind()`
- `rewind_unlocked()`

Using `fseek()` to seek past the end of a memory file extends the file using null characters. This may cause z/OS XL C/C++ to attempt to allocate more storage than is available as it tries to extend the memory file.

When you use the `fseek()` function with memory files, it supports byte offsets from `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`.

All file positions from `ftell()` are relative byte offsets from the beginning of the file. `fseek()` supports these values as offsets from `SEEK_SET`.

`fgetpos()`, `fseek()` with an offset of `SEEK_CUR`, and `ftell()` handle `ungetc()` characters unless you have set the `_EDC_COMPAT` environment variable, in which case `fgetpos()` and `fseek()` do not. See Chapter 28, “Using environment variables,” on page 327 for more information about `_EDC_COMPAT`. If in handling these characters, if the current position goes beyond the start of the file, `fgetpos()` returns the EOF value, and `ftell()` returns -1.

`fgetpos()` values generated by code from previous releases of the z/OS XL C/C++ compiler are not supported by `fsetpos()`.

Closing files

Use the `fclose()` library function to close a regular or hiperspace memory file. See [z/OS C/C++ Runtime Library Reference](#) for more information on this library function. z/OS XL C/C++ automatically closes memory files at the termination of the C root main environment.

Performance tips

You should use hiperspace memory files instead of regular memory files when they will be large (1MB or greater).

Regular memory files perform more efficiently if large amounts of data (10K or more) are written in one request (that is, if you pass 10K or more of data to the `fwrite()` function). You should use `fopen("*, "type=memory")` both to generate a name for a memory file and to open the file instead of calling `fopen()` with a name returned by `tmpnam()`. You can acquire the file's generated name by using `fldata()`.

Removing memory files

The memory file remains accessible until the file is removed by the `remove()` or `clrmemf()` library functions or until the root program has terminated. You cannot remove an open memory file, except when you use `clrmemf()`. See [z/OS C/C++ Runtime Library Reference](#) for more information on these library functions.

fldata() behavior

The `fldata()` function is used to retrieve information about an open stream; it has the following format:

```
int fldata(FILE *file, char *filename, fldata_t *info);
```

The name of the file is returned in `filename` and other information is returned in the `fldata_t` structure, shown in [Figure 25](#) on [page 151](#). Values specific to this category of I/O are shown in the comment beside the structure element. Additional notes pertaining to this category of I/O follow. For more information on the `fldata()` function, refer to [z/OS C/C++ Runtime Library Reference](#).

```

struct __fileData {
    unsigned int    __recfmF      : 1, /* always on          */
                  __recfmV      : 1, /* always off         */
                  __recfmU      : 1, /* always off         */
                  __recfmS      : 1, /* always off         */
                  __recfmBlk     : 1, /* always off         */
                  __recfmASA     : 1, /* always off         */
                  __recfmM      : 1, /* always off         */
                  __dsorgP0      : 1, /* N/A -- always off */
                  __dsorgPDsmem  : 1, /* N/A -- always off */
                  __dsorgPDsdir  : 1, /* N/A -- always off */
                  __dsorgPS      : 1, /* N/A -- always off */
                  __dsorgConcat  : 1, /* N/A -- always off */
                  __dsorgMem     : 1, /*                    */
                  __dsorgHiper   : 1, /*                    */
                  __dsorgTemp     : 1, /* N/A -- always off */
                  __dsorgVSAM    : 1, /* N/A -- always off */
                  __dsorgHFS     : 1, /* N/A -- always off */
                  __openmode     : 2, /* __BINARY           */
                  __modeflag     : 4, /* combination of:    */
                                /* __READ             */
                                /* __WRITE            */
                                /* __APPEND           */
                                /* __UPDATE           */
                                __dsorgPDSE: 1, /* N/A -- always off */
                                __reserve2 : 8; /*                    */
    __device_t      __device; /* one of:            */
                                /* __MEMORY           */
                                /* __HIPERSPACE       */
    unsigned long    __blksize, /*                    */
                  __maxreclen; /*                    */
    unsigned short   __vsamtype; /* N/A                */
    unsigned long    __vsamkeylen; /* N/A                */
    unsigned long    __vsamRKP; /* N/A                */
    char *           __dsname; /*                    */
    unsigned int     __reserve4; /*                    */
};
typedef struct __fileData fldata_t;

```

Figure 25. *fldata()* structure

Notes:

1. The *filename* is the fully qualified version of the filename specified on the `fopen()` or `freopen()` function call. There are no quotation marks. However, if the filename specified on the `fopen()` or `freopen()` function call begins with an `*`, a unique filename is generated in the format `((n))`, where *n* is an integer.
2. The `__dsorgMem` bit will be set on only for regular memory files.
3. The `__dsorgHiper` bit will be set on only for hiperspace memory files.
4. The `__dsname` is identical to the *filename* value.

Example program

The following examples show the use of a memory file. Program CCNGMF3 (Figure 26 on page 152) creates a memory file, calls program CCNGMF4 (Figure 27 on page 152), and redirects the output of the called program to the memory file. When control returns to the first program, the program reads and prints the string in the memory file.

For more information on the `system()` library function, see [z/OS C/C++ Runtime Library Reference](#).

```

/* this example demonstrates the use of a memory file */
/* part 1 of 2-other file is CCNGMF4 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;
    char buffer[20];
    char *rc;

    /* Open the memory file to create it */
    if ((fp = fopen("PROG.DAT","wb+",type=memory)) != NULL)
    {
        /* Close the memory file so that it can be used as stdout */
        fclose(fp);

        /* Call CCNGMF4 and redirect its output to memory file */
        /* CCNGMF4 must be an executable MODULE */
        system("CCNGMF4 >PROG.DAT");

        /* Now print the string contained in the file */

        fp = fopen("PROG.DAT","rb");
        rc = fgets(buffer,sizeof(buffer),fp);
        if (rc == NULL)
        {
            perror(" Error reading from file ");
            exit(99);
        }
        printf("%s", buffer);
    }

    return(0);
}

```

Figure 26. Memory file example, part 1

Figure 27 on page 152 redirects the output of the called program to the memory file.

```

/* this example demonstrates the use of a memory file */
/* part 2 of 2-other file is CCNGMF3 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char item1[] = "Hello World\n";
    int rc;

    /* Write the data to the stdout which, at this point, has been
    redirected to the memory file */
    rc = fputs(item1,stdout);
    if (rc == EOF) {
        perror("Error putting to file ");
        exit(99);
    }

    return(0);
}

```

Figure 27. Memory file example, part 2

Chapter 13. Performing CICS Transaction Server I/O operations

Restriction: This chapter does not apply to AMODE 64.

z/OS XL C/C++ under CICS Transaction Server for z/OS (CICS TS) supports only three kinds of I/O:

CICS I/O

z/OS XL C/C++ applications can access the CICS I/O commands through the CICS command level interface.

Files

Memory files are the only type of file that z/OS XL C/C++ supports under CICS. Hiperspace files are not supported. VSAM files can be accessed through the CICS command level interface.

CICS data queues

Under CICS, z/OS XL C/C++ implements the standard output (`stdout`) and standard error (`stderr`) streams as CICS transient data queues. These data queues must be defined in the CICS Destination Control table (DCT) by the CICS system administrator before the CICS cold start. Output from all users' transactions that use `stdout` (or `stderr`) is written to the queue in the order of occurrence.

To help differentiate the output, place a user's terminal name, the CICS transaction identifier, and the time at the beginning of each line printed to the queue. The queues are as follows:

Stream	Queue
<code>stdout</code>	CESO
<code>stderr</code>	CESE
<code>stdin</code>	Not supported

To access any other queues, you must use the command level interface.

Note: If you are using the C++ I/O stream classes, the standard stream `cout` maps to `stdout`, which maps to CESO. The standard stream `cerr` and `clog` both map to `stderr`, which maps to CESE. The standard stream `cin` is not supported under CICS.

For more general information about C++ I/O streaming, see Chapter 4, “Using the Standard C++ Library I/O Stream Classes,” on page 21. For more detailed information, see [Standard C++ Library Reference](#), which discusses the Standard C++ I/O stream classes

For information about using z/OS XL C/C++ and z/OS XL C/C++ under CICS TS, see Chapter 46, “Using the CICS Transaction Server (CICS TS),” on page 575. For information on using wide characters in the CICS TS environment, see Chapter 7, “z/OS XL C support for the double-byte character set,” on page 29.

Chapter 14. Language Environment Message file operations

This chapter describes input and output with the z/OS Language Environment message file. This file is write-only. That is, it is nonreadable and nonseekable.

Restriction: This chapter does not apply to AMODE 64. There is no MSGFILE runtime option in AMODE 64. In AMODE 64, the `stderr` stream does not get directed to the Language Environment message file. Anything that would normally go to the Language Environment message file is now directed to the C `stderr` stream, including when `stderr` is directed to `stdout`. For more information on AMODE 64 see Chapter 20, “z/OS 64-bit environment,” on page 221.

The default open mode for the z/OS Language Environment message file is text. Binary and record I/O modes are not supported.

This chapter also describes C I/O streams as they can be used within C++ programs. If you want to use the C++ I/O stream classes instead, see Chapter 4, “Using the Standard C++ Library I/O Stream Classes,” on page 21 for general information. For more detailed information, see *Standard C++ Library Reference*, which discusses the Standard C++ I/O stream classes.

The standard stream `stderr` defaults to using the z/OS Language Environment message file. `stderr` will be directed to file descriptor 2, which is typically your terminal if you are running under one of the z/OS UNIX shells. There are some exceptions, however:

- If the application has allocated the `ddname` in the `MSGFILE (ddname)` runtime parameter, your output will go there. The default is `MSGFILE (SYSOUT)`.
- If the application has issued one of the POSIX `exec()` functions, or it is running in an address space created by the POSIX `fork()` function and the application has not dynamically allocated a `ddname` for `MSGFILE`, then the default is to use file descriptor 2, if one exists. If it doesn't, then the default is to create a message file in the user's current working directory. The message file will have the name that is specified on the message file runtime option, the default being `SYSOUT`.

Opening files

The default is for `stderr` to go to the message file automatically. The message file is available only as `stderr`; you cannot use the `fopen()` or `freopen()` library function to open it.

- `freopen()` with the null string (“”) as filename string will fail.
- Record format (RECFM) is always treated as undefined (U). Logical record length (LRECL) is always treated as 255 (the maximum length defined by z/OS Language Environment message file system write interface).

Reading from files

The z/OS Language Environment message file is nonreadable.

Writing to files

- Data written to the z/OS Language Environment message file is always appended to the end of the file.
- When the data written is longer than 255 bytes, it is written to the z/OS Language Environment message file 255 bytes at a time, with the last write possibly less than 255 bytes. No truncation will occur.
- When the output data is shorter than the actual LRECL of the z/OS Language Environment message file, it is padded with blank characters by the z/OS Language Environment system write interface.

- When the output data is longer than the actual LRECL of the z/OS Language Environment message file, it is split into multiple records by the z/OS Language Environment system write interface. The z/OS Language Environment system write interface splits the output data at the last blank before the LRECL-th byte, and begins writing the next record with the first non-blank character. Note that if there are no blanks in the first LRECL bytes (DBCS for instance), the z/OS Language Environment system write interface splits the output data at the LRECL-th byte. It also closes off any DBCS string on the first record with a X'0F' character, and begins the DBCS string on the next record with a X'0E' character.
- The hex characters X'0E' and X'0F' have special meaning to the z/OS Language Environment system write interface. The z/OS Language Environment system write interface removes adjacent pairs of these characters (normalization).
- You can set up a SIGIOERR handler to catch system write errors. See [Chapter 16, “Debugging I/O programs,”](#) on page 161 for more information.

Flushing buffers

The `fflush()` function has no effect on the z/OS Language Environment message file.

Repositioning within files

The `ftell()`, `fgetpos()`, `fseek()`, and `fsetpos()` functions are not allowed, because z/OS Language Environment message file is a nonseekable file. The `frewind()` function only resets error flags.

You cannot call `fseek()` on `stderr` when it is mapped to MSGFILE (the default routing of `stderr`).

Closing files

Do not use the `fclose()` library function to close the z/OS Language Environment message file. z/OS XL C/C++ automatically closes files on normal program termination and attempts to do so under abnormal program termination or abend.

Chapter 15. CELQPIPI MSGRTN file operations

Restriction: This chapter only applies to AMODE 64 when using preinitialization services.

If the CELQPIPI MSGRTN service routine is specified and the standard stream `stderr` has not been redirected, it defaults to using the CELQPIPI MSGRTN. The remainder of this chapter describes the behavior of the standard stream `stderr` when used in the application.

The CELQPIPI MSGRTN file is write-only. That is, the file is nonreadable and nonseekable. The mode for this file is text append. Binary and record I/O modes are not supported.

Opening files

The default behavior is that standard stream `stderr` will go to the CELQPIPI MSGRTN. You cannot use the `fopen()` or `freopen()` library functions to open or reopen the CELQPIPI MSGRTN.

- Using `fopen()` to assign a stream to `stderr` directs `stderr` away from the CELQPIPI MSGRTN.
- A call to `freopen()` with the null string ("") as the filename string in an attempt to reopen the CELQPIPI MSGRTN using a different mode will fail.
- Record format (RECFM) is always treated as undefined (U). Logical record length (LRECL) is always treated as 255, which is the maximum length defined by the write interface of the CELQPIPI MSGRTN file system.

Reading from files

The CELQPIPI MSGRTN file is nonreadable.

Writing to files

- Data written to the CELQPIPI MSGRTN file is always appended to the end of the file.
- When the data written is longer than 255 bytes, it is written to the CELQPIPI MSGRTN file 255 bytes at a time, with the last write possibly less than 255 bytes. No truncation will occur.
- When the output data is shorter than the actual LRECL of the CELQPIPI MSGRTN file, it is padded with blank characters by the z/OS Language Environment system write interface.

Flushing buffers

The `fflush()` function has no effect on the CELQPIPI MSGRTN file.

Repositioning within files

The `ftell()`, `fgetpos()`, `fseek()`, and `fsetpos()` functions are not allowed, because CELQPIPI MSGRTN file is nonseekable. The `rewind()` function only resets error flags.

Closing files

Do not use the `fclose()` library function to close the CELQPIPI MSGRTN file. z/OS XL C/C++ automatically closes files on normal program termination and attempts to do so under abnormal program termination or abend.

fldata() behavior

The `fldata()` function is used to retrieve information about an open stream. The name of the file is returned in *filename* and other information is returned in the `fldata_t` structure, shown in [Figure 28 on](#)

page 158. Values specific to this category of I/O are shown in the comment beside the structure element. For more information about the `fldata()` function, see [z/OS C/C++ Runtime Library Reference](#).

```

struct __fileData {
    unsigned int  __recfmF : 1,          /*          */
                  __recfmV : 1,          /*          */
                  __recfmU : 1,          /* always on */
                  __recfmS : 1,          /*          */
                  __recfmBlk : 1,        /*          */
                  __recfmASA : 1,        /*          */
                  __recfmM : 1,          /*          */
                  __dsorgP0 : 1,          /*          */
                  __dsorgPDsmem : 1,      /*          */
                  __dsorgPDsdir : 1,      /*          */
                  __dsorgPS : 1,          /* always on */
                  __dsorgConcat : 1,      /*          */
                  __dsorgMem : 1,         /*          */
                  __dsorgHiper : 1,       /*          */
                  __dsorgTemp : 1,        /*          */
                  __dsorgVSAM : 1,        /*          */
                  __dsorgHFS : 1,         /*          */
                  __openmode : 2,         /* __TEXT   */
                  __modeflag : 4,         /* __APPEND  */
                  __dsorgPDSE : 1,        /*          */
                  __vsamRLS : 3,          /*          */
                  __vsamEA : 1,          /*          */
                  __reserve2 : 4;         /*          */
    __device_t    __device;              /* __MSGRTN  */
    unsigned long __blksize,              /* 255       */
                  __maxreclen;           /* 255       */
    union {
        struct {
            unsigned short __vsam_type;  /*          */
            unsigned long  __vsam_keylen; /*          */
            unsigned long  __vsam_RKP;   /*          */
        } __vsam;
        struct {
            unsigned char  __disk_access_method; /*          */
            unsigned char  __disk_noseek_to_seek; /*          */
            long           __disk_reserve[2];     /*          */
        } __disk;
    } __device_specific;
    char *      __dsname;                 /*          */
    unsigned int __reserve4;              /*          */
};
typedef struct __fileData fldata_t;

```

Figure 28. `fldata()` structure

`fldata()` example

Figure 29 on page 158 example shows the retrieval of the file name in *filename* from the open stream pointed to by `stderr` that maps to the CELQPIPI MSGRTN file.

```

#include <stdio.h>
int main(void) {
    FILE *stream;
    char filename[100];
    fldata_t fileinfo;
    int rc;
    stream = stderr;
    rc = fldata(stream, filename, &fileinfo);
    if (rc != 0)
        printf("fldata failed\n");
    else
        printf("filename is %s\n", filename);
}

```

Figure 29. `fldata()` example

The program in Figure 29 on page 158 produces the following output:

```
filename is ((MSGRTN))
```

The example in [Figure 30 on page 159](#) shows the use of the CELQPIPI MSGRTN file as the default stderr.

```
#include <stdio.h>
#include <errno.h>
int main(void) {
    FILE *stream;

    stream = stderr;

    fprintf(stream, "fprintf: Output redirects to the CELQPIPI MSGRTN file.\n");

    errno = ETIME;
    perror("perror: Output redirects to the CELQPIPI MSGRTN file.");
}
```

Figure 30. CELQPIPI MSGRTN example

Chapter 16. Debugging I/O programs

This chapter will help you locate and diagnose problems in programs that use input and output. It discusses several diagnostic methods specific to I/O. Diagnostic methods for I/O errors include:

- Using return codes from I/O functions
- Using errno values and the associated `error()` message
- Using the `__amrc` structure
- Using the `__amrc2` structure

The information provided with the return code of I/O functions and with the `error()` message associated with errno values may help you locate the source of errors and the reason for program failure. Because return codes and errno values do not exist for every possible system I/O failure, return codes and errno values are not useful for diagnosing all I/O errors. This chapter discusses the use of the `__amrc` structure and the `__amrc2` structure. For information on return codes from I/O functions see [z/OS C/C++ Runtime Library Reference](#). For information on errno values and the associated `error()` message see [z/OS Language Environment Debugging Guide](#).

Using the `__amrc` structure

`__amrc` is a structure defined in `stdio.h` (when the compile-time option `LANGLVL (EXTENDED)` or `LANGLVL (LIBEXT)` is in effect) to help you determine errors resulting from an I/O operation. This structure is changed during system I/O and some C specific error situations. When looking at `__amrc`, be sure to copy the structure into a temporary structure of `__amrc` type since any I/O function calls will change the value of `__amrc`. [Figure 31 on page 161](#) shows the `__amrc` structure as it appears in `stdio.h`.

Note: `__amrc` is not used to record I/O errors in UNIX file system files.

```
#if __TARGET_LIB__ >= __EDC_LE
    typedef struct __amrc_type {
#else
    typedef struct {
#endif
        /* The error or warning value from
        * an I/O operation is in __error,
        * __abend, __feedback or __alloc.
        * Look at the value in __last_op
        * to determine how to interpret
        * the __code union. */
        union {
            int __error; /* error from OPEN/CLOSE,
            * GENCB/MODCB/TESTCB/SHOWCB */
            struct {
                unsigned short __syscode, /* system abend code */
                __rc; /* return code */
            } __abend;
        };
    };
};
```

__amrc structure (Part 1 of 2)

Figure 31. `__amrc` structure

```

    struct {
        unsigned char __fdbk_fill,
                     __rc, /* reg 15 */
                     __ftncd, /* function code */
                     __fdbk; /* feedback code */
    } __feedback; 4
    struct {
        unsigned short __svc99_info,
                     __svc99_error;
    } __alloc; 5
} __code;
unsigned int __RBA; 6
/* RBA value returned by VSAM after *
 * an ESDS or KSDS record is written *
 * out; for RRDS it is a calculated *
 * value from the record number. *
 * It may be used in a subsequent *
 * call to flocate. */
unsigned int __last_op; 7
/* #defined below */
struct {
    unsigned int __len_fill; /* __len + 4 */
    unsigned int __len; /* length of msg in __str */
    char __str[120]; /* the actual data */
    unsigned int __parmr0; /* parameter save area (R0) */
    unsigned int __parmr1; /* parameter save area (R1) */
    unsigned int __fill2[2]; /* non-printable bytes */
    char __str2[64]; /* the actual data */
} __msg; 8
/* error message */

#if __EDC_TARGET >= 0x22080000
    unsigned char __rplfdbwd[4]; 9 /* rpl feedback word */
#endif
/* __EDC_TARGET >= 0x22080000 */
#if __EDC_TARGET >= 0x41080000
#ifdef __LP64 10
    unsigned long __XRBA; /* 8 byte RBA */
#elif defined(__LL)
    unsigned long long __XRBA; /* 8 byte RBA */
#else
    unsigned int __XRBA1; /* high half of 8 byte RBA */
    unsigned int __XRBA2; /* low half of 8 byte RBA */
#endif
/* QSAM to BSAM switch reason */
    unsigned char __amrc_noseek_to_seek; 11
    /* padding to make amrc 256 bytes */
    char __amrc_pad[23];
#endif
} __amrc_type;

```

__amrc structure (Part 2 of 2)

1 union{ ... } __code

The error or warning value from an I/O operation is in either `__error`, `__abend`, `__feedback`, or `__alloc`. You must look at `__last_op` to determine how to interpret the `__code` union.

2 __error

`__error` contains the return code from the system macro or utility. Refer to [Table 23 on page 165](#) for further information.

3 __abend

This struct contains the abend code when `errno` is set to indicate a recoverable I/O abend. `__syscode` is the system abend code and `__rc` is the return code. For more information on the abend codes, see the System Codes manual as listed in [z/OS Information Roadmap](#). The macros `__abendcode()` and `__rsncode()` may be set to the abend code and reason code of a TSO CLIST or command when invoked with `system()`.

4 __feedback

This struct is used for VSAM only. The `__rc` stores the VSAM register 15, `__fdbk` stores the VSAM error code or reason code, and `__RBA` stores the RBA after some operations.

5 __alloc

This struct contains errors during `fopen()` or `freopen()` calls when defining files to the system using SVC 99. See the Systems Macros manual, as listed in [z/OS Information Roadmap](#), for more information on these fields as set by SVC 99.

6 __RBA

This is the RBA value returned by VSAM after an ESDS or KSDS record is written out. For a RRDS, it is the calculated value from the record number. It may be used in subsequent calls to `flocate()`. The `__amrc.__RBA` field is defined as an unsigned int, and therefore will only contain a 4-byte RBA value. This field will be set to -1 when the RBA is beyond 4GB in an extended addressable VSAM data set. In this case, the `__XRBA` field should be used.

7 __last_op

Contains a value that indicates the last I/O operation being performed by z/OS XL C/C++ at the time the error occurred. These values are shown in [Table 23 on page 165](#).

8 __msg

This may contain the system error messages from read or write operations emitted from the BSAM SYNADAF macro instruction. This field will not always be filled. If you print this field using the `%s` format, you should print the string starting at the sixth position because of possible null characters found in the first 6 characters. Special messages for PDSEs are contained in the positions 136 through 184. See the Data Administration manual as listed in [z/OS Information Roadmap](#) for more information. This field is used by the SIGIOERR handler.

9 __rplfdbwd

Contains feedback information related to a VSAM RLS failure. This is the feedback code from the IFGRPL control block.

10 __XRBA

This is the 8 byte relative byte address returned by VSAM after an ESDS or KSDS record is written out. For an RRDS, it is the calculated value from the record number. It may be used in subsequent calls to `flocate()`.

11 __amrc_noseek_to_seek

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

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

Sample program CCNGDI1 (Figure 32 on page 164) demonstrates how to print the `__amrc` structure after an error has occurred to get information that may help you to diagnose an I/O error. The program writes to a file until it is full. When the file is full, the program fails. Following the I/O failure, the program makes

a copy of the `__amrc` structure, and prints the number of successful writes to the file, the `errno`, the `__last_op` code, the `abend` system code and the return code.

```
/* this example demonstrates how to print the __amrc structure */
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    FILE *fp;
    __amrc_type save_amrc;
    char buffer[80];
    int i = 0;

    /* open an MVS binary file */

    fp = fopen("testfull.file", "wb, recfm=F, lrecl=80");
    if (fp == NULL) exit(99);

    memset(buffer, 'A', 80);

    /* write to MVS file until it runs out of extents */

    while (fwrite(buffer, 1, 80, fp) == 80)
        ++i;

    save_amrc = *__amrc; /* need copy of __amrc structure */

    printf("number of successful fwrites of 80 bytes = %d\n", i);
}
```

Example of printing the `__amrc` structure (Part 1 of 2)

Figure 32. Example of printing the `__amrc` structure

```
printf("last fwrite errno=%d lastop=%d syscode=%X rc=%d\n",
      errno,
      save_amrc.__last_op,
      save_amrc.__code.__abend.__syscode,
      save_amrc.__code.__abend.__rc);

return 0;
}
```

Example of printing the `__amrc` structure (Part 2 of 2)

Using the `__amrc2` structure

The `__amrc2` structure is an extension of `__amrc`. Only 2 fields are defined for `__amrc2`. Like the `__amrc` structure, `__amrc2` is changed during system I/O and some C specific error situations. [Figure 33 on page 164](#) shows the `__amrc2` structure as it appears in `stdio.h`.

Note: See “Using the SIGIOERR signal” on page 168 for information on restrictions that exist when comparing file pointers if you are using the `__amrc2` structure.

```
struct {
    int      __error2;
    FILE     *__fileptr;
    int      __reserved[6];
}
1
2
*/
*/
```

Figure 33. `__amrc2` structure

1

This field is a secondary error code that is used to store the reason code from specific macros. The `__last_op` codes that can be returned to `__amrc2` are `__BSAM_STOW`, `__BSAM_BLDL`, `__IO_LOCATE`, `__IO_RENAME`, `__IO_CATALOG` and `__IO_UNCATALOG`. For information on the macros associated with these codes, see [Table 23 on page 165](#). For more information about the macros, see [z/OS DFSMSdfp Diagnosis](#).

2

This field, `__fileptr`, of the `__amrc2` structure is used by the signal `SIGIOERR` to pass back a `FILE` pointer that can then be passed to `fldata()` to get the name of the file causing the error. The `__amrc2__fileptr` will be `NULL` if a `SIGIOERR` is raised before the file has been successfully opened.

Using `__last_op` codes

The `__last_op` field is the most important of the `__amrc` fields. It defines the last I/O operation z/OS XL C/C++ was performing at the time of the I/O error. You should note that the structure is neither cleared nor set by non-I/O operations so querying this field outside of a `SIGIOERR` handler should only be done immediately after I/O operations. [Table 23 on page 165](#) lists `__last_op` codes you may receive and where to look for further information.

Table 23. `__last_op` codes and diagnosis information

Code	More Information
<code>__BSAM_BLDL</code>	Sets <code>__error</code> with return code from OS BLDL macro.
<code>__BSAM_CLOSE</code>	Sets <code>__error</code> with return code from OS CLOSE macro.
<code>__BSAM_CLOSE_T</code>	Sets <code>__error</code> with return code from OS CLOSE TYPE=T.
<code>__BSAM_NOTE</code>	NOTE returned 0 unexpectedly, no return code.
<code>__BSAM_OPEN</code>	Sets <code>__error</code> with return code from OS OPEN macro.
<code>__BSAM_POINT</code>	This will not appear as an error <code>lastop</code> .
<code>__BSAM_READ</code>	No return code (either <code>__abend(errno == 92)</code> or <code>__msg(errno == 66)</code> filled in).
<code>__BSAM_STOW</code>	Sets <code>__error</code> with return code from OS STOW macro.
<code>__BSAM_WRITE</code>	No return code (either <code>__abend(errno == 92)</code> or <code>__msg(errno == 65)</code> filled in).
<code>__C_CANNOT_EXTEND</code>	This occurs when an attempt is made to extend a file that allows writing, but cannot be extended. Typically, this is a member of a partitioned data set being opened for update.
<code>__C_DBCS_SI_TRUNCATE</code>	This occurs only when there was not enough room to start a DBCS string and data was written anyway, with an SI to end it. Cannot happen if <code>MB_CUR_MAX</code> is 1.
<code>__C_DBCS_SO_TRUNCATE</code>	This occurs when there is not enough room in a record to start any DBCS string or else when a redundant SO is written to the file before an SI. Cannot happen if <code>MB_CUR_MAX</code> is 1.
<code>__C_DBCS_TRUNCATE</code>	This occurs when writing DBCS data to a text file and there is no room left in a physical record for anymore double byte characters. A new-line is not acceptable at this point. Truncation will continue to occur until an SI is written or the file position is moved. Cannot happen if <code>MB_CUR_MAX</code> is 1.
<code>__C_DBCS_UNEVEN</code>	This occurs when an SI is written before the last double byte character is completed, thereby forcing z/OS XL C/C++ to fill in the last byte of the DBCS string with a padding byte 'X'FE'. Cannot happen if <code>MB_CUR_MAX</code> is 1.
<code>__C_FCBCHECK</code>	Set when z/OS XL C/C++ FCB is corrupted. This is due to a pointer corruption somewhere. File cannot be used after this.

Table 23. `__last_op` codes and diagnosis information (continued)

Code	More Information
<code>__CICS_WRITEQ_TD</code>	Sets <code>__error</code> with error code from EXEC CICS WRITEQ TD.
<code>__C_TRUNCATE</code>	Set when z/OS XL C/C++ truncates output data. Usually this is data written to a text file with no newline such that the record fills up to capacity and subsequent characters cannot be written. For a record I/O file this refers to an <code>fwrite()</code> writing more data than the record can hold. Truncation is always of rightmost data. There is no return code.
<code>__HSP_CREATE</code>	Indicates last op was a DSPSERV CREATE to create a hiperspace for a hiperspace memory file. If CREATE fails, stores abend code in <code>__amrc.__code.__abend.__syscode</code> , reason code in <code>__amrc.__code.__abend.__rc</code> .
<code>__HSP_DELETE</code>	Indicates last op was a DSPSERV DELETE to delete a hiperspace for a hiperspace memory file during termination. If DELETE fails, stores abend code in <code>__amrc.__code.__abend.__syscode</code> , reason code in <code>__amrc.__code.__abend.__rc</code> .
<code>__HSP_EXTEND</code>	Indicates last op was a HSPSERV EXTEND during a write to a hiperspace. If EXTEND fails, stores abend code in <code>__amrc.__code.__abend.__syscode</code> , reason code in <code>__amrc.__code.__abend.__rc</code> .
<code>__HSP_READ</code>	Indicates last op was a HSPSERV READ from a hiperspace. If READ fails, stores abend code in <code>__amrc.__code.__abend.__syscode</code> , reason code in <code>__amrc.__code.__abend.__rc</code> .
<code>__HSP_WRITE</code>	Indicates last op was a HSPSERV WRITE to a hiperspace. If WRITE fails, stores abend code in <code>__amrc.__code.__abend.__syscode</code> , reason code in <code>__amrc.__code.__abend.__rc</code> .
<code>__IO_CATALOG</code>	Sets <code>__error</code> with return code from I/O CAMLST CAT. The associated macro is CATALOG.
<code>__IO_DEVTYPE</code>	Sets <code>__error</code> with return code from I/O DEVTYPE macro.
<code>__IO_INIT</code>	Will never be seen by SIGIOERR exit value given at initialization.
<code>__IO_LOCATE</code>	Sets <code>__error</code> with return code from I/O CAMLST LOCATE.
<code>__IO_OBTAIN</code>	Sets <code>__error</code> with return code from I/O CAMLST OBTAIN.
<code>__IO_RDJFCB</code>	Sets <code>__error</code> with return code from I/O RDJFCB macro.
<code>__IO_RENAME</code>	Sets <code>__error</code> with return code from I/O CAMLST RENAME.
<code>__IO_TRKCALC</code>	Sets <code>__error</code> with return code from I/O TRKCALC macro.
<code>__IO_UNCATALOG</code>	Sets <code>__error</code> with return code from I/O CAMLST UNCAT. The associated macro is CATALOG.
<code>__LFS_CLOSE</code>	Sets <code>__error</code> with reason code from UNIX file system services. Reason code from UNIX file system services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference .
<code>__LFS_FSTAT</code>	Sets <code>__error</code> with reason code from UNIX file system services. Reason code from UNIX file system services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference .
<code>__LFS_LSEEK</code>	Sets <code>__error</code> with reason code from UNIX file system services. Reason code from UNIX file system services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference .

Table 23. `__last_op` codes and diagnosis information (continued)

Code	More Information
<code>__LFS_OPEN</code>	Sets <code>__error</code> with reason code from UNIX file system services. Reason code from UNIX file system services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference .
<code>__LFS_READ</code>	Sets <code>__error</code> with reason code from UNIX file system services. Reason code from UNIX file system services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference .
<code>__LFS_STAT</code>	Sets <code>__error</code> with reason code from UNIX file system services. Reason code from UNIX file system services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference .
<code>__LFS_WRITE</code>	Sets <code>__error</code> with reason code from UNIX file system services. Reason code from UNIX file system services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference .
<code>__OS_CLOSE</code>	Sets <code>__error</code> to result of OS CLOSE macro.
<code>__OS_OPEN</code>	Sets <code>__error</code> to result of OS OPEN macro.
<code>__QSAM_FREEPOOL</code>	This is an intermediate operation. You will only see this if an I/O abend occurred.
<code>__QSAM_GET</code>	<code>__error</code> is not set (if abend (<code>errno</code> == 92), <code>__abend</code> is set, otherwise if read error (<code>errno</code> == 66), look at <code>__msg</code> .
<code>__QSAM_PUT</code>	<code>__error</code> is not set (if abend (<code>errno</code> == 92), <code>__abend</code> is set, otherwise if write error (<code>errno</code> == 65), look at <code>__msg</code> .
<code>__QSAM_TRUNC</code>	This is an intermediate operation. You will only see this if an I/O abend occurred.
<code>__SVC99_ALLOC</code>	Sets <code>__alloc</code> structure with info and error codes from SVC 99 allocation.
<code>__SVC99_ALLOC_NEW</code>	Sets <code>__alloc</code> structure with info and error codes from SVC 99 allocation of NEW file.
<code>__SVC99_UNALLOC</code>	Sets <code>__alloc</code> structure with info and error codes from SVC 99 unallocation. The <code>__QSAM_CLOSE</code> and <code>__QSAM_OPEN</code> codes do not exist. They should be <code>__OS_CLOSE</code> and <code>__OS_OPEN</code> instead.
<code>__TGET_READ</code>	Sets <code>__error</code> with return code from TSO TGET macro.
<code>__TPUT_WRITE</code>	Sets <code>__error</code> with return code from TSO TPUT macro.
<code>__VSAM_CLOSE</code>	Set when the last op was a low level VSAM CLOSE; if the CLOSE fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.
<code>__VSAM_ENDREQ</code>	Set when the last op was a low level VSAM ENDREQ; if the ENDREQ fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.
<code>__VSAM_ERASE</code>	Set when the last op was a low level VSAM ERASE; if the ERASE fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.
<code>__VSAM_GENCB</code>	Set when a low level VSAM GENCB macro fails, sets <code>__rc</code> and <code>__fdbk</code> fields in the <code>__amrc</code> struct.
<code>__VSAM_GET</code>	Set when the last op was a low level VSAM GET; if the GET fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.
<code>__VSAM_MODCB</code>	Set when a low level VSAM MODCB macro fails, sets <code>__rc</code> and <code>__fdbk</code> fields in the <code>__amrc</code> struct.
<code>__VSAM_OPEN_ESDS</code>	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.

Table 23. `__last_op` codes and diagnosis information (continued)

Code	More Information
<code>__VSAM_OPEN_ESDS_PATH</code>	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.
<code>__VSAM_OPEN_FAIL</code>	Set when a low level VSAM OPEN fails, sets <code>__rc</code> and <code>__fdbk</code> fields in the <code>__amrc</code> struct.
<code>__VSAM_OPEN_KSDS</code>	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.
<code>__VSAM_OPEN_KSDS_PATH</code>	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.
<code>__VSAM_OPEN_RRDS</code>	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.
<code>__VSAM_POINT</code>	Set when the last op was a low level VSAM POINT; if the POINT fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.
<code>__VSAM_PUT</code>	Set when the last op was a low level VSAM PUT; if the PUT fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.
<code>__VSAM_SHOWCB</code>	Set when a low level VSAM SHOWCB macro fails, sets <code>__rc</code> and <code>__fdbk</code> fields in the <code>__amrc</code> struct.
<code>__VSAM_TESTCB</code>	Set when a low level VSAM TESTCB macro fails, sets <code>__rc</code> and <code>__fdbk</code> fields in the <code>__amrc</code> struct.

Using the SIGIOERR signal

SIGIOERR is a signal used by the library to pass control to an error handler when an I/O error occurs. The default action for this signal is SIG_IGN. Setting up a SIGIOERR handler is like setting up any other error handler. Example program CCNGDI2 (Figure 34 on page 169) adds a SIGIOERR handler to the example shown in Figure 32 on page 164. Note the way `fldata()` and the `__amrc2` field `__fileptr` are used to get the name of the file that caused the error.


```

#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

#ifdef __cplusplus
extern "C" {
#endif
void iohdlr(int);

#ifdef __cplusplus
}
#endif

int main(void) {
    FILE *fp;
    char buffer[80];
    int i = 0;

    signal(SIGIOERR, iohdlr);

    /* open an MVS binary file */

    fp = fopen("testfull.file", "wb, recfm=F, lrecl=80");
    if (fp == NULL) exit(99);

    memset(buffer, 'A', 80);

    /* write to MVS file until it runs out of extents */

    while (fwrite(buffer, 1, 80, fp) == 80)
        ++i;

    printf("number of successful fwrites of 80 bytes = %d\n", i);

    return 0;
}

void iohdlr (int signum) {
    __amrc_type save_amrc;
    __amrc2_type save_amrc2;
    char filename[FILENAME_MAX];
    fldata_t info;

```

Example of using SIGIOERR (Part 1 of 2)

Figure 34. Example of using SIGIOERR

```

    save_amrc = *__amrc;    /* need copy of __amrc structure */
    save_amrc2 = *__amrc2; /* need copy of __amrc2 structure */

    /* get name of file causing error from fldata */

    if (fldata(save_amrc2.__fileptr, filename, &info) == 0)
        printf("error on file %s\n", filename);

    perror("io handler"); /* give errno message */
    printf("lastop=%d syscode=%X rc=%d\n",
        save_amrc.__last_op,
        save_amrc.__code.__abend.__syscode,
        save_amrc.__code.__abend.__rc);

    signal(SIGIOERR, iohdlr);
}

```

Example of using SIGIOERR (Part 2 of 2)

When control is given to a SIGIOERR handler, the `__amrc2` structure field `__fileptr` will be filled in with a file pointer. The `__amrc2 __fileptr` will be NULL if a SIGIOERR is raised before the file has been successfully opened. The only operation permitted on the file pointer is `fldata()`. This operation can be used to extract information about the file that caused the error. Other than `freopen()` and `fclose()`, all I/O operations will fail since the file pointer is marked invalid. Do not issue `freopen()` or `fclose()` in a SIGIOERR handler that returns control. This will result in unpredictable behavior, likely an abend.

If you choose not to return from the handler, the file is still locked from all operations except `fldata()`, `freopen()`, or `fclose()`. The file is considered open and can prevent other incorrect access, such as an MVS sequential file opened more than once for a write. Like all other files, the file is closed automatically at program termination if it has not been closed explicitly already.

When you exit a SIGIOERR handler and do not return, the state of the file at closing is indeterminate. The state of the file is indeterminate because certain control block fields are not set correctly at the point of error and they do not get corrected unless you return from the handler.

For example, if your handler were invoked due to a truncation error and you performed a `longjmp()` out of your SIGIOERR handler, the file in error would remain open, yet inaccessible to all I/O functions other than `fldata()`, `fclose()`, and `freopen()`. If you were to close the file or it was closed at termination of the program, it is still likely that the record that was truncated will not appear in the final file.

You should be aware that for a standard stream passed across a `system()` call, the state of the file will be indeterminate even after you return to the parent program. For this reason, you should not jump out of a SIGIOERR handler.

I/O with files other than the file causing the error is perfectly valid within a SIGIOERR handler. For example, it is valid to call `printf()` in your SIGIOERR handler if the file causing the error is not `stdout`. Comparing the incoming file pointer to the standard streams is not a reliable mechanism of detecting whether any of the standard streams are in error. This is because the file pointer in some cases is only a pointer to a file structure that points to the same `__file` as the stream supplied by you. The FILE pointers will not be equal if compared, but a comparison of the `__file` fields of the corresponding FILE pointers will be. See the `stdio.h` header file for details of type FILE.

If `stdout` or `stderr` are the originating files of a SIGIOERR, you should open a special log file in your handler to issue messages about the error.

File I/O trace

The file I/O trace is an EBCDIC encoded trace that can be used as a debugging aid for file I/O failures. The files that are traced, the level of detail provided by the trace, and the trace table size can be controlled with the environment variable `_EDC_IO_TRACE`. For more information about `_EDC_IO_TRACE`, see “`_EDC_IO_TRACE`” on page 352. Figure 35 on page 171 shows the result of running a file I/O application with the displayed environment variable setting.

```
EDCTRACE   File I/O Trace   z/OS XL C/C++ Release: 410B0000   09/28/08 09:14:43 AM
```

```
Trace details for /u/bryntco/hle7760/B49300/ut/tst/myfile.dat:
```

```
Trace detail level: 2
Trace buffer size: 2048K
```

```
fopen(/u/bryntco/hle7760/B49300/ut/tst/myfile.dat,w)
```

```
fldata:
```

--recfmF:1..... 0	--dsorgVSAM:1..... 0
--recfmV:1..... 0	--dsorgHFS:1..... 1
--recfmU:1..... 1	--openmode:2..... 1
--recfmS:1..... 0	--modeflag:4..... 2
--recfmBlk:1..... 0	--dsorgPDSE:1..... 0
--recfmASA:1..... 0	--reserve2:4..... 0
--recfmM:1..... 0	--device..... 9
--dsorgP0:1..... 0	--blksize..... 0
--dsorgPDsmem:1... 0	--maxreclen..... 0
--dsorgPDsdir:1... 0	--vsamtype..... 0
--dsorgPS:1..... 0	--vsamkeylen..... 0
--dsorgConcat:1... 0	--vsamRKP..... 0
--dsorgMem:1..... 0	--access_method... 0 (0)
--dsorgHiper:1.... 0	--noseek_to_seek.. 0 (0)
--dsorgTemp:1..... 0	

```
FILE pointer..... 21A34030
```

```
Trace Entries:
```

```
Function Name
openhfs
fwrite
mwphfs
wp124
__pcloseall
cp124
```

```
Trace Type
Entry
Entry
Entry
Entry
Entry
Entry
Entry
```

Figure 35. Sample File I/O Trace

Locating the file I/O trace

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

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

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

The name of the file uses the following format:

```
/path/EDCTRACE.Date.Time.Pid
```

path

Path determined from the previous algorithm.

Date

Date that the trace is taken, appearing in the format `YYYYMMDD`. For example, 20071122 is for November 22, 2007.

Time

Time that the trace is taken, appearing in the format HHMMSS. For example, 115601 is for 11:56:01 a.m.

Pid

Process ID in which the application is running when the trace is taken.

Part 3. Interlanguage calls with z/OS XL C/C++

This part describes z/OS XL C/C++ considerations about interlanguage calls in the z/OS Language Environment. For complete information about interlanguage calls (ILC) with z/OS XL C/C++ and z/OS Language Environment, refer to [z/OS Language Environment Writing Interlanguage Communication Applications](#).

- [Chapter 17, “Using linkage specifications in C or C++,” on page 175](#)

Chapter 17. Using linkage specifications in C or C++

This information describes how you can make calls between C or C++ programs and assembler, COBOL, PL/I, or FORTRAN programs, or other C or C++ programs. For complete information on making interlanguage calls to and from C or C++, see [z/OS Language Environment Writing Interlanguage Communication Applications](#).

With XPLINK compilation, the linkage and parameter-passing mechanisms for C and C++ are identical. If you link to a C function from a C++ program, you should still specify `extern "C"` to avoid name mangling. For more information about XPLINK, see [z/OS Language Environment Programming Guide](#).

Syntax for Linkage in C or C++

You can specify one of the following linkage types:

Linkage Type	Description
C	C linkage (C++ only)
C++	C++ linkage (C++ only, the default for C++)
COBOL	Previously used for linkage to COBOL routines. Maintained for compatibility with COBOL/370 and VS COBOL II. With newer COBOL products, use the REFERENCE, OS, or C linkage type instead.
FORTRAN	FORTRAN linkage
OS	Operating System linkage
OS_DOWNSTACK	XPLINK-enabled operating system linkage
OS_NOSTACK	Minimal operating system linkage (for use with XPLINK)
OS_UPSTACK	Complete operating system linkage (for use with XPLINK)
OS31_NOSTACK	Same as OS_NOSTACK
PLI	Maintained for compatibility with PL/I products prior to the Enterprise PL/I for z/OS product. With newer PL/I products use the C linkage type instead.
REFERENCE	A Language Environment reference linkage that has the same syntax and semantics with and without XPLINK. Unlike OS linkage, REFERENCE linkage is not affected by the OSCALL suboption of XPLINK. It is equivalent to OS_DOWNSTACK in XPLINK mode and OS_UPSTACK in non-XPLINK mode.

Syntax for linkage in C

You can create linkages between C and other languages by using linkage specifications with the following `#pragma linkage` directive, where *identifier* specifies the name of the function and *linkage* specifies the linkage associated with the function.

```
#pragma linkage(identifier, linkage)
```

Syntax for linkage in C++

You can create linkages between C++ and other languages by using linkage specifications with the following syntax, where *linkage* specifies the linkage associated with the function. If z/OS XL C++ does not recognize the linkage type, it uses C linkage.

```
extern "linkage" { [declaration-list] }
extern "linkage" declaration

declaration-list:
```

declaration
declaration-list declaration

Kinds of linkage used by C or C++ interlanguage programs

Table 24 on page 176 describes the kinds of linkage used by C++ interlanguage programs.

Table 24. Linkage used by C or C++ Interlanguage Programs

What calls or is called by a C or C++ program	Linkage used	Description of linkage	C++ Example
GDDM, ISPF, or non-Language Environment conforming assembler	OS	Basic linkage defined by the operating system. OS Linkage allows integer, pointer, and floating point return types.	<code>extern "OS" { ... }</code>
Language Environment conforming assembler, NOXPLINK-compiled C or C++ declared with OS linkage (or C linkage, passing each parameter as a pointer) is to be called from XPLINK-compiled C or C++. Cannot be used on a function definition in XPLINK-compiled code.	OS_UPSTACK	This is the same as OS linkage in NOXPLINK-compiled programs. It is declared this way by the caller when the caller is XPLINK-compiled. The compiler will call glue code to transition from the XPLINK caller to the non-XPLINK callee. Also, see the OSCALL suboption of the XPLINK option in z/OS XL C/C++ User's Guide .	<code>extern "OS_UPSTACK" { ... }</code>
Assembler that does not follow Language Environment conventions.	OS_NOSTACK, OS31_NOSTACK	The compiler does not generate any glue code for this call. It provides the called program with a 72-byte save area pointed to by Register 13, as does OS_UPSTACK, but the save area may not be initialized. In particular, the Language Environment Next Available Byte (NAB) field may not be present. On entry to the called function, Register 15 contains the entry point address and Register 14 contains the return address. Register 1 points to an OS-style argument list. Typically a program would declare an operating system or subsystem assembler routine with this linkage, where such a routine was not Language Environment enabled.	<code>extern "OS31_NOSTACK" { ... }</code>
XPLINK-compiled C or C++ + using OS_DOWNSTACK linkage, or XPLINK-enabled assembler.	OS_DOWNSTACK	As with OS linkage in NOXPLINK-compiled C or C++, the parameters are passed by reference rather than by value. However, parameter and stack management use XPLINK conventions. Also, see the OSCALL suboption of the XPLINK option in z/OS XL C/C++ User's Guide .	<code>extern "OS_DOWNSTACK" { ... }</code>

Table 24. Linkage used by C or C++ Interlanguage Programs (continued)

What calls or is called by a C or C++ program	Linkage used	Description of linkage	C++ Example
<p>The following programs, using by-reference parameter passing:</p> <ul style="list-style-type: none"> • XPLINK-compiled C/C++ programs calling XPLINK functions (C, C++, or Language Environment conforming assembler) • NOXPLINK-compiled C/C++ programs calling NOXPLINK functions (C, C++, or Language Environment conforming assembler) <p>A Language Environment conforming stack frame is always provided. This is not affected by the OSCALL suboption of XPLINK.</p>	REFERENCE	<p>This is the same as OS_DOWNSTACK linkage in XPLINK-compiled programs and OS_UPSTACK in NOXPLINK-compiled programs. Use this for Language Environment-conforming assembler linkage.</p>	<pre>extern "REFERENCE" { ... }</pre>
PL/I	PLI	<p>Modification of OS linkage. It forces the compiler to read and write parameter lists using PL/I linkage conventions. This linkage type extends OS linkage by allowing structures as return types. (When the return type is a structure, the caller allocates a buffer large enough to receive the returned structure and passes it, by reference, as a hidden final argument.)</p> <p>This linkage type is maintained for compatibility with PL/I products prior to the Enterprise PL/I for z/OS product. With newer PL/I products use the C linkage type instead.</p>	<pre>extern "PLI" { ... }</pre>
COBOL	COBOL	<p>Forces the compiler to read and write parameter lists using COBOL linkage conventions. All calls from C++ to COBOL must be void functions.</p> <p>This linkage type is maintained for compatibility with COBOL/370 and VS COBOL II. With newer COBOL products, you can call COBOL functions with the REFERENCE and OS linkage types, which allow integer return types. If the COBOL routine receives parameters by value (a pragmaless call), you can use the C linkage type.</p>	<pre>extern "COBOL" { ... }</pre>
FORTRAN	FORTRAN	<p>Forces the compiler to read and write parameter lists using FORTRAN linkage conventions.</p>	<pre>extern "FORTRAN" { ... }</pre>

Table 24. Linkage used by C or C++ Interlanguage Programs (continued)

What calls or is called by a C or C++ program	Linkage used	Description of linkage	C++ Example
C	C	<p>Use in C++ to force the compiler to read and write parameter lists using C linkage conventions. C code and the Data Window Services (DWS) product both use C linkage.</p> <p>With XPLINK, C and C++ use the same linkage conventions. When this linkage is specified in C++ code, the specified function is known by its function name alone rather than its name and argument types. It cannot be overloaded.</p>	<code>extern "C" { ... }</code>

Using Linkage Specifications in C++

In the following example, a function is prototyped in a piece of C++ code and uses, by default, C++ linkage.

```
void CXX_FUNC (int);    // C++ linkage
```

Note that C++ is case-sensitive, but PL/I, COBOL, assembler, and FORTRAN are not. In these languages, external names are mapped to uppercase. To ensure that external names match across interlanguage calls, code the names in uppercase in the C++ program, supply an appropriate `#pragma map` specification, or use the `NOLONGNAME` compiler option. This will truncate and uppercase names for functions without C++ linkage.

To reference functions defined in other languages, you should use a linkage specification with a literal string that is one of the following:

- C
- COBOL
- FORTRAN
- OS
- OS_DOWNSTACK
- OS_NOSTACK
- OS_UPSTACK
- OS31_NOSTACK
- PLI
- REFERENCE

For example, the following specification declares the two functions `ASMFUNC1` and `ASMFUNC2` to have operating system linkage. The function names are case-sensitive and must match the definition exactly. You should also limit identifiers to 8 or fewer characters.

```
extern "OS" {
    int ASMFUNC1(void);
    int ASMFUNC2(int);
}
```

Use the reference type parameter (`type&`) in C++ prototypes if the called language does not support pass-by-value parameters or if the called routine expects a parameter to be passed by reference.

Note: To have your program be callable by any of these other languages, include an `extern` declaration for the function that the other language will call.

Part 4. Coding: Advanced topics

This part contains the following coding topics:

- [Chapter 18, “Building and using Dynamic Link Libraries \(DLLs\),” on page 181](#)
- [Chapter 19, “Building complex DLLs,” on page 199](#)
- [Chapter 20, “z/OS 64-bit environment,” on page 221](#)
- [Chapter 21, “Using threads in z/OS UNIX applications,” on page 247](#)
- [Chapter 22, “Reentrancy in z/OS XL C/C++,” on page 261](#)
- [Chapter 23, “IEEE Floating-Point ,” on page 267](#)
- [Chapter 24, “Handling error conditions, exceptions, and signals,” on page 273](#)
- [Chapter 25, “Network communications under UNIX System Services,” on page 293](#)
- [Chapter 26, “Interprocess communication using z/OS UNIX,” on page 317](#)
- [Chapter 27, “Using templates in C++ programs,” on page 319](#)
- [Chapter 28, “Using environment variables,” on page 327](#)
- [Chapter 29, “Using hardware built-in functions,” on page 361](#)
- [Chapter 31, “XL C++ 98 applications and C99,” on page 415](#)
- [Chapter 32, “Writing applications for Single UNIX Specification, Version 3,” on page 417](#)
- [Chapter 33, “Saved compile-time options information,” on page 421](#)

Chapter 18. Building and using Dynamic Link Libraries (DLLs)

A dynamic link library (DLL) is a collection of one or more functions or variables in an executable module that is executable or accessible from a separate application module. In an application without DLLs, all external function and variable references are resolved statically at bind time. In a DLL application, external function and variable references are resolved dynamically at run time.

This chapter defines DLL concepts and shows how to build simple DLLs. [Chapter 19, “Building complex DLLs,”](#) on page 199 shows how to build complex DLLs and discusses some of the compatibility issues of DLLs.

There are two types of DLLs: simple and complex. A simple DLL contains only DLL code in which special code sequences are generated by the compiler for referencing functions and external variables, and using function pointers. With these code sequences, a DLL application can reference imported functions and imported variables from a DLL as easily as it can non-imported ones.

A complex DLL contains mixed code, that is, some DLL code and some non-DLL code. A typical complex DLL would contain some C++ code, which is always DLL code, and some C object modules compiled with the NODLL compiler option bound together.

The object code generated by the z/OS XL C++ compiler is always DLL code. Also, the object code generated by the z/OS XL C compiler with either the DLL compiler option or the XPLINK compiler option is DLL code. Other types of object code are non-DLL code. For more information about compiler options for DLLs, see the [z/OS XL C/C++ User's Guide](#).

XPLINK compiled code and non-XPLINK compiled code cannot be statically mixed (with the exception of OS_UPSTACK and OS_NOSTACK (or OS31_NOSTACK) linkages). The XPLINK compiled code can only be bound together with other XPLINK-compiled code. You can mix non-XPLINK compiled DLLs with XPLINK compiled DLLs (the same is true for routines which you load with `fetch()`). The z/OS XL C++ runtime library manages the transitions between the two different linkage styles across the DLL and `fetch()` boundaries.

Notes:

1. There is inherent performance degradation when the z/OS XL C++ runtime library transitions across these boundaries. In order for your application to perform well, these transitions should be made infrequently. When using XPLINK, recompile all parts of the application with the XPLINK compiler option wherever possible.
2. As of z/OS V1R9, all support for the C/C++ IBM Open Class® Library is removed. For new code and enhancements to existing applications, the Standard C++ Library should be used.

Support for DLLs

DLL support is available for applications running under the following systems:

- z/OS batch
- CICS
- IMS
- TSO
- z/OS UNIX

It is not available for applications running under SPC, CSP or MTF.

Note: All potential DLL executable modules are registered in the CICS PPT control table in the CICS environment and are invoked at run time.

DLL concepts and terms

Table 25 on page 182 summarizes important concepts and terminology about DLLs.

Table 25. Summary of DLL concepts and terms

Term	Definition
Application	All the code executed from the time an executable program module is invoked until that program, and any programs it directly or indirectly calls, is terminated.
DLL	An executable module that exports functions, variable definitions, or both, to other DLLs or DLL applications.
DLL application	An application that references imported functions, imported variables, or both, from other DLLs.
DLL code	Object code resulting when C source code is compiled with the DLL or XPLINK compiler options. C++ code is always DLL code.
Executable program (or executable module)	A file that can be loaded and executed on the computer. z/OS supports two types: Load module An executable residing in a PDS. Program object An executable residing in a PDSE or in the UNIX file system.
Exported functions or variables	Functions or variables that are defined in one executable module and can be referenced from another executable module. When an exported function or variable is referenced within the executable module that defines it, the exported function or variable is also non-imported.
Function descriptor	An internal control block containing information needed by compiled code to call a function.
Imported functions and variables	Functions and variables that are not defined in the executable module where the reference is made, but are defined in a referenced DLL.
Non-imported functions and variables	Functions and variables that are defined in the same executable module where a reference to them is made.
Object code (or object module)	A file output from a compiler after processing a source code module, which can subsequently be used to build an executable program module.
Source code (or source module)	A file containing a program written in a programming language.
Variable descriptor	An internal control block containing information about the variable needed by compiled code.
Writable Static Area (WSA)	An area of memory that is modifiable during program execution. Typically, this area contains global variables and function and variable descriptors for DLLs.
XPLINK application	An application that is made up of C and/or C++ object modules that were compiled with the XPLINK compiler option. XPLINK applications are always DLL applications. Since the C/C++ runtime library for XPLINK is packaged as a DLL, any XPLINK executable module that calls a C/C++ runtime library is also importing from a DLL.
XPLINK code	Object code resulting when C or C++ source code is compiled with the XPLINK compiler option. XPLINK code is always DLL code.

Loading a DLL

A DLL is loaded implicitly when an application references an imported variable or calls an imported function. DLLs can be explicitly loaded by calling `dllload()` or `dlopen()`. Due to optimizations performed, the DLL implicit load point may be moved and the DLL will be loaded only if the actual reference occurs.

Loading a DLL implicitly

When an application uses functions or variables defined in a DLL, the compiled code loads the DLL. This implicit load is transparent to the application. The load establishes the required references to functions and variables in the DLL by updating the control information contained in function and variable descriptors.

If the DLL contains static classes, constructors are run when the DLL is loaded. This loading may occur before `main()`; in this case, the corresponding destructors are run once when `main()` returns.

To implicitly load a DLL, do one of the following:

1. Statically initialize a variable pointer to the address of an exported DLL variable.
2. Reference a function pointer that points to an exported function.
3. Call an exported function.
4. Reference (use, modify, or take the address of) an exported variable.
5. Call through a function pointer that points to an exported function.

In the first situation, the DLL is loaded before `main()` is invoked, and if the DLL contains C++ code, constructors are run before `main()` is invoked. In the other situations, the DLL loading may be delayed until the time of the implicit call, although optimization may move this load earlier.

If the DLL application references (imports) an exported DLL variable, that DLL may be implicitly loaded before that DLL application is invoked (not necessarily before `main()` is invoked). With XPLINK, the DLL will always be implicitly loaded before invoking the DLL application that references (imports) a DLL variable or takes the address of a DLL function.

Note: When a DLL is loaded, its writable static is initialized. If the DLL load module contains C++ code, static constructors are run once at initial load time, and static destructors are run once at program termination. Static destructors are run in the reverse order of the static constructors.

Loading a DLL explicitly

The use of DLLs can also be explicitly controlled by the application code at the source level. The application uses explicit source-level calls to one or more runtime services to connect the reference to the definition. The connections for the reference and the definition are made at run time.

The DLL application writer can explicitly call the following runtime services:

- `dllload()`, which loads the DLL and returns a handle to be used in future references to this DLL
- `dllqueryfn()`, which obtains a pointer to a DLL function
- `dllqueryvar()`, which obtains a pointer to a DLL variable
- `dllfree()`, which frees a DLL loaded with `dllload()`

The following runtime services are also available as part of the Single UNIX Specification, Version 3:

- `dlopen()`, which loads the DLL and returns a handle to be used in future references to this DLL
- `dlsym()`, which obtains a pointer to an exported function or exported variable
- `dlclose()`, which frees a DLL that was loaded with `dlopen()`
- `dlerror()`, which returns information about the last DLL failure on this thread that occurred in one of the `dlopen()` family of functions

While you can use both families of explicit DLL services in a single application, you cannot mix usage across those families. So a handle returned by `dllload()` can only be used with `dllqueryfn()`, `dllqueryvar()`, or `dllfree()`. And a handle returned by `dlopen()` can only be used with `dlsym()` and `dlclose()`.

Because the `dlopen()` family of functions is part of the Single UNIX Specification, Version 3, it should be used in new applications whenever cross-platform portability is a concern.

For more information about the runtime services, see [z/OS C/C++ Runtime Library Reference](#).

To explicitly call a DLL in your application:

- Determine the names of the exported functions and variables that you want to use. You can get this information from the DLL provider's documentation or by looking at the definition side-deck file that came with the DLL. A definition side-deck is a directive file that contains an `IMPORT` control statement for each function and variable exported by that DLL.
- If you are using the `dllload()` family of functions, include the DLL header file `<dll.h>` in your application. If you are using the `dlopen()` family of functions, include the DLL header file `<dlfcn.h>` in your application.
- Compile your source as usual.
- Bind your object with the binder using the same `AMODE` value as the DLL.

Note: You do not need to bind with the definition side-deck if you are calling the DLL explicitly with the runtime services, since there are no references from the source code to function or variable names in the DLL, for the binder to resolve. Therefore the DLL will not be loaded until you explicitly load it with the `dllload()` or `dlopen()` runtime service.

Examples of explicit use of a DLL in an application

The following examples show explicit use of a DLL in an application. The first example in [Figure 36 on page 185](#) uses the `dllload()` family of functions.


```

#include <dll.h>
#include <stdio.h>
#include <string.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef int (DLL_FN)(void);

#ifdef __cplusplus
}
#endif

#define FUNCTION      "FUNCTION"
#define VARIABLE      "VARIABLE"

static void Syntax(const char* progName) {
    fprintf(stderr, "Syntax: %s <DLL-name> <type> <identifier>\n"
        "      where\n"
        "      <DLL-name> is the DLL to load,\n"
        "      <type> can be one of FUNCTION or VARIABLE\n"
        "      and <identifier> is the function or variable\n"
        "      to reference\n", progName);
    return;
}

main(int argc, char* argv[]) {
    int value;
    int* varPtr;
    char* dll;
    char* type;
    char* id;
    dllhandle* dllHandle;

    if (argc != 4) {
        Syntax(argv[0]);
        return(4);
    }
    dll = argv[1];
    type = argv[2];
    id = argv[3];

    dllHandle = dllload(dll);
    if (dllHandle == NULL) {
        perror("DLL-Load");
        fprintf(stderr, "Load of DLL %s failed\n", dll);
        return(8);
    }
}

```

Explicit use of a DLL in an application using the `dllload()` family of functions (Part 1 of 2)

Figure 36. Explicit use of a DLL in an application using the `dllload()` family of functions

```

if (strcmp(type, FUNCTION)) {
    if (strcmp(type, VARIABLE)) {
        fprintf(stderr,
            "Type specified was not " FUNCTION " or " VARIABLE "\n");
        Syntax(argv[0]);
        return(8);
    }
    /*
     * variable request, so get address of variable
     */
    varPtr = (int*)(dllqueryvar(dllHandle, id));
    if (varPtr == NULL) {
        perror("DLL-Query-Var");
        fprintf(stderr, "Variable %s not exported from %s\n", id, dll);
        return(8);
    }
    value = *varPtr;
    printf("Variable %s has a value of %d\n", id, value);
}
else {
    /*
     * function request, so get function descriptor and call it
     */
    DLL_FN* fn = (DLL_FN*) (dllqueryfn(dllHandle, id));
    if (fn == NULL) {
        perror("DLL-Query-Fn");
        fprintf(stderr, "Function %s() not exported from %s\n", id, dll);
        return(8);
    }
    value = fn();
    printf("Result of call to %s() is %d\n", id, value);
}
dllfree(dllHandle);

return(0);
}

```

Explicit use of a DLL in an application using the `dllload()` family of functions (Part 2 of 2)

[Figure 37 on page 187](#) shows an example that uses the `dlopen()` family of functions.

```

#define _UNIX03_SOURCE

#include <dlfcn.h>
#include <stdio.h>
#include <string.h>

#ifdef __cplusplus
extern "C" {
#endif

    typedef int (DLL_FN)(void);

#ifdef __cplusplus
}
#endif

#define FUNCTION        "FUNCTION"
#define VARIABLE        "VARIABLE"

static void Syntax(const char* progName) {
    fprintf(stderr, "Syntax: %s <DLL-name> <type> <identifier>\n"
        "      where\n"
        "      <DLL-name> is the DLL to open,\n"
        "      <type> can be one of FUNCTION or VARIABLE,\n"
        "      and <identifier> is the symbol to reference\n"
        "      (either a function or variable, as determined by"
        "      <type>)\n", progName);
    return;
}

main(int argc, char* argv[]) {
    int value;
    void* symPtr;
    char* dll;
    char* type;
    char* id;
    void* dllHandle;

    if (argc != 4) {
        Syntax(argv[0]);
        return(4);
    }

    dll = argv[1];
    type = argv[2];
    id = argv[3];

    dllHandle = dlopen(dll, 0);
    if (dllHandle == NULL) {
        fprintf(stderr, "dlopen() of DLL %s failed: %s\n", dll, dlerror());
        return(8);
    }
}

```

Explicit use of a DLL in an application using the dlopen() family of functions (Part 1 of 2)

Figure 37. Explicit use of a DLL in an application using the dlopen() family of functions

```

/*
 * get address of symbol (may be either function or variable)
 */
symPtr = (int*)(dlsym(dllHandle, id));
if (symPtr == NULL) {
    fprintf(stderr, "dlsym() error: symbol %s not exported from %s: %s\n",
            id, dll, dlerror());
    return(8);
}

if (strcmp(type, FUNCTION)) {
    if (strcmp(type, VARIABLE)) {
        fprintf(stderr,
            "Type specified was not " FUNCTION " or " VARIABLE "\n");
        Syntax(argv[0]);
        return(8);
    }
}
/*
 * variable request, so display its value
 */
value = *(int *)symPtr;
printf("Variable %s has a value of %d\n", id, value);
}
else {
    /*
     * function request, so call it and display its return value
     */
    value = ((DLL_FN *)symPtr)();
    printf("Result of call to %s() is %d\n", id, value);
}
dlclose(dllHandle);

return(0);
}

```

Explicit use of a DLL in an application using the `dlopen()` family of functions (Part 2 of 2)

Managing the use of DLLs when running DLL applications

This section describes how z/OS XL C/C++ manages loading, sharing and freeing DLLs when you run a DLL application.

Loading DLLs

When you load a DLL for the first time, either implicitly or by an explicit `dllload()` or `dlopen()`, writable static is initialized. If the DLL is written in C++ and contains static objects, then their constructors are run.

You can load DLLs from a UNIX file system as well as from conventional data sets. The following list specifies the order of a search for unambiguous and ambiguous file names.

- **Unambiguous file names**

- If the file has an unambiguous z/OS UNIX file system name (it starts with a `./` or contains a `/`), the file is searched for only in the UNIX file system.
- If the file has an unambiguous MVS name, and starts with two slashes (`//`), the file is only searched for in MVS.

- **Ambiguous file names**

For ambiguous cases, the settings for POSIX are checked.

- When specifying the `POSIX(ON)` runtime option, the runtime library attempts to load the DLL as follows:

1. An attempt is made to load the DLL from the UNIX file system. This is done using the system service BPX1LOD. For more information on this service, see [z/OS UNIX System Services Programming: Assembler Callable Services Reference](#).

If the environment variable LIBPATH is set, each directory listed will be searched for the DLL. See Chapter 28, “Using environment variables,” on page 327 for information on LIBPATH. Otherwise the current directory will be searched for the DLL. Note that a search for the DLL in the UNIX file system is case-sensitive.

- If the DLL is found and contains an external link name of eight characters or less, the uppercase external link name is used to attempt a LOAD from the caller's MVS load library search order. If the DLL is not found or the external link name is more than eight characters, then the load fails.
 - If the DLL is found and its sticky bit is on, any suffix is stripped off. Next, the name is converted to uppercase, and the base DLL name is used to attempt a LOAD from the caller's MVS load library search order. If the DLL is not found or the base DLL name is more than eight characters, the version of the DLL in the UNIX file system is loaded.
 - If the DLL is found and does not fall into one of the previous two cases, a load from the UNIX file system is attempted.
2. If the DLL could not be loaded from the UNIX file system, an attempt is made to load the DLL from the caller's MVS load library search order. This is done by calling the LOAD service with the DLL name, which must be eight characters or less (it will be converted to uppercase). LOAD searches for it in the following sequence:
 - a. Runtime library services (if active)
 - b. Job Pack Area (JPA)
 - c. TASKLIB
 - d. STEPLIB or JOBLIB. If both are allocated, the system searches STEPLIB and ignores JOBLIB.
 - e. LPA
 - f. Libraries in the linklist

For more information, see [z/OS MVS Initialization and Tuning Guide](#).

- When POSIX(OFF) is specified the sequence is reversed.
 - An attempt to load the DLL is made from the caller's MVS load library search order.
 - If the DLL could not be loaded from the caller's MVS load library then an attempt is made to load the DLL from the UNIX file system.

Recommendation: All DLLs used by an application should be referred to by unique names, whether ambiguous or not. Using multiple names for the same DLL (eg. aliases or symlinks) may result in a decrease in DLL load performance. The use of UNIX file system symbolic links by themselves will not degrade performance, as long as the application refers to the DLL solely through the symbolic link name. To help ensure this, when building an application with implicit DLL references always use the same side deck for each DLL. Also, make sure that explicit DLL references with `dllload()` or `dlopen()` specify the same DLL name (case matters for UNIX file system loads).

Changing the search order for DLLs while the application is running (eg. changing LIBPATH) may result in errors if ambiguous file names are used.

Sharing DLLs

DLLs are shared at the enclave level (as defined by the z/OS Language Environment). A referenced DLL is loaded only once per enclave and only one copy of the writable static is created or maintained per DLL per enclave. Thus, one copy of a DLL serves all modules in an enclave regardless of whether the DLL is loaded implicitly or explicitly. You can access the same DLL within an enclave both implicitly and by explicit runtime services.

All accesses to a variable in a DLL in an enclave refer to the only copy of that variable. All accesses to a function in a DLL in an enclave refer to the only copy of that function.

Although only one copy of a DLL is maintained per enclave, multiple logical loads are counted and used to determine when the DLL can be deleted. For a given DLL in a given enclave, there is one logical load for each explicit `dllload()` or `dlopen()` request. DLLs that are referenced implicitly may be logically loaded at application initialization time if the application references any data exported by the DLL, or the logical load may occur during the first implicit call to a function exported by the DLL.

DLLs are not shared in a nested enclave environment. Only the enclave that loaded the DLL can access functions and variables.

Freeing DLLs

You can free explicitly loaded DLLs with a `dllfree()` or `dllclose()` request. This request is optional because the DLLs are automatically deleted by the runtime library when the enclave is terminated.

Implicitly loaded DLLs cannot be deleted from the DLL application code. They are deleted by the runtime library at enclave termination. Therefore, if a DLL has been both explicitly and implicitly loaded, the DLL can only be deleted by the run time when the enclave is terminated.

Creating a DLL or a DLL application

Building a DLL or a DLL application is similar to creating a C or C++ application. It involves the following steps:

1. Writing your source code
2. Compiling your source code
3. Binding your object modules

Building a simple DLL

This section shows how to build a simple DLL in C and C++, using techniques that export externally-linked functions and variables to DLL users. These techniques include:

- The `#pragma export` directive
- The `_Export` keyword
- The `EXPORTALL` compiler option

Both the `_Export` keyword and the `export` directive are used to specify functions and variables.

The `EXPORTALL` compiler option is used to export all defined functions and variables. Using the `EXPORTALL` compiler option means that all defined functions and variables are accessible by all users of the given DLL.

Notes:

1. If the `EXPORTALL` compiler option is used, then neither `#pragma export` nor `_Export` is required in your code.
2. Exporting all functions and variables has a performance penalty, especially when the IPA compiler option is used to build the DLL.

For more information, see:

- The `EXPORTALL` | `NOEXPORTALL` compiler option in [z/OS XL C/C++ User's Guide](#)
- The `_Export` qualifier (C++ only) in [z/OS XL C/C++ Language Reference](#)
- The `#pragma export` directive in [z/OS XL C/C++ User's Guide](#)

Example of building a simple C DLL

To build a simple C DLL, use the `#pragma export` directive to export specific external functions and variables as shown in [Figure 38 on page 191](#).

```

#pragma export(bopen)
#pragma export(bclose)
#pragma export(bread)
#pragma export(bwrite)
int bopen(const char* file, const char* mode) {
    ...
}
int bclose(int) {
    ...
}
int bread(int bytes) {
    ...
}
int bwrite(int bytes) {
    ...
}
#pragma export(berror)
int berror;
char buffer[1024];
...

```

Figure 38. Using #pragma export to create a DLL executable module named BASICIO

This example exports the functions `bopen()`, `bclose()`, `bread()`, `bwrite()`, and the variable `berror`. The variable `buffer` is not exported.

Compiling with the `EXPORTALL` compiler option would export all the functions and the `buffer` variable.

Example of building a simple C++ DLL

To build a simple C++ DLL, use the `_Export` keyword or the `#pragma export` directive to export specific external functions and variables. Ensure that classes and class members are exported correctly, especially if they use templates.

For example, [Figure 39 on page 191](#) shows how to create a DLL executable module named `triangle` using the `#pragma export` directive. This example exports the functions `getarea()`, `getperim()`, the static member `objectCount`, and the constructor for class `triangle`.

```

class triangle
{
public:
    static int objectCount;
    getarea();
    getperim();
    triangle(void);
};
#pragma export(triangle::objectCount)
#pragma export(triangle::getarea())
#pragma export(triangle::getperim())
#pragma export(triangle::triangle(void))

```

Figure 39. Using #pragma export to create the triangle DLL executable module

Similarly, [Figure 40 on page 191](#) shows how to create a DLL executable module named `triangle` using the `_Export` keyword:

```

{
public:
    static int _Export objectCount;
    double _Export getarea();
    double _Export getperim();
    _Export triangle::triangle(void);
};

```

Figure 40. Using _Export to create the triangle DLL executable module

There are some restrictions when using the `_Export` keyword.

- Do not inline the function if you apply the `_Export` keyword to the function declaration, as in [Figure 40 on page 191](#)
- Always export constructors and destructors

If you apply the `_Export` keyword to a class, then it automatically exports the static members, defined functions, constructors, and destructors of that class, as shown in the following example. This behavior is the same as using the `EXPORTALL` compiler option.

```
class triangle
{
public:
    static int objectCount;
    double getarea();
    double getperim();
    triangle(void);
};
```

Compiling your code

For C source code compiled **without** using the `DLL` or `XPLINK` compiler options, that code cannot reference (import) functions or variables that are exported by a DLL. `NODLL` is the default when compiling C source code, and the `XPLINK` compiler option is not used. C source code compiled with the `DLL` or `XPLINK` compiler options, and all C++ source code, can reference exported functions and variables. Source code that can reference exported functions and variables is called DLL application code. It need not itself be a DLL, in that it may not itself export any functions or variables.

When compiling DLL application source code, the compiler generates object code in such a way that references to external functions and variables can be resolved statically or dynamically (that is, resolved to a DLL). If you are uncertain whether non-`XPLINK` C source code references a DLL, you should specify the `DLL` or `XPLINK` compiler options. Compiling source code as DLL application code eliminates the potential compatibility problems that may occur when binding DLL application code with non-DLL application code. See [Chapter 19, “Building complex DLLs,” on page 199](#) for more information on compatibility issues.

The decision to use `XPLINK` needs to be made independently from the decision to build a DLL application. While `XPLINK` compiled code is always DLL application code, the `XPLINK` and non-`XPLINK` function call linkages are different. There is DLL compatibility for `XPLINK` and non-`XPLINK` at the DLL boundary, but `XPLINK` and non-`XPLINK` object modules cannot be mixed in the same DLL. Also, there is a performance penalty when transitioning between `XPLINK` and non-`XPLINK` DLLs (and vice versa). It is best to have a DLL application made up of all `XPLINK` or all non-`XPLINK` executable modules to the extent that is possible. For more information on `XPLINK`, see [“Using the XPLINK option” on page 457](#).

Binding your code

When creating a DLL, the binder automatically creates a definition side-deck that describes the functions and the variables that can be imported by DLL applications. You must provide the generated definition side-deck to all users of the DLL. Any DLL application that implicitly loads the DLL must include the definition side-deck when they bind. For information about creating a side-deck, refer to [Binding z/OS XL C/C++ programs in z/OS XL C/C++ User's Guide](#).

Note: You can choose to store your DLL in a PDS load library, but only if it is non-`XPLINK`. Otherwise, it must be stored in a PDSE load library or in the UNIX file system. To target a PDS load library, prelink and link your code rather than using the binder. For information on prelinking and linking, see [Prelinker and linkage editor options in z/OS XL C/C++ User's Guide](#).

When binding the C object module as shown in [Figure 38 on page 191](#), the binder generates the following definition side-deck:

```
IMPORT CODE,BASICIO,'bopen'  
IMPORT CODE,BASICIO,'bclose'  
IMPORT CODE,BASICIO,'bread'  
IMPORT CODE,BASICIO,'bwrite'  
IMPORT DATA,BASICIO,'beerror'
```

Note: You should also provide a header file containing the prototypes for exported functions and external variable declarations for exported variables.

When binding the C++ object modules shown in [Figure 39 on page 191](#), the binder generates the following definition side-deck.

```
IMPORT CODE,TRIANGLE,'getarea__8triangleFv'  
IMPORT CODE,TRIANGLE,'getperim__8triangleFv'  
IMPORT CODE,TRIANGLE,'__ct__8triangleFv'
```

You can edit the definition side-deck to remove any functions and variables that you do not want to export. You must maintain the file as a binary file with fixed format and a record length of 80 bytes. Also, use proper binder continuation rules if the IMPORT statement spans multiple lines, and you change the length of the statement. In the above example, if you do not want to expose `getperim()`, remove the control statement `IMPORT CODE,TRIANGLE,'getperim__8triangleFv'` from the definition side-deck.

Notes:

1. Removing functions and variables from the definition side-deck does not minimize the performance impact caused by specifying the EXPORTALL compiler option.
2. Editing the side-deck is not recommended. If the DLL name needs to be changed, you should bind using the appropriate name. Instead of using the EXPORTALL compiler option, you should remove unnecessary IMPORT statements by using explicit `#pragma export` statements or `_Export` directives.

The definition side-deck contains mangled names of exported C++ functions, such as `getarea__8triangleFv`. To find the original function or variable name in your source module, review the compiler listing, the binder map, or use the CXXFILT utility, if you do not have access to the listings. This will permit you to see both the mangled and demangled names. For more information, see [filter utility](#) in [z/OS XL C/C++ User's Guide](#).

Building a simple DLL application

A simple DLL application contains object modules that are made up of only DLL-code. The application may consist of multiple source modules. Some of the source modules may contain references to imported functions, imported variables, or both.

Steps for using an implicitly loaded DLL in your simple DLL application

About this task

Perform the following steps to use an implicitly loaded DLL (sometimes called a load-on-call DLL) in your simple DLL application:

Procedure

1. Write your code as you would if the functions were statically bound.
2. Compile as follows:
 - Compile your non-XPLINK application C source files with the following compiler options:
 - DLL

- RENT
- LONGNAME

These options instruct the compiler to generate special code when calling functions and referencing external variables. If you are using z/OS UNIX, RENT and LONGNAME are already the defaults, so compile as:

```
c89 -W c,DLL ...
```

- Compile your C++ source files normally. A C++ application is always DLL code.
- For XPLINK, compile your C and C++ source files with the XPLINK compiler option. XPLINK compiled C and C++ source is always DLL code.

3. Bind your object modules as follows:

- If you are using z/OS Batch, use the IBM-supplied procedure when you bind your object modules. You must choose the appropriate procedures for XPLINK or non-XPLINK.
- If you are not using the IBM-supplied procedure, specify the RENT, DYNAM(DLL), and CASE(MIXED) binder options when you bind your object modules.

Note: XPLINK and non-XPLINK use different z/OS Language Environment libraries, and XPLINK requires the C runtime library side-deck for resolution of C runtime library function calls. For more information, see "Planning to Link-Edit and Run" in *z/OS Language Environment Programming Guide*.

- If you are using z/OS UNIX specify the following option for the bind step for c89 or c++.

```
c89 -W 1,DLL
```

If you are using XPLINK, also add the XPLINK option, so that the c89 utility will use the correct z/OS Language Environment libraries and side-decks:

```
c89 -W 1,DLL,XPLINK ...
```

- Include the definition side-deck from the DLL provider in the set of object modules to bind. The binder uses the definition side-deck to resolve references to functions and variables defined in the DLL. If you are referencing multiple DLLs, you must include multiple definition side-decks.

Note: Definition side-decks can not be resolved by automatic library call (autocall) processing, so you must specify an INCLUDE statement to explicitly include a definition side-deck for each referenced DLL.

Results

The following code fragment illustrates how an application can use the DLL described previously. Compile normally and bind with the definition side-deck provided with the TRIANGLE DLL.

```
extern int getarea(); /* function prototype */
main () {
    ...
    getarea();        /* imported function reference */
    ...
}
```

See [Figure 41 on page 195](#) for a summary of the processing steps required for the application (and related DLLs).

Creating and using DLLs

[Figure 41 on page 195](#) summarizes the use of DLLs for both the DLL provider and for the writer of applications that use them. In this example, application ABC is referencing functions and variables from two DLLs, XYZ and PQR. The connection between DLL preparation and application preparation is shown. Each DLL shown contains a single compilation unit. The same general scheme applies for DLLs composed

of multiple compilation units, except that they have multiple compiles and a single bind for each DLL. For simplicity, this example assumes the following:

- ABC does not export variables or functions.
- XYZ and PQR do not use other DLLs.
- The application is completely non-XPLINK and written in C.

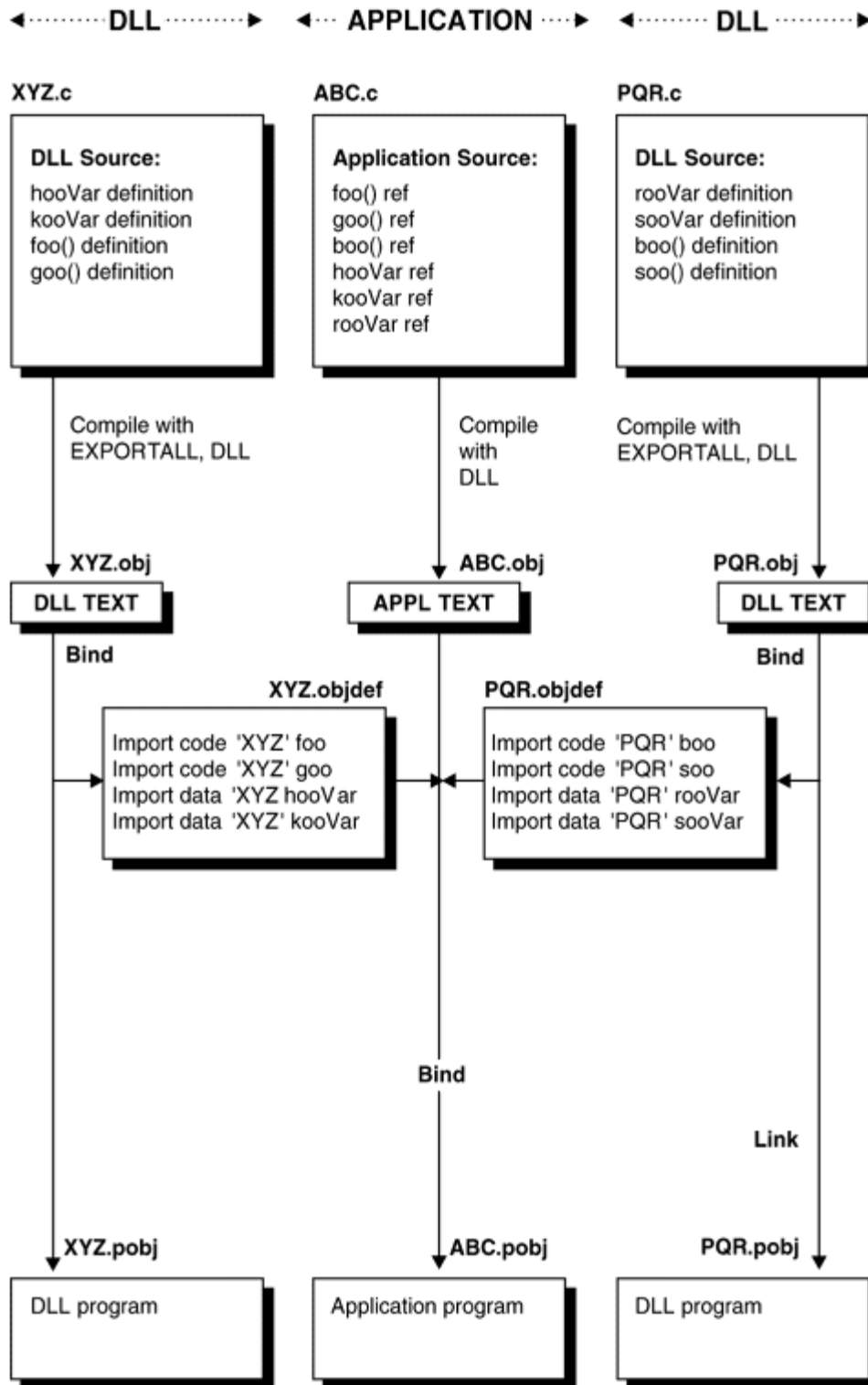


Figure 41. Summary of DLL and DLL application preparation and usage

DLL restrictions

Consider the following restrictions when creating DLLs and DLL applications:

- The entry point for a DLL must be either z/OS XL C/C++ or Language Environment conforming. An entry point is considered Language Environment conforming if it includes CEESTART or if it was compiled using a Language Environment conforming compiler.

Note: If the entry point for a DLL does not meet either of the above conditions, Language Environment issues an error and terminates the application.

- In a DLL application that contains `main()`, `main()` cannot be exported.
- The AMODE of a DLL application must be the same as the AMODE of the DLL that it calls.
- DLL facilities are not available:
 - Under MTF, CSP or SPC
 - To application programs with `main()` written in PL/I that dynamically call z/OS XL C functions
- You cannot implicitly or explicitly perform a physical load of a DLL while running C++ static destructors. However, a logical load of a DLL (meaning that the DLL has previously been loaded into the enclave) is allowed from a static destructor. In this case, references from the load module containing the static destructor to the previously-loaded DLL are resolved.
- If a DLL contains static objects, the constructors are called during DLL load. ISO C++ requires that the global objects must be defined within the same compilation unit, but does not specify any order for these to be called; hence the objects are constructed in the order that they are defined. z/OS XL C/C++ enhances the standard behavior by providing `#pragma priority` to control the construction order for all global objects within the same execution load module. For more information, see [#pragma priority \(C++ only\)](#) in *z/OS XL C/C++ Language Reference* for the details of this pragma. A DLL is one execution load module and the `#pragma priority` allows you to control global object construction within a single DLL. On the other hand, you still have no control over the initialization order across different DLLs, or across a DLL application and the DLLs it references. If such order is important, the DLL provider has to define a protocol for applications to follow so that the interaction between the DLL and the applications happens in the required manner. The protocol must be part of the DLL interface design. Take note of the restriction in the previous bullet when defining such a protocol. A simple example would be requiring an application to call a `setup()` function, which is exported by a DLL, before any other references to the same DLL are made. More elaborate designs are possible. The techniques for controlling static initialization are well-discussed in C++ literature; you can reference, for example, Item 47 of Scott Meyers's *Effective C++, 50 Specific Ways to Improve Your Programs and Designs*.
- You cannot use the functions `set_new_handler()` or `set_unexpected()` in a DLL if the DLL application is expected to invoke the new handler or unexpected function routines.
- When using the explicit DLL functions in a multithreaded environment, avoid any situation where one thread frees a DLL while another thread calls any of the DLL functions. For example, this situation occurs when a `main()` function uses `dllload()` or `dlopen()` to load a DLL, and then creates a thread that uses the `ftw()` function. The `ftw()` target function routine is in the DLL. If the `main()` function uses `dllfree()` or `dlclose()` to free the DLL, but the created thread uses `ftw()` at any point, you will get an abend.

To avoid a situation where one thread frees a DLL while another thread calls a DLL function, do either of the following:

- Do not free any DLLs by using `dllfree()` or `dlclose()` (the z/OS Language Environment will free them when the enclave is terminated).
- Have the `main()` function call `dllfree()` or `dlclose()` only after all threads have been terminated.
- For DLLs to be processed by IPA, they must contain at least one function or method. Data-only DLLs will result in a compilation error.
- Use of circular DLLs may result in unpredictable behavior related to the initialization of non-local static objects. For example, if a static constructor (being run as part of loading DLL "A") causes another DLL

"B" to be loaded, then DLL "B" (or any other DLLs that "B" causes to be loaded before static constructors for DLL "A" have completed) cannot expect non-local static objects in "A" to be initialized (that is what static constructors do). You should ensure that non-local static objects are initialized before they are used, by coding techniques such as counters or by placing the static objects inside functions.

- DLLs are enclave-level resources and, when opening and closing DLLs in a multithreaded environment, an application must control DLL load ordering with its own serialization mechanism to avoid unpredictable results.

Example: Unless the application controls the order of DLL loads, unpredictable results can occur when different threads perform the following operations at the same time:

- One thread uses a global symbol object handle, obtained via `dlopen()`, to search for a symbol whose name has been defined in various DLLs with different values.
- Another thread closes the DLL that defines the symbol whose value is being sought.

Improving performance

This section contains some hints on using DLLs efficiently. Effective use of DLLs may improve the performance of your application. Following are some suggestions that may improve performance:

- If you are using a particular DLL frequently across multiple address spaces, the DLL can be installed in the LPA or ELPA. When the DLL resides in a PDSE, the dynamic LPA services should be used (this will always be the case for XPLINK applications). Installing in the LPA/ELPA may give you the performance benefits of a single rather than multiple load of the DLL.
- When writing XPLINK applications, avoid frequent calls from XPLINK to non-XPLINK DLLs, and vice-versa. These transitions are expensive, so you should build as much of the application as possible as either XPLINK or non-XPLINK. When there is a relatively large amount of function calls compared to the rest of the code, the performance of an XPLINK application can be significantly better than non-XPLINK. It is acceptable to make calls between XPLINK and non-XPLINK, when a relatively large amount of processing will be done once the call is made.
- Be sure to specify the RENT option when you bind your code. Otherwise, each load of a DLL results in a separately loaded DLL with its own writable static. Besides the performance implications of this, you are likely to get incorrect results if the DLL exports variables (data).
- Group external variables into one external structure.
- When using z/OS UNIX avoid unnecessary load attempts.

z/OS Language Environment supports loading a DLL residing in the UNIX file system or a data set. However, the location from which it tries to load the DLL first varies depending whether your application runs with the runtime option `POSIX(ON)` or `POSIX(OFF)`.

If your application runs with `POSIX(ON)`, z/OS Language Environment tries to load the DLL from the UNIX file system first. If your DLL is a data set member, you can avoid searching the UNIX file system directories. To direct a DLL search to a data set, prefix the DLL name with two slashes (`//`), as shown in the following example.

```
//MYDLL
```

If your application runs with `POSIX(OFF)`, z/OS Language Environment tries to load your DLL from a data set. If your DLL is a UNIX file system file, you can avoid searching a data set. To direct a DLL search to the UNIX file system, prefix the DLL name with a period and slash (`./`), as shown in the following example.

```
./mydll
```

Note: DLL names are case sensitive in the UNIX file system. If you specify the wrong case for your DLL that resides in the UNIX file system, it will not be found in the UNIX file system.

- For IPA, you should only export subprograms (functions and C++ methods) or variables that you need for the interface to the final DLL. If you export subprograms or variables unnecessarily (for example, by using the `EXPORTALL` option), you severely limit IPA optimization. In this case, global variable

coalescing and pruning of unreachable or 100% inlined code does not occur. To be processed by IPA, DLLs must contain at least one subprogram. Attempts to process a data-only DLL will result in a compilation error.

- The suboption NOCALLBACKANY of the compiler option DLL is more efficient than the CALLBACKANY suboption. The CALLBACKANY option calls z/OS Language Environment at run time. This runtime service enables direct function calls. Direct function calls are function calls through function pointers that point to actual function entry points rather than function descriptors. The use of CALLBACKANY will result in extra overhead at every occurrence of a call through a function pointer. This is unnecessary if the calls are not direct function calls.

Chapter 19. Building complex DLLs

Before you attempt to build complex DLLs, it is important to understand the differences between the terms DLL, DLL code, and DLL application.

A **DLL** (Dynamic Link Library) is a file containing executable code and data bound to a program at run time. The code and data in a DLL can be shared by several applications simultaneously. It is important to note that compiling code with the DLL option does not mean that the produced executable will be a DLL. To create a DLL, you must use the `#pragma export` or `EXPORTALL` compiler option.

DLL code is code that can use a DLL. The following are DLL code:

- C++ code
- C code compiled using the DLL or XPLINK option

Code written in languages other than C++ and compiled without the DLL or XPLINK option is non-DLL code.

A **DLL application** is an application that can use exported functions or variables that are bound with DLL code. All of the source files that make up a DLL application do not need to be compiled with the DLL or XPLINK option, only the source files that reference exported functions and exported global variables.

If you link DLL code with non-DLL code, the resulting DLL or DLL application is called **complex**. You might compile your code as non-DLL for the following reasons:

- Source modules do not use C or C++.
- To prevent problems which occur when a non-DLL function pointer call uses DLL code. This problem takes place when a function makes a call through a function pointer that points to a function entry rather than a function descriptor.

For complex DLLs and DLL applications that you compile without XPLINK, you can use the CBA suboption of the DLL|NODLL compiler option. With this suboption, a call is made, through a function pointer, to the z/OS Language Environment, for each function call, at run time. This call eliminates the error that would occur when a non-DLL function pointer passes a value to DLL code.

Note: In this book, unless otherwise specified, all references to the DLL|NODLL compiler option assume suboption NOCBA. For more information, see [DLL](#) in *z/OS XL C/C++ User's Guide*.

If you specify the XPLINK compiler option, the CBA and NOCBA suboptions of DLL and NODLL are ignored.

There are two ways to combine XPLINK and non-XPLINK code in the same application:

- Compile each entire DLL with XPLINK or without XPLINK. The only interaction between XPLINK and non-XPLINK code occurs at a DLL or `fetch()` boundary.
- Use the `OS_UPSTACK`, `OS_NOSTACK`, and `OS31_NOSTACK` linkage directive. For more information, see the description of the linkage pragma in [z/OS XL C/C++ Language Reference](#).

The steps for creating a complex DLL or DLL application are:

1. Determining how to compile your source modules.
2. Modifying the source modules that do not meet all the DLL rules.
3. Compiling the source modules to produce DLL code and non-DLL code as determined in the previous steps.
4. Binding your DLL or DLL application.

The focus of this chapter is step 1 and step 2. You perform step 3 the same way you would for any other C or C++ application. [“Binding your code” on page 192](#) explains step 4.

Rules for compiling source code with XPLINK

This section provides guidelines for compiling with the XPLINK and NOXPLINK compiler options. See [XPLINK](#) in *z/OS XL C/C++ User's Guide* for the details of this option.

XPLINK applications

XPLINK provides compatibility with non-XPLINK functions when calls are made across executable modules, using either the DLL or `fetch()` call mechanism. You should make a reference from XPLINK code into non-XPLINK code only if the reference is by an imported function or variable, or the function pointer is a parameter into the XPLINK code. This prevents incompatible references to a non-XPLINK function entry point.

Non-XPLINK code can expose a function entry point directly to the XPLINK code:

- as a global variable
- as part of a structure that is passed as a parameter
- by passing an explicit return value

A function pointer from a non-XPLINK application can be used as a callback by passing it as an argument into the XPLINK function, or as a member of a structure that is itself an argument to the XPLINK function.

Prior to z/OS V1R8, a function entry point from a non-XPLINK application only could be passed explicitly as an argument into a XPLINK function. This restriction did not apply if you used the compiler option `XPLINK(CALLBACK)` or the `__callback` qualifier where any such function pointers were used. Existing DLLs compiled using one of these options do not need to be recompiled. The use of these options can only be discontinued if the owner of the XPLink-compiled DLL is certain that any non-XPLink-compiled DLL callers have been recompiled with z/OS XL C/C++ V1R8 targeting z/OS Language Environment V1R8 or later, and those applications are targetted for and running on z/OS Language Environment V1R8 or later.

Also, note the following restrictions:

- DLLs must be created using the binder.
- C/C++ source modules must be compiled code using the DLL and GOFF options without the XPLink or LP64 option.
- Non-XPLink assembler DLLs are not supported.

Modifying noncompliant source

For each function pointer, make sure that one of the following is true:

- The function pointer is passed as a parameter to the XPLINK code.
- The indirectly-referenced function pointer was imported by this DLL.
- The indirectly-referenced function pointer was imported by another XPLINK or non-XPLINK DLL.

Non-XPLINK applications

To create a complex DLL or DLL application, you must comply with the following rules that dictate how you compile source modules. The first decision you must make is how you should compile your code. You determine whether to compile with either the DLL or NODLL compiler option based on whether or not your code references any other DLLs. Even if your code is a DLL, it is safe to compile your code with the NODLL compiler option if your code does not reference other DLLs.

The second decision you must make is whether to compile with the default compiler suboption for DLL|NODLL, which is NOCBA, or use the alternative suboption CBA. This decision is based upon your knowledge of the code you reference. If you are sure that you do not reference any function calls through function pointers that point to a function entry rather than a function descriptor, use the NOCBA suboption. Otherwise, you should use the CBA suboption.

As of V2R4 of OS/390® C/C++, use the following options to ensure that you do not have undefined results as a result of the function pointer pointing to a function entry rather than a function descriptor:

1. Compile your source module with the CBA suboption of DLL | NODLL. This option inserts extra code whenever you have a function call through a function pointer. The inserted code invokes a runtime service of z/OS Language Environment which enables direct function calls through C/C++ function pointers. Direct function calls are function calls through function pointers that point to actual function entry points rather than function descriptors. The drawback of this method is that your code will run slower. This occurs because whenever you have function calls through function pointers z/OS Language Environment is called at run time to enable direct function calls. See [Figure 50 on page 209](#) for an example of the CBA suboption and an explanation of what the called z/OS Language Environment routine does at run time when using the CBA suboption.
2. Compile your C source module with the NOCBA suboption of DLL | NODLL. This option has the benefit of faster running but with more restrictions placed on your coding style. If you do not follow the restrictions, your code may behave unpredictably. See [“DLL restrictions” on page 196](#) for more information.

Compile your C source modules as DLL when:

1. Your source module calls imported functions or imported variables by name.
2. Your source module contains a comparison of function pointers that may be DLL function pointers.
The comparisons shown in [“Function pointer comparison in non-DLL code” on page 211](#) are undefined. To obtain valid comparisons, compile the source modules as DLL code.
3. Your source module may pass a function pointer to DLL code through a parameter or a return value.
If the `sort()` routine in [Figure 49 on page 208](#) is compiled as DLL code instead of non-DLL code, non-DLL applications can no longer call it. To be able to call the DLL code version of `sort()`, the original non-DLL application must be recompiled as DLL code.
4. Your source module may define a global function pointer and another source module changes it.

Consider the example programs shown in [Table 26 on page 201](#). You have the following two options when compiling them.

- a. If source module 1 is compiled as DLL code, source module 2 must also be compiled as DLL code.
- b. Alternately, you can compile source module 1 as DLL and source module 2 as NODLL(CBA).

Table 26. Example programs to demonstrate compiling options

Source module 1	Source module 2
<pre>void (*fp)(void); extern void goo (void); void main() { goo(); (*fp)(); /* call hello function */ }</pre>	<pre>#include <stdio.h> extern void (*fp)(void); void hello(void) { printf("hello\n"); } void goo(void) { fp = hello; }</pre>

[Table 27 on page 201](#) summarizes some of the ways that you could compile the two source modules and list the results. Both modules are linked into a single executable.

Table 27. Examples of how to compile two source modules and list result

How Modules Were Compiled	Result
<ul style="list-style-type: none"> • Source module 1 NODLL (NOCBA) • Source module 2 DLL(NOCBA) 	fp contains a function descriptor. Execution of fp will succeed because it is valid to the address of a function descriptor.

Table 27. Examples of how to compile two source modules and list result (continued)

How Modules Were Compiled	Result
<ul style="list-style-type: none"> Source module 1 DLL(NOCBA) Source module 2 NODLL(NOCBA) 	fp contains the address of hello . The execution of fp would abend because source module 1 expects fp to contain a function descriptor for hello.
<ul style="list-style-type: none"> Source module 1 DLL(CBA) Source module 2 DLL(NOCBA) 	fp contains a function descriptor. The generated code will function correctly. It will run slower than if the source modules were compiled as DLL(NOCBA) because it will use Language Environment to make the function call.
<ul style="list-style-type: none"> Source module 1 NODLL(CBA) Source module 2 DLL(NOCBA) 	A call to Language Environment made by the function call through the function pointer prevents a problem that would have occurred had a direct function call been made.

If you do not use the DLL compiler option, and your source module calls imported functions or imported variables by name, there will be unresolved references to these variables and functions at bind time. A DLL or DLL application that does not comply with these rules may produce undefined runtime behavior. For a detailed explanation of incompatibilities between DLL and non-DLL code, see [“Compatibility issues between DLL and non-DLL code” on page 202](#).

Modifying noncompliant source

Sometimes source modules of a complex DLL or DLL application do not simultaneously meet all the DLL rules. These rules are documented in the section [“Rules for compiling source code with XPLINK” on page 200](#). When these situations occur, you can use the following methods to solve the problem:

- Use the CBA suboption.
- Rewrite the source in C. Only C source can be compiled as either DLL or non-DLL code. C++ source code is always DLL code.
- Split a C source module in two so that one of the new files is compiled as DLL code and the other is compiled as non-DLL code.

Note: In rare cases, you may have to split a function into two functions before you can successfully split the file.

An example of noncompliant source is a C++ source module that contains a function call through a pointer that may be either a DLL pointer to a function descriptor or a direct function pointer. Convert it to C code and compile as non-DLL code or, preferably, as DLL(CBA) and recompile.

Compatibility issues between DLL and non-DLL code

This section describes the differences between DLL code and non-DLL code, and discusses the related compatibility issues for linking them to create complex DLLs.

Note: This section does not apply to XPLINK applications. XPLINK code is always DLL code.

Table 28 on page 202 and Figure 42 on page 203 illustrate DLL code referencing functions and variables.

Table 28. Referencing functions and external variables

Function or Variable	DLL
Imported Functions	A function descriptor is created by the binder. The descriptor is in the WSA class and contains the address of the function and the address of the writable static area associated with that function. The function address and the address of the WSA associated with the function is resolved when the DLL is loaded. 1
Nonimported Functions	Also called through the function descriptor but the function address is resolved at link time. 3

Table 28. Referencing functions and external variables (continued)

Function or Variable	DLL
Imported Variables	A variable descriptor is created in the WSA by the binder. It contains addressing information for accessing an imported variable. The address is resolved when the DLL is loaded. 2
Nonimported Variables	Direct access 4

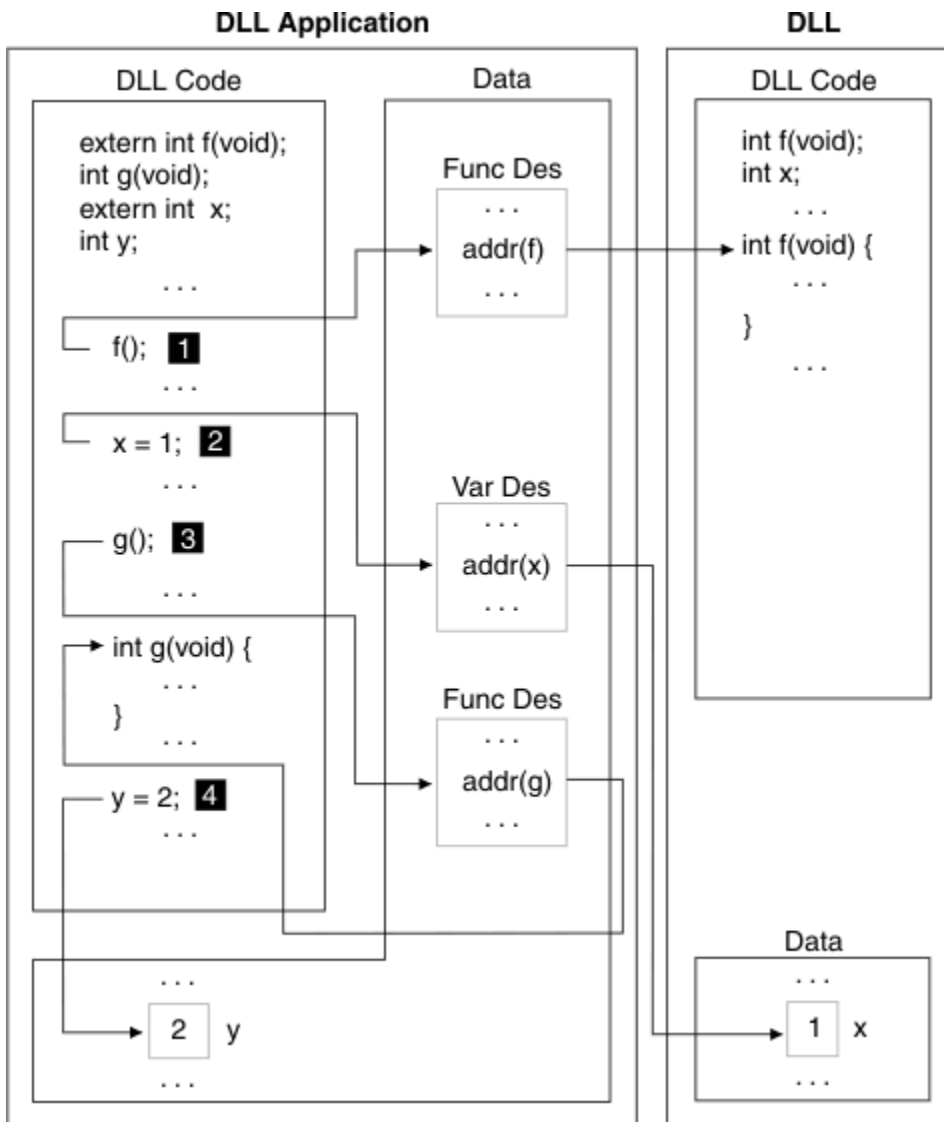


Figure 42. Referencing functions and external variables in DLL code

Pointer assignment

In DLL code and non-DLL code, the actual address of a variable is assigned to a variable pointer. A valid variable pointer always points to the variable itself and causes no compatibility problems.

Function pointers

In non-DLL code, the actual address of a nonimported function is assigned to a function pointer. In DLL code, the address of a function descriptor is assigned to a function pointer.

If you assign the address of an imported function to a pointer in non-DLL code, the link step will fail with an unresolved reference. In a complex DLL or DLL application, a pointer to a function descriptor may be

passed to non-DLL code. A direct function pointer (pointer to a function entry point) may be passed to DLL code. A parameter, a return value, or an external variable can pass a function pointer or an external variable.

In a complex DLL or DLL application, a function pointer may point either to a function descriptor or to a function entry, depending on the origin of the code. The different ways of dereferencing a function pointer causes the compatibility problem in linking DLL code with non-DLL code.

In Figure 43 on page 204, **1** assigns the address of the descriptor for the imported function `f` to `fp`. **2** assigns the address of the imported variable `x` to `xp`. **3** assigns the address of the descriptor for the nonimported function `g` to `gp`. **4** assigns the address of the non-imported variable `y` to `yp`.

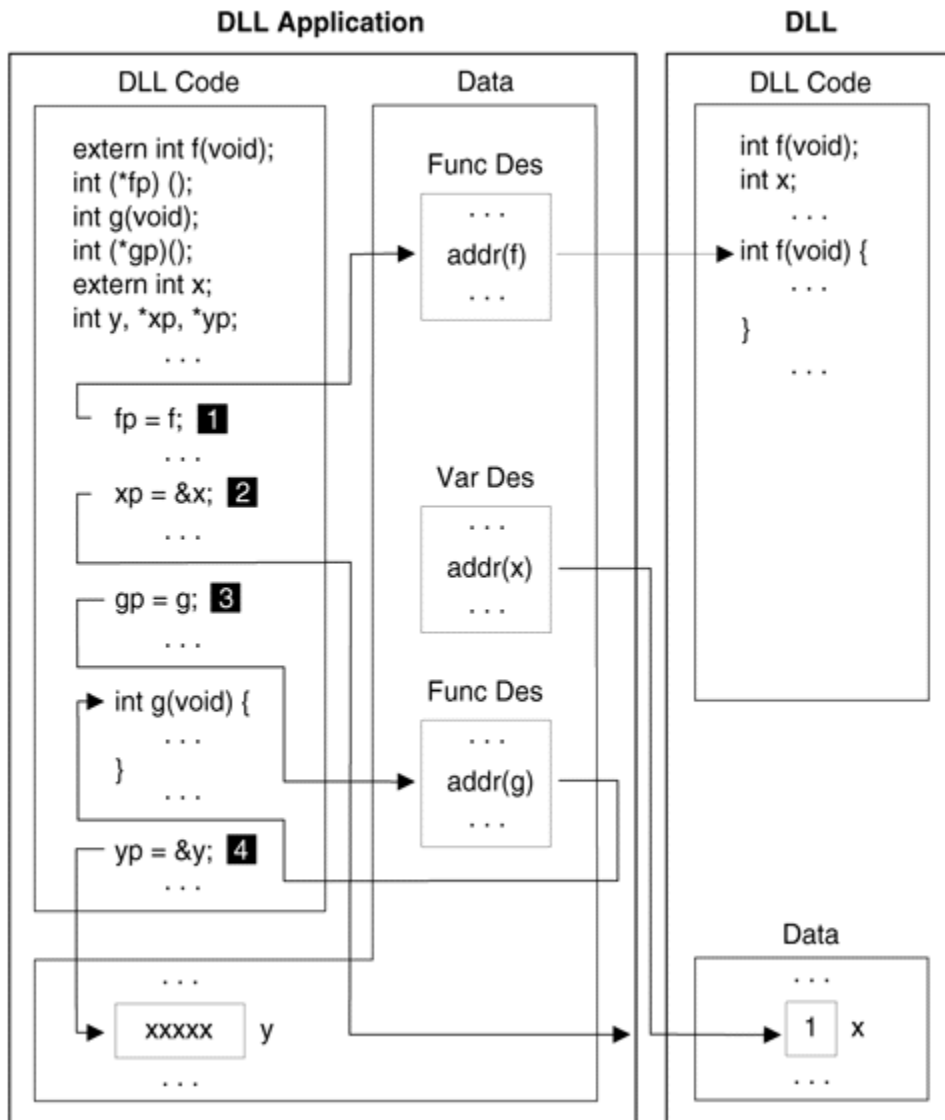


Figure 43. Pointer Assignment in DLL code

In Figure 44 on page 205, **1** causes a bind error because the assignment to `fp` is undefined. **2** causes a binder error because the assignment to `xp` is undefined. **3** assigns `gp` to the address of the nonimported function, `g`. **4** assigns the address of the nonimported variable `y` to `yp`.

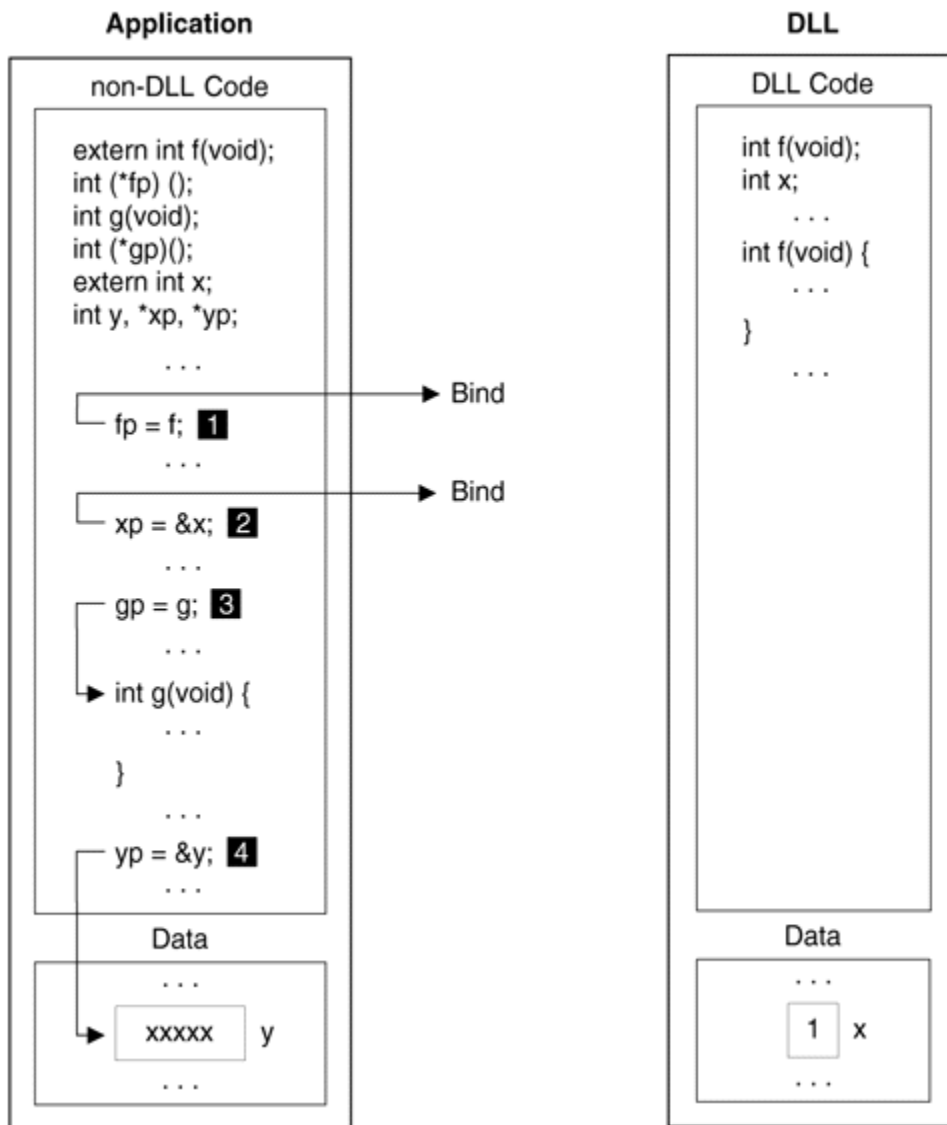


Figure 44. Pointer assignment in non-DLL code

DLL function pointer call in non-DLL code

Because z/OS XL C/C++ supports a DLL function pointer call in non-DLL code, you are able to create a DLL to support both DLL and non-DLL applications. The z/OS XL C/C++ compiler inserts *glue code* at the beginning of a function descriptor to allow branching to a function descriptor. Glue code is special code that enables function pointer calls from non-DLL code to DLL code, including XPLINK code.

A function pointer in non-DLL code points to the function entry and a function pointer call branches to the function address. However, a DLL function pointer points to a function descriptor. A call made through this pointer in non-DLL code results in branching to the descriptor.

z/OS XL C/C++ executes a DLL function pointer call in non-DLL code by branching to the descriptor and executing the glue code that invokes the actual function.

The following examples and Figure 49 on page 208 show a DLL function pointer call in non-DLL code, where a simplified `sort()` routine is used. Note that the `sort()` routine compiled as non-DLL code can be called from both a DLL application and a non-DLL application.

C example

File 1 and File 2 are bound together to create application A. File 1 is compiled with the NODLL option. File 2 is compiled with the DLL option (so that it can call the DLL function `sort()`). File 3 is compiled as DLL to create application B. Application A and B can both call the imported function `sort()` from the DLL in file 4.

Figure 45 on page 206 shows how a file (File 1) of a complex DLL application is compiled with the NODLL option

```
typedef int CmpFP(int, int);
void sort(int* arr, int size, CmpFP*); /* sort routine in DLL */
void callsort(int* arr, int size, CmpFP* fp); /* routine compiled as DLL */
/* which can call DLL */
/* routine sort() */

int comp(int e1, int e2) {
    if (e1 == e2) {
        return(0);
    }
    else if (e1 < e2) {
        return(-1);
    }
    else {
        return(1);
    }
}

main() {
    CmpFP* fp = comp;
    int a[2] = {2,1};
    callsort(a, 2, fp);
    return(0);
}
```

Figure 45. File 1. Application A

Figure 46 on page 206 shows how a file (File 2) of a complex DLL application is compiled with the DLL option.

```
typedef int CmpFP(int, int);
void sort(int* arr, int size, CmpFP*); /* sort routine in DLL */
void callsort(int* arr, int size, CmpFP* fp) {
    sort(arr, size, fp);
}
```

Figure 46. File 2. Application A

Figure 47 on page 206 shows how a simple DLL application is compiled with the DLL option.

```
int comp(int e1, int e2) {
    if (e1 == e2)
        return(0);
    else if (e1 < e2)
        return(-1);
    else
        return(1); }
int (*fp)(int e1, int e2);
main()
{
    int a[2] = { 2, 1 };
    fp = comp; /* assign function address */
    sort(a, 2, fp); /* call sort */
}
```

Figure 47. File 3. Application B

Figure 48 on page 207 shows how a DLL is compiled with the NODLL option. File 4 is compiled as NODLL and bound into a DLL. The function `sort()` will be exported to users of the DLL.

```

typedef int CmpFP(int, int);
int sort(int* arr, int size, CmpFP* fp) {
    int i,j,temp,rc;

    for (i=0; i<size; ++i) {
        for (j=1; j<size-1; ++j) {
            rc = fp(arr[j-1], arr[j]); /* call 'fp' which may be DLL or no-DLL code */
            if (rc > 0) {
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
            }
        }
    }
    return(0);
}
#pragma export(sort)

```

Figure 48. File 4. DLL

Non-DLL function pointers can only safely be passed to a DLL if the function referenced is naturally reentrant, that is, it is C code compiled with the RENT compiler option, or is C code with no global or static variables. See the discussion on the CBA option to see how to make a DLL that can be called by applications that pass constructed reentrant function pointers.

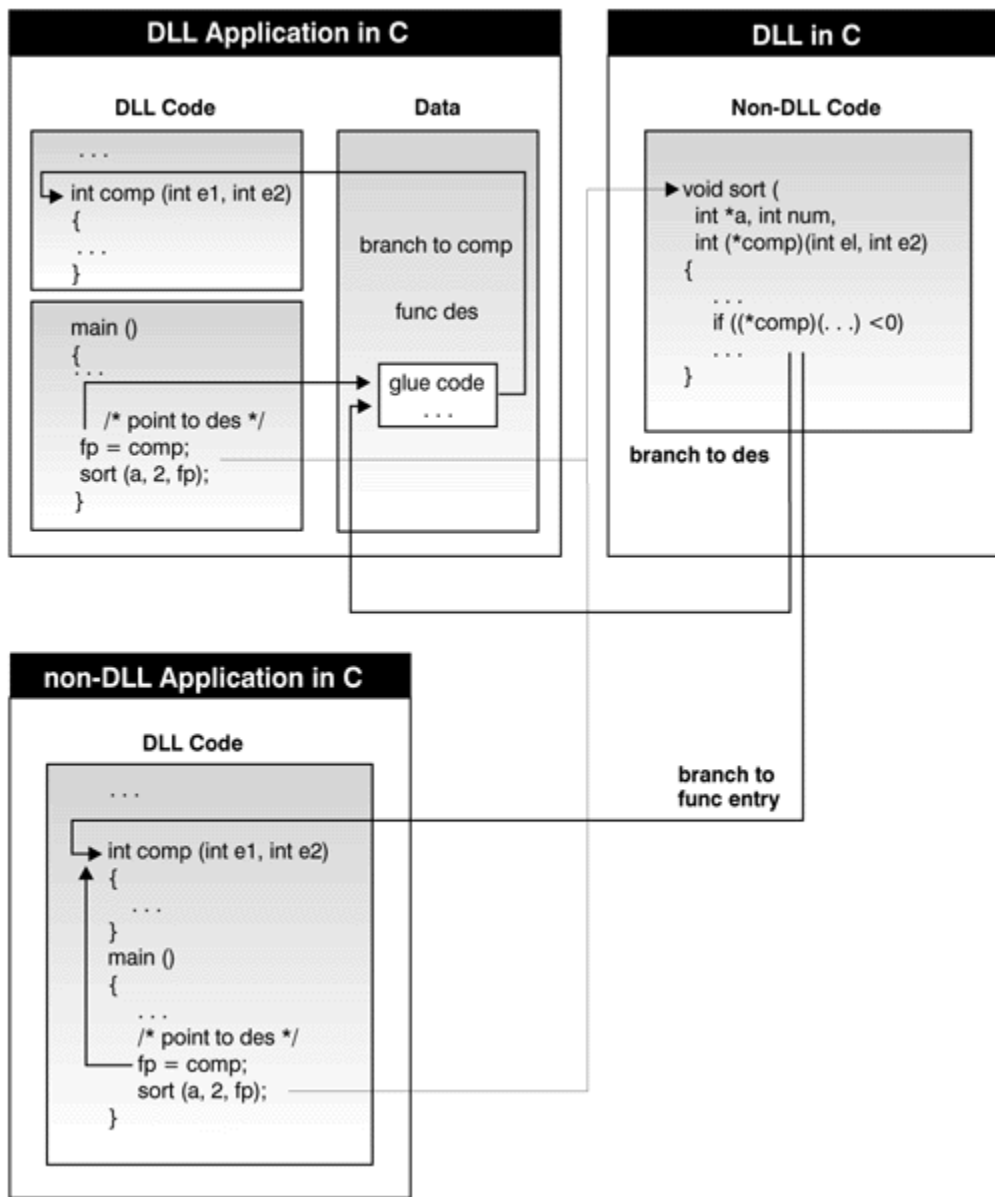


Figure 49. DLL function pointer call in non-DLL code

Non-DLL function pointer call in DLL(CBA) code

Figure 50 on page 209 illustrates one situation where you could use the CBA suboption. In the example, the DLL provider provides stub routines which the application programmer can bind with their applications. These stub routines allow an application programmer to use a DLL without recompiling the application with the DLL option. This is an important consideration for library providers that want to move from a static version of a library to a dynamic one. Stub routines are not mandatory, however if they are provided, the application programmer only needs to rebind, but not recompile the application. If stub routines are not provided by the DLL provider, the application programmer must recompile the application.

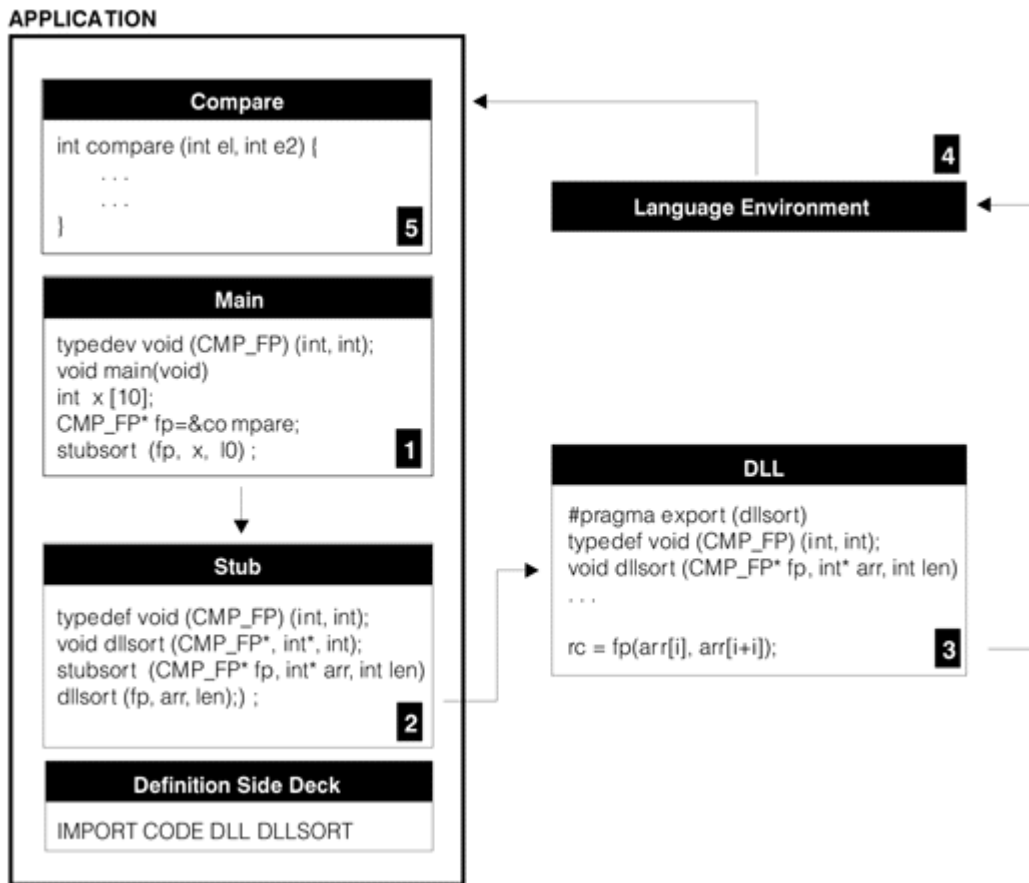


Figure 50. DLL function pointer call in non-DLL code

In the previous example, the DLL provider:

- Compiles the DLL parts as either DLL(CBA) or NODLL(CBA).
- Exports function `dllsort()` for use by other applications.
- Binds the DLL to produce a DLL executable module and a DLL definition side-deck.
- Creates a stub function for every function exported from the DLL. The stub function calls a corresponding function in the DLL. This routine is compiled with the DLL option. The stub functions are provided to the application programmer in a static library to be bound with the application.

The Application Programmer:

- Codes the program using any of the following compiler options;
 - DLL
 - NODLL
 - RENT
 - NORENT
- Calls the stub routines, not the exported functions.

The stub routines must be called because the application programmer may have compiled his code with the NODLL compiler option. Otherwise, references to the DLL functions will be unresolved at bind time. Providing the stub routines allows an application programmer to use a DLL without recompiling the application with the DLL option. This is an important consideration for library providers that want to move from a static version of a library to a dynamic one. Providing stub routines requires the application programmer to rebind but not recompile the application.

- Statically binds the definition side-deck, provided by the DLL provider, and the stub routines with their program.

- Binds the DLL to produce a DLL executable module and a DLL definition side-deck
- Creates a stub function for every function exported from the DLL. The stub function calls the DLL directly

The reference keys in [Figure 50 on page 209](#) illustrate the sequence of events. Note that in [3](#), the user does not explicitly make a call to Language Environment. The generated code for the fp function call makes the call to z/OS Language Environment. z/OS Language Environment does the following at point [4](#) in the figure:

- Saves the DLL environment
- Establishes the application environment
- Branches to the user's function
- Reestablishes the DLL environment after execution of the function
- Returns control to the DLL.

Non-DLL function pointer call in DLL code

In DLL code, it is assumed that a function pointer points to a function descriptor. A function pointer call is made by first obtaining the function address through dereferencing the pointer; and then, branching to the function entry. When a non-DLL function pointer is passed to DLL code, it points directly to the function entry. An attempt to dereference through such a pointer produces an undefined function address. The subsequent branching to the undefined address may result in an exception.

The following is an example of passing a non-DLL function pointer to DLL code using an external variable. Its behavior is undefined as shown in [Figure 51 on page 210](#).

```
#include <stdio.h>
extern void (*fp)(void);
void hello(void) {
    printf("hello\n");
}
void goo(void) {
    fp = hello; /* assign address of hello, to fp */
               /* (refer to
Figure 44 on page 205). */
}
```

Figure 51. C non-DLL code

[Figure 52 on page 210](#) shows how dereferencing through a pointer produces an undefined function address in C.

```
extern void goo(void);
void (*fp)(void);
void main (void) {
    goo();
    (*fp)(); /* Expect a descriptor, but get a function address, */
            /* so it dereferences to an undefined address and */
            /* call fails */
}
```

Figure 52. C DLL code

[Figure 53 on page 211](#) shows how dereferencing through a pointer produces an undefined function address in C++.

```

extern "C" void goo(void);
void (*fp)(void);
void main (void) {
    goo();
    (*fp)(); /* Expect a descriptor, but get a function address, */
            /* so it dereferences to an undefined address and */
            /* call fails */
}

```

Figure 53. C++ DLL code

In Figure 54 on page 211, a non-DLL function pointer call to an assembler function is resolved.

```

/*
 * This function must be compiled as DLL(CBA)
 */

extern "OS" {
    typedef void OS_FP(char *, int *);
}
extern "OS" OS_FP* ASMFN(char*);

int CXXFN(char* p1, int* p2) {
    OS_FP* fptr;

    fptr = ASMFN("ASM FN"); /* returns pointer to address of function */
    if (fptr) {
        fptr(p1, p2); /* call asm function through fn pointer */
    }
    return(0);
}

```

Figure 54. C++ DLL code calling an Assembler function

Function pointer comparison in non-DLL code

In non-DLL code, the results of the following function pointer comparisons are undefined:

- Comparing a DLL function pointer to a non-DLL function pointer
- Comparing a DLL function pointer to another DLL function pointer
- Comparing a DLL function pointer to a constant function address

Comparing a DLL function pointer to a non-DLL function pointer

In Figure 55 on page 211, both the DLL function pointer and the non-DLL function pointer point to the same function; but the pointers, when compared, are unequal.

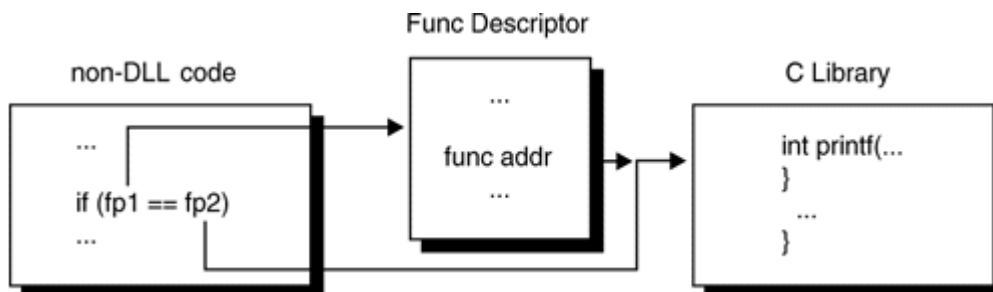


Figure 55. Comparison of function pointers in non-DLL code

Example of comparing a DLL function pointer to a non-DLL function pointer using C

In the examples shown in [Figure 56 on page 212](#) and [Figure 57 on page 212](#), DLL code and non-DLL code can reside either in the same executable file or in different executable files.

```
#include <stdio.h>
extern int foo(int (*fp1)(const char *, ...));
main ()
{
    int (*fp)(const char *, ...);
    fp = printf; /* assign address of a descriptor that */
                /* points to printf. */
    if (foo(fp))
        printf("Test result is undefined\n");
}
```

Figure 56. C DLL code

```
int foo(int (*fp1)(const char *, ...))
{
    int (*fp2)(const char *, ...);
    fp2 = printf; /* assign the address of printf. */
    if (fp1 == fp2) /* comparing address of descriptor to */
                    /* address of printf results in unequal.*/
        return(0);
    else
        return(1);
}
```

Figure 57. C non-DLL code

Comparing a DLL function pointer to another DLL function pointer

The example in [Figure 61 on page 213](#) compares addresses of function descriptors. In the following examples, both of the DLL function pointers point to the same function, but they compare unequal.

Comparison of two DLL function pointers in non-DLL code

The following example shows a comparison of two DLL function pointers in non-DLL code. In this example, File 1 ([Figure 58 on page 212](#)) and File 2 ([Figure 59 on page 213](#)) reside in different executable modules. File 3 ([Figure 60 on page 213](#)) can reside in the same executable module as File 1 or File 2 or it can reside in a different executable module. In all cases, the addresses of the function descriptors will not compare equally.

```
#include <stdio.h>
extern int goo(int (*fp1)(const char *, ...));
main ()
{
    int (*fp)(const char *, ...);
    fp = printf; /* assign address of a descriptor that */
                /* points to printf. */
    if (goo(fp))
        printf("Test result is undefined\n");
}
```

Figure 58. File 1 C DLL code

```

#include <stdio.h>
extern int foo(int (*fp1)(const char *, ...),
               int (*fp2)(const char *, ...));
int goo(int (*fp1)(const char *, ...))
{
    int (*fp2)(const char *, ...);
    fp2 = printf; /* assign address of a different      */
                  /* descriptor that points to printf. */
    return (foo(fp1, fp2));
}

```

Figure 59. File 2 C DLL code

```

int foo(int (*fp1)(const char *, ...),
        int (*fp2)(const char *, ...))
{
    if (fp1 == fp2) /* comparing the addresses of two      */
                    /* descriptors results in unequal.    */
        return(0);
    else
        return(1);
}

```

Figure 60. File 3 C non-DLL code

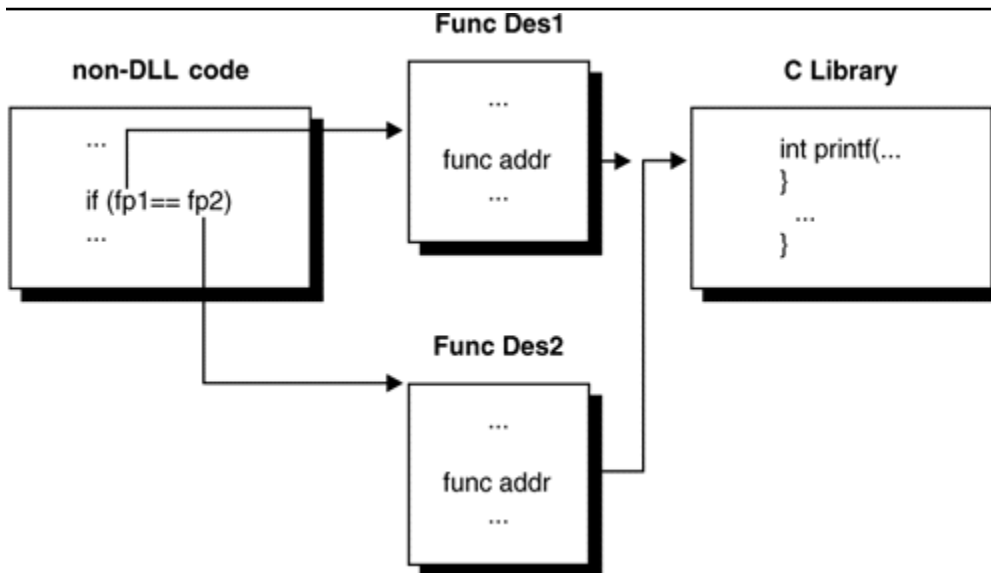


Figure 61. Comparison of two DLL function pointers in non-DLL code

Comparing a DLL function pointer to a constant function address other than NULL

Here, you are comparing the constant function address to an address of a function descriptor.

Note: Comparing a DLL function pointer to NULL is well defined, because when a pointer variable is initialized to NULL in DLL code, it has a value zero.

Function pointer comparison in DLL code

In XPLINK code, function pointers are compared using the address of the descriptor. No special considerations, such as dereferencing, are required to initialize the function pointer prior to comparison. Function descriptors are guaranteed to be unique throughout the XPLINK application for a particular function, so this comparison of function descriptor addresses will yield the correct results even if the function pointer is passed between executable modules within the XPLINK application. The remainder of this section does not apply to XPLINK applications.

In non-XPLINK DLL code, a function pointer must be NULL before it is compared. For a non-NULL pointer, the pointer is further dereferenced to obtain the function address that is used for the comparison. For an uninitialized function pointer that has a non-zero value, the dereference can cause an exception to occur. This happens if the storage that the uninitialized pointer points to is read-protected.

Usually, comparing uninitialized function pointers results in undefined behavior. You must initialize a function pointer to NULL or the function address (from source view). Two examples follow.

Figure 62 on page 214 shows undefined comparison in DLL code (C or C++).

```
#include <stdio.h>
int (*fp2)(const char *, ...) /* Initialize to point to the */
                             = printf; /* descriptor for printf */

int goo(void);
int (*fp2)(void) = goo;
int goo(void) {
    int (*fp1)(void);
    if (fp1 == fp2)
        return (0);
    else
        return (1);
}

void check_fp(void (*fp)()) {
    /* exception likely when -1 is dereferenced below */
    if (fp == (void (*)())-1)
        printf("Found terminator\n");
    else
        fp();
}

void dummy() {
    printf("In function\n");
}

main() {
    void (*fa[2])();
    int i;

    fa[0] = dummy;
    fa[1] = (void (*)())-1;

    for(i=0; i<2; i++)
        check_fp(fa[i]);
}
```

Figure 62. Undefined comparison in DLL code (C or C++)

Figure 63 on page 214 shows that, when fp1 points to a read-protected memory block, an exception occurs.

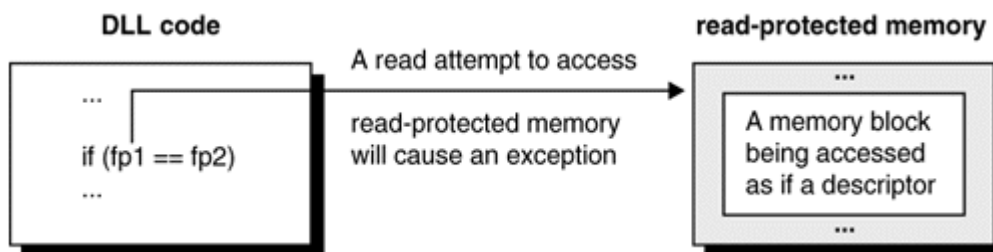


Figure 63. Comparison of function pointers in DLL code (C or C++)

Figure 64 on page 215 is an example of valid comparisons in DLL code.

```

#include <stdio.h>
int (*fp1)(const char *, ...); /* An extern variable is implicitly*/
                                /* initialized to zero           */
                                /* if it has not been explicitly  */
                                /* initialized in source.         */
int (*fp2)(const char *, ...) /* Initialize to point to the    */
                                = printf; /* descriptor for printf */
int foo(void) {
    if (fp1 != fp2 )
        return (0);
    else
        return (1);
}

```

Figure 64. Valid comparisons in DLL code (C or C++)

Using DLLs that call each other

An application can use DLLs that call each other. There are two methods for building these applications, as illustrated in the examples that follow:

- In the first method, the loop is broken by manually creating IMPORT statements for the referenced DLLs, when binding one of the DLLs (CCNGA2D3).
- In the second method, an initial bind is done on CCNGA2D3 using the binder NCAL parameter, which will be done again after the referenced DLLs are built.

In both cases, the result is that the side-deck is produced for CCNGA2D3, so that the DLLs that reference CCNGA2D3 can be built.

The CCNGA2 application (Figure 65 on page 215) imports functions and variables from three DLLs: (Figure 66 on page 216, Figure 67 on page 216, and Figure 68 on page 216). It is an example of an application that uses DLLs that call each other.

```

#include <stdlib.h>

extern int  var1_d1;           /*imported from CCNGA2D1 */
extern int  func1_d1(int);     /*imported from CCNGA2D1 */

extern int  var1_d2;           /*imported from CCNGA2D2 */
extern int  func1_d2(int);     /*imported from CCNGA2D2 */

extern int  var1_d3;           /*imported from CCNGA2D3 */
extern int  func1_d3(int);     /*imported from CCNGA2D3 */

int main() {
    int rc = 0;

    printf("+CCNGA2::main() starting \n");
    /* ref DLL1 */
    if (var1_d1 == 100) {
        printf("| var1_d1=<%d>\n", var1_d1++);
        func1_d1(var1_d1);
    }
    /* ref DLL2 */
    if (var1_d2 == 200) {
        printf("| var1_d2=<%d>\n", var1_d2++);
        func1_d2(var1_d2);
    }
    /* ref DLL3 */
    if (var1_d3 == 300) {
        printf("| var1_d3=<%d>\n", var1_d3++);
        func1_d3(var1_d3);
    }

    printf("+CCNGA2::main() Ending \n");
}

```

Figure 65. Application CCNGA2

Figure 66 on page 216 shows application CCNGA2D1, which imports functions from [Figure 67 on page 216](#) and [Figure 68 on page 216](#).

```
#include <stdio.h>

int func1_d1();          /* A function to be externalized */
int var1_d1 = 100;       /* export this variable          */

extern int func1_d2(int); /*imported from CCNGA2D2    */
extern int func1_d3(int); /*imported from CCNGA2D3    */

int func1_d1 (int input)
{
    int rc2 = 0;
    int rc3 = 0;
    printf("| +-CCNGA2D1() func1_d1() starting. Input is %d\n", input);
    rc2 = func1_d2(200);
    rc3 = func1_d3(300);
    printf("| | func1_d1() dl11 - rc2=<%d> rc3=<%d>\n", rc2,
rc3);
    printf("| +-CCNGA2D1() func1_d1() ending. \n");
}
```

Figure 66. Application CCNGA2D1

Figure 67 on page 216 shows application CCNGA2D2, which imports a function from [Figure 68 on page 216](#).

```
#include <stdio.h>

int func1_d2();          /* A function to be externalized */
int var1_d2 = 200;

extern int func1_d3(int); /* import this function          */

int func1_d2 (int input)
{
    int rc3 =0;
    printf("| | +-CCNGA2D2() func1_d2() starting. Input is %d\n",
input);
    rc3 = func1_d3(300);
    printf("| | | func1_d2() dl12 - rc3=<%d>\n", rc3);
    printf("| | +-CCNGA2D2() func1_d2() ending\n");
}
```

Figure 67. Application CCNGA2D2

Application CCNGA2D3 ([Figure 68 on page 216](#)) imports variables from [Figure 66 on page 216](#) and [Figure 67 on page 216](#).

```
#include <stdio.h>

int func1_d3();          /* A function to be externalized */
int var1_d3 = 300;

extern int var1_d1;       /* imported variable from CCNGA2D1 */
extern int var1_d2;       /* imported variable from CCNGA2D2 */

int func1_d3 (int input)
{
    printf("| | | +-CCNGA2D3()-func1_d3() starting. Input is %d\n",
input);
    printf("| | | value of var1_d1=%d var1_d2=%d\n",
var1_d1, var1_d2);
    printf("| | | +-CCNGA2D3()-func1_d3() ending\n");
}
```

Figure 68. Application CCNGA2D3

The first method uses the JCL in [Figure 69 on page 217](#). The following processing occurs:

1. CCNGA2D3 is compiled and bound to create a DLL. The binder uses the control cards that are supplied through SYSIN to import variables from CCNGA2D1 and CCNGA2D2. The binder also generates a side-deck CCNGA2D3 that is used in the following steps.
2. CCNGA2D2 is compiled and bound to create a DLL. The binder uses the control cards that are supplied through SYSIN to include the side-deck from CCNGA2D3. The following steps use the binder that generates the side-deck CCNGA2D2.
3. CCNGA2D1 is compiled and bound to create a DLL. The binder uses the control cards that are supplied through SYSIN to include the side-decks from CCNGA2D2 and CCNGA2D3. The following steps show the binder generating the side-deck CCNGA2D1.
4. CCNGA2 is compiled, bound, and run. The binder uses the control statements that are supplied through SYSIN to include the side-decks from CCNGA2D1, CCNGA2D2, and CCNGA2D3.

```
//jobcard information...
//PROC JCLLIB ORDER=(CBC.SCCNPRC, CEE.SCEEPROC)
//*
/* CBDLL3: -Compile and bind CCNGA2D3
/* -Explicit import of variables from CCNGA2D1 and CCNGA2D2
/* -Generate the side-deck CCNGA2D3
/*
/*CBDLL3 EXEC EDCCB,INFILE='CBC.SCCNSAM(CCNGA2D3)',
/* CPARM='SO,LIST,DLL,EXPO,RENT,LONG',
/* OUTFILE='myid.LOAD,DISP=SHR'
/*BIND.SYSIN DD *
IMPORT DATA CCNGA2D1 var1_d1
IMPORT DATA CCNGA2D2 var1_d2
NAME CCNGA2D3(R)
/*
/*BIND.SYSDEFSD DD DSN=myid.IMPORT(CCNGA2D3),DISP=SHR
/*
/*CDDL2: -Compile and bind CCNGA2D2
/* -Include the side-deck CCNGA2D3
/* -Generate the side-deck CCNGA2D2
/*
/*CBDLL2 EXEC EDCCB,INFILE='CBC.SCCNSAM(CCNGA2D2)',
/* CPARM='SO,LIST,DLL,EXPO,RENT,LONG',
/* OUTFILE='myid.LOAD,DISP=SHR'
/*BIND.SYSIN DD *
INCLUDE DSD(CCNGA2D3)
NAME CCNGA2D2(R)
/*
/*BIND.SYSDEFSD DD DSN=myid.IMPORT(CCNGA2D2),DISP=SHR
/*BIND.DSD DD DSN=myid.IMPORT,DISP=SHR
/*
/* CBDLL1: -Compile and bind CCNGA2D1
/* -Include the side-deck CCNGA2D2 and CCNGA2D3
/* -Generate the side-deck CCNGA2D1
/*
/*CBDLL1 EXEC EDCCB,INFILE='CBC.SCCNSAM(CCNGA2D1)',
/* CPARM='SO,LIST,DLL,EXPO,RENT,LONG',
/* OUTFILE='myid.LOAD,DISP=SHR'
/*BIND.SYSIN DD *
INCLUDE DSD(CCNGA2D2)
INCLUDE DSD(CCNGA2D3)
NAME CCNGA2D1(R)
/*
/*BIND.SYSDEFSD DD DSN=myid.IMPORT(CCNGA2D1),DISP=SHR
/*BIND.DSD DD DSN=myid.IMPORT,DISP=SHR
/*
/* CBAPP2: -Compile, bind and run CCNGA2
/* -Include the side-deck CCNGA2D1, CCNGA2D2 and CCNGA2D3
/*
/*CBAPP2 EXEC EDCCBG,INFILE='CBC.SCCNSAM(CCNGA2)',
/* CPARM='SO,LIST,DLL,RENT,LONG',
/* OUTFILE='myid.LOAD(CCNGA2),DISP=SHR'
/*BIND.SYSIN DD *
INCLUDE DSD(CCNGA2D1)
INCLUDE DSD(CCNGA2D2)
INCLUDE DSD(CCNGA2D3)
NAME CCNGA2(R)
/*
/*BIND.DSD DD DSN=myid.IMPORT,DISP=SHR
/*GO.STEPLIB DD
/* DD DSN=myid.LOAD,DISP=SHR
```

Figure 69. CCNGA2M1

The second method uses the JCL in [Figure 70 on page 218](#). The following processing occurs:

1. After compiled, the object module CCNGA2D2 is saved for the following steps.
2. CCNGA2D1 is compiled. The object module is saved for the following steps.
3. CCNGA2D3 is compiled and bound to generate the side-deck and the object module is not used in the following steps. The load module for this step is not saved, as it is not being used. The load module for CCNGA2D3 is generated at a later step.
4. CCNGA2D2 is bound to create a DLL. The binder takes as input the object module CCNGA2D2 and the side-deck CCNGA2D3. It also generates the side-deck CCNGA2D2 that is used in the following steps.
5. CCNGA2D1 is bound to create a DLL. The binder takes as input the object module CCNGA2D1 and the side-decks CCNGA2D3 and CCNGA2D2. It also generates the side-deck CCNGA2D1 that is used in the following steps.
6. CCNGA2D3 is bound to create a DLL. The binder takes as input the object module CCNGA2D3 and the side-decks CCNGA2D1 and CCNGA2D2. It also generates the side-deck CCNGA2D3 that is used in the following step.
7. CCNGA2 is compiled, bound, and run. The binder takes as input the object module CCNGA2 and the side-decks CCNGA2D1, CCNGA2D2, and CCNGA2D3.
8. If you do not specify the OUTFILE parameter, the load module member GO is generated and executed by default. So it is recommended that you use **NAME GO** in the bind statements if you do not specify the OUTFILE parameter. If you specify **OUTFILE='dataset(member)', DISP=SHR** in the EDCCBG invocation, the member name overrides what was specified in the bind.

```
//jobcard information...
//PROC JCLLIB ORDER=(CBC.SCCNPRC, CEE.SCEEPROC)
//* CDLL2: -Compile CCNGA2D2
//*
//CDLL2 EXEC EDCC,INFILE='CBC.SCCNSAM(CCNGA2D2)',
// OUTFILE='myid.OBJ(CCNGA2D2),DISP=SHR',
// CPARM='SO,LIST,DLL,EXPO,RENT,LONG'
//*
//* CDLL1: -Compile CCNGA2D1
//*
//CDLL1 EXEC EDCC,INFILE='CBC.SCCNSAM(CCNGA2D1)',
// OUTFILE='myid.OBJ(CCNGA2D1),DISP=SHR',
// CPARM='SO,LIST,DLL,EXPO,RENT,LONG'
//*
//* CBDLL3: -Compile and bind CCNGA2D3 with NCAL
//* -Generate the side-deck CCNGA2D3
//* -The load module will not be kept, as it will not be
//* used
//*
//CBDLL3 EXEC EDCCB,INFILE='CBC.SCCNSAM(CCNGA2D3)',
// CPARM='SO,LIST,DLL,EXPO,RENT,LONG',
// BPARM='NCAL'
//COMPILE.SYSLIN DD DSN=myid.OBJ(CCNGA2D3),DISP=SHR
//BIND.SYSLIN DD DSN=myid.OBJ(CCNGA2D3),DISP=SHR
//BIND.SYSLIN DD *
//INCLUDE OBJ(CCNGA2D2)
//INCLUDE OBJ(CCNGA2D1)
//NAME CCNGA2D3(R)
//*
//BIND.SYSDEFSD DD DSN=myid.IMPORT(CCNGA2D3),DISP=SHR
//BIND.OBJ DD DSN=myid.OBJ,DISP=SHR
//*
```

CCNGA2M2 (Part 1 of 2)

Figure 70. CCNGA2M2

```

/*
/** BDLL2: -Bind CCNGA2D2
/** -Generate the side-deck CCNGA2D2
/*
/**
//BDLL2 EXEC CBCB,INFILE='myid.OBJ(CCNGA2D2)',
// BPARM='CALL',
// OUTFILE='myid.LOAD(CCNGA2D2),DISP=SHR'
//BIND.SYSIN DD DSN=myid.IMPORT(CCNGA2D3),DISP=SHR
//BIND.SYSDEFS DD DSN=myid.IMPORT(CCNGA2D2),DISP=SHR
/*
/**
/** BDLL1: -Bind CCNGA2D1
/** -Generate the side-deck CCNGA2D1
/*
//BDLL1 EXEC CBCB,INFILE='myid.OBJ(CCNGA2D1)',
// BPARM='CALL',
// OUTFILE='myid.LOAD(CCNGA2D1),DISP=SHR'
//BIND.SYSIN DD *
INCLUDE DSD(CCNGA2D2)
INCLUDE DSD(CCNGA2D3)
/*
//BIND.SYSDEFS DD DSN=myid.IMPORT(CCNGA2D1),DISP=SHR
//BIND.DSD DD DSN=myid.IMPORT,DISP=SHR
/*
/** BDLL3: -Bind CCNGA2D3
/** -Generate the side-deck CCNGA2D3
/*
//BDLL3 EXEC CBCB,INFILE='myid.OBJ(CCNGA2D3)',
// BPARM='CALL',
// OUTFILE='myid.LOAD(CCNGA2D3),DISP=SHR'
//BIND.SYSIN DD *
INCLUDE DSD(CCNGA2D1)
INCLUDE DSD(CCNGA2D2)
NAME CCNGA2D3(R)
/*
//BIND.SYSDEFS DD DSN=myid.IMPORT(CCNGA2D3),DISP=SHR
//BIND.DSD DD DSN=myid.IMPORT,DISP=SHR
/*
/** CBAPP2: -Compile, bind and run CCNGA2
/** -Input the side-decks CCNGA2D1, CCNGA2D2 and CCNGA2D3
/*
//CBAPP2 EXEC EDCCBG,INFILE='CBC.SCCNSAM(CCNGA2)',
// CPARM='SO,LIST,DLL,RENT,LONG',
// OUTFILE='myid.LOAD(CCNGA2),DISP=SHR'
//BIND.SYSIN DD *
INCLUDE DSD(CCNGA2D1)
INCLUDE DSD(CCNGA2D2)
INCLUDE DSD(CCNGA2D3)
NAME CCNGA2(R)
/*
//BIND.DSD DD DSN=myid.IMPORT,DISP=SHR
//GO.STEPLIB DD
// DD DSN=myid.LOAD,DISP=SHR

```

CCNGA2M2 (Part 2 of 2)

Chapter 20. z/OS 64-bit environment

Implementation of the 64-bit environment has not changed the default behavior of the compiler; the default compilation environment is 32-bit, which is specified by the ILP32 compiler option.

The compiler changes the behavior of code only when compiling for the 64-bit environment, which is specified by the LP64 compiler option.

Related information

- [Introduction to AMODE 31 and AMODE 64 programs interoperability in z/OS Language Environment Vendor Interfaces](#)

Differences between the ILP32 and LP64 environments

The ILP32 and LP64 environments are differentiated by:

- [Addressing capability](#)
- [Data model](#)

ILP32 and LP64 addressing capabilities

Table 29 on page 221 shows the differences in addressing capabilities that are available in each environment. 31-bit refers to the addressing mode, or AMODE. In z/OS XL C/C++, pointer sizes in this mode are always 4 bytes. In AMODE 31, 31 bits of the pointer are used to form the address, which is defined by the term "31-bit addressing mode". Occasionally, we also use the term "32-bit mode". Strictly speaking, 31-bit is an architectural characteristic referring to the addressing capability, while 32-bit is a programming language aspect referring to the data model. The latter is also referred to as ILP32 (int-long-pointer 32). When there is no ambiguity, we use the term "32-bit mode".

Table 29. Comparison of ILP32 and LP64 addressing capabilities	
ILP32 (32-bit environment)	LP64 (64-bit environment)
2 GB of address space	1 million TB of address space
31-bit execution mode ¹	64-bit execution mode

ILP32 and LP64 data models and data type sizes

Table 30 on page 221 compares data models and data type sizes of ILP32 and LP64 environments.

Table 30. Comparison of ILP32 and LP64 data models	
ILP32 (32-bit environment)	LP64 (64-bit environment)
Data model ILP32 (32-bit pointer)	Data model LP64 (64-bit pointer)
int, long, ptr, and off_t are all 32 bits (4 bytes) in size.	int is 32 bits in size. long, ptr, and off_t are all 64 bits (8 bytes) in size.

The 32-bit data model for z/OS XL C/C++ compilers is ILP32 plus long long. This data model uses the 4/4/4 data type size model and includes a long long type. [Table 31 on page 222](#) compares the type sizes for the different models.

LP64 is the 64-bit data model chosen by the Aspen working group (formed by X/OPEN and a consortium of hardware vendors). LP64 is short for long-pointer 64. It is commonly referred to as the 4/8/8 data type size model and includes the integer/long/pointer type sizes, measured in bytes.

Table 31. ILP32 and LP64 type size comparisons for signed and unsigned data types			
Data Type	32-bit sizes (in bytes)	64-bit sizes (in bytes)	Remarks
char	1	1	
short	2	2	
int	4	4	
long	4	8	
long long	8	8	
float	4	4	
double	8	8	
long double	16	16	
pointer	4	8	
wchar_t	2	4	Other UNIX platforms usually have wchar_t 4 bytes for both 32-bit and 64-bit mode.
size_t	4	8	This is an unsigned type.
ptrdiff_t	4	8	This is a signed type.

Advantages and disadvantages of the LP64 environment

A major advantage of using a 64-bit environment is the increase in the virtual addressing space. A 64-bit program can handle large tables as arrays without putting temporary files in secondary storage. LP64 provides:

- 64-bit addressing with 8-byte pointers
- Large object support (8-byte longs)
- Backward compatibility (4-byte integers)

Note: Integers are the same size under the ILP32 and LP64 data models.

LP64 application performance and program size

You can use the 64-bit address space to dramatically improve the performance of applications that manipulate large amounts of data, whether the data is created within the application or obtained from files. Generally, the performance gain comes from the fact that the 64-bit application can contain the data in its 64-bit address space (either created in data structures or mapped into memory), when it would not have fit into a 32-bit address space. The data would need to be multiple GBs in size or larger to show this benefit.

If the same source code is used to create a 32-bit and a 64-bit application, the 64-bit application will typically be larger than the 32-bit application. The 64-bit application is unlikely to run faster than the 32-bit application unless it makes use of the larger 64-bit addressability. Because most C programs are pointer-intensive, a 64-bit application can be close to twice as large as a 32-bit application, depending on how many global pointers and longs are declared. A 64-bit C++ program uses almost twice the data as a 32-bit C++ program, due to the large number of pointers the compiler uses to implement virtual function tables, objects, templates, and so on. That is why the appropriate choice is to create a 32-bit application, unless 64-bit addressability is required by the application or can be used to dramatically improve its performance.



Attention: Even though the address space is increased significantly, the amount of hardware physical memory is still limited by your installation. Data that is not immediately required by the program is subject to system paging. Programs that use large data tables therefore require a large amount of paging space. For example, if a program requires 3 GB of address space, the

system must have 3 GB of paging space. 64-bit applications might require paging I/O tuning to accommodate the large data handling benefit.

LP64 restrictions

The following restrictions apply under LP64:

- The ILP32 statement `type=memory(hiperspace)` is treated as `type=memory` under LP64.

Hiperspace memory files are treated as regular memory files in a 64-bit environment. All behavior is the same as for regular memory files.

- The IMS and CICS environments are not supported under LP64.

References to these environments are valid under ILP32 only.

- User-supplied buffers are ignored for all but UNIX file system files under LP64.

References to user-supplied buffers are valid under ILP32 only.

- Under 64-bit data models, pointer sizes are always 64 bits.

The C Standard does not provide a mechanism for specifying mixed pointer size. However, it might be necessary to specify the size of a pointer type to help migrate a 32-bit application (for example, when libraries share a common header between 32-bit and 64-bit applications).

Migrating applications from ILP32 to LP64

This section describes:

- [When to migrate applications to LP64](#)
- [Pre-migration checklist](#)
- [Post-migration checklist](#)

When to migrate applications to LP64

The LP64 strategy is to strike a balance between maximizing the robustness of 64-bit capabilities while minimizing the effort of migrating many programs.

Typically, a 32-bit application should be ported only if either of the following is true:

- It is required by a DLL or a supporting utility
- It must have 64-bit addressability

This is because:

- Porting programs to a 64-bit environment presents a modest technical effort where good coding practices are used. Poor coding practices greatly increase the programming effort.
- There is no clear performance advantage to recompiling an existing 32-bit program in 64-bit mode. In fact, a small slowdown is possible. This is due to:
 - An increase in module size because instructions are larger
 - An increase in size of the writable static area (WSA) and the stack because pointers and longs are larger
 - Issues related to runtime requirements (for example, when you port a program that is compiled with `NORENT` and `NODLL` to a 64-bit environment, you must code the program to use the `RENT` and `DLL` options, which are required in the 64-bit environment)

Checklist for ILP32-to-LP64 pre-migration activities

Use the following checklist prior to migrating an application from ILP32 to LP64. After migration, test the code and confirm that its behavior is the same under LP64 as it was under ILP32. If you see any difference, debug the code and use the checklist again.

1. Search the source code for patterns that might indicate migration issues. These include:
 - `printf` specifiers that involve long data types
 - `0xffffffff`
 - `2147483647`
2. Verify that all functions are properly prototyped.

Note: The C compiler assumes that an unprototyped function returns the `int` type. This could cause undesirable behavior under LP64 while remaining undetectable under ILP32.
3. Examine all types to determine whether the types should be 4-byte or 8-byte.
 - For system types, the type will be the appropriate size for use with library/system calls.
 - For user-defined types:
 - 4-byte types should be defined based upon `int` or unsigned `int` or some system type that is 4 bytes long under LP64.
 - 8-byte types should be defined based upon `long` or unsigned `long` or some system type that is 8 bytes long.
4. Change all types to the chosen type.

Note: When doing so, examine all arithmetic calculations to make sure that expansion and truncation of data values is done appropriately. Make sure that no assumption is made that pointer values will fit into integer types.
5. Use the INFO compiler option to identify the following potential problems:
 - Functions not prototyped - Function prototypes allow the compiler to check for mismatched parameters.
 - Functions not prototyped - Return parameter mis-matched, especially when the code expects a pointer. (For example, `malloc` and `family`)
 - Assignment of a long or a pointer to an `int` - This type of assignment could cause truncation. Even assignments with an explicit cast will be flagged.
 - Assignment of an `int` to a pointer - If the pointer is referenced it might be invalid.

Checklist for ILP32-to-LP64 post-migration activities

After migrating a program, test the code and confirm that its behavior is the same under LP64 as it was under ILP32. Use the following checklist to test the code. If you see any difference, debug the code and use the pre-migration checklist again.

1. Verify that all output produced is contained in the 4-byte range.

If this is not possible, then any other application using this data needs to be ported to LP64 or, at least, be made 8-byte-aware.
2. Verify that any user-provided process containing the `wchar_t` type definition did not produce unexpected results.

UNIX `wchar_t` data types are typically defined as four bytes under both 32-bit and 64-bit environments. *The size difference applies to the ILP32 model, not the LP64 model.* The new environment was an opportunity to increase the size for future development. Because `wchar_t` is a type definition, user-provided methods are a likely problem area. A carefully-written application should not require changes.

Using compiler diagnostics to ensure portability of code

This section describes:

- Using the INFO option to ensure that numbers are suffixed
- Using the WARN64 option to identify potential portability problems

Using the INFO option to ensure that numbers are suffixed

The INFO C and C++ option provides general diagnostics about program code and is not specific to migrations from ILP32 to LP64. Before migrating, use the appropriate option to ensure that the following items have been expunged from the code:

- Functions not prototyped - Function prototypes allow the compiler to check for mismatched parameters.
- Functions not prototyped - Return parameter mis-matched, especially when the code expects a pointer. (For example, malloc and family)
- Assignment of a long or a pointer to an int - This type of assignment could cause truncation. Even assignments with an explicit cast will be flagged.
- Assignment of an int to a pointer - If the pointer is referenced it might be invalid.

Table 32. Example of diagnostic messages generated from code that is not ready to be migrated from ILP32 to LP64

Source:	<pre>1 #include <stdio.h> 2 #include <limits.h> 3 4 void main(void) { 5 int foo_i; 6 long foo_l; 7 int *foo_pt; 8 9 foo_l = boo(1); 10 foo_l = foo_l << 1; 11 foo_l = 0xFFFFFFFF; 12 foo_l = (foo_l & 0xFFFFFFFF); 13 foo_l = LONG_MAX; 14 foo_l = (long)foo_i; 15 foo_i = (int) &foo_l; 16 17 foo_pt = (int *)foo_i; 18 } 19 long boo(long boo_l) { 20 return(boo_l); 21 }</pre>
Output:	<pre>WARNING CCN3304 sample.c:9 No function prototype was given for boo. INFORMATIONAL CCN3419 sample.c:11 Converting 4294967295 to type long int does not preserve its value. INFORMATIONAL CCN3438 sample.c:14 The value of the variable foo_i may be used before being set. INFORMATIONAL CCN3491 sample.c:17 The automatic variable foo_pt is set but never referenced. WARNING CCN3343 sample.c:19 Redeclaration of boo differs from the declaration on line 9 of /home/ts43218/sample2.c. INFORMATIONAL CCN3050 sample.c:19 Return type long in the redeclaration is not compatible with the previous return type int. INFORMATIONAL CCN3470 sample.c:21 Function main should return int, not void.</pre>
Note: Lines 9,11 and 14 are affected by porting the code to LP64.	

Using the WARN64 option to identify potential portability problems

Under ILP32, both int and long data types are 32 bits in size. Because of this coincidence, these types might have been used interchangeably. As shown in [Table 31 on page 222](#), the data type long is 8 bytes in length under LP64.

A general guideline is to review the existing use of long data types throughout the source code. If the values to be held in such variables, fields, and parameters will fit in the range of `[-231 . . 231-1]` or `[0 . . 232-1]`, then it is probably best to use `int` or `unsigned int` instead. Also, review the use of the `size_t` type (used in many subroutines), since its type is defined as `unsigned long`.

When you migrate a program from ILP32 to LP64, the data model differences might result in unexpected behavior at execution time. Under LP64, the size of pointers and long data types are 8 bytes, which can lead to conversion or truncation problems. The `WARN64` option can be used to detect these portability errors.

The `WARN64` option provides general diagnostics about program code that might behave differently under ILP32 and LP64. However the checking is not exhaustive. Use it to look for potential migration problems, such as the following common problems:

- Truncation due to explicit or implicit conversion of long types into `int` types
- Unexpected results due to explicit or implicit conversion of `int` types into long types
- Invalid memory references due to explicit conversion by cast operations of pointer types into `int` types
- Invalid memory references due to explicit conversion by cast operations of `int` types into pointer types
- Problems due to explicit or implicit conversion of constants into long types
- Problems due to explicit or implicit conversion by cast operations of constants into pointer types

There are a few problems that `WARN64` cannot find. For example, unions that use longs or pointers that work under ILP32 might not work under LP64 .

```
union {
    int *p; /* 32 bits / 64 bits */
    int i; /* 32 bits / 32 bits */
};

union {
    double d; /* 64 bits / 64 bits */
    long l[2]; /* 64 bits / 128 bits */
};
```

ILP32-to-LP64 portability issues

Before migrating applications, consider the following:

- The sizes of the `long`, `pointer` and `wchar_t` types are different under LP64 than they are under ILP32. You must check application behavior, especially if the logic depends on data size.
- Data model differences can result in unexpected behavior at execution time. Under LP64, the size of pointers and long data type are 8 bytes long. This can lead to conversion or truncation problems.

Note: You can use the `WARN64` option to help detect these portability errors. See [“Using the `WARN64` option to identify potential portability problems” on page 225](#).

- A migration issue can exist if the program assumes that `int`, `long` and `pointer` type are all the same size. The number of cases where program logic relies on this assumption varies from application to application, depending on the coding style and functionality of the application.

Note: Most unexpected behaviors occur at the limits of a type's value range.

- 32-bit applications that rely implicitly on internal data representations (for example, those that cast a float pointer to an integer pointer, then manipulate the bit patterns directly and encode such knowledge directly into the program logic) can be difficult to migrate. In this case, certain assumptions are made about the internal structure of a float representation and the size of `int`.
- Code must be checked to ensure that any shifting and masking operations that manipulate long integers still work properly with a 64-bit long.
- Input and output file dependencies are relevant when you migrate an application that is in the middle of a pipeline of applications, where each application reads the previous application's output as input, and then passes its output to the next application in the pipe. Before migrating one of these applications

to a 64-bit environment, you must verify that the output will not produce values outside of the 32-bit range. Typically, once an application is ported to a 64-bit environment, all downstream applications (that is, any application that depends on output from the ported application) must be ported to a 64-bit environment.

- Extending functions is sometimes included as part of a migration project to exploit the benefit and to justify the cost of migrating to a 64-bit environment. You might have to change code for using expanded limits after extending functions.
- You cannot mix 32-bit and 64-bit object files during binding. The only object file format supported under LP64 bit is GOFF, and the only linkage convention is XPLINK.

IPA(LINK) option and exploitation of 64-bit virtual memory

As of z/OS V1R8 XL C/C++, IPA(LINK) makes use of 64-bit virtual memory, which will cause an XL C/C++ compiler ABEND if there is insufficient storage. The default MEMLIMIT system parameter size in the SMFPRMx parmlib member should be at least 3000 MB. The default MEMLIMIT value takes effect whenever the job does not specify one of the following:

- MEMLIMIT in the JCL JOB or EXEC statement
- REGION=0 in the JCL

Note: The MEMLIMIT value specified in an IEFUSI exit routine overrides all other MEMLIMIT settings.

The z/OS UNIX System Services **ulimit** command can be used to set the MEMLIMIT default. For information, see *z/OS UNIX System Services Command Reference*. For additional information about the MEMLIMIT system parameter, see *z/OS MVS Programming: Extended Addressability Guide*.

As of z/OS V1R8 XL C/C++, the EDCI, EDCXI, EDCQI, CBCI, CBCXI, and CBCQI cataloged procedures, which are used for IPA Link, contain the variable IMEMLIM, which can be used to override the default MEMLIMIT value.

Availability of suboptions

Table 33 on page 227 shows a comparison of the compiler and runtime options that are available in each environment. For example, if you are developing a program to run in either a 32-bit or a 64-bit environment, you must code it to ensure that the high-performance linkage (XPLINK) option is in effect regardless of whether the program is running under ILP32 or LP64.

Table 33. Comparison of ILP32 and LP64 processing and runtime options	
ILP32 (32-bit environment)	LP64 (64-bit environment)
XPLINK or non-XPLINK	XPLINK only
32-bit dynamic linked libraries (DLLs)	64-bit DLLs

Potential changes in structure size and alignment

The LP64 specification changes the size and alignment of certain structure elements, which affects the size of the structure itself. In general, all structures that use long integers and pointers must be checked for size and alignment dependencies.

It is not possible to share a data structure between 32-bit and 64-bit processes, unless the structure is devoid of pointer and long types. Unions that attempt to share long and int types (or overlay pointers onto int types) will be aligned differently or will be corrupted. For example, the virtual function table pointer, inherent in many C++ objects, is a pointer and will change the size and alignment of many C++ objects. In addition, the size and composition of the compiler-generated virtual function table will change.

Note: The issue of changing structure size and alignment should not be a problem unless the program makes assumptions about the size and/or composition of structures.

z/OS basic rule of alignment

The basic rule of alignment in z/OS is that a data structure is aligned in accordance with its size and the strictest alignment requirement for its largest member. An 8-byte alignment is more stringent than a 4-byte alignment. In other words, members that can be placed on a 4-byte boundary can also be placed on an 8-byte boundary, but not vice versa.

Note: The only exception is a `long double`, which is always aligned on an 8-byte boundary.

You can satisfy the rule of alignment by inserting pad members both between members and at the end of a structure, so that the overall size of the structure is a multiple of the structure's alignment.

Examples of structure alignment differences under ILP32 and LP64

This section provides examples of three structures that illustrate the impact of the ILP32 and LP64 programming environments on structure size and alignment.

In accordance with the z/OS rule of alignment (see “z/OS basic rule of alignment” on page 228), the length of each data member produced by the source code depends on the runtime environment, as shown in Table 34 on page 228.

Table 34. Comparison of data structure member lengths produced from the same code	
Source:	<pre>#include <stdio.h> int main(void) { struct li{ long la; int ia; } li; struct lii{ long la; int ia; int ib; } lii; struct ili{ int ia; long la; int ib; } ili; printf("length li = %d\n",sizeof(li)); printf("length lii = %d\n",sizeof(lii)); printf("length ili = %d\n",sizeof(ili)); }</pre>
ILP32 member lengths:	<pre>length li = 8 1 length lii = 12 3 length ili = 12 3</pre>
LP64 member lengths:	<pre>length li = 16 2 length lii = 16 3 length ili = 24 3</pre>

Table 34. Comparison of data structure member lengths produced from the same code (continued)

Notes:

1. In a 32-bit environment, both `int` and `long int` have 4-byte alignments, so each of these members is aligned on 4-byte boundary. In accordance with the z/OS rule of alignment, the structure as a whole has a 4-byte alignment. The size of `struct li` is 8 bytes. See [Figure 71 on page 230](#).
2. In a 64-bit environment, `int` has a 4-byte alignment and `long int` has an 8-byte alignment. In accordance with the z/OS rule of alignment, the structure as a whole has an 8-byte alignment. See [Figure 71 on page 230](#).
3. The `struct lii` and the `struct ili` have the same members, but in a different member order. See [Figure 72 on page 231](#) and [Figure 73 on page 232](#). Because of the padding differences in each environment:
 - Under ILP32:
 - The size of `struct lii` is 12 bytes (4-byte long + 4-byte int + 4-byte int)
 - The size of `struct ili` is 12 bytes (4-byte int + 4-byte long + 4-byte int)
 - Under LP64:
 - The size of `struct lii` is 16 bytes (8-byte long + 4-byte int + 4-byte int)
 - The size of `struct ili` is 24 bytes (4-byte int + 4-byte pad + 8-byte long + 4-byte int + 4-byte pad)

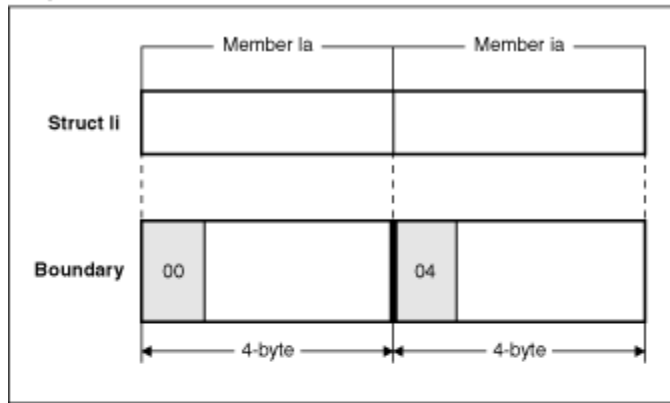
The ILP32 and LP64 alignments for the structs defined by the code shown in [Table 34 on page 228](#) are compared in [Figure 71 on page 230](#), [Figure 72 on page 231](#), and [Figure 73 on page 232](#).

[Figure 71 on page 230](#) compares how `struct li` is aligned under ILP32 and LP64. The structure has two members:

- The first (member `la`) is of type `long`
- The second (member `ia`) is of type `int`

Under ILP32, each member is 4 bytes long and is aligned on a 4-byte boundary, making the structure 8 bytes long. Under LP64, member `la` is 8 bytes long and is aligned on an 8-byte boundary. Member `ia` is 4 bytes long, so the compiler inserts 4 padding bytes to ensure that the structure is aligned to the strictest alignment requirement for its largest member. Then, the structure can be used as part of an array under LP64.

ILP32



LP64

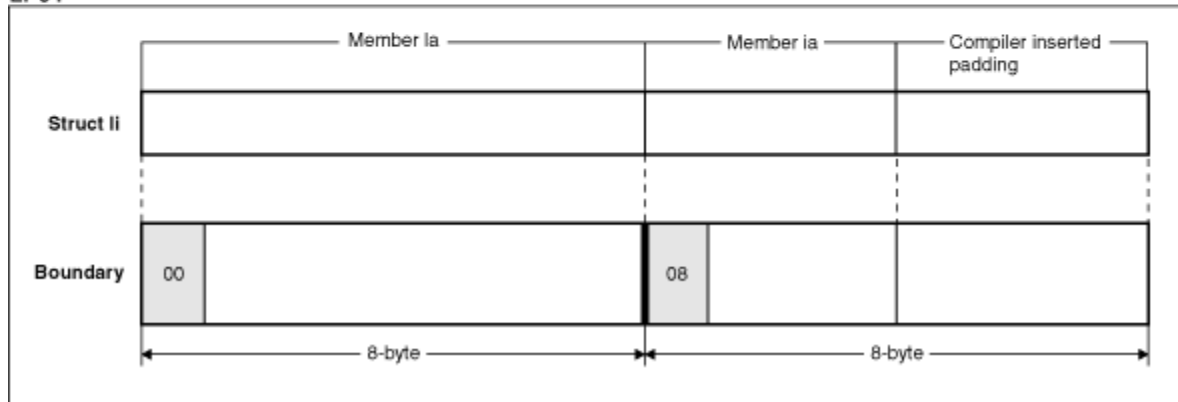


Figure 71. Comparison of struct li, alignments under ILP32 and LP64

Figure 72 on page 231 and Figure 73 on page 232 show structures that have the same members, but in a different order. Compare these figures to see how the order of the members impacts the size of the structures in each environment.

Figure 72 on page 231 compares how `struct lii` is aligned under ILP32 versus LP64. `struct lii` has three members:

- The first (member `la`) is of type `long`
- The second (member `ia`) and third (member `ib`) are of type `int`

Under ILP32, each member is 4 bytes long and is aligned on a 4-byte boundary, making the structure 12 bytes long. Under LP64, member `la` is 8 bytes long and is aligned on an 8-byte boundary. Member `ia` and member `ib` are each 4 bytes long, so the structure is 16 bytes long and can align on an 8-byte boundary without padding.

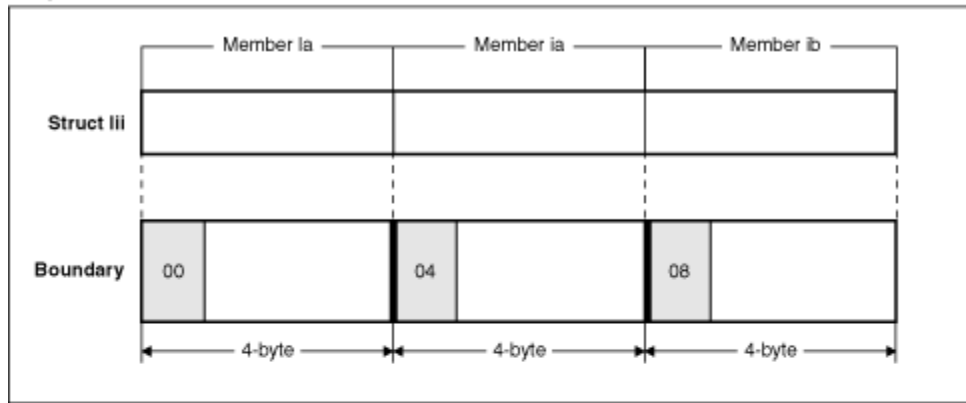
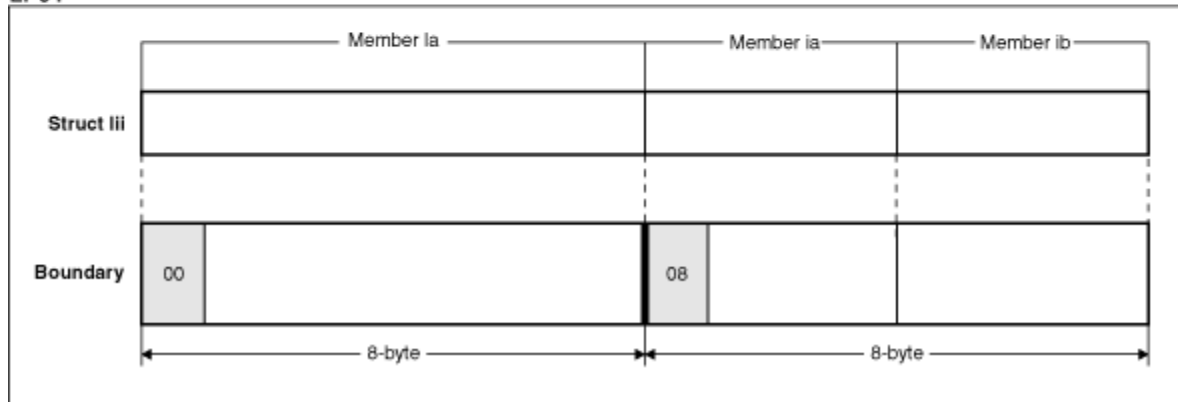
ILP32**LP64**

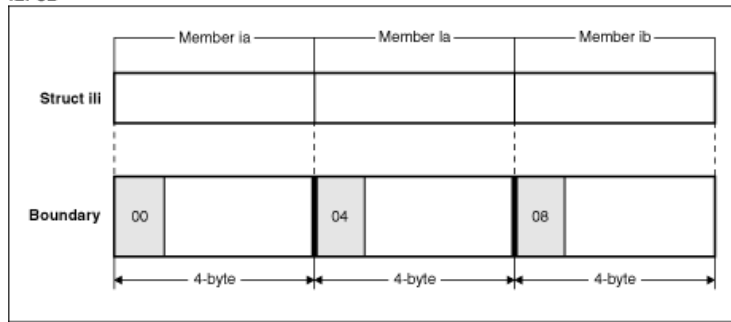
Figure 72. Comparison of struct ili alignments under ILP32 and LP64

Figure 73 on page 232 compares how `struct ili` is aligned under ILP32 and LP64. `struct ili` has three members:

- The first (member ia) is of type `int`
- The second (member la) is of type `long`
- The third (member ib) is of type `int`

Under ILP32, each member is 4 bytes long and is aligned on a 4-byte boundary, making the structure 12 bytes long. Under LP64, the compiler inserts padding after both member ia and member ib, so that each member with padding is 8 bytes long (member la is already 8 bytes long) and are aligned on 8-byte boundaries. The structure is 24 bytes long.

ILP32



LP64

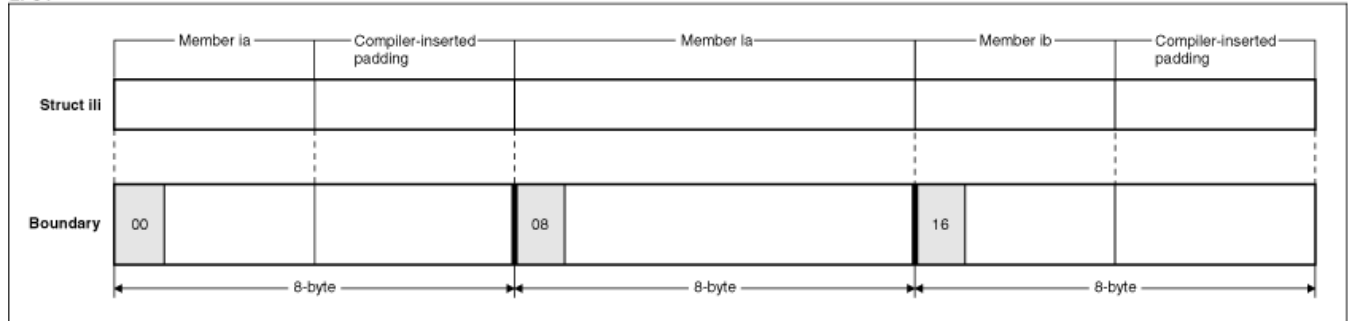


Figure 73. Comparison of struct `ili` alignments under ILP32 and LP64

Data type assignment differences under ILP32 and LP64

Under ILP32, `int`, `long`, and pointer types have the same size and can be freely assigned to one another. Under LP64, all pointer types are 8 bytes in size. However, the pointer size might change when it is qualified by the pointer size qualifier `__ptr32` or `__ptr64` or the `_far` type qualifier.

For objects of different sizes, assigning pointers to `int` types and back again can result in a invalid address, and passing pointers to a function that expects an `int` type will result in truncation. For example, the following statement show an incorrect assignment.

```
int i;
int *p;
i = (int)p;
```

Note: The problem is harder to detect when casts are used. Although there is no warning message, the problem still exists.

Avoid making any of the following assumptions:

- A pointer type or a C long type can fit into a C integer type.
- A type that is derived from a pointer type can fit into a type derived from an integer type.
- The number of bits in a C long type object is assumed, especially when shifting bits or doing bitwise operations.
- A C integer can be passed to an unprototyped long or pointer parameter.
- A function that is not a prototype can return a pointer or long.

Portability issues with data types long and int

Under LP64, types `long` and `int` are not interchangeable. The `long` type (and types derived from it) is 64 bits in size.

You should consider all types related to the `long` and unsigned `long` types. For example, `size_t`, used in many subroutines, is defined under LP64 as unsigned `long`.

Because of the difference in size for `int` and `long` under LP64, conversions to `long` from other integral types might be executed differently than it was under ILP32.

Example of possible change of result after conversion from signed number to unsigned long

When a signed `char`, signed `short`, or signed `int` is converted to unsigned `long`, sign extension might result in a different unsigned value in 64-bit mode. The example in Table 35 on page 233 will yield 4294967295 (0xffffffff) under ILP32 but 18446744073709551615 (0xffffffffffffffff) under LP64, because of sign extension.

Table 35. Example of possible change of result after conversion from signed number to unsigned long	
Source:	<pre>#include<stdio.h> void foo(int i) { unsigned long l = i; printf("%lu (0x%lx)\n", l, l); } void main() { foo(-1); }</pre>
Compiler options:	<code>cc -Wc,"flag(i),warn64" -c warn2.c</code>
Output:	<pre>INFORMATIONAL CCN3743 ./warn2.c:4 64-bit portability: possible change of result through conversion of int type into unsigned long int type.</pre>

Example of possible change of result after conversion from unsigned int variable to signed long

When an unsigned `int` variable with values greater than `INT_MAX` is converted to signed `long`, the results depend on whether the application is executed under ILP32 or under LP64. In the example in Table 36 on page 233:

- Under ILP32, the value `INT_MAX+1` will wrap around and yield -2147483648 (0x80000000)
- Under LP64, the value `INT_MAX+1` can be represented by an 8-byte signed `long` and will result in the correct value 2147483648 (0x80000000)

Table 36. Example of possible change of result after conversion from unsigned int variable to signed long	
Source:	<pre>#include<stdio.h> #include<limits.h> void foo(unsigned int i) { long l = i; printf("%ld (0x%lx)\n", l, l); } void main() { foo(INT_MAX + 1); }</pre>
Compiler options:	<code>cc -Wc,"flag(i),warn64" -c warn3.c</code>

Table 36. Example of possible change of result after conversion from unsigned int variable to signed long (continued)

Output:	INFORMATIONAL CCN3743 ./warn3.c:5 64-bit portability: possible change of result through conversion of unsigned int type into long int type.
----------------	---

Example of possible change of result after conversion from signed long long variable to unsigned long

When a signed long long variable with values either greater than UINT_MAX or less than 0 is converted to unsigned long, truncation will not occur under LP64. The example in [Table 37](#) on page 234 will yield:

- 4294967295 (0xffffffff) 0 (0x0) under ILP32
- 18446744073709551615 (0xffffffffffffffff) 4294967296 (0x100000000) under LP64

Table 37. Example of possible change of result after conversion from signed long long variable to unsigned long

Source:	<pre>#include<stdio.h> #include<limits.h> void foo(signed long long ll) { unsigned long l = ll; printf("%lu (0x%x)\n", l, l); } void main() { foo(-1); foo(UINT_MAX + 1ll); }</pre>
Compiler options:	cc -Wc,"flag(i),warn64" -c warn4.c
Output:	INFORMATIONAL CCN3743 ./warn4.c:564-bit portability: possible change of result through conversion of long long int type into unsigned long int type.

Example of possible change of result after conversion from unsigned long long variable to unsigned long

Under LP64, when an unsigned long long variable with values greater than UINT_MAX is converted to unsigned long, truncation will not occur.

Table 38. Example of possible change of result after conversion from unsigned long long variable to unsigned long

Source:	<pre>#include<stdio.h> #include<limits.h> void foo(unsigned long long ll) { unsigned long l = ll; printf("%ld (0x%x)\n", l, l); } void main() { foo(UINT_MAX + 1ull); }</pre>
----------------	---

Table 38. Example of possible change of result after conversion from unsigned long long variable to unsigned long (continued)

ILP32 output:	0 (0x0) Note: The higher order word is truncated.
LP64 output:	4294967296 (0x100000000) Note: There is no truncation.

Example of possible change of result after conversion from signed long long variable to signed long

Under LP64, when a signed long long variable with values less than INT_MIN or greater than INT_MAX is converted to signed long, truncation does not occur.

Table 39. Example of possible change of result after conversion from signed long long variable to signed long

Source:	<pre>#include<stdio.h> #include<limits.h> void foo(signed long long ll) { signed long l = ll; printf("%ld (0x%lx)\n", l, l); } void main() { foo(INT_MIN - 1ll); foo(INT_MAX + 1ll); }</pre>
Compiler options:	cc -Wc,"flag(i),warn64" -c warn5.c
ILP32 output:	<p>INFORMATIONAL CCN3743 ./warn5.c:5 64-bit portability: possible change of result through conversion of long long int type into long int type.</p> <p>2147483647 (0x7fffffff) -2147483648 (0x80000000)</p> <p>Note: The higher order word is truncated.</p>
LP64 output:	<p>INFORMATIONAL CCN3743 ./warn5.c:5 64-bit portability: possible change of result through conversion of long long int type into long int type.</p> <p>-2147483649 (0xffffffff7fffffff) 2147483648 (0x80000000)</p> <p>Note: There is no truncation.</p>

Example of possible change of result after conversion from unsigned long long variable to signed long

Under LP64, when an unsigned long long variable with values greater than INT_MAX is converted to signed long, truncation does not occur.

Table 40. Example of possible change of result after conversion from unsigned long long variable to signed long

Source:	<pre>#include<stdio.h> #include<limits.h> void foo(unsigned long long ll) { signed long l = ll; printf("%ld (0x%lx)\n", l, l); } void main() { foo(INT_MAX + 1ull); }</pre>
Compiler options:	cc -Wc,"flag(i),warn64" -c warn6.c
ILP32 output:	<p>INFORMATIONAL CCN3743 ./warn6.c:5 64-bit portability: possible change of result through conversion of unsigned long long int type into long int type.</p> <p>-2147483648 (0x80000000)</p> <p>Note: The value INT_MAX+1ull will wrap around.</p>
LP64 output:	<p>INFORMATIONAL CCN3743 ./warn6.c:5 64-bit portability: possible change of result through conversion of unsigned long long int type into long int type.</p> <p>2147483648 (0x80000000)</p> <p>Note: The value INT_MAX+1ull can be represented by an 8-byte signed long and will result in the correct value.</p>

Pointer declarations when 32-bit and 64-bit applications share header files

In 64-bit data models, pointer sizes are always 64 bits. There is no standard language syntax for specifying mixed pointer size. However, it might be necessary to specify the size of a pointer type to help migrate a 32-bit application (for example, when libraries share a common header between 32-bit and 64-bit applications).

The z/OS XL C/C++ compiler reserves two pointer size qualifiers:

- `__ptr32`
- `__ptr64`

The size qualifier `__ptr32` declares a pointer to be 32 bits in size. This is ignored under **ILP32**. The size qualifier `__ptr64` declares a pointer to be 64 bits in size. This is ignored under **LP64**.

Examples of pointer declarations that can be made under **LP64**:

```
int * __ptr32 p; /* 32-bit pointer */ 1, 3
int * r;        /* 64-bit pointer, default to the model's size */ 4
int * __ptr32 const q; /* 32-bit const pointer */ 1, 2, 3
```

Notes:

1. The qualifier qualifies the `*` before it.
2. `q` is a 32-bit constant pointer to an integer.
3. When `__ptr32` is used, the program expects that the address of the pointer variable is less than or equal to 31 bits. You might need to ensure this by calling a special runtime function, such as the

Language Environment runtime function `__malloc31`. You can call `__malloc31` whenever you use your own assembler routine to get storage, and want to keep the addresses in structures and unions to a length of four bytes.

4. If a pointer declaration does not have the size qualifier, it defaults to the size of the data model.

Potential pointer corruption

When porting a program from ILP32 to LP64, be aware of the following potential problems:

- An invalid address might be the result of either of the following actions:
 - Assigning an integer (4 bytes) or a 4-byte hexadecimal constant to a pointer type variable (8 bytes)
 - Casting a pointer to an integer type
- Note:** An invalid address causes errors when the pointer is dereferenced.
- If you compare an integer to a pointer, you might get unexpected results.
- Data truncation might result if you convert pointers to signed or unsigned integers with the expectation that the pointer value will be preserved.
- If return values of functions that return pointers are assigned to an integer type, those return values will be truncated.
- If code assumes that pointers and integers are the same size (in an arithmetic context), there will be problems. Pointer arithmetic is often a source of problems when migrating code. The ISO C and C++ standards dictate that incrementing a pointer adds the size of the data type to which it points to the pointer value. For example, if the variable `p` is a pointer to `long`, the operation `(p+1)` increments the value of `p` by 4 bytes (in 32-bit mode) or by 8 bytes (in 64-bit mode). Therefore, casts between `long*` and `int*` are problematic because of the size differences between pointer objects (32 bits versus 64 bits).

Potentially incorrect pointer-to-int and int-to-pointer conversions

Before porting code, It is important to test the ILP32 code to determine if any code paths would have incorrect results under LP64. For example:

- When a pointer is explicitly converted to an integer, truncation of the high-order word occurs.
- When an integer is explicitly converted to a pointer, the pointer might not be correct, which could result in invalid memory access when the pointer is dereferenced.

Table 41. Example of source code that explicitly converts an integer to a pointer	
Source:	<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 int main() 4 { 5 int i, *p, *q; 6 p = (int*)malloc(sizeof(int)); 7 i = (int)p; 8 q = (int*)i; 9 p[0] = 55; 10 printf("p = %p q = %p\n", p, q); 11 printf("p[0] = %d q[0] = %d\n", p[0], q[0]); 12 }</pre>
Compiler options:	<code>c89 -Wc,"flag(i),warn64" -c warn7.c</code>

Table 41. Example of source code that explicitly converts an integer to a pointer (continued)

Output:	<pre> INFORMATIONAL CCN3744 ./warn7.c:7 64-bit portability: possible truncation of pointer through conversion of pointer type into int type. INFORMATIONAL CCN3745 ./warn7.c:8 64-bit portability: possible incorrect pointer through conversion of int type into pointer. </pre>
----------------	--

Notes:

1. Under ILP32, the pointers p and q are pointing to the same memory location.
2. Under LP64, the pointer q is likely pointing to an invalid address, which could result in a segmentation fault when q is dereferenced.
3. Warning messages are generated for invalid conversions, as shown in [Table 41 on page 237](#).

Potential truncation problem with a pointer cast conversion

As [Table 42 on page 238](#) shows, truncation problems can occur when converting between 64-bit and 32-bit data objects. Because `int` and `long` are both 32 bits under ILP32, a mixed assignment or conversion between these data types did not represent any problem. However, under LP64, a mixed assignment or conversion does present problems because `long` is larger in size than `int`. Without an explicit cast, the compiler is unable to determine whether the narrowing of assignment is intended. If the value `l` is always within the range representable by an `int`, or if the truncation is intended by design, use an explicit cast to silent the `WARN64` message that you will receive for this code.

Table 42. Example of truncation problem with a pointer cast conversion

Source:	<pre> void foo(long l) { int i = l; } </pre>
Compiler options:	<pre>cc -Wc,"flag(i),warn64" -c warn1.c</pre>
Output:	<pre> WARNING CCN3742 ./warn1.c:3 64-bit portability: possible loss of digits through conversion of long int type into int type. </pre>

Potential loss of data in constant expressions

A loss of data can occur in some constant expressions because of lack of precision. These types of problems are very hard to find and might be unnoticed. It is possible to write data-neutral code that can be compiled under both ILP32 and LP64.

When coding constant expressions, you must be very explicit about specifying types and use the constant suffixes `{u,U,l,L,ll,LL}` to specify types, as shown in [Table 43 on page 239](#). You could also use casts to specify the type of a constant expression.

It is especially important to code constant expressions carefully when you are porting programs to a 64-bit environment because integer constants might have different types when compiled in 64-bit mode. The ISO C and C++ standards state that the type of an integer constant, depending on its format and suffix, is the first (that is, smallest) type in the corresponding list that will hold the value. The number of leading zeros does not influence the type selection. [Table 43 on page 239](#) describes the type of an integer constant according to the ISO standards.

Table 43. Type of an integer constant		
Suffix	Decimal constant	Octal or hexadecimal constant
unsuffixed	int long unsigned long	int unsigned int long unsigned long
u or U	unsigned int unsigned long	unsigned int unsigned long
l or L	long unsigned long	long unsigned long
Both u or U and l or L	unsigned long	unsigned long
ll or LL	long long	long long unsigned long long
Both u or U and ll or LL	unsigned long long	unsigned long long

Note: Under LP64, a change in the type of a constant in an expression might cause unexpected results because long is equal to long long. For example, an unsuffixed hexadecimal constant that can be represented only by an unsigned long in 32-bit mode can fit within a long in 64-bit mode.

Data alignment problems when structures are shared

Modern processor designs usually require data in memory to be aligned to their natural boundaries, in order to gain the best possible performance. In most cases, the compiler ensures proper alignment by inserting padding bytes immediately in front of the misaligned data. Although the padding bytes do not affect the integrity of the data, they might result in an unexpected layout, which affects the size of structures and unions.

Because both pointer size and long size are doubled in 64-bit mode, structures and unions containing them as members are larger than they are in 32-bit mode.



Attention: The example in Table 44 on page 239 is for illustrative purposes only. Sharing pointers between 32-bit and 64-bit processes is *not recommended* and will likely yield incorrect results.

Table 44. An attempt to share pointers between 32-bit and 64-bit processes	
 Attention: Source:	<pre> #include <stdio.h> #include <stddef.h> int main() { struct T { char c; int *p; short s; } t; printf("sizeof(t) = %d\n", sizeof(t)); printf("offsetof(t, c) = %d sizeof(c) = %d\n", offsetof(struct T, c), sizeof(t.c)); printf("offsetof(t, p) = %d sizeof(p) = %d\n", offsetof(struct T, p), sizeof(t.p)); printf("offsetof(t, s) = %d sizeof(s) = %d\n", offsetof(struct T, s), sizeof(t.s)); } </pre>

Table 44. An attempt to share pointers between 32-bit and 64-bit processes (continued)

ILP32 output:	<pre> sizeof(t) = 12 offsetof(t, c) = 0 sizeof(c) = 1 offsetof(t, p) = 4 sizeof(p) = 4 offsetof(t, s) = 8 sizeof(s) = 2 </pre>
LP64 output:	<pre> sizeof(t) = 24 offsetof(t, c) = 0 sizeof(c) = 1 offsetof(t, p) = 8 sizeof(p) = 8 offsetof(t, s) = 16 sizeof(s) = 2 </pre>

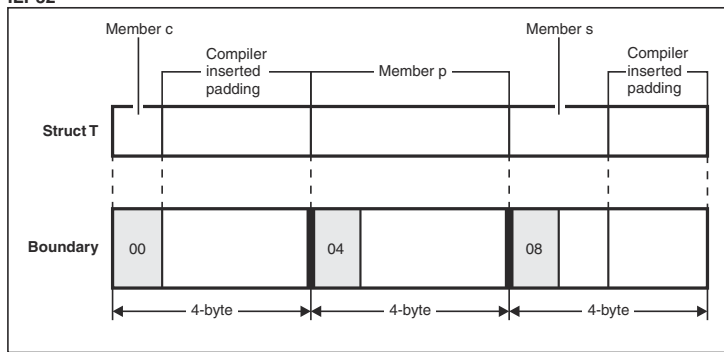
Notes:

1. When the source is compiled and executed under ILP32, the result indicates that paddings have been inserted before the member p, and after the member s. Three padding bytes have been inserted before the member p to ensure that p is aligned to its natural 4-byte boundary. The alignment of the structure itself is the alignment of its strictest member. In this example, it is a 4-byte alignment because the member p has the strictest alignment. Two padding bytes are inserted at the end of the structure to make the total size of the structure a multiple of 4 bytes. This is required so that if you declare an array of this structure, each element of the array will be aligned properly.
2. When the source is compiled and executed under LP64, the size of the structure doubles because additional padding is required to force the member p to fall on a natural alignment boundary of 8-bytes.

Figure 74 on page 241 illustrates how the compiler treats the source code shown in Table 44 on page 239 under ILP32 and LP64. Because the pointer is a different size in each environment, they are aligned on different boundaries. This means that if the code is compiled under both ILP32 and LP64, there are likely to be alignment problems. Figure 75 on page 245 illustrates the solution, which is to define pad members of type character that prevent the possibility of data misalignment. Table 47 on page 244 shows the necessary modifications to the code in Table 44 on page 239.

If the structure in Table 44 on page 239 is shared or exchanged among 32-bit and 64-bit processes, the data fields (and padding) of one environment will not match the expectations of the other, as shown in Figure 74 on page 241.

ILP32



LP64

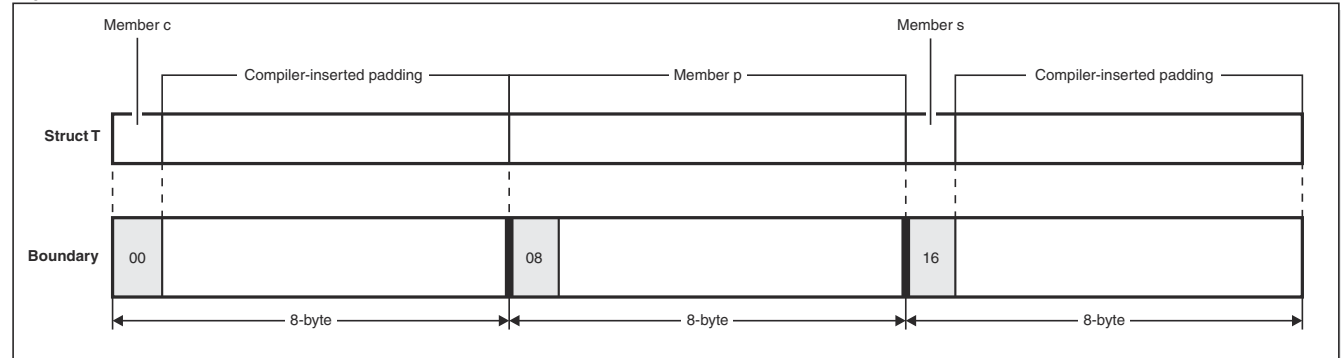


Figure 74. Example of potential alignment problems when a struct is shared or exchanged among 32-bit and 64-bit processes

Portability issues with unsuffixed numbers

When porting code, be aware that:

- Unsuffixed constants are more likely to become 8 bytes long if they are in hexadecimal.
- All constants that can impact any constant assignment must be explicitly suffixed.

Example of unexpected behavior resulting from use of unsuffixed numbers

This causes some operations, such as one that compares `sizeof(4294967295)` to another value, to return 8. If you add the suffix `U` to the number (`4294967295U`), the compiler can parse it as unsigned `int`.

Table 45. Example of unexpected behavior resulting from use of unsuffixed numbers

Source:	<pre>#include <stdio.h> #include <limits.h> void main(void) { long l = LONG_MAX; printf("size(2147483647) = %d\n", sizeof(2147483647)); printf("size(2147483648) = %d\n", sizeof(2147483648)); printf("size(4294967295U) = %d\n", sizeof(4294967295U)); printf("size(-1) = %d\n", sizeof(-1)); printf("size(-1L) = %d\n", sizeof(-1L)); printf("LONG_MAX = %d\n", l); }</pre>
----------------	---

Table 45. Example of unexpected behavior resulting from use of unsuffixed numbers (continued)	
ILP32 output:	<pre> size(2147483647) = 4 size(2147483648) = 4 size(4294967295U) = 4 size(-1) = 4 size(-1L) = 4 LONG_MAX = 2147483647 </pre>
LP64 output:	<pre> size(2147483647) = 4 size(2147483648) = 8 size(4294967295U) = 4 size(-1) = 4 size(-1L) = 8 LONG_MAX = -11 </pre>

Example of how a suffix causes the compiler to parse the number differently under ILP32 than under LP64

Example: A number like 4294967295 (UINT_MAX), when parsed by the compiler, will be

- An unsigned long under ILP32
- A signed long under LP64

Using a LONG_MAX macro in a printf subroutine

The printf subroutine format string for a 64-bit integer is different than the string used for a 32-bit integer. Programs that do these conversions must use the proper format specifier.

Under LP64, you must also consider the maximum number of digits of the long and unsigned long types. The ULONG_MAX is twenty digits long, and the LONG_MAX is nineteen digits.

In [Table 46 on page 242](#), the code assumes that the long type is the same size as the int type (as it would be under ILP32). That is, %d is used instead of %ld.

Table 46. Example of using LONG_MAX macros in a printf subroutine	
Source:	<pre> #include <stdio.h> int main(void) { printf("LONG_MAX(d) = %d\n", LONG_MAX); printf("LONG_MAX(x) = %x\n", LONG_MAX); printf("LONG_MAX(lu) = %lu\n", LONG_MAX); printf("LONG_MAX(lx) = %lx\n", LONG_MAX); } </pre>
LONG_MAX value:	9,223,372,036,854,775,807
Output:	<pre> LONG_MAX(d) = -1 LONG_MAX(x) = ffffffff LONG_MAX(lu) = 9223372036854775807 LONG_MAX(lx) = 7fffffffffffffff </pre>

Notes:

1. Under LP64:
 - %ld must be used
 - %x will give incorrect results and must be replaced by %p or %lx
2. A similar example would produce the same results for an unsigned long with a ULONG_MAX value of 18,446,744,073,709,551,615.

Programming for portability between ILP32 and LP64

When you want to program for portability between the ILP32 and LP64 environments, you can use the following strategies:

- [Header files to provide type definitions](#)
- [Suffixes and explicit types to prevent unexpected behavior](#)
- [Defining pad members to avoid data alignment problems](#)
- [Prototypes to avoid debugging problems](#)
- [Conditional compiler directive for preprocessor macro selection](#)
- [Converters](#)
- [Locales](#)

Using header files to provide type definitions

The header file `inttypes.h` provides type definitions for integer types that are guaranteed to have a specific size (for example, `int32_t` and `int64_t`, and their unsigned variations). Consider using those type definitions if your program code relies on types with specific sizes.

There are many ways to use headers to handle code that is portable between ILP32 and LP64. You can minimize the amount of conditional compilation code and avoid having totally different sections of code for a ILP32 and LP64 structure definitions if you adopt a coding convention that suits your environment.

If you provide a library to your application users and ship header files that define the application programming interface of the library, consider shipping a single set of headers that can support both 32-bit and 64-bit versions of your library. You can use the type definitions in `inttypes.h`. For example, if you are currently shipping 32-bit versions of your header files, you could:

- Replace all fields of type `long` with type `int32_t` (or another 32-bit type)
- Similarly replace all fields for the unsigned variation
- If you cannot let a 64-bit application use a 64-bit pointer for a field, use the `__ptr32` qualifier.

Using suffixes and explicit types to prevent unexpected behavior

The C language limit (in `limits.h`) is different under LP64 than it is under ILP32. As the following example shows, you can prevent unexpected behavior by an application by using suffixes and explicit types with all numbers.

```
#ifndef _LP64
#define LONG_MAX (9223372036854775807L)
#define LONG_MIN (-LONG_MAX - 1)
#define ULONG_MAX (18446744073709551615U)
#else
#define LONG_MAX INT_MAX
#define LONG_MIN INT_MIN
#define ULONG_MAX (UINT_MAX)
#endif /* _LP64 */
```

Note: The output for `LONG_MAX` is not really -1. The reason for the -1 is that:

- The `printf` subroutine handles it as an integer
- `(LONG_MAX == (int) LONG_MAX)` returns a negative value

Defining pad members to avoid data alignment problems

If you want to allow the structure to be shared, you might be able to reorder the fields in the data structure to get the alignments in both 32-bit and 64-bit environments to match (as shown in [Table 34](#) on [page 228](#)), depending on the data types used in the structure and the way in which the structure as

a whole is used (for example, whether the structure is used as a member of another structure or as an array).

If you are unable to reorder the members of a structure, or if reordering alone cannot provide correct alignment, you can define paddings that force the members of the structure to fall on their natural boundaries regardless of whether it is compiled under ILP32 or LP64. A conditional compilation section is required whenever a structure uses data types that have different sizes in 32-bit and 64-bit environments.

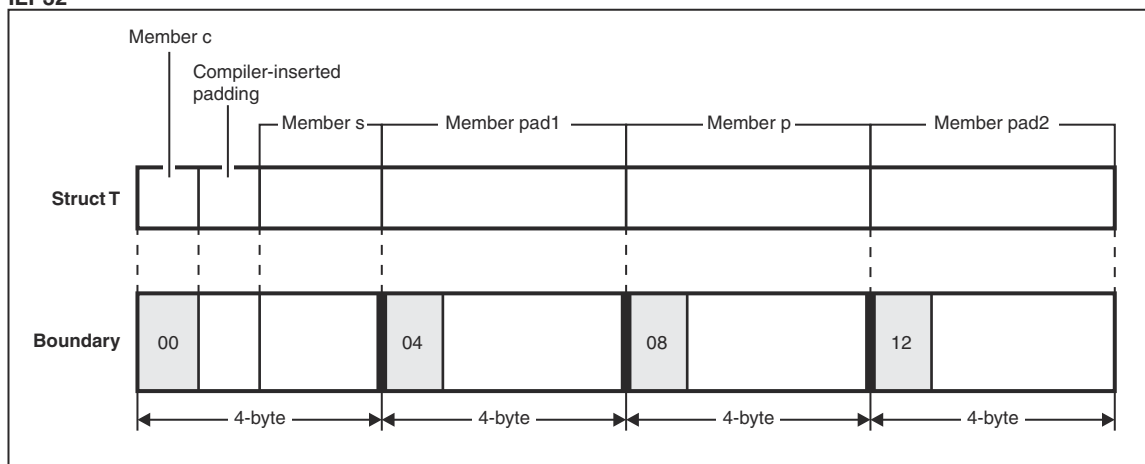
The example in Table 47 on page 244 shows how the source code in Table 44 on page 239 can be modified to avoid the data alignment problem.

Table 47. Example of source code that successfully shares pointers between ILP32 and LP64 programs	
Source:	<pre> struct T { char c; short s; #if !defined(_LP64) char pad1[4]; #endif int *p; #if !defined(_LP64) char pad2[4]; #endif } t </pre>
ILP32/ LP64 size and member layout:	<pre> sizeof(t) = 16 offsetof(t, c) = 0 sizeof(c) = 1 offsetof(t, s) = 2 sizeof(s) = 2 offsetof(t, p) = 8 sizeof(p) = 4 </pre>

Figure 75 on page 245 shows the member layout of the structure with user-defined padding. Because the pointer is a different size in each environment, it is aligned on different a boundary in each environment. This means that if the code is compiled under both ILP32 and LP64, there are likely to be alignment problems. This figure illustrates the solution, which is to define pad members of type character that prevent the possibility of data misalignment.

Note: When inserting paddings into structures, use an array of characters. The natural alignment of a character is 1-byte, which means that it can reside anywhere in memory.

ILP32



LP64

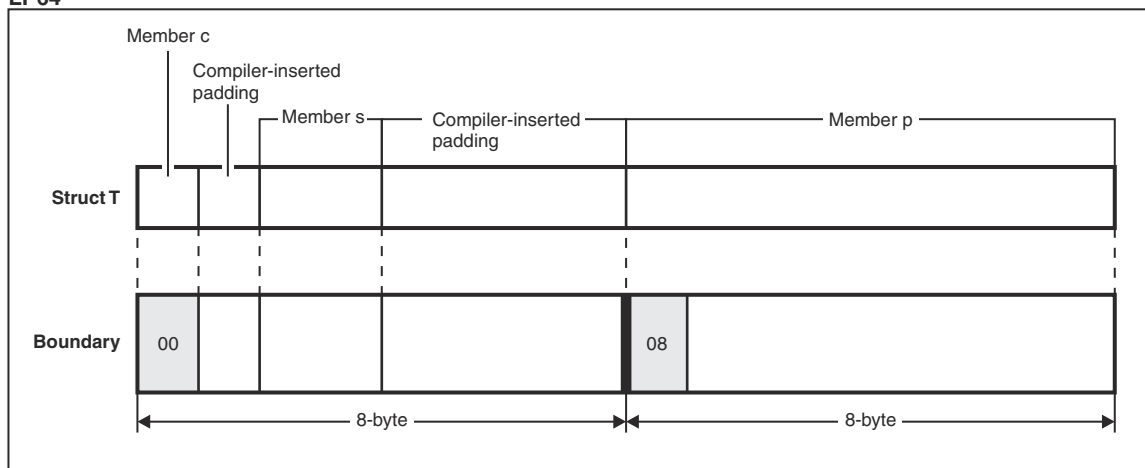


Figure 75. Example of user-defined data padding for a structure that is shared or exchanged among 32-bit and 64-bit processes

Using prototypes to avoid debugging problems

You can avoid complex debugging problems by ensuring that all functions are prototyped.

The C language provides a default prototype. If a function is not prototyped, it defaults to a function which returns an integer and has no information about the parameters.

The C++ language does not provide a default and always requires a prototype. However, C++ has an implicit integer return type extension for legacy code.

A common problem is that the default return type of `int` might not remain the same size as an associated pointer. For example, the function `malloc()` can cause truncation when an unprototyped function returns a pointer. This is because an unprototyped function is assumed to return an `int` (4 bytes).

Using a conditional compiler directive for preprocessor macro selection

When the compiler is invoked with the LP64 option, the preprocessor macro `_LP64` is defined. When the compiler is invoked with the ILP32 option, the macro `_ILP32` is defined.

You can use a conditional compiler directive such as `#if defined _LP64` or `#ifndef _LP64` to select lines of code (such as `printf` statements) that are appropriate for the data model that is invoked.

Using converters under ILP32 or LP64

Both table-driven converters (such as `EDCGNXLT proc`) and indirect UCS-2 converters (such as the `uconvdef` UNIX System Services utility) function the same in both 32-bit and 64-bit environments. The naming convention requires that dataset member names must begin with `CEQ`.

Notes:

1. GENXLT converters are shipped only in data sets.
2. The converter objects that are shipped with z/OS 2.5 allow existing applications to work at a basic level only. You might need to build customized objects.

Using locales under ILP32 or LP64

The locale objects that are shipped with z/OS 2.5 allow existing applications to work at a basic level only. You might need to build customized objects.

Customized 64-bit locales

If you need to create 64-bit locales, you must use the UNIX System Services `localedef` utility with the new `-6` compiler option.

- If the locales are dataset members, they must have the `CEQ` prefix.
- If the locales are zFS-resident or UNIX file system-resident, they must have the `.lp64` suffix.

Note: There is no batch or TSO `LOCALDEF` support for 64-bit locales.

Old SAA locales

Old SAA locales (such as `EDC$FRAN`) are not supported by the LP64 model.

Chapter 21. Using threads in z/OS UNIX applications

A thread is a single flow of control within a process. The following section describes some of the advantages of using multiple threads within a single process, and functions that can be used to maintain this environment.

Models and requirements

Threads are efficient in applications that allow them to take advantage of any underlying parallelism available in the host environment. This underlying parallelism in the host can be exploited either by forking a process and creating a new address space, or by using multiple threads within a single process. There are advantages and disadvantages to both techniques, but it primarily comes down to a compromise between the efficiency of using multiple threads versus the security of working in separate address spaces. The POSIX(ON) runtime option must be specified to use threads.

Functions

Table 48 on page 247 lists the functions provided to implement a multi-threaded application.

Table 48. Functions used in creating multi-threaded applications	
Function	Purpose
pthread_create()	Create a thread
pthread_join()	Wait for thread termination
pthread_exit()	Terminate a thread normally
pthread_detach()	Detach a thread
pthread_self()	Get your thread ID
pthread_equal()	Compare thread IDs
pthread_once()	Run a function once per process
pthread_yield()	Yield the processor

Creating a thread

To use a thread you must first create a thread attribute object with the pthread_attr_init() function. A thread attribute object defines the modifiable characteristics that a thread may have. Refer to the description of pthread_attr_init() in *z/OS C/C++ Runtime Library Reference* for a list of the attributes and their default values. When the thread attribute object has been created, you may use the functions listed in Table 49 on page 247 to change the default attributes.

Table 49. Functions to change default attributes	
Function	Purpose
pthread_attr_init()	Initialize a thread attribute object
pthread_attr_destroy()	Delete a thread attribute object
pthread_attr_getguardsize()	Gets the threadstack guardsize from the thread attribute object
pthread_attr_setguardsize()	Sets the threadstack guardsize in the thread attribute object
pthread_attr_getschedparam()	Gets the schedparam from the thread attribute object
pthread_attr_setschedparam()	Sets the schedparam in the thread attribute object

Table 49. Functions to change default attributes (continued)

Function	Purpose
<code>pthread_attr_getstack()</code>	Gets the stacksize and base storage address of application-managed stack
<code>pthread_attr_setstack()</code>	Sets the stacksize and base storage address of application-managed stack
<code>pthread_attr_getstacksize()</code>	Gets the stacksize for thread attribute object
<code>pthread_attr_setstacksize()</code>	Sets the stacksize for thread attribute object
<code>pthread_attr_getdetachstate()</code>	Returns current value of detachstate for thread attribute object
<code>pthread_attr_setdetachstate()</code>	Alters the current detachstate of thread attribute object
<code>pthread_attr_getweight_np()</code>	Obtains the current weight of thread setting
<code>pthread_attr_setweight_np()</code>	Alters the current weight of thread setting
<code>pthread_attr_getsynctype_np()</code>	Returns the current synctype setting of thread attribute object
<code>pthread_attr_setsynctype_np()</code>	Alters the synctype setting of thread attribute object

The attribute object is only used when the thread is created. You can reuse it to create other threads with the same attributes, or you can modify it to create threads with other attributes. You can delete the attribute object with the `pthread_attr_destroy()` function.

After you create the thread attribute object, you can then create the thread with the `pthread_create()` function.

When a daughter thread is created, the function specified on the `pthread_create()` as the start routine begins to execute concurrently with the thread that issued the `pthread_create()`. It may use the `pthread_self()` function to determine its thread ID. The daughter thread will continue to execute until a `pthread_exit()` is issued, or the start routine ends. The function that issued the `pthread_create()` resumes as soon as the daughter thread is created. The daughter thread ID is returned on a successful `pthread_create()`. This thread ID, for example, can be used to send a signal to the daughter thread using `pthread_kill()` or it can be used in `pthread_join()` to cause the initiating thread to wait for the daughter thread to end.

Table 50 on page 248 lists functions that can be used to control the behavior of the individual threads in a multi-threaded application. Refer to [z/OS C/C++ Runtime Library Reference](#) for more information on these functions.

Table 50. Functions used to control individual threads in a multi-threaded environment

Function	Purpose
<code>pthread_equal()</code>	Compares two thread IDs
<code>pthread_yield()</code>	Allows threads to give up control

Synchronization primitives

This section covers the control of multiple threads that may share resources. In order to maintain the integrity of these resources, a method must exist for the threads to communicate their use of, or need to use, a resource. The threads can be within a common process or in different processes.

Models

Mutexes, condition variables, and read-write locks are used to communicate between threads. These constructs may be used to synchronize the threads themselves, or they can also be used to serialize access to common data objects shared by the threads.

- The *mutex*, which is the simple type of lock, is exclusive. If a thread has a mutex locked, the next thread that tries to acquire the same mutex is put in a wait state. This is beneficial when you want to serialize access to a resource. This might cause contention however if several threads are waiting for a thread to unlock a mutex. Therefore, this form of locking is used more for short durations. If the mutex is a shared mutex, it must be obtained in shared memory accessible among the cooperating processes.

A thread in mutex wait will not be interrupted by a signal.

- A *condition variable* provides a mechanism by which a thread can suspend execution when it finds some condition untrue, and wait until another thread makes the condition true. For example, threads could use a condition variable to insure that only one thread at a time had write access to a data set.

Threads in condition wait can be interrupted by signals.

- A *read-write lock* can allow many threads to have simultaneous read-only access to data while allowing only one thread at a time to have write access. The read-write lock must be allocated in memory that is writable. If the read-write lock is a shared read-write lock, it must be obtained in shared memory accessible among the cooperating processes.

Functions

Table 51 on page 249 lists functions that allow for synchronization between threads.

Table 51. Functions that allow for synchronization between threads	
Function	Purpose
<code>pthread_mutex_init()</code>	Initialize a Mutex
<code>pthread_mutex_destroy()</code>	Destroy a Mutex
<code>pthread_mutexattr_init()</code>	Initialize Default Attribute Object for a Mutex
<code>pthread_mutexattr_destroy()</code>	Destroy Attribute Object for a Mutex
<code>pthread_mutexattr_getkind_np()</code>	Get Kind Attribute for a Mutex
<code>pthread_mutexattr_setkind_np()</code>	Set Kind Attribute for a Mutex
<code>pthread_mutexattr_gettype()</code>	Get Type Attribute for a Mutex
<code>pthread_mutexattr_settype()</code>	Set Type Attribute for a Mutex
<code>pthread_mutexattr_getpshared()</code>	Get Process-shared Attribute for a Mutex
<code>pthread_mutexattr_setpshared()</code>	Set Process-shared Attribute for a Mutex
<code>pthread_mutex_lock()</code>	Acquire a Mutex Lock
<code>pthread_mutex_unlock()</code>	Release a Mutex Lock
<code>pthread_mutex_trylock()</code>	Allows lock to be tested
<code>pthread_cond_init()</code>	Initialize a Condition Variable
<code>pthread_cond_destroy()</code>	Destroy a Condition Variable
<code>pthread_condattr_init()</code>	Initialize Default Attribute Object for a Condition Variable
<code>pthread_condattr_destroy()</code>	Destroy Attributes Object for a Condition Variable
<code>pthread_condattr_getkind_np()</code>	Get Attribute for Condition Variable object
<code>pthread_condattr_setkind_np()</code>	Set Attribute for Condition Variable object
<code>pthread_condattr_getpshared()</code>	Get the Process-shared Condition Variable Attribute
<code>pthread_condattr_setpshared()</code>	Set the Process-shared Condition Variable Attribute
<code>pthread_cond_wait()</code>	Wait for a Condition Variable
<code>pthread_cond_timedwait()</code>	Timed wait for a Condition Variable

Table 51. Functions that allow for synchronization between threads (continued)

Function	Purpose
<code>pthread_cond_signal()</code>	Signal a Condition Variable
<code>pthread_cond_broadcast()</code>	Broadcast a Condition Variable
<code>pthread_rwlock_init()</code>	Initialize a Read-Write Lock
<code>pthread_rwlock_destroy()</code>	Destroy a Read-Write Lock
<code>pthread_rwlock_rdlock()</code>	Wait for a Read Lock
<code>pthread_rwlock_tryrdlock()</code>	Allows Read Lock to be Tested
<code>pthread_rwlock_trywrlock()</code>	Allows Read-Write Lock to be Tested
<code>pthread_rwlock_unlock()</code>	Release a Read-Write Lock
<code>pthread_rwlock_wrlock()</code>	Wait for a Read-Write Lock
<code>pthread_rwlockattr_init()</code>	Initialize Default Attribute Object for a Read-Write Lock
<code>pthread_rwlockattr_destroy()</code>	Destroy Attribute Object for a Read-Write Lock
<code>pthread_rwlockattr_getpshared()</code>	Get Process-shared Attribute for a Read-Write Lock
<code>pthread_rwlockattr_setpshared()</code>	Set Process-shared Attribute for a Read-Write Lock

Creating a mutex

To use the mutex lock you must first create a mutex attribute object with the `pthread_mutexattr_init()` function. A mutex attribute object defines the modifiable characteristics that a mutex may have. Refer to the description of `pthread_mutexattr_init()` in [z/OS C/C++ Runtime Library Reference](#) for a list of these attributes and their defaults.

After the mutex attribute object has been created, you can use the following functions to change the default attributes.

- `pthread_mutexattr_getkind_np()`
- `pthread_mutexattr_setkind_np()`
- `pthread_mutexattr_gettype()`
- `pthread_mutexattr_settype()`
- `pthread_mutexattr_getpshared()`
- `pthread_mutexattr_setpshared()`

The mutex attribute object is used only when creating the mutex. It can be used to create other mutexes with the same attributes or modified to create mutexes with different attributes. You can delete a mutex attribute object with the `pthread_mutexattr_destroy()` function.

After the mutex attribute object has been created, the mutex can be created with the `pthread_mutex_init()` function.

While using mutexes as the locking device, the following functions can be used:

- `pthread_mutex_lock()`
- `pthread_mutex_unlock()`
- `pthread_mutex_trylock()`

To remove the mutex, use the `pthread_mutex_destroy()` function.

Note: Before freeing up the storage containing the `pthread_mutexattr_t` object, be sure to destroy it by calling `pthread_mutexattr_destroy()`. If the `pthread_mutexattr_t` object is not destroyed before the storage is reused, the results are undefined.

Creating a condition variable

Before creating a condition variable, you need to create a mutex (as shown above), then you must use the `pthread_condattr_init()` function to create a condition variable attribute object. This attribute object, like the mutex attribute object, defines the modifiable characteristics that a condition variable may have. Refer to the description of `pthread_condattr_init()` in [z/OS C/C++ Runtime Library Reference](#) for a list of these attributes and their defaults.

After the condition variable attribute object has been created, you may use the following functions to change the default attributes:

- `pthread_condattr_getkind_np()`
- `pthread_condattr_setkind_np()`
- `pthread_condattr_getpshared()`
- `pthread_condattr_setpshared()`

The condition variable attribute object is used only when creating the condition variable. It can be used to create other condition variables with the same attributes or modified to create condition variables with different attributes. You can delete a condition variable attribute object with the `pthread_condattr_destroy()` function.

After a condition variable attribute object has been created, the condition variable itself can be created with the `pthread_cond_init()` function.

Condition variables can then be used as a synchronization primitive using the following functions:

- `pthread_cond_wait()`
- `pthread_cond_timedwait()`
- `pthread_cond_signal()`
- `pthread_cond_broadcast()`

The condition variable can be removed with the `pthread_cond_destroy()` function.

Creating a read-write lock

To use a read-write lock you must first create a read-write attribute object with the `pthread_rwlockattr_init()` function. A read-write attribute object defines the modifiable characteristics that a read-write lock may have. Refer to the description of `pthread_rwlockattr_init()` in [z/OS C/C++ Runtime Library Reference](#) for a list of these attributes and their defaults.

After the read-write lock attribute object has been created, you can use the following functions to change the default attributes.

- `pthread_rwlockattr_getpshared()`
- `pthread_rwlockattr_setpshared()`

The read-write lock attribute object is used only when creating the read-write lock. It can be used to create other read-write locks with the same attributes or modified to create read-write locks with different attributes. You can delete a read-write attribute object with the `pthread_rwlockattr_destroy()` function.

After the read-write attribute has been created, the read-write lock can be created with the `pthread_rwlock_init()` function.

While using read-write locks as the locking device, the following functions can be used:

- `pthread_rwlock_rdlock()`
- `pthread_rwlock_tryrdlock()`
- `pthread_rwlock_wrlock()`
- `pthread_rwlock_trywrlock()`

- `pthread_rwlock_unlock()`

To remove the read-write lock, use the `pthread_rwlock_destroy()` function.

Thread-specific data

While all threads can access the same memory, it is sometimes desirable to have data that is (logically) local to a specific thread. The *key/value* mechanism provides for global (process-wide) keys with value bindings that are unique to a thread.

You can also use the `pthread_tag_np()` function to set and query 65 bytes of thread tag data associated with the caller's thread.

Model

The *key/value mechanism* associates a data key with each data item. When the association is made, the key identifies the data item with a particular thread. This data key is a transparent data object of type `pthread_key_t`. The contents of this key are not exposed to the user.

The user gets a key by issuing the `pthread_key_create()` function. One of the arguments on the `pthread_key_create()` function is a pointer to a local variable of type `pthread_key_t`. This variable is then used with the `pthread_setspecific()` function to establish a unique key value.

`pthread_key_create()` creates a unique identifier (a key) that is visible to all of the threads in a process. This data key is returned to the caller of `pthread_key_create()`. Threads can associate a thread unique data item with this key using the `pthread_setspecific()` call. A thread can get its unique data value for a key using the `pthread_getspecific()` call. In addition, a key can have an optional "destructor" routine associated with it. This routine is executed during thread termination and is passed the value of the key for the thread being terminated. A typical use of a key and destructor is to have storage obtained by a thread using `malloc()` and returned within the destructor at thread termination by using `free()`.

`pthread_key_delete()` deletes a thread-specific data key. Once a key has been deleted, it may not be passed to `pthread_getspecific()` or `pthread_setspecific()`. Any destructor function associated with the key when it was created will no longer be called. The application must perform any cleanup needed for values associated with the key.

Functions

Table 52 on page 252 lists functions that are used with thread-specific data.

Table 52. Functions used with thread-specific data	
Function	Purpose
<code>pthread_key_create()</code>	Create a thread-specific data key
<code>pthread_key_delete()</code>	Delete a thread-specific data key
<code>pthread_getspecific()</code>	Retrieve the value associated with a thread-specific key
<code>pthread_setspecific()</code>	Associate a value with a thread-specific key
<code>pthread_tag_np()</code>	Set and query the contents of the calling thread's tag data

Creating thread-specific data

Figure 76 on page 253 shows the example program CCNGTH1, which uses thread-specific data to insure that storage acquired by a specific thread is freed when the thread ends.

```

#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>
pthread_key_t mykey;          /* A place to get the key */
void mydestruct(void *value); /* My destructor routine */
main()
{
    char * thddataptr;
    /* Create a key, getting back the key from pthread_key_create(),
       and associate a function to be executed at thread termination
       for this key */

    (void)pthread_key_create(&mykey,&mydestruct);

    /* Obtain some storage which this thread will manage (remember,
       the main is also a thread), which we want freed by our
       destructor upon thread termination. Associate the storage
       pointer with the key using pthread_setspecific.
    */
    thddataptr = (char *) malloc(100);
    (void)pthread_setspecific(mykey,thddataptr);

    /* the body of the function

    /* now, the thread exits, causing the thread termination
       key data destructor to be executed.
    */
    pthread_exit((void *)0);
}
/* The key data destructor function
*/
void mydestruct(void * value) {
    /* value is the value in the key/value binding that is unique
       to the thread being terminated. Thus, in the example,
       it represents the pointer to the storage needing freed.
    */
    free(value);
}

```

Figure 76. Referring to thread-specific data

Signals

Each thread has an associated signal mask. The signal mask contains a flag for each signal defined by the system. The flag determines which signals are to be blocked from being delivered to a particular thread.

Unlike the signal mask, there is one signal action per signal for all of the threads in the process. Some signal functions work on the process level, having an impact on multiple threads, while others work on the thread level, and only affect one particular thread. For example, the function `kill()` operates at the process level, whereas the functions `pthread_kill()` and `sigwait()` operate at the thread level.

The following are some other signal functions that operate on the process level and can influence multiple threads:

- `alarm()`
- `bsd_signal()`
- `kill()`
- `killpg()`
- `raise()`
- `sigaction()`
- `siginterrupt()`
- `signal()`
- `sigset()`

Generating a signal

A signal can be generated explicitly with the `raise()`, `kill()`, `killpg()`, or `pthread_kill()` functions or implicitly with functions such as `alarm()` or by the system when certain events occur. In all cases, the signal will be directed to a specific thread running in a process.

The two primary functions for controlling signals are `sigaction()` and `sigprocmask()`. `sigaction()` also includes `bsd_signal()`, `signal()`, and `sigset()`.

sigaction()

`sigaction()` specifies the action when a signal is processed by the system. This function is process-scoped instead of thread-specific. When a signal is generated for a process, the state of each thread within that process determines which thread is affected. The three types of signal actions are:

catcher

Specifies the address of a function that will get control when the signal is delivered

SIG_DFL

Specifies that the system should perform default processing when this signal type is generated

SIG_IGN

Specifies that the system should ignore all signals of this type.

Attention: If a signal whose default action is to terminate is delivered to a thread running in a process where there are multiple threads running, and no signal catcher is designated for the signal, the entire process is terminated. You can avoid this by blocking each of the terminating signals, or by establishing a signal catcher for each of them.

In a multi-threaded application, when a signal is generated by a function or action that is not thread specific, and the process has some threads set up for signals and some threads that are not set up for signals, then the kernel's signal processing determines which thread has the most interest in the signal.

The following is a list of signal interest rules in their order of priority:

1. When threads are found in a `sigwait()` for this signal type, the signal is delivered to the first thread found in a `sigwait()`.
2. When all threads are blocking this signal type, the signal is left pending in the kernel at the process level. The `sigpending` function moves blocked pending signals at the process level to the thread level.
3. When all of the following are true:
 - One or more threads are set up for signals
 - All threads set up for signals have the signal blocked
 - A thread not set up for signals has not blocked the signal

The signal is left pending in the kernel on the first thread set up for signals. The signal remains pending on that thread until the thread unblocks the signal.

4. When the signal action is to catch, the signal is delivered to one of the threads that has the signal unblocked.

sigprocmask()

`sigprocmask()` specifies a way to control which set of signals interrupt a specific thread. Because `sigprocmask()` is thread-scoped, it blocks the signal for only the thread that issues the function.

Thread cancellation

When multiple threads are running in a process, thread cancellation permits one thread to cancel another thread in that process. This is done with the `pthread_cancel()` function, which causes the system to generate a cancel interrupt and direct it to the thread specified on the `pthread_cancel()`. Each thread can control how the system generates this cancel interrupt by altering the interrupt state and type.

A thread may have the following interrupt states, in descending order of control:

disabled

For short code sequences, the entire code sequence can be disabled to prevent cancel interrupts. The `pthread_setintr()` and `pthread_setcancelstate()` functions enable or disable cancel interrupts in this manner.

controlled

For larger code sequences where you want some control over the interrupts but cannot be entirely disabled, set the interrupt type to controlled/deferred and the interrupt state to enabled. The `pthread_setintrtype()` and `pthread_setcancelttype()` functions allow for this type of managed interrupt delivery by introducing the concept of cancellation points.

Cancellation points consist of calls to a limited set of library functions, documented below.

The user program can implicitly or explicitly solicit interrupts by invoking one of the library functions in the set of cancellation points, thus allowing the user to control the points within their application where a cancel may occur.

asynchronous

For code sequences where you do not need any control over the interrupt, set `pthread_setintr()/pthread_setcancelstate()` to enable and `pthread_setintrtype()/pthread_setcancelttype()` to asynchronous. This will allow cancel interrupts to occur at any point within your program.

For example, if you have a critical code section (a sequence of code that needs to complete), you would turn cancel off or prevent the sequence from being interrupted. If the code is relatively long, consider running using the `control` interrupt and as long as the critical code section doesn't contain any of the functions that are considered cancellation points, it will not be unexpectedly canceled.

For C++, destructors for automatic objects on the stack are run when a thread is cancelled. The stack is unwound and the destructors are run in reverse order.

Cancellation Points

The library functions listed in [Table 53 on page 255](#), and any of their callers, will introduce cancellation points into a thread's execution.

Table 53. Cancellation point summary

accept()	<code>aio_suspend()</code>	<code>close()</code>	<code>connect()</code>
creat()	<code>fcntl()</code>	<code>fsync()</code>	<code>getmsg()</code>
getpmsg()	<code>lockf()</code>	<code>msync()</code>	<code>open()</code>
pause()	<code>poll()</code>	<code>pread()</code>	<code>putpmsg()</code>
pwrite()	<code>read()</code>	<code>readv()</code>	<code>recv()</code>
recvfrom()	<code>recvmsg()</code>	<code>select()</code>	<code>send()</code>
sendmsg()	<code>sendto()</code>	<code>sigpause()</code>	<code>sigsuspend()</code>
sigtimedwait()	<code>sigwait()</code>	<code>tcdrain()</code>	<code>usleep()</code>
wait()	<code>waitid()</code>	<code>waitpid()</code>	<code>write()</code>
writv()			

Functions

[Table 54 on page 256](#) lists functions that are used to control the cancellability of a thread.

Table 54. Functions used to control cancellability

Function	Purpose
<code>pthread_cancel()</code>	Cancel a thread
<code>pthread_setintr()</code>	Set thread cancellability state
<code>pthread_setintrtype()</code>	Set thread cancellability type
<code>pthread_testintr()</code>	Establish a cancellability point
<code>pthread_setcancelstate()</code>	Set thread cancellability state
<code>pthread_setcanceltype()</code>	Set thread cancellability type
<code>pthread_testcancel()</code>	Establish a cancellability point

Cancelling a thread

Three possible scenarios may cancel a thread, one for each of the interrupt states of the thread being canceled.

- One thread issues `pthread_cancel()` to another thread whose cancellability state is enabled and controlled. In this case the thread being canceled continues to run until it reaches an appropriate cancellation point. When the thread is eventually cancelled, just prior to termination of the thread, any cleanup handlers which have been pushed and not yet popped will be executed. Then if the thread has any thread-specific data, the destructor functions associated with this data will be executed.
- One thread issues `pthread_cancel()` to another thread whose interruption state is enabled and asynchronous. In this case the thread being canceled is terminated immediately, after any cleanup handlers and thread-specific data destructor functions are executed, as in the first scenario.
- One thread issues `pthread_cancel()` to another thread whose interruption state is disabled. In this case the cancel request is ignored and the thread being canceled continues to run normally.

In the first two interrupt states, the caller of `pthread_cancel()` may get control back before the thread is actually canceled.

Cleanup for threads

Cleanup handlers are routines written by the user that include any special processing the user finds necessary for termination of a thread. As the user's routine executes, it pushes cleanup handlers on to a stack. As the thread continues to run and the routine progresses, these cleanup handlers can be taken off of the stack by the user's routine.

A list or stack of cleanup handlers is maintained for each thread. When the thread ends, all pushed but not yet popped cleanup routines are popped from the cleanup stack and executed in last-in-first-out (LIFO) order. This occurs when the thread:

- Calls `pthread_exit()`
- Does a return from or reaches the end of the start routine (that gets controls as a result of a `pthread_create()`)
- Is canceled because of a `pthread_cancel()`.

The first thread in a process to call `pthread_create()` becomes the initial pthread-creating task (IPT). When exiting back to the operating system from the IPT, the caller may receive an A03 abend if any `pthread_created` tasks are still running. These tasks may still be running even if the IPT has called `pthread_join()` for all the threads that it created. To avoid the A03 abend, the IPT should call `_exit()` when it is ready to return to the operating system. `_exit()` ends the IPT and all of its `pthread_created` subtasks without causing an A03 abend to occur.

Functions

Table 55 on page 257 lists functions that are used for thread clean up.

Table 55. Functions used for cleanup purposes

Function	Purpose
<code>pthread_cleanup_push()</code>	Establish a cleanup handler
<code>pthread_cleanup_pop()</code>	Remove a cleanup handler

Thread stack attributes

Three attributes allow POSIX applications to control their threads' stack usage: `stackaddr`, `stacksize`, and `guardsize`.

The `stackaddr` attribute contains the address of application-provided storage to be used as the initial stack segment. This storage is referred to as an "application-managed stack".

The `stacksize` attribute controls the size of the stack. In an SUSv3 application, calls to `pthread_attr_setstacksize()` or `pthread_attr_setstack()` must provide a `stacksize` of at least `PTHREAD_STACK_MIN`. If not provided, the initial increment size is derived from the `STACK64/THREADSTACK64` runtime options in `AMODE 64`, or `STACK/THREADSTACK` otherwise.

When used in conjunction with application-managed stack, the size of the storage must be a multiple 4K on ILP32 (`AMODE31`) and 1M on L64 (`AMODE64`), and at least `PTHREAD_STACK_MIN` in length.

For portability, `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` may modify the `guardsize` in the thread attribute object, but the `guardsize` attribute is not honored on z/OS for system-managed stack. In the case of application-managed stack, guarding is an application responsibility.

The `guardsize` attribute is not honored on z/OS.

When a thread is created using an attribute object with a `stackaddr` attribute set, behavior is undefined if the application ever accesses the same storage again, except through normal stack access in that thread. An attribute object with `stackaddr` attribute set may not be used more than once, unless it is destroyed and re-initialized, or its `stackaddr` attribute changed.

Behaviors and restrictions in z/OS UNIX applications

The following are implementation-specified behaviors and restrictions that apply to the XL C/C++ library functions when running a multi-threaded z/OS UNIX application.

Using threads with MVS files

MVS files that are opened by data-set names or ddnames are thread-specific in the following ways:

MVS files opened in update mode where repositioning functions are used after the files have been extended are also restricted to the owning thread. This is because a reposition might need to reopen the read DCB in order to be able to see the new EOF marker. The runtime library does not enforce this restriction.

Multivolume data sets, files that are part of a concatenated ddname, and hiperspace memory files are further restricted in multithreaded applications. All I/O operations are restricted to the thread on which the file is opened.

When standard streams are directed to MVS files, they are governed by the previous restrictions. Standard streams are directed to MVS files in one of two ways:

- By default when a `main()` program is run from the TSO ready prompt or by a `JCL EXEC PGM=` statement, that is, whenever it is not initiated by the `exec()` function. This is regardless of whether you are running with `POSIX(ON)` or `POSIX(OFF)`. In these cases, the owning thread is the initial processing thread (IPT), the thread on which `main()` is executed.
- By explicit action when the user redirects the streams by using command line redirection, `fopen()`, or `freopen()`. The thread that is redirected (the IPT, if you are using command line redirection) becomes

the owning thread of the particular standard stream. The usual MVS file thread affinity restrictions outlined above apply until the end of program or until the stream is redirected to the UNIX file system.

Any operation that violates these restrictions causes SIGIOERR to be raised and `errno` to be set with the following associated message:

```
EDC5024I: An attempt was made to close a file that had been
opened on another thread.
```

All MVS files opened from a given thread and still open when the thread is terminated are closed automatically by the library during thread termination.

Having more than one writer use separate file pointers to a single data set or ddname is prohibited as always, regardless of whether the file pointers are used from multiple threads or a single thread.

Note: These restrictions specifically do not apply to UNIX file system. All opens and closes by the C library that result in calls to an underlying access method for a given MVS file must occur on the same thread. Therefore, the following specific functions are prohibited from any thread except the owning thread (the one that does the initial `fopen()` of the file:

- `fclose()`
- `freopen()`
- `rewind()`

Multithreaded I/O

The `getc()`, `getchar()`, `putc()`, and `putchar()` functions have two versions, one that is defined in the header file, `stdio.h`, which is a macro and the other which is an actual library routine. The macros have better performance than their respective function versions, but these macros are not thread safe, so in a multithreaded application where `_OPEN_THREADS` feature test macro is defined, the macro version of these functions are not exposed. Instead, the library functions are used. This is done to ensure thread safety while multiple threads are executing.

The `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` functions and macros are functionally equivalent to the `getc()`, `getchar()`, `putc()`, and `putchar()` functions and macros. These functions and macros can safely be used in a multi-thread environment if and only if called from a thread that owns the `FILE*` object, such as after a successful call to `flockfile()` or `ftrylockfile()`.

Use of the `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, or `putchar_unlocked()` functions can have unpredictable behavior when used on a thread that has not locked the file.

It is the application's responsibility to prevent deadlocks or looping. For example, deadlock or looping may occur if a `FILE*` object is closed, or a thread is terminated, before relinquishing all locked `FILE*` objects

Thread-scoped functions

Thread-scoped functions are functions that execute independently on each thread without sharing intermediate state information across threads. For example, `strtok()` preserves pointers to tokens independently on each thread, regardless of the fact that multiple threads may be examining the same string in a `strtok()` operation. Some examples of thread-scoped functions are:

- `strtok()`
- `rand()`, `srand()`
- `mblen()`, `mbtowc()`
- `strerror()`
- `asctime()`, `ctime()`, `gmtime()`, `localtime()`
- `clock()`

The following are examples of process-scoped functions, which means that a call to these functions on one thread influences the results of calls to the same function on another thread. For example, `tmpnam()` is required to return a unique name for every invocation during the life of the process, regardless of which thread issues the call.

- `tmpnam()`
- `getenv()`
- `setenv()`
- `clearenv()`
- `putenv()`

Unsafe thread functions

The following functions are not thread-safe. In a multithreaded application, therefore, they should only be used before the first invocation of `pthread_create()`.

- `setlocale()` - (returns NULL if issued after `pthread_create()`)
- `tzset()`

Fetches functions and writable statics

Fetches functions are recorded globally at the process level. Therefore, a function fetched from one thread can be executed from any thread.

Module boundary crossings are thread-scoped. Writable statics have a scope between process and thread. They are process-scoped except that module crossings are thread-scoped. This means that:

- All threads initially inherit the writable statics of the creating thread at the time of the creation.
- When any thread executes a function pointer supplied by the `fetch()` function and crosses a module boundary, only that thread has access to the writable statics of the fetched module.

MTF and z/OS UNIX threading

MTF is not supported from applications running under POSIX(ON). A return value of EWRONGOS is issued when running in a POSIX(ON) environment. An application that requires multithreading must either use MTF with POSIX(OFF) or `pthread_create()` with POSIX(ON).

Thread queuing function

The thread queuing function allows you to control whether or not threads should be queued up while waiting for TCBs to become available. You can accomplish this by switching the `synctype` attribute of a thread between synchronous and asynchronous mode. With synchronous mode for example, if a process can only have 50 TCBs active at any one time, then only 50 threads can be created. The 51st thread create results in an error. With asynchronous mode, however, you can set the `synctype` attribute for a thread such that the 51st thread is created. This thread will not start until one of the other threads finishes and releases a TCB.

Functions that relate to the ability to control thread queuing are:

- `pthread_set_limit_np()`
- `pthread_attr_getsynctype_np()`
- `pthread_attr_setsynctype_np()`

Thread scheduling

You can use the `pthread_attr_setweight_np()` and `pthread_attr_setsynctype_np()` functions to establish priorities for threads. The `pthread_attr_setweight_np()` *threadweight* variable can be set to the following:

__MEDIUM_WEIGHT

Each thread runs on a task. When the current thread exits, the task waits for another thread to do a `pthread_create()`. The new thread runs on that task.

__HEAVY_WEIGHT

The task is attached on `pthread_create()` and terminates when the thread exits. When the thread exits, the associated task can no longer request threads to process, and full MVS EOT resource manager cleanup occurs.

You can use the `pthread_addt_setsynctype_np()` function to set the `__PTATASYNCHRONOUS` value. This enables you to create more threads than there are TCBs available. For example, you could run 50 TCBs and create hundreds of threads. The kernel queues the threads until a task is available. This frees your application from managing the work. While a thread is queued and not executing on an MVS task, you can still interact with the thread via `pthread` functions, such as `pthread_join()` and `pthread_kill()`.

iconv() family of functions

The conversion descriptor returned from a successful `iconv_open()` may be used safely within a single thread for conversion purposes. It may, however, be opened on one thread (`iconv_open()`), closed on another thread (`iconv_close()`), and used on a third thread (`iconv()`). However, it is the user's responsibility to ensure operations are synchronized if they are used across multiple threads.

Note: The `iconv_open()` function tolerates converter names without a dash in the name for all converter names containing dashes. For example, `iconv_open()` tolerates the name `IBM1047` for the `IBM-1047` converter.

Threads and recoverable resources

When a Language Environment application terminates with an ABEND, outstanding updates to recoverable resources are rolled back for all contexts that are associated with the initial `pthread`-creating task (IPT) and any `pthread_created` tasks that are active.

When a Language Environment application terminates normally or with a nonzero return code, outstanding updates to recoverable resources are committed for all contexts that are associated with the initial `pthread`-creating task (IPT) and any `pthread_created` tasks that are active.

MEMLIMIT for 64-bit multithreaded applications

In the preinitialized AMODE64 environment, every thread requires at least 2049 KB of virtual storage for its stack storage and control blocks. Otherwise, every thread that runs in AMODE64 requires at least 3 MB of virtual storage for its stack storage and control blocks. For applications with large number of threads, you need to set `MEMLIMIT` to a large enough value for thread stack virtual storage.

Chapter 22. Reentrancy in z/OS XL C/C++

This information describes the concept of reentrancy. It tells you how to use reentrancy in C programs to help make your programs more efficient, and how C++ achieves constructed reentrancy.

Reentrant programs are structured to allow multiple users to share a single copy of an executable module or to use an executable module repeatedly without reloading. C and C++ achieve reentrancy by splitting your program into two parts, which are maintained in separate areas of memory until the program terminates:

- The first part, which consists of executable code and constant data, does not change during program execution.
- The second part contains persistent data that can be altered. This part includes the dynamic storage area (DSA) and a piece of storage known as the writable static area.

For XPLINK, the writable static area is further logically subdivided into areas called environments. Environments are optional, and each function can have its own environment. When an XPLINK function is called, the caller must load general purpose register 5 with the address of the environment of the called function before control is given to the entry point of the called function.

If the program is installed in the Link Pack Area (LPA) or Extended Link Pack Area (ELPA) of your operating system, only a single copy of the first (constant or reentrant) part exists within a single address space. This occurs regardless of the number of users that are running the program simultaneously. This reentrant part may be shared across address spaces or across sessions. In this case, the executable module is loaded only once. Separate concurrent invocations of the program share or reenter the same copy of the write-protected executable module. If the program is not installed in the LPA or ELPA area, each invocation receives a private copy of the code part, but this copy may not be write-protected.

The modifiable writable static part of the program contains:

- All program variables with the `static` storage class
- All program variables receiving the `extern` storage class
- All writable strings
- All function linkage descriptors for all referenced DLL functions
- Function linkage descriptors for all referenced DLL functions that are used by multiple compilation units in the program, but are not imported (XPLINK, RENT)
- All variable pointers for imported variables (non-XPLINK)
- All function pointers for imported functions (XPLINK, RENT)
- All variable linkage descriptors to reference imported variables (non-XPLINK)

Each user running the program receives a private copy of the second (data or non-reentrant) part. This part, the data area, is modifiable by each user.

The code part of the program contains:

- Executable instructions
- Read-only constants
- Global objects compiled with the `#pragma variable(identifier, NORENT)`

Note: The ROCONST compiler option implicitly inserts a `#pragma variable(identifier, NORENT)` for `const` qualified variables.

Natural or constructed reentrancy

Natural reentrancy

C programs that contain no references to the writable static objects listed in a previous section have natural reentrancy. You do not need to compile naturally reentrant C programs with the RENT compiler option or bind them with the binder.

Constructed reentrancy

C++ programs, and C programs that contain references to writable static objects, can have constructed reentrancy. You must bind these programs with the binder. For C programs, you must use the RENT compiler option.

If you use the XPLINK option, RENT is the default. If you override this default by specifying NORENT, any parts of the program that are normally stored in the writable static area go instead into a static area. If this static area is write-protected, you will get a runtime failure because the function pointers for imported functions cannot be modified to point to the function when the DLL containing the function is loaded and the function address determined. For programs that are both XPLINK and NORENT, all functions must be statically bound or explicitly loaded (`dllload()`, or `fetch()`).

Limitations of constructed reentrancy for C programs

Even if a C program is large and will have more than one user at the same time, there are also these limitations to consider:

- The binder is required for code that you compile with XPLINK.
- If the prelinker, rather than the binder, will process code that is compiled with NOXPLINK, RENT:
 - The resultant load module referring to the writable area cannot be reprocessed.
 - The resultant program may reside in a PDS.
- If the binder is used, and not the prelinker, the resultant program must reside in a PDSE or UNIX file system. If a PDSE member should be installed into LPA or ELPA, it can only be installed into dynamic LPA.
- A system programmer can install only the shared portion of your program in the LPA or ELPA of your operating system.

Controlling external static in C programs

Certain program variables with the `extern` storage class may be constant and never written. If this is the case, every user does not need to have a separate copy of these variables. In addition, there may be a need to share constant program variables between C and another language.

You can force an external variable to be the part of the program that includes executable code and constant data by using the `#pragma variable(varname, NORENT)` directive. The program fragment in [Figure 77 on page 262](#) illustrates how this is accomplished.

```
#pragma options(RENT)

#pragma variable(rates, NORENT)
extern float rates[5] = { 3.2, 83.3, 13.4, 3.6, 5.0 };

extern float totals[5];

int main(void) {
    /* ... */
}
```

Figure 77. Controlling external static

In this example, the source file is compiled with the RENT option. The external variable `rates` are included in the executable code because `#pragma variable(rates, NORENT)` is specified. The variable `totals`

are included with the writable static. Each user has a copy of the array totals, and the array rates are shared among all users of the program.

The `#pragma variable(varname, NORENT)` does not apply to, and has no effect on, program variables with the static storage class. Program variables with the static storage class are always included in the writable static. An informational message will appear if you do try to write to a non-reentrant variable when you specify the CHECKOUT compiler option.

When specifying `#pragma variable(varname, NORENT)`, ensure that this variable is never written; if it is written, program exceptions or unpredictable program behavior may result. In addition, you must include `#pragma variable(varname, NORENT)` in every source file where the variable is referenced or defined. It is good practice to put these pragmas in a common header file.

Note: You can also use the keyword `const` to ensure that a variable is not written. See the `const` type qualifier in [z/OS XL C/C++ Language Reference](#) for more information.

The ROCONST compiler option has the same effect as specifying the `#pragma variable (var_name, NORENT)` for all constant variables (i.e. `const` qualified variables). The option gives the compiler the choice of allocating `const` variables outside of the Writable Static Area (WSA). For more information, see [ROCONST | NOROCONST](#) in [z/OS XL C/C++ User's Guide](#).

Controlling writable strings

In a large number of C programs, character strings may be constant and never written to. If this is the case, every user does not need a separate copy of these strings.

You can force all strings in a given source file to be the part of the program that includes executable code and constant data by using `#pragma strings(readonly)` or the ROSTRING compiler option. [Figure 78 on page 263](#) illustrates one way to make the strings constant.

[Figure 78 on page 263](#) shows a sample program (CCNGRE1) that makes strings constant. The string "hello world\n" is included with the executable code because `#pragma strings(readonly)` is specified. This can yield a performance and storage benefit.

```
/* this example demonstrates how to make strings constant */
#pragma strings(readonly)
#include <stdio.h>

int main(void)
{
    printf("hello world\n");

    return(0);
}
```

Figure 78. Making strings constant

Ensure that you do not write to read-only strings. The following code tries to overwrite the literal string "abcd" because 'chrs' is just a pointer:

```
char chrs[] = "abcd";
memcpy(chrs, "ABCD", 4);
```

Program exceptions or unpredictable program behavior may result if you attempt to write to a string constant.

The ROSTRING compiler option has the same effect as `#pragma strings(readonly)` in the program source. For more information, see [ROSTRING | NOROSTRING](#) in [z/OS XL C/C++ User's Guide](#).

Controlling the memory area in C++

In C++, some objects may be constant and never modified. If your program is reentrant, having such objects exist in the code part is a storage and performance benefit.

As a programmer, you control where objects with global names and string literals exist. You can use the `#pragma variable(objname, NORENT)` directive to specify that the memory for an object with a global name is to be in the code area. You can use the `ROCONST` compiler option to specify that all `const` variables go into the code area.

In Figure 79 on page 264, the variable `RATES` exists in the executable code area because `#pragma variable(RATES, NORENT)` has been specified. The variable `totals` exists in writable static area. All users have their own copies of the array `totals`, but the array `RATES` is shared among all users of the program.

```
/*-----*/
/* RATES is constant and in code area */
#pragma variable(RATES, NORENT)
const float RATES[5] = { 1.0, 1.5, 2.25, 3.375, 5.0625 };
float totals[5];
/*-----*/
```

Figure 79. Example of controlling the memory area

When you specify `#pragma variable(objname, NORENT)` for an object, and the program is to be reentrant, you must ensure that this object is never modified, even by constructors or destructors. Program exceptions or unpredictable behavior may result. Also, you must include `#pragma variable(objname, NORENT)` in every source file where the object is referenced or defined. Otherwise, the compiler will generate inconsistent addressing for the object, sometimes in the code area and sometimes in the writable static area.

Controlling where string literals exist in C++ code

In z/OS XL C/C++, the string literals exist in the code part by default, and are not modifiable if the code is reentrant. In a large number of programs, string literals may be constant. In this case, every user does not need a separate copy of these strings.

By using the `#pragma strings(writable)` directive, you can ensure that the string literals for that compilation unit will exist in the writable static area and be modifiable. Figure 80 on page 264, which shows sample program `CCNGRE2`, illustrates how to make the string literals modifiable.

```
/* this example demonstrates how to make string literals modifiable */
#pragma strings(writable)
#include <iostream>
using namespace std;

int main(void)
{
    char * s;
    s = "wall\n";          // point to string literal
    *(s+3) = 'k';          // modify string literal
    cout << s;             // output "walk\n"
}
```

Figure 80. How to Make String Literals Modifiable

In this example, the string `"wall\n"` will exist in the writable static area because `#pragma strings(writable)` is specified. This modifies the fourth character.

Using writable static in Assembler code

Programming in C or C++ can eliminate most of the need to code in assembler. However, in cases where you must code in assembler, you may have a need to modify data in the writable static area of a C or C++ program, from within an assembler program.

Notes:

1. To call assembler from C++, you must use extern "OS" as documented in [Chapter 17, "Using linkage specifications in C or C++,"](#) on page 175.
2. The following macros, and access to writable static data from assembler are not supported for XPLINK programs.
 - EDCDXD
 - EDCLA
 - EDCDPLNK

One way to modify data in the writable static area is to pass the address of the writable static data item as a parameter to the assembler program. This may be difficult in some cases. The following assembler macros makes this easier:

- EDCDXD
- EDCLA
- EDCDPLNK

These are in CEE.SCEEMAC (EDCDXD, EDCLA, EDCDPLNK). The restriction on the names of writable static objects accessible in assembler code is that they are S-names. This means that they may be at most 8 characters long and may contain only characters allowed in external names by the assembler code.

The macro EDCDXD declares a writable static data item. EDCLA loads the address of the writable static data item into a register. Using the EDCLA macro in assembler code necessitates coding EDCDXD as well.

The EDCDPLNK macro defines reference writable static data with the z/OS binder. This macro must appear before the first executable control section is initiated in the assembler source module. If there is more than one assembler source program in the input file, EDCDPLNK must precede every assembler source program in any input file that defines or references writable static data. [Figure 81 on page 265](#) illustrates their use.

```
*****
* this example shows how to reference objects in the writable *
* static area, from assembler code                          *
* part 1 of 2(other file is CCNGRE4)                        *
*                                                           *
* parameters: none                                         *
* return: none                                           *
* action: store contents of register 13 ( callers dynamic *
* storage area) in variable DSA which exists in          *
* the writable static area                               *
*                                                           *
* Macros: EDCPRLG, EDCEPIL, EDCDXD, EDCLA in CEE.SCEEMAC *
*****
XOBJHDR EDCDPLNK ;generate an XOBJ header
GETDSA CSECT
GETDSA AMODE ANY
GETDSA RMODE ANY
EDCPRLG ;prolog (save registers etc.)
EDCLA 1, DSA ;load register 1 with address of DSA
ST 13, 0(, 1) ;store contents of reg 13 in DSA
EDCEPIL ;epilog (restore registers etc.)
DSA EDCDXD 0F ;declaration of DSA in writable static
TBLDSA EDCDXD 20F ;definition of TBLDSA in writable static
END
```

Figure 81. Referencing objects in the writable static area, Part 1

In this example, the external variable TBLDSA is declared using the EDCDXD macro. The size value of 0F (zero fullwords) indicates that DSA will be treated as an extern declaration in C or C++. Because TBLDSA is an extern declaration and not a definition, DSA must be defined in another C, C++, or assembler program. The EDCLA macro loads the general purpose register 1 with the address of DSA, which exists in the writable static area.

The external variable TBLDSA is declared using the EDCDXD macro. It is defined because its size is 20F (20 fullwords or 80 bytes) and corresponds to an external data definition in C or C++. When the program

starts, TBDLSA is initialized to zero. Because TBDLSA is an external data definition, there should not be another definition of it in a C++, C, or assembler program.

When these macros are used, these pseudo-registers cannot be used within the same assembler program.

There are no assembler macros for static initialization of a variable with a nonzero value. You can do this by defining and initializing the variable in C or C++ and making an extern declaration for it in the assembler program. In the example assembler program, DSA is declared this way.

Figure 82 on page 266 shows sample program CCNGRE4, which illustrates how to call the assembler program in Figure 81 on page 265.

```
/* this example shows how to reference objects in the writable */
/* static area, from assembler code */
/* part 2 of 2 (other file is CCNGRE3) */

#include <stdio.h>

#ifdef __cplusplus
    extern "C" {
#endif
void GETDSA(void);          /* assembler routine modifies DSA */
#ifdef __cplusplus
    }
#endif

const int sz = 20;          /* maximum call depth */
extern void * TBLDSA[sz];   /* defined in assembler program */
void * DSA;                /* define it here, source name */
                           /* same as assembler name */

/* call yourself deeper and deeper */
/* save DSA pointers as you go */
void deeper( int i)
{
    if (i >= sz)            /* if deep enough just return */
        return;
    GETDSA();               /* assign value to DSA */
    TBLDSA[i] = DSA;        /* save value in table */
    deeper(i+1);            /* go deeper in call chain */
}

int main(void) {
    int i;
    deeper(0);
    for(i=0; i<sz; i++)
        printf("depth %3d, DSA was at %p\n", i, TBLDSA[i]);
    return 0;
}
```

Figure 82. Referencing objects in the writable static area, Part 2

Chapter 23. IEEE Floating-Point

Starting with OS/390 V2R6 (including the Language Environment and C/C++ components), support was added for IEEE binary floating-point (IEEE floating-point) as defined by the ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic. For more information on floating-point support, see:

- [*z/Architecture Principles of Operation*](#)
- [*z/OS XL C/C++ User's Guide*](#)
- [*z/OS XL C/C++ Language Reference*](#)
- [*z/OS C/C++ Runtime Library Reference*](#)
- [*z/OS Language Environment Vendor Interfaces*](#)

Starting with z/OS V1R9 (including the Language Environment and C/C++ components), support was added for IEEE decimal floating-point, as defined by the ANSI/IEEE Standard P754/D0.15.3, IEEE Standard for Floating-Point Arithmetic.

Floating-point numbers

The format of floating-point numbers can be either base 16 S/390® hexadecimal format, base 2 IEEE-754 binary format, or base 10 IEEE-754 decimal format. The formats are based on three operand lengths for hexadecimal and binary: short (32 bits), long (64 bits), and extended (128 bits). The formats are also based on three operand lengths for decimal: `_Decimal32` (32 bits), `_Decimal64` (64 bits), and `_Decimal128` (128 bits).

A floating-point operand may be numeric or, for binary and decimal floating-point only, positive or negative infinity, or nonnumeric (Not a Number, or NaN). A floating-point number, has three components: a sign bit, a signed binary exponent, and a significand. The significand consists of an implicit unit digit to the left of an implied radix point, and an explicit fraction field to the right. The significand digits are based on the radix, 2 (for binary floating-point), 10 (for decimal floating-point), or 16 (for hexadecimal floating-point). The magnitude (an unsigned value) of the number is the product of the significand and the radix raised to the power of the exponent. The number is positive or negative depending on whether the sign bit is zero or one, respectively. A nonnumeric binary or decimal floating-point operand also has a sign bit, signed exponent, and fraction field.

Hexadecimal floating-point operands have formats that provide for exponents that specify powers of the radix 16 and significands that are hexadecimal numbers. The exponent range is the same for the short, long, and extended formats. The results of most operations on hexadecimal floating-point data are truncated to fit into the target format, but there are instructions available to round the result when converting to a narrower format. For hexadecimal floating-point operands, the implicit unit digit of the significand is always zero. Because the value if the significand and fraction are the same, hexadecimal floating-point operations are described in terms of the fraction, and the term significand is not used.

Binary floating-point operands have formats which provide for exponents that specify powers of the radix 2 and significands that are binary numbers. The exponent range differs for different formats, the range being greater for the longer formats. In the long and extended formats, the exponent range is significantly greater for binary floating-point data than for hexadecimal floating-point data. The results of operations performed on binary floating-point data are rounded automatically to fit into the target format; the manner of rounding is determined by a program-settable rounding mode.

Decimal floating-point operands have formats that provide for exponents that specify powers of the radix 10 and significands that are decimal numbers. The exponent range differs for different formats, the range being greater for the longer formats. The results of operations performed on decimal floating-point data are rounded automatically to fit into the target format; the manner of rounding is determined by a program-settable rounding mode.

C/C++ compiler support

The 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 binary floating-point or hexadecimal floating-point format. For details on the z/OS XL C/C++ support, see the description of the `FLOAT` option in *z/OS XL C/C++ User's Guide*. In addition, two related sub-options, `ARCH(3)` and `TUNE(3)`, support IEEE binary floating-point data. Refer to the [ARCHITECTURE](#) and [TUNE](#) compiler options in *z/OS XL C/C++ User's Guide* for details.

The *z/OS XL C/C++ Language Reference* contains additional information on floating-point in the following sections:

- [Floating-point literals](#)
- [Floating-point types](#)
- [Floating-point conversions](#)

Notes:

1. You must have OS/390 Release 6 or higher to use the IEEE binary floating-point instructions. In Release 6, the base control program (BCP) is enhanced to support the new IEEE binary floating-point hardware in the IBM S/390 Generation 5 Server. This enables programs running on OS/390 Release 6 to use the IEEE binary floating-point instructions and 16 floating-point registers. In addition, the BCP provides simulation support for all the new floating-point hardware instructions. This enables applications that make light use of IEEE binary floating-point, and can tolerate the overhead of software simulation, to execute on OS/390 V2R6 without requiring an IBM S/390 Generation 5 Server.
2. The terms binary floating-point and IEEE binary floating-point are used interchangeably. The abbreviations BFP and HFP, which are used in some function names, refer to binary floating-point and hexadecimal floating-point respectively.
3. Under hexadecimal floating-point format, the rounding mode is set to round toward 0. Under IEEE binary floating-point format, the rounding mode is set to round toward the nearest integer.

Using IEEE floating-point

IEEE binary floating-point is provided primarily to enhance interoperability and portability between IBM Z[®] and other platforms. It is anticipated that IEEE binary floating-point will be most commonly used for new and ported applications. Customers should not migrate existing applications that use hexadecimal floating-point to IEEE binary floating-point, unless there is a specific reason.

IBM does not suggest mixing Binary and Hexadecimal floating-point formats in an application. However, for applications which must handle both formats, the C/C++ runtime library does offer some support. Reference information for IEEE binary floating-point can also be found in *z/OS XL C/C++ Language Reference*.

You should use IEEE binary floating-point in the following situations:

- You deal with data that are already in IEEE binary floating-point format
- You need the increased exponent range (see *z/OS XL C/C++ Language Reference* for information on exponent ranges with IEEE-754 floating-point)
- You want the changes in programming paradigm provided by infinities and NaN (Not a Number)

For more information about the IEEE format, refer to the IEEE 754-1985 IEEE Standard for Binary Floating-Point Arithmetic.

When you use IEEE binary floating-point, make sure that you are in the same rounding mode at compile time (specified by the `ROUND(mode)` option), as at run time. Entire compilation units will be compiled with the same rounding mode throughout the compilation. If you switch runtime rounding modes inside a function, your results may vary depending upon the optimization level used and other characteristics of your code; switch rounding mode inside functions with caution.

If you have existing data in hexadecimal floating-point (the original base 16 S/390 hexadecimal floating-point format), and have no need to communicate these data to platforms that do not support this format, there is no reason for you to change to IEEE binary floating-point format.

Applications that mix the two formats are not supported.

IEEE binary and decimal floating-point are fully supported in a CICS environment only if CICS TS Version 4 or later is in use.

For information on the C/C++ functions that support floating-point, see the functions listed in [Table 56](#) on page 269. For more information, see [z/OS C/C++ Runtime Library Reference](#).

Table 56. C/C++ functions that support floating-point

Functions			
absf()	absl()	acos()	acosd32()
acosd64()	acosd128()	acosf()	acosh()
acoshd32()	acoshd64()	acoshd128()	acoshf()
acoshl()	acosl()	asin()	asind32()
asind64()	asind128()	asinf()	asinh()
asinhd32()	asinhd64()	asinhd128()	asinhf()
asinhl()	asinl()	atan()	atand32()
atand64()	atand128()	atanf()	atanh()
atanhd32()	atanhd64()	atanhd128()	atanhf()
atanhl()	atanl()	__atanpid32()	__atanpid64()
__atanpid128()	atan2()	atan2d32()	atan2d64()
atan2d128()	atan2f()	atan2l()	cabs()
cabsf()	cabsl()	cacos()	cacosf()
cacosh()	cacoshf()	cacoshl()	cacosl()
carg()	cargf()	cargl()	casin()
casinf()	casinh()	casinhf()	casinhf()
casinl()	catan()	catanf()	catanh()
catanhf()	catanhl()	catanl()	cbrt()
cbrtd32()	cbrtd64()	cbrtd128()	cbrtf()
cbrtl()	ccos()	ccosf()	ccosh()
ccoshf()	ccoshl()	ccosl()	ceil()
ceild32()	ceild64()	ceild128()	ceilf()
ceill()	cexp()	cexpf()	cexpl()
cimag()	cimagf()	cimagl()	clog()
clogf()	clogl()	conj()	conjf()
conjl()	copysign()	copysignd32()	copysignd64()
copysignd128()	copysignf()	copysignl()	cos()
cosd32()	cosd64()	cosd128()	cosf()
cosh()	coshd32()	coshd64()	coshd128()

Table 56. C/C++ functions that support floating-point (continued)

Functions

coshf()	coshl()	cosl()	__cospid32()
__cospid64()	__cospid128()	cotan()	cotanhf()
cotanf()	cpow()	cpowf()	cpowl()
cproj()	cprojf()	cprojl()	creal()
crealf()	creall()	csin()	csinf()
csinh()	csinhf()	csinhl()	csinl()
csqrt()	csqrtf()	csqrtl()	ctan()
ctanf()	ctanh()	ctanhf()	ctanhl()
ctanl()	erf()	erfc()	erfcd32()
erfcd64()	erfcd128()	erfcf()	erfcl()
erfd32()	erfd64()	erfd128()	erff()
erfl()	exp()	expf()	expl()
expm1()	expm1d32()	expm1d64()	expm1d128()
expm1f()	expm1l()	exp2()	exp2d32()
exp2d64()	exp2d128()	exp2f()	exp2l()
expd32()	expd64()	expd128()	fabs()
fabsd32()	fabsd64()	fabsd128()	fabsf()
fabsl()	fdim()	fdimd32()	fdimd64()
fdimd128()	fdimf()	fdiml()	feclearexcept()
fe_dec_getround()	fe_dec_setround()	fegetenv()	fegetexceptflag()
fegetround()	feholdexcept()	feraiseexcept()	fesetenv()
fesetexceptflag()	fesetround()	fetestexcept()	feupdateenv()
finite()	floor()	floord32()	floord64()
floord128()	floorf()	floorl()	fma
fmad32()	fmad64()	fmad128()	fmaf()
fmal()	fmax()	fmaxd32()	fmaxd64()
fmaxd128()	fmaxf()	fmaxl()	fmin()
fmind32()	fmind64()	fmind128()	fminf()
fminl()	fmod()	fmodd32()	fmodd64()
fmodd128()	fmodf()	fmodl()	fpclassify()
fp_read_rnd()	fp_swap_rnd()	frexp()	frexpd32()
frexpd64()	frexpd128()	frexpf()	frexpl()
gamma()	gamma_r()	hypot()	hypotd32()
hypotd64()	hypotd128()	hypotf()	hypotl()
ilogb()	ilogbd32()	ilogbd64()	ilogbd128()
ilogbf()	ilogbl()	isfinite()	isgreater()

Table 56. C/C++ functions that support floating-point (continued)

Functions			
isgreaterqual()	isinf()	isless()	islessequal()
islessgreater()	isnan()	isnormal()	isunordered()
jn()	j0()	j1()	ldexp()
ldexpd32()	ldexpd64()	ldexpd128()	ldexpf()
ldexpl()	lgamma()	lgammad32()	lgammad64()
lgammad128()	lgammaf()	lgammal()	lgamma_r()
llrint()	llrintd32()	llrintd64()	llrintd128()
llrintf()	llrintl()	llround()	llroundd32()
llroundd64()	llroundd128()	llroundf()	llroundl()
log()	logb()	logbd32()	logbd64()
logbd128()	logbf()	logbl()	logf()
logl()	log10()	log10d32()	log10d64()
log10d128()	log10f()	log10l()	log1p()
log1pd32()	log1pd64()	log1pd128()	log1pf()
log1pl()	log2()	log2d32()	log2d64()
log2d128()	log2f()	log2l()	logd32()
logd64()	logd128()	lrint()	lrintd32()
lrintd64()	lrintd128()	lrintf()	lrintl()
lround()	lroundd32()	lroundd64()	lroundd128()
lroundf()	lroundl()	matherr()	modf()
modff()	modfl()	modfd32()	modfd64()
modfd128()	nan()	nand32()	nand64()
nand128()	nanf()	nanl()	nearbyint()
nearbyintd32()	nearbyintd64()	nearbyintd128()	nearbyintf()
nearbyintl()	nextafter()	nextafterd32()	nextafterd64()
nextafterd128()	nextafterf()	nextafterl()	nexttoward()
nexttowardd32()	nexttowardd64()	nexttowardd128()	nexttowardf()
nexttowardl()	pow()	powd32()	powd64()
powd128()	powf()	powl()	quantexpd32()
quantexpd64()	quantexpd128()	quantized32()	quantized64()
quantized128()	remainder()	remainderd32()	remainderd64()
remainderd128()	remainderf()	remainderl()	remquo()
__remquod32()	__remquod64()	__remquod128()	remquof()
remquol()	rint()	rintd32()	rintd64()
rintd128()	rintf()	rintl()	round()
roundd32()	roundd64()	roundd128()	roundf()

Table 56. C/C++ functions that support floating-point (continued)

Functions			
roundl()	samequantumd32()	samequantumd64()	samequantumd128()
scalb()	scalbln()	scalblnd32()	scalblnd64()
scalblnd128()	scalblnf()	scalblnl()	scalbn()
scalbnd32()	scalbnd64()	scalbnd128()	scalbnf()
scalbnl()	signbit()	significand()	sin()
sind32()	sind64()	sind128()	sinf()
sinh()	sinhd32()	sinhd64()	sinhd128()
sinhf()	sinhl()	sinl()	__sinpid32()
__sinpid64()	__sinpid128()	sqrt()	sqrtd32()
sqrtd64()	sqrtd128()	sqrtf()	sqrtl()
tan()	tand32()	tand64()	tand128()
tanf()	tanhd32()	tanhd64()	tanhd128()
tanl()	tanh()	tanhf()	tanhl()
tgamma()	tgammad32()	tgammad64()	tgammad128()
tgammaf()	tgammal()	trunc()	truncd32()
truncd64()	truncd128()	truncf()	truncl()
yn()	y0()	y1()	

In *z/OS Language Environment Vendor Interfaces*, the chapter on [C/C++ special purpose interfaces for IEEE floating-point](#) provides information on the following functions.

Table 57. Special purpose C/C++ functions that support floating-point

Functions			
__chkbfp()	__fp_btoh()	__fp_cast()	__fp_htob()
__fp_level()	__fp_read_rnd()	__fp_setmode()	__fp_swapmod()
__fp_swap_rnd()	__fpc_rd()	__fpc_rs()	__fpc_rw()
__fpc_sm()	__fpc_wr()	__isBFP()	

Chapter 24. Handling error conditions, exceptions, and signals

This chapter discusses how to handle error conditions, exceptions, and signals with z/OS XL C/C++. It describes how to establish, enable and raise a signal, and provides a list of signals supported by z/OS XL C/C++.

In 31-bit applications, there are two basic ways to handle program checks and ABENDs:

- POSIX or ISO C/C++ signals (SIGABND, SIGFPE, SIGILL, SIGSEGV)
- User condition handlers registered using CEEHDLR interface or the USRHDLR runtime option.

In 31-bit applications, z/OS Language Environment uses a stack-based model to handle error conditions. This environment establishes a last-in, first-out (LIFO) queue of 0 or more user condition handlers for each stack frame. The z/OS Language Environment condition handler calls the user condition handler at each stack frame to handle error conditions when they are detected. For more information about the callable services in z/OS Language Environment, refer to [“Handling signals using Language Environment callable services” on page 278](#).

In AMODE 64 applications, user condition handlers are not available. The basic ways to handle program checks and ABENDs in AMODE 64 applications are:

- POSIX or ISO C/C++ signals (SIGABND, SIGFPE, SIGILL, SIGSEGV)
- Exception handlers registered using the `__set_exception_handler()` C runtime library function. See [“AMODE 64 exception handlers” on page 276](#) for more information.

The C error handling approach using signals is supported in a z/OS XL C/C++ program, but there are some restrictions (refer to [“Handling C software exceptions under C++” on page 273](#)). See [“Signal handlers” on page 278](#) for more information.

C++ exception handling is supported in all z/OS environments that are supported by C++ (including CICS and IMS); you must run your application with the TRAP (ON) runtime option. To turn off C++ exception handling, use the compiler option NOEXH. For more information on this compiler option, see [z/OS XL C/C++ User's Guide](#).

Note: If C++ exception handling is turned off you will get code which runs faster but is not ISO C/C++ conformant.

This chapter also describes some aspects of C++ object-oriented exception handling. The object-oriented approach uses the try, throw, and catch mechanism. Refer to [z/OS XL C/C++ Language Reference](#) for a complete description. Some library functions (`abort()`, `atexit()`, `exit()`, `setjmp()` and `longjmp()`) are affected by C++ exception handling; refer to [z/OS C/C++ Runtime Library Reference](#) for more information.

Handling C software exceptions under C++

Using the C and C++ condition handling schemes together in a C++ program may result in undefined behavior. This applies to the use of try, throw and catch with `signal()` and `raise()`, with z/OS Language Environment condition handlers such as CEEHDLR, or with CICS HANDLE ABEND under CICS in 31-bit mode. The behavior with respect to running destructors for automatic objects is undefined, due to control being transferred to non-C++ exception handlers (such as signal handlers) and stacks being collapsed. If a C software exception is not handled and results in program termination, the behavior for destructors for static non-local objects will also be undefined.

With z/OS UNIX, in a multithreaded environment, z/OS XL C++ exception stacks are managed on a per-thread basis. This means an exception thrown on one thread cannot be caught on another thread, including the IPT where `main()` was started. If the exception is not handled by the thread from which it was thrown, then the `terminate()` function is called.

Handling hardware exceptions under C++

You cannot use `try`, `throw`, and `catch` to handle hardware exceptions.

If a hardware exception resulting in abnormal termination occurs in a z/OS XL C++ program, destructors for static and automatic objects are not run. If a hardware exception occurs, and a handler was registered for the exception using `signal()`, the behavior of destructors for automatic objects is undefined.

Tracebacks under C++

A traceback is not produced if a thrown object was caught and handled.

If an object is thrown, and no catch clauses exist that will handle the thrown object, the program will call `terminate()`. By default, `terminate()` calls `abort()`, and the traceback produced will show that this has occurred. The traceback will not show the point from which the object was originally thrown. Instead, it will show that the object was thrown from the last encountered catch clause.

In sample routine CCNGCH1 ([Figure 83 on page 275](#)), `sub1()` throws object `a`. Because `sub1()` does not have any catch clauses to handle `a`, C++ attempts to find a suitable catch clause in the calling sub function, and then in the `main` function. Because no catch clauses can be found to handle object `a`, the traceback will show that object `a` was thrown from `main()`.

```

/* example of C++ exception handling */

#include <stdlib.h>
#include <iostream>
using namespace std;

class A {
    int i;
public:
    A(int j) { i = j; cout << "A ctor: i= " << i << '\n'; }
    A() { cout << "A dtor: i= " << i << '\n'; }
};

class B {
    char c;
public:
    B(char d) { c = d; cout << "B ctor: c= " << c << '\n'; }
    B() { cout << "B dtor: c= " << c << '\n'; }
};

void sub(void);
void sub1(void);

main() {
    try {
        sub();
        //traceback will show that the thrown object was from here because
        //no catch clauses match the thrown object and the last rethrow
        //occurred here.
        catch(int i) { cout << "caught an integer" << '\n'; }
        catch(char c) { cout << "caught a character" << '\n'; }
        exit(55);
    }

    void sub() {
        try {
            sub1();
        }
        //neither catch clause will catch object a, so again a will be
        //rethrown
        catch(double d) { cout << "caught a double" << '\n'; }
        catch(float f) { cout << "caught a float" << '\n'; }
        return;
    }

    void sub1() {
        A a(3001);
        try {
            throw(a);
        }
        //neither catch clause will catch object a, so a will be rethrown
        catch(B b) { cout << "caught a B object" << '\n'; }
        catch(short s) { cout << "caught a short" << '\n'; }
        return;
    }
}

```

Figure 83. Example illustrating C++ exception handling/traceback

If an object is thrown and a catch clause catches but then rethrows that object, or throws another object, and no catch clauses exist for the rethrown or subsequently thrown object, the traceback starts at the point from which the rethrow or subsequent throw occurred. The first object thrown is considered to have been caught and handled.

In sample routine CCNGCH2 (Figure 84 on page 276), the traceback would show that the `testeh` function rethrows an integer. Because there is no catch clause to handle the rethrown integer, the traceback will also show that `terminate()` and then `abort()` were called.

```

/* example of C++ exception handling */

#include <stdlib.h>
#include <iostream>
using namespace std;

int testeh(void);
class A {
    int i;
    public:
        A(int j) { i = j; cout << "A ctor: i= " << i << '\n'; }
        A() { cout << "A dtor: i= " << i << '\n'; }
};
class B {
    char c;
    public:
        B(char d) { c = d; cout << "B ctor: c= " << c << '\n'; }
        B() { cout << "B dtor: c= " << c << '\n'; }
};
A staticA(333);
B staticB('z');
void sub();

main() {
    sub();
    return(55);
}

void sub()
{
    A c(3001);
    try {
        cout << "calling testeh" << '\n';
        testeh(); // int will be rethrown from testeh()
    }
    // no catch clauses for the rethrown int
    catch(char c) { cout << "caught char" << '\n'; }
    catch(short s) { cout << "caught short s = " << s << '\n'; }
    cout << "this line should not be printed" << '\n';
    return;
}

testeh()
{
    A a(2001), a1(1001);
    B b('k');
    short k=12;
    int j=0, l=0;

    try {
        cout << "testeh running" << '\n';
        throw (6); // first throw: an int
    }
    catch(char c) { cout << "testeh caught char" << '\n'; }
    catch(int j) { cout << "testeh caught int j = " << j << '\n';
        try { // int should be caught here
            cout << "testeh again rethrowing" << '\n';
            throw; // rethrow the int
        }
        catch(char d) { cout << "char d caught" << '\n'; }
    }
    cout << "this line should not be printed" << '\n';
    return(0);
}

```

Figure 84. Example illustrating C++ exception handling/traceback

AMODE 64 exception handlers

In AMODE 64 applications, exception handlers are registered using the `__set_exception_handler()` C runtime library function. When no exception handler is registered, program checks and ABENDs cause POSIX/ISO C/C++ signals to be raised. These signals can be caught by user-written signal catchers, where suitable recovery can be done. When an exception handler is registered, no signal is generated when a program check or ABEND occurs. Instead, the active exception handler is invoked. Since program checks and ABENDs do not generate signals, the blocked/unblocked/ignored/caught settings for SIGABND, SIGFPE, SIGILL, and SIGSEGV make no difference. When an exception handler is active, all non-program-check and non-ABEND signal processing still occurs as described by POSIX or ISO C/C++. Only signals normally generated by program checks or ABENDs are suppressed.

Scope and nesting of exception handlers

Exception handlers apply only to the thread they are registered on. In a multi-threaded application, it is possible to have a mixture of threads, some with exception handlers registered, and some without. Program checks and ABENDs occurring on threads without active exception handlers cause the usual ISO C/C++/POSIX signal generation. Program checks and ABENDs occurring on threads with active exception handlers will bypass signal generation and will cause the active exception handler to be invoked.

Exception handlers are also stack-frame based, much like 31-bit user condition handlers. If function `a()` registers an exception handler, future program checks and ABENDs will drive that handler, until the handler is de-registered. This includes program checks occurring in `a()` (after the registration), and in any called functions. Function `a()` can deregister the handler using `__reset_exception_handler()`. After this is done, program checks and ABENDs once again cause signals to be raised. If function `a()` returns without calling `__reset_exception_handler()` to deregister its handler, the handler will be automatically removed when `a()` returns.

If function `a()` registers handler `ah()`, and calls function `b()`, program checks and ABENDs in `b()` will also go to `ah()`. However, `b()` can register its own handler, `bh()`, in which case any program checks and ABENDs in `b()` or any functions it calls will go to `bh()`. Exception handlers can be nested in this way as deep as required. If they are not explicitly deregistered by calling `__reset_exception_handler()`, they are automatically removed when the registering function returns. They are also removed, whenever a longjump-type function (`longjmp()`, `_longjmp()`, `siglongjmp()`, `setcontext()`, or C++ throw) causes control to jump back past the function that registered the handler. (Example: `a()` registers handler `ah()`, and calls `b()`, which registers handler `bh()`, and calls `c()`. Function `c()` longjumps back into `a()`. In this case, `bh()` will be removed, but `ah()` will remain.)

Note: Whenever a program check or ABEND occurs, no more than one exception handler will ever be driven, even when several nested handlers have been registered. The active handler is the one that was most recently registered, and not de-registered/removed. It will usually be the handler registered by the most deeply-nested routine at the time of the program check or ABEND.

During C++ throw processing, as the Language Environment stack is unwound and destructors for automatic C++ object are invoked, handlers registered by more-deeply nested functions are temporarily bypassed, in case program checks or ABENDs occur in the destructors. Example: `a()` registers handler `ah()`, and calls `b()`. Function `b()` has a dynamic object with destructor `bd()`. Function `b()` calls `c()`, which has a dynamic object with destructor `cd()`, and it registers handler `ch()`. Function `c()` then calls `d()`, which registers handler `dh()`, and then throws a C++ exception that will eventually get caught back in `a()`. As the C++ destructors are run, program checks/ABENDs in `cd()` go to handler `ch()`, and program checks/ABENDs in `bd()` go to `ah()`. By the time control resumes in the catch clause in `a()`, `dh()` and `ch()` are gone, and `ah()` is the active exception handler. This same type of exception handler scoping occurs after `pthread_exit()` is called and all outstanding C++ dynamic destructors still left on the stack are run.

If a program does `pthread_exit()` while an exception handler is active, that exception handler remains active while any `pthread_keycreate()` destructor routines and any `pthread_cleanup_push()` routines are invoked. These routines can register their own exception handlers, too, if required.

When `atexit()` routines or C++ static destructors are run, any active exception handlers at the time of the `exit()` or `pthread_exit()` have already been removed. If these routines need recovery, they can register their own exception handlers.

Handling exceptions

When the active exception handler is called after a program check or ABEND, it receives a pointer to the CIB (Condition Information Block) for the error. It can examine the CIB and associated MCH (Machine Check Handler record) to determine what the error is. The handler can fix up whatever is required or take dumps, etc. When it is finished, the only valid things it can do are:

- Long jump back to some earlier pre-defined recovery point (any of the several longjump-type functions may be used -- `longjmp()`, `_longjmp()`, `siglongjmp()`, `setcontext()`, or C++ throw.)
- Issue `exit()` or `_exit()`
- Issue `pthread_exit()`

- Issue `__cabend()`, `abort()`, etc

What it cannot do is return. If it returns, the system will automatically do `pthread_exit(-1)` if `POSIX(ON)` is in effect, or `exit(-1)` if not.

When the active exception handler is given control, the handler is suspended, along with all other handlers already registered. This means that any future program checks/ABENDs will cause the usual signal processing to occur. The active handler is re-enabled once it longjumps back. If it exits or returns, it is not re-activated, and termination starts with no active exception handler. If an exception handler needs exception handling recovery for its own program checks or ABENDs, it must register its own exception handler. As usual, this new handler will become active, and will get control for any program checks/ABENDs occurring in the outer exception handler or any routines it calls.

Signal handlers

The basis for error handling in z/OS UNIX XL C/C++ application programs is the generation, delivery, and handling of signals. Signals can be generated and delivered as a result of system events or application programming. You can code your application program to generate and send signals and to handle and respond to signals delivered to it.

Two types of signal handling are supported for catching signals: ISO C and POSIX.1. Each of these has standard signal delivery rules, which are discussed in this chapter. Asynchronous signal delivery under z/OS UNIX is also discussed. For additional information on the subject of POSIX-conforming signals, see *The POSIX.1 Standard: A Programmer's Guide*, by Fred Zlotnick, (Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991).

Handling signals with POSIX(OFF) using `signal()` and `raise()`

The z/OS XL C environment provides two functions that alter the signal handling capabilities available in the runtime environment: `signal()` and `raise()`. The `signal()` function registers a condition handler and the `raise()` function raises the condition.

In general, for C++ programs you are encouraged to use `try`, `throw`, and `catch` to perform exception handling. However, you can also use the z/OS XL C `signal()` and `raise()` functions.

You can use the `signal()` function to perform one of the following actions:

- Ignore the condition. For example, use the `SIG_IGN` condition to specify `signal(SIGFPE, SIG_IGN)`.
- Reset the Global Error Table for default handling. For example, use the `SIG_DFL` condition to specify `signal(SIGSEGV, SIG_DFL)`.
- Register a function to handle the specific condition. For example, pass a pointer to a function for the specific condition with `signal(SIGILL, cfunc1)`. The function registered for `signal()` must be declared with C linkage.

Handling signals using Language Environment callable services

In 31-bit mode, you can set up user signal handlers with the z/OS Language Environment condition handling services. Some of the z/OS Language Environment callable services available for condition handling are:

CEEHDLR

Register a user-written condition handler.

CEEHDLU

Remove a registered user-written condition handler.

CEESGL

Raise z/OS Language Environment condition.

In addition, with z/OS Language Environment, when an exception occurs after an interlanguage call, the exception may be handled where it occurs, or percolated to its caller (written in any z/OS Language Environment-conforming language), or promoted. For more information on how to handle exceptions

under the z/OS Language Environment condition handling model, refer to *z/OS Language Environment Programming Guide*.

Specific considerations for C and C++ under z/OS Language Environment :

1. The TRAP runtime option (equivalent to the former C/370 runtime options SPIE and STAE) determines how the z/OS Language Environment condition manager is to act upon error conditions and program interrupts. If the TRAP (OFF) runtime option is in effect, conditions detected by the operating system, often due to machine interrupts, will not be handled by the z/OS Language Environment environment and thus cannot be handled by a z/OS XL C/C++ program.

Note: TRAP (OFF) only blocks the handling of hardware (program checks) and operating system (abend) conditions. It does not block software conditions such those that are associated with a raise or CEESGL (31-bit mode). Any conditions that are blocked because of TRAP (OFF) are not presented to any handlers (whether registered by a signal or by CEEHDLR). In particular, even for TRAP (OFF), conditions that are initiated by a signal or by CEESGL (31-bit mode) are presented to handlers registered by either `signal()` or CEEHDLR.

The use of the TRAP (OFF) option is not recommended; refer to *z/OS Language Environment Programming Reference* for more information.

2. You can use the ERRCOUNT runtime option to specify how many errors are to be tolerated during the execution of your program before an abend occurs. The counter is incremented by one for every severity 2, 3, or 4 condition that occurs. Both hardware-generated and software-generated signals increment the counter.

If your C++ program uses `try`, `throw`, and `catch`, it is recommended that you specify either `ERRCOUNT(0)` (31-bit mode), which allows an unlimited number of errors, or `ERRCOUNT(n)` (31-bit mode), where `n` is a fairly high number. This is because z/OS XL C++ generates a severity 3 condition for each thrown object. In addition, each `catch` clause has the potential to rethrow an object or to throw a new object. In a large C++ program, many conditions can be generated as a result of objects being thrown, and thus the `ERRCOUNT` can be exceeded if the value used for it is too low. The default used for `ERRCOUNT` is usually a low number.

Note: The z/OS XL C/C++ registered condition handlers (those registered by `signal()` and `raise()`), are activated after the z/OS Language Environment registered condition handlers for the current stack frame are activated. This means that if there are condition handlers for both z/OS XL C/C++ and z/OS Language Environment, the z/OS Language Environment handlers are activated first.

Combining C++ condition handling (using `try`, `throw`, and `catch`), with z/OS Language Environment condition handling may result in undefined behavior.

Handling signals using z/OS UNIX with POSIX(ON)

z/OS UNIX signal processing allows flags to control the behavior of signal processing. Using these flags, you can simulate these signals and a wide variety of other signals such as ISO C/C++, POSIX.1, and BSD.

ISO C has the following standard signal delivery rules:

- Traditionally, signal actions are established only through the `signal()`.
- During signal delivery, the signal action is reset to `SIG_DFL` before the user signal action catcher function receives control.
- During signal delivery to a user signal catcher function, the signal mask is not changed.

POSIX.1 has the following standard signal delivery rules:

- Signal actions are typically established through the `sigaction()` function. With the addition of XPG4 support, there are a number of new flags that have been defined for `sigaction()` that extend its flexibility.
- During signal delivery, the signal action is not changed.
- During signal delivery to a user signal catcher function, the signal mask is changed to the *union* of:

- The signal mask at the time of the interruption
- A signal mask that blocks the signal type being delivered

The signal mask is restored when the signal catcher function returns.

BSD signals for the most part are consistent with the POSIX rules above except for the following:

- BSD signal mask is a 31-bit mask whereas the z/OS UNIX signal mask is a AMODE 64 mask. The relationship of the bits to specific signals is not the same. Therefore, we recommend you change to use the sigset manipulation functions, such as, `sigadd()`, `sigdelete()`, `sigempty()`.
- Traditionally, for BSD to generate a signal action, the `signal()` function was used. However, because the `signal()` function is used in ISO C/C++, BSD applications should be changed to use the `bsd_signal()` function.
- During signal delivery, the signal action is not changed.
- During signal delivery to a user signal catcher function, the signal mask is changed to the *union* of:
 - The signal mask at the time of the interruption
 - The signal mask specified in the `sa_mask` field of the `sigaction()` function

The signal mask is restored once the signal catcher function returns.

For compatibility, z/OS XL C/C++ supports the three standards listed above, and additional functions provided by XPG4.

Under z/OS XL C/C++, the primary function for establishing signal action is the `sigaction()` function. However, there are a number of other functions that you can use to effect signal processing. All signal types are accessible regardless of the function used to establish the signal action.

[Table 58 on page 280](#) includes functions that will establish a signal handler for a signal action.

Table 58. Functions that establish a signal handler

BSD Function	Purpose
<code>bsd_signal()</code>	BSD version of <code>signal()</code>
<code>sigaction()</code>	Examine and/or change a signal action
<code>sigignore()</code>	Set disposition to ignore a signal
<code>sigset()</code>	Change a signal action and/or a thread's signal mask
<code>signal()</code>	Specify signal handling

[Table 59 on page 280](#) lists other signal-related functions.

Table 59. Other signal-related functions

Signal Related Functions	Purpose
<code>abort()</code>	Stop a program
<code>kill()</code>	Send a signal to a process
<code>pthread_kill()</code>	Send a signal to a thread
<code>raise()</code>	Send a signal to yourself
<code>sigaddset()</code>	Add a signal to a signal set
<code>sigdelset()</code>	Delete a signal from a signal set
<code>sigemptyset()</code>	Initialize a signal set to exclude all signals
<code>sigfillset()</code>	Initialize a signal set to include all signals
<code>sighold()</code>	Add a signal to a thread's signal mask
<code>siginterrupt()</code>	Allow signals to interrupt functions

Table 59. Other signal-related functions (continued)

Signal Related Functions	Purpose
sigismember()	Test if a signal is in a signal set
sigpause()	Unblock a signal and wait for a signal
sigprocmask()	Examine and/or change a thread's signal mask
sigqueue()	Queue a signal to a process
sigrelse()	Remove a signal from a thread's signal mask
sigstack()	Set and/or get signal stack context
sigaltstack()	Set and/or get signal alternate stack context
sigsuspend()	Change mask and suspend the thread
sigwait()	Wait for asynchronous signal
sigpending()	Examine pending signals
sigtimedwait()	Wait for queued signals
sigwaitinfo()	Wait for queued signals

Asynchronous signal delivery under z/OS UNIX

Your z/OS UNIX application program might require its active processes to be able to react and respond to events occurring in the system or resulting from the actions of other processes communicating with its processes. One way of accomplishing such interprocess communication is for you to code your application program to identify signal conditions and determine how to react or respond when a signal condition is received from another application process.

Before you attempt to code your z/OS UNIX C/C++ application program to deliver and handle signals, you should identify all the processes that might cause signal conditions to be received by your application program's processes. You also need to know which signal condition codes are valid for your z/OS UNIX C/C++ application program and where the `signal.h` header file will be located and available to your application program. Your system programmer or the application program's designer should provide this information.

Note: Signal condition codes are defined in the `signal.h` include file.

A *signal* is a mechanism by which a process can be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* also refers to an event itself.

The POSIX.1-defined `sigaction()` function allows a calling application process to examine a specific signal condition and specify the processing action to be associated with it.

You can code your application program to use the `sigaction()` function in different ways. Two simplistic examples of using signals within z/OS UNIX C/C++ application programs follow:

1. A process is forked but the process is *aborted* if the signal handler receives an incorrect value.
2. A request is received from a *client* process to provide information from a database. The *server* process is a single point of access to the database.

If coded properly for handling and delivering interprocess signals, your application program can receive signals from other processes and interpret those signals such that the appropriate processing procedure occurs for each specific signal condition received. Your application program also can send signals and wait for responses to signal handling events from other application processes. Note that signals are not the best method of interprocess communication, because they can easily be lost if more than one is delivered at the same time. You may want to use other methods of interprocess communication, such as pipes, message queues, shared memory, or semaphores.

For descriptions of the supported z/OS XL C/C++ signal handling functions, see [z/OS C/C++ Runtime Library Reference](#).

Note: If your z/OS UNIX C/C++ application program calls a program written in a high-level language other than z/OS UNIX C/C++, you need to disable signal handling to block all signals from the z/OS UNIX C/C++ application program. If the called program encounters a program interrupt check situation, the results are unpredictable.

C signal handling features under z/OS XL C/C++

The terms used to describe implementation features and concepts are:

- Establishing a signal handler
- Enabling a signal
- Interrupting a program
- Raising a signal

Establishing a signal handler

A signal handler for a signal, `sig_num`, becomes established when `signal(sig_num, sig_handler)` is executed. (Two values of `sig_handler` are reserved: `SIG_IGN` and `SIG_DFL`. They are special values that establish the action taken.) `sig_handler` is a pointer to a function to be called when the signal is raised. This function is also known as a *signal handler*. Under C++, the signal handler function must have C linkage, by declaring it as `extern "C"`. Under C, the function must be written in C with the default linkage in effect. That is, `sig_handler` cannot have OS, PLI, C++, or COBOL linkage. The signal handler for the signal ceases to be established when:

- The signal is explicitly reset to the system default by using `signal(sig_num, SIG_DFL)`.
- You indicate that a signal is to be ignored by using `signal(sig_num, SIG_IGN)`.
- The signal is implicitly reset to the system default when the signal is raised. When `sig_handler` is called, signal handling is reset to the default as if an implicit `signal(sig_num, SIG_DFL)` had been executed. Depending on the purpose of the signal handler, you may want to reestablish the signal from within the signal handler.
- Under C, a loaded executable is deleted using the `release()` function and a signal handler for the signal resides in the executable. In this case, default handling will be reset for all the affected signals.
- A DLL module is explicitly loaded using `dllload()`, a function pointer in that module is obtained using `dllqueryfn()`, a signal handler is establishing using that function, and the DLL module is then explicitly deleted using `dllfree()`. Default handling will be reset for the affected signal.

Note: A C signal handler can be written in C, or can be written in C++ and declared as `extern "C"` so that it has C linkage.

Enabling a signal

A signal is enabled when the occurrence of the condition will result in either the execution of an established signal handler or the default system response. The signal is disabled when the occurrence is to be ignored, such as, when the signal action is `SIG_IGN`. This can be done by making the call `signal(sig_num, SIG_IGN)`. Using z/OS UNIX with POSIX(ON), `SIG_IGN` may be set with several other functions, such as, `sigaction()`. In addition to changing the signal action to `SIG_IGN`, the signal can be enabled or disabled (blocked) using the `sigprocmask()` function.

Interrupting a program

Program interrupts or errors detected by the hardware and identified to the program by operating system mechanisms are known as hardware signals. For example, the hardware can detect a divide by zero and this result can be raised to the program.

Raising a signal

Signals that are explicitly raised by the user, by using the `raise()` function or using z/OS UNIX with POSIX(ON) using the `kill()`, `killpg()`, or `pthread_kill()` functions, are known as software signals.

Identifying hardware and software signals

The following signals are a list of signals supported with z/OS XL C/C++ under POSIX(OFF):

SIGABND

System abend.

SIGABRT

Abnormal termination (software only).

SIGFPE

Erroneous arithmetic operation (hardware and software).

SIGILL

Invalid object module (hardware and software).

SIGINT

Interactive attention interrupt by `raise()` (software only).

SIGIOERR

Serious software error such as a system read or write. You can assign a signal handler to determine the file in which the error occurs or whether the condition is an abort or abend. This minimizes the time required to locate the source of a serious error.

SIGSEGV

Invalid access to memory (hardware and software).

SIGTERM

Termination request sent to program (software only).

SIGUSR1

Reserved for user (software only).

SIGUSR2

Reserved for user (software only).

The following signals are a list of signals supported with z/OS XL C/C++ under POSIX(ON):

SIGABND

System abend.

SIGABRT

Abnormal termination (software only).

SIGALRM

Asynchronous timeout signal generated as a result of an `alarm()`.

SIGBUS

Bus error.

SIGCHLD

Child process terminated or stopped.

SIGCLD

Child process terminated or stopped.

SIGCONT

Continue execution, if stopped.

SIGDANGER

Shutdown imminent.

SIGDUMP

Take a SYSMDUMP.

SIGFPE

Erroneous arithmetic operation (hardware and software).

SIGHUP

Hangup, when a controlling terminal is suspended or the controlling process ended.

SIGILL

Invalid object module (hardware and software).

SIGINT

Asynchronous CNTL-C from one of the z/OS UNIX shells or a software generated signal.

SIGIO

Completion of input or output.

SIGIOERR

Serious software error such as a system read or write. Assign a signal handler to determine the file in which the error occurs or whether the condition is an abort or abend. Minimize the time required to locate the source of a system error.

SIGKILL

An unconditional terminating signal.

SIGPIPE

Write on a pipe with no one to read it.

SIGPOLL

Pollable event.

SIGPROF

Profiling timer expired.

SIGQUIT

Terminal quit signal.

SIGSEGV

Invalid access to memory (hardware and software).

SIGSTOP

The process is stopped.

SIGSYS

Bad system call.

SIGTERM

Termination request sent to program (software only).

SIGTHCONT

The specific thread is resumed.

SIGTHSTOP

The specific thread is stopped.

SIGTRACE

UNIX System Services syscall trace toggle signal.

SIGTRAP

Debugger event.

SIGTSTP

Terminal stop signal.

SIGTTIN

Background process attempting read.

SIGTTOU

Background process attempting write.

SIGURG

High bandwidth is available at a socket.

SIGUSR1

Reserved for user (software only).

SIGUSR2

Reserved for user (software only).

SIGVTALRM

Virtual timer expired.

SIGWINCH

Window size has changed.

SIGXCPU

CPU time limit exceeded.

SIGXFSZ

File size limit exceeded.

The applicable hardware signals or exceptions are listed in [Table 60 on page 285](#). It also lists those hardware exceptions that are not supported (for example, fixed-point overflow) and are masked.

The applicable software signals or exceptions that are supported with POSIX(OFF) are listed in [Table 61 on page 285](#) (see [Table 62 on page 287](#) for the POSIX(ON) signals).

Table 60. Hardware exceptions - Default runtime messages and system actions

C Signal	Hardware Exception	Default Runtime Message with z/OS Language Environment	Default System Action with z/OS Language Environment Library
SIGILL	Operation exception	CEE3201	Abnormal termination MVS rc=3000
	Privileged operation exception	CEE3202	
	Execute exception	CEE3203	
SIGSEGV	Protection exception	CEE3204	Abnormal termination MVS rc=3000
	Addressing exception	CEE3205	
	Specification exception	CEE3206	
SIGFPE	Data exception	CEE3207	Abnormal termination MVS rc=3000
	Fixed-point divide	CEE3209	
	Decimal overflow (for C only)	CEE3210	
	Decimal divide	CEE3211	
	Exponent overflow	CEE3212	
	Floating point divide	CEE3215	
	Compare and Trap Data Exception	CEE3234	

Note: Under TSO, SIGINT will not be raised if you press the attention key. It must be raised using raise().

The default runtime program mask is enabled for decimal overflow exceptions.

[Table 61 on page 285](#) shows software signals with POSIX(OFF) or exceptions, their origin, default runtime messages and default system actions.

Table 61. Software exceptions - Default runtime messages and system actions with POSIX(OFF)

C Signal	Software Exception	Default Runtime Message with z/OS Language Environment	Default System Action with z/OS Language Environment Library
SIGFPE	raise(SIGFPE)	EDC6000	Abnormal Termination MVS rc=3000
SIGILL	raise(SIGILL)	EDC6001	Abnormal Termination MVS rc=3000
SIGSEGV	raise(SIGSEGV)	EDC6002	Abnormal Termination MVS rc=3000
SIGFPE	raise(SIGFPE)	EDC6002	Abnormal Termination MVS rc=3000

Table 61. Software exceptions - Default runtime messages and system actions with POSIX(OFF) (continued)

C Signal	Software Exception	Default Runtime Message with z/OS Language Environment	Default System Action with z/OS Language Environment Library
SIGABND	raise(SIGABND)	EDC6003	Abnormal Termination MVS rc=3000
SIGTERM	raise(SIGTERM)	EDC6004	Abnormal Termination MVS rc=3000
SIGINT	raise(SIGINT)	EDC6005	Abnormal Termination MVS rc=3000
SIGABRT	raise(SIGABRT)	EDC6006	Abnormal Termination MVS rc=2000
SIGUSR1	raise(SIGUSR1)	EDC6007	Abnormal Termination MVS rc=3000
SIGUSR2	raise(SIGUSR2)	EDC6008	Abnormal Termination MVS rc=3000
SIGIOERR	raise(SIGIOERR)	EDC6009	Signal is ignored

SIGABND considerations

When the SIGABND signal is registered with an address of a C handler using the `signal()` function, control cannot resume at the instruction following the abend or the invocation of `raise()` with SIGABND. If the C signal handler is returned, the abend is percolated and the default behavior occurs. The `longjmp()` or `exit()` function can be invoked from the handler to control the behavior.

If SIG_IGN is the specified action for SIGABND and an abend occurs (or SIGABND was raised), the abend will not be ignored because a resume cannot occur. The abend will percolate and the default action will occur.

Two macros are available in `signal.h` header file that provide information about an abend. The `__abendcode()` macro returns the abend that occurred and `__rsnocode()` returns the corresponding reason code for the abend. These values are available in a C signal handler that has been registered with the SIGABND signal. If you are looking for the abend and reason codes, using these macros, they should only be checked when in a signal handler. The values returned by the `__abendcode()` and `__rsnocode()` macros are undefined if the macros are used outside a registered signal handler.

SIGIOERR considerations

When the SIGIOERR signal is raised, codes for the last operation will be set in the `__amrc` structure to aid you in error diagnosis.

Default handling of signals

The runtime environment performs default handling of a given signal unless the signal is established (`signal(sig_num, sig_handler)`) or the signal is disabled (`signal(sig_num, SIG_IGN)`). A user can set or reset default handling by coding `signal(sig_num, SIG_DFL)`.

The default handling depends upon the signal that was raised. For more information about the default handling of a given signal, see [Table 60 on page 285](#), [Table 61 on page 285](#), and [Table 62 on page 287](#).

Using z/OS UNIX

[Table 62 on page 287](#) describes the default actions for signals that may be delivered to C/C++ application programs running POSIX(ON).

Table 62. Default signal processing with POSIX(ON)

Signal	Default Action
SIGABND	Clean up the z/OS XL C/C++ runtime library, issue message CEE5204, and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. If the signal is generated as a result of an abend condition, as opposed to being software generated by a <code>raise()</code> , <code>kill()</code> , or <code>pthread_kill()</code> function, the CEE5204 message is issued along with a trace-back message indicating a user function was in control when the abend occurred.
SIGABRT	Clean up the z/OS XL C/C++ runtime library, issue message CEE5207 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGALRM	Clean up the z/OS XL C/C++ runtime library, issue message CEE5214 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGCHLD	The signal is ignored.
SIGCLD	The signal is ignored.
SIGCONT	The process is continued if it was stopped. Otherwise, the signal is ignored.
SIGDANGER	The signal is ignored.
SIGDUMP	The system will obtain a user address space dump.
SIGFPE	Clean up the z/OS XL C/C++ runtime library, issue message CEE5201, and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. If the signal is generated as a result of an abend condition, as opposed to being software generated by a <code>raise()</code> , <code>kill()</code> , or <code>pthread_kill()</code> function, the CEE5201 message is issued along with a trace-back message indicating a user function was in control when the abend occurred.
SIGHUP	Clean up the z/OS XL C/C++ runtime library, issue message CEE5210 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGILL	Clean up the z/OS XL C/C++ runtime library, issue message CEE5202, and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. If the signal is generated as a result of an abend condition, as opposed to being software generated by a <code>raise()</code> , <code>kill()</code> , or <code>pthread_kill()</code> function, the CEE5202 message is issued along with a trace-back message indicating a user function was in control when the abend occurred.
SIGINT	Clean up the z/OS XL C/C++ runtime library, issue message CEE5206 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGIO	The signal is ignored.
SIGIOERR	The signal is ignored. In a POSIX application running on z/OS UNIX SIGIOERR is not supported directly by the kernel. Instead, z/OS XL C/C++ maps SIGIOERR to SIGIO. Any application using SIGIOERR should not also use SIGIO.
SIGKILL	End the process with no z/OS XL C/C++ runtime cleanup.

Table 62. Default signal processing with POSIX(ON) (continued)

Signal	Default Action
SIGPIPE	Clean up the z/OS XL C/C++ runtime library, issue message CEE5213 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGQUIT	Clean up the z/OS XL C/C++ runtime library, issue message CEE5220 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGSEGV	Clean up the z/OS XL C/C++ runtime library, issue message CEE5203 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGSTOP	The process is stopped.
SIGTERM	Clean up the z/OS XL C/C++ runtime library, issue message CEE5205 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGTHCONT	The specific thread is resumed.
SIGTHSTOP	The specific thread is stopped.
SIGTRACE	The UNIX System Services syscall trace is toggled.
SIGTRAP	Clean up the z/OS XL C/C++ runtime library, issue message CEE5222 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGTSTP	The process is stopped.
SIGTTIN	The process is stopped.
SIGTTOU	The process is stopped.
SIGUSR1	Clean up the z/OS XL C/C++ runtime library, issue message CEE5208 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. In past releases, the default action for this signal was to ignore the signal.
SIGUSR2	Clean up the z/OS XL C/C++ runtime library, issue message CEE5209 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. In past releases, the default action for this signal was to ignore the signal.
SIGPOLL	Clean up the z/OS XL C/C++ runtime library, issue message CEE5225 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGURG	The signal is ignored.
SIGBUS	Clean up the z/OS XL C/C++ runtime library, issue message CEE5227 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.

Table 62. Default signal processing with POSIX(ON) (continued)

Signal	Default Action
SIGSYS	Clean up the z/OS XL C/C++ runtime library, issue message CEE5228 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGWINCH	The signal is ignored.
SIGXCPU	Clean up the z/OS XL C/C++ runtime library, issue message CEE5230 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGXFSZ	Clean up the z/OS XL C/C++ runtime library, issue message CEE5231 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGVTALRM	Clean up the z/OS XL C/C++ runtime library, issue message CEE5232 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGPROF	Clean up the z/OS XL C/C++ runtime library, issue message CEE5233 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
Dubbed Process: A process that is not from a call to a <code>fork()</code> function or to a program <code>main()</code> function through an <code>exec()</code> function.	

Summary of C error handling

Procedure

1. Signal is raised. Is SIG_IGN set for the signal? Or is the signal blocked?

Option	Description
Yes	See “2” on page 289
No	See “5” on page 289

2. Is the signal for a SIGABND?

Option	Description
Yes	See “3” on page 289
No	See “4” on page 289

3. a. Condition is percolated for default behavior.
4. a. Resume at the next instruction.
5. a. Continue at [“6” on page 289](#).
6. Is the signal asynchronous (or previously blocked)?

Option	Description
Yes	See “7” on page 290
No	See “12” on page 290

7. Is a C handler established for the signal?

Option	Description
Yes	See “8” on page 290
No	See “11” on page 290

8. Was the C handler established by `signal()` or `sigaction()` with the `SA_OLD_STYLE` or `SA_RESETHAND` flag set?

Option	Description
Yes	See “9” on page 290
No	See “10” on page 290

9. a. Run C handler using ISO C/C++ rules and transfer control to the next instruction following asynchronous interrupt.
10. a. Run C handler using POSIX rules and transfer control to the next instruction following the asynchronous interrupt.
11. a. Perform default processing.

12. Is z/OS Language Environment user handler registered?

Option	Description
Yes	See “13” on page 290
No	See “14” on page 290

13. a. Run z/OS Language Environment user handler. The handler can resume, percolate or promote the signal. See [z/OS Language Environment Programming Guide](#) for more details.
14. Is a C handler established for the signal by `signal()` or `sigaction()` with the `SA_OLD_STYLE` or `SA_RESETHAND` flag set?

Option	Description
Yes	See “15” on page 290
No	See “16” on page 290

15. a. Run C handler using ISO C/C++ rules and resume at the next instruction.
16. a. Continue at [“17” on page 290](#).

17. At stack frame 0?

Option	Description
Yes	See “18” on page 290
No	See “21” on page 291

18. Was a C handler established?

Option	Description
Yes	See “19” on page 290
No	See “20” on page 290

19. a. Run C handler using POSIX signal delivery rules and resume at next instruction.
20. a. Perform default processing.

21. a. Default handling for the signal and percolate to next stack frame.

Signal considerations using z/OS UNIX

The following restrictions and inconsistencies exist for z/OS UNIX XL C/C++ application program signal handling:

- Signal processing is blocked by the kernel when an application program is running on a request block (RB) other than the one the `main()` routine was started on.
- An application program should not use the `longjmp()` function to exit from a signal catcher established through the use of `sigaction()`. The `sigsetjmp()` and `siglongjmp()` functions should be used instead of `setjmp()` and `longjmp()`. The `longjmp()` function can be used if the `signal()` function was used to establish the signal catcher.
- An application program must not use the macro versions of the `getc()`, `putc()`, `getchar()`, and `putchar()` functions to perform I/O to the same file from an asynchronous signal catcher function.
- Floating point registers are saved before a call to the signal catcher function and restored when the signal catcher returns. This is done for all signals.
- For z/OS UNIX XL C/C++ application programs, the `errno` value is saved before a call to the signal catcher function and restored when the signal catcher returns.

Example of C signal handling under z/OS XL C or z/OS XL C++

In example program CCNGEC1 (Figure 85 on page 291), the call to `signal()` in `main()` establishes the function `signal_handler` to process the interrupt signal when it occurs. An error value returned from this call to `signal()` causes the program to end with a printed error message. The `signal_handler` function asks you to enter a `y` or `Y` from the keyboard if you want to halt the program. Entering any other character causes the program to resume operation.

```
/* this example demonstrates signal handling */

#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

#ifdef __cplusplus /* __cplusplus is implicitly defined when */
    extern "C" { /* the program is compiled with the z/OS C++ */
#endif /* compiler */

void handler(int);

#ifdef __cplusplus
}
#endif

int main(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        perror("Could not set SIGINT");
        abort();
    }
    /* add code here if desired */
    raise(SIGINT);
    /* add code here if desired */
    return(0);
}

void handler(int sig_num) {
    char ch;

    signal(SIGINT, handler);
    printf("End processing?\n");
    ch = getchar();
    if (ch == 'y' || ch == 'Y')
        exit(0);
}
```

Figure 85. Example illustrating signal handling

Chapter 25. Network communications under UNIX System Services

This chapter discusses interprocess communication, including MVS Sockets for z/OS UNIX and the X/Open Transport Interface (XTI) for z/OS UNIX and the internetworking involved.

Many products today supply a socket interface. The types of application programming interfaces (APIs) for the sockets which will be covered in this chapter are:

- **X/Open Socket**
- **Berkeley Socket**

If you are running with some other socket API, this material will not necessarily apply.

Your z/OS UNIX XL C/C++ application program can take advantage of sockets or XTI to communicate with a related application (server or client).

The X/Open Transport Interface (XTI) defines an independent transport service interface that allows multiple users to communicate at the transport level of the OSI reference model. More information can be found at the end of this chapter.

Understanding z/OS UNIX sockets and internetworking

z/OS UNIX provides support for an enhanced version of an industry-accepted protocol for client/server communication known as *sockets*. The three types of application programming interfaces (API), for the sockets which will be covered in this chapter are:

- **X/Open Socket:** The API type of socket as defined by X/Open in XPG4.2.
- **Berkeley Socket:** The socket API that represents a migration path for programs coded under the HOT1120 and HOT1130 elements. It allows use of the BSD4.3 interface and function in the X/Open environment. Its purpose is to expedite the porting of existing BSD4.3 applications.

The z/OS UNIX socket API provides support for both UNIX domain sockets and Internet domain sockets. UNIX domain sockets, or *local sockets*, allow interprocess communication within MVS independent of TCP/IP. Local sockets behave like traditional UNIX-domain sockets and allow processes to communicate with one another on a single system. Internet sockets allow application programs to communicate with others in the network using TCP/IP.

This chapter provides some background information about z/OS UNIX sockets and about network communication in general. It is intended to provide an overview of the programming concepts associated with using z/OS UNIX sockets and network communication.

For information about using the socket API, see [*z/OS C/C++ Runtime Library Reference*](#).

Basics of network communication

This section looks at network communication from a very high level and defines some terms used throughout the book. For more detailed information on z/OS network communication and TCP/IP sockets, see [*z/OS Communications Server: IP Configuration Guide*](#) and [*z/OS Communications Server: IP Programmer's Guide and Reference*](#). For more detailed information on IPv6 network communication and AF_INET6 sockets, see [*z/OS Communications Server: IPv6 Network and Appl Design Guide*](#).

Network communication, or *internetworking*, defines a set of protocols (that is, rules and standards) that allow application programs to talk with each other without regard to the hardware and operating systems where they are run. Internetworking allows application programs to communicate independently of their physical network connections.

The internetworking technology called *TCP/IP* is named after its two main protocols: Transmission Control Protocol (TCP) and Internet Protocol (IP). To understand TCP/IP, you should be familiar with the following terms:

client

A process that requests services on the network.

server

A process that responds to a request for service from a client.

datagram

The basic unit of information, consisting of one or more data packets, which are passed across an Internet at the transport level.

packet

The unit or block of a data transaction between a computer and its network. A packet usually contains a network header, at least one high-level protocol header, and data blocks. Generally, the format of data blocks does not affect how packets are handled. Packets are the exchange medium used at the Internetwork layer to send data through the network.

Transport protocols for sockets

A *protocol* is a set of rules or standards that each host must follow to allow other hosts to receive and interpret messages sent to them. There are two general types of transport protocols:

- A *connectionless protocol* is a protocol that treats each datagram as independent from all others. Each datagram must contain all the information required for its delivery.

An example of such a protocol is *User Datagram Protocol (UDP)*. UDP is a datagram-level protocol built directly on the IP layer and used for application-to-application programs on a TCP/IP host. UDP does not guarantee data delivery, and is therefore considered unreliable. Application programs that require reliable delivery of streams of data should use TCP.

- A *connection-oriented protocol* requires that hosts establish a logical connection with each other before communication can take place. This connection is sometimes called a *virtual circuit*, although the actual data flow uses a packet-switching network. A connection-oriented exchange includes three phases:

1. Start the connection
2. Transfer data
3. End the connection

An example of such a protocol is *Transmission Control Protocol (TCP)*. TCP provides a reliable vehicle for delivering packets between hosts on an Internet. TCP breaks a stream of data into datagrams, sends each one individually using IP, and reassembles the datagrams at the destination node. If any datagrams are lost or damaged during transmission, TCP detects this and retransmits the missing datagrams. The data stream that is received is therefore a reliable copy of the original.

These types of protocols are illustrated in [Figure 87 on page 302](#) and in [Figure 88 on page 303](#).

What is a socket?

A *socket* can be thought of as an endpoint in a two-way communication channel. Socket routines create the communication channel, and the channel is used to send data between application programs either locally or over networks. Each socket within the network has a unique name associated with it called a *socket descriptor*—a fullword integer that designates a socket and allows application programs to refer to it when needed.

Using an electrical analogy, you can think of the communication channel as the electrical wire with its plug and think of the port, or socket, as the electrical socket or outlet, as shown in [Figure 86 on page 295](#).

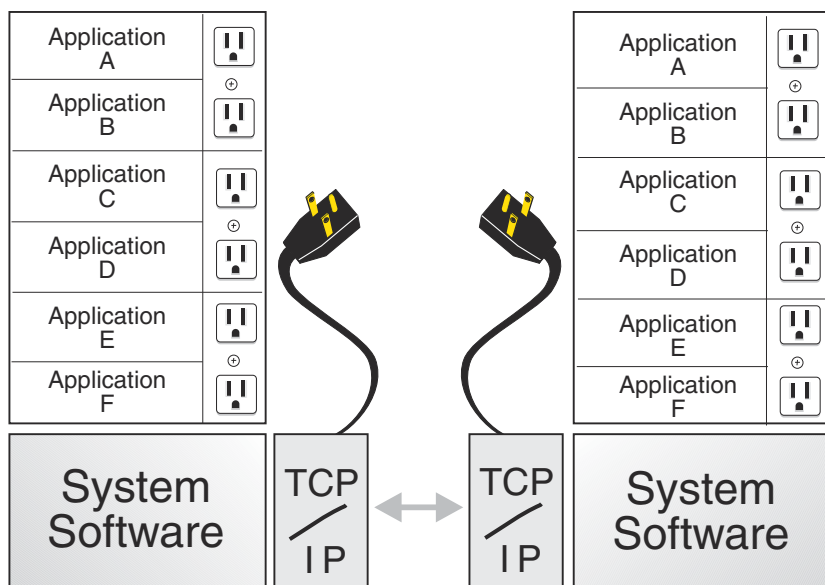


Figure 86. An electrical analogy showing the socket concept

This figure shows many application programs running on a client and many application programs on a server. When the client starts a socket call, a socket connection is made between an application on the client and an application on the server.

Another analogy used to describe socket communication is a telephone conversation. Dialing a phone number from your telephone is similar to starting a socket call. The telephone switching unit knows where to logically make the correct switch to complete the call at the remote location. During your telephone conversation, this connection is present and information is exchanged. After you hang up, the connection is broken and you must start it again. The client uses the `socket()` function call to start the logical switch mechanism to connect to the server.

As with file access, user processes ask the operating system to create a socket when one is needed. The system returns an integer, the socket descriptor (`sd`), that the application uses every time it wants to refer to that socket. The main difference between sockets and files is that the operating system binds file descriptors to a file or device when the `open()` call creates the file descriptor. With sockets, application programs can choose to either specify the destination each time they use the socket—for example, when sending datagrams—or to bind the destination address to the socket.

Sockets behave in some respects like UNIX files or devices, so they can be used with such traditional operations as `read()` or `write()`. For example, after two application programs create sockets and open a connection between them, one program can use `write()` to send a stream of data, and the other can use `read()` to receive it. Because each file or socket has a unique descriptor, the system knows exactly where to send and to receive the data.

You can wait on a socket using the following asynchronous I/O functions:

- `aio_read()` - Asynchronous read from a socket
- `aio_write()` - Asynchronous write to a socket
- `aio_cancel()` - Cancel an asynchronous I/O request
- `aio_suspend()` - Wait for an asynchronous I/O request
- `aio_error()` - Retrieve error status for an asynchronous I/O operation
- `aio_return()` - Retrieve return status for an asynchronous I/O operation

You can suspend the invoking thread until a specified asynchronous I/O event, timeout, or signal occurs. These functions are described in [z/OS C/C++ Runtime Library Reference](#).

z/OS UNIX Socket families

In z/OS UNIX, the following socket families are supported:

- UNIX Domain Sockets, known as *local sockets*, which are part of the UNIX Address Family (AF_UNIX)
- Internet Protocol Sockets, which are part of the Internet Address Family (AF_INET for IPv4 and AF_INET6 for IPv6)

AF_UNIX sockets provide communication between processes on a single system. This socket family supports two types of sockets—stream and datagram sockets. These socket types are described in the next section.

AF_INET and AF_INET6 sockets provide a means of communicating between application programs that are on different systems using the Transport Control Protocol provided by a TCP/IP product. This socket family supports both stream and datagram sockets. Each of these socket types is described in the next section.

z/OS UNIX Socket types

The z/OS UNIX socket API provides application programs with a network interface that hides the details of the physical network. The socket API supports both *stream sockets* and *datagram sockets*, each providing different services for application programs. Stream and datagram sockets interface to the transport layer protocols, UDP and TCP. You choose the appropriate interface for an application.

Stream sockets

Stream sockets act like streams of information. There are no boundaries between data, so communicating processes must agree on their own mechanism to distinguish information. Usually, the process sending information sends the length of the data, followed by the data itself. The process receiving information reads the length and then loops, accepting data until all of it has been transferred. Stream sockets guarantee delivery of the data in the order it was sent and without duplication. The stream socket interface defines a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent. Flow control is built in, to avoid data overruns. No boundaries are imposed on the data; the data is considered to be a stream of bytes.

Stream sockets are more common, because the burden of transferring the data reliably is handled by the system rather than by the application.

Datagram sockets

The *datagram socket* interface defines a connectionless service. Datagrams are sent as independent packets. The service provides no guarantees; data can be lost or duplicated, and datagrams can arrive out of order. The size of a datagram is limited to the size that can be sent in a single transaction. No disassembly and reassembly of packets is performed.

Guidelines for using socket types

This section describes criteria to help you choose the appropriate socket type for an application program.

If you are communicating with an existing application program, you must use the same protocols as the existing application program. For example, if you communicate with an application that uses TCP, you must use stream sockets. For other application programs, you should consider the following factors:

- **Reliability.** Stream sockets provide the most reliable connection. Datagram sockets are unreliable, because packets can be discarded, corrupted, or duplicated during transmission. This may be acceptable if the application program does not require reliability, or if the application program implements the reliability on top of the sockets interface. The trade-off is the increased performance available with datagram sockets.
- **Performance.** The overhead associated with reliability, flow control, packet reassembly, and connection maintenance degrade the performance of stream sockets in comparison with datagram sockets.

- **Data transfer.** Datagram sockets impose a limit on the amount of data transferred in a single transaction. If you send less than 2048 bytes at a time, use datagram sockets. As the amount of data in a single transaction increases, use stream sockets.

Addressing within sockets

The following sections describe the different ways to address within the socket API.

Address families

Address families define different styles of addressing. All hosts in the same address family use the same scheme for addressing socket endpoints. z/OS UNIX supports three address families—AF_INET, AF_INET6, and AF_UNIX. The AF_INET and AF_INET6 address families define addressing in the IP domain. The AF_UNIX address family defines addressing in the z/OS UNIX domain. In the z/OS UNIX domain, address spaces can use the socket interface to communicate with other address spaces on the same host.

Note: In this case, the z/OS UNIX domain is used in much the same way as the UNIX domain on other UNIX-type systems.

Socket address

A socket address is defined by the *sockaddr* structure in the `sys/socket.h` include file. The structure has three fields, as shown in the following example:

```
struct sockaddr {
    unsigned char sa_len;
    unsigned char sa_family;
    char          sa_data[14];    /* variable length data */
};
```

The *sa_len* field contains the length of the *sa_data* field. The *sa_family* field contains the address family. It is AF_INET or AF_INET6 for the Internet domain and AF_UNIX for the UNIX domain. The *sa_data* field is different for each address family. Each address family defines its own structure, which can be overlaid on the *sockaddr* structure. See [“Addressing within the AF_INET domain” on page 298](#) and [“Addressing within the AF_INET6 domain” on page 298](#) for more information about the Internet domain, and [“Addressing within the AF_UNIX domain” on page 299](#) for more information about the UNIX domain.

Internet addresses

Internet addresses represent a network interface. Every Internet address within an administered AF_INET domain must be unique. On the other hand, it is not necessary that every host have a unique Internet address; in fact, a host has as many Internet addresses as it has network interfaces.

Ports

A port is used to distinguish between different application programs using the same network interface. It is an additional qualifier used by the system software to get data to the correct application program. Physically, a port is a 16-bit integer. Some ports are reserved for particular application programs or protocols and are called *well-known ports*.

Network byte order

Ports and addresses are usually specified to calls using the network byte ordering convention. This convention is a method of sorting bytes under specific machine architectures. There are two common methods:

- *Big-endian* byte ordering places the most significant byte first. This method is used in IBM mainframe processors.
- *Little-endian* byte ordering places the least significant byte first. This method is used in Intel microprocessors.

Using network byte ordering for data exchanged between hosts allows hosts using different architectures to exchange address information. See references in figures [Figure 90 on page 304](#), [Figure 91 on page 305](#), and [Figure 93 on page 305](#) for examples of using the `htons()` call to put ports into network byte order. For more information about network byte order, see [z/OS C/C++ Runtime Library Reference](#).

Note: The socket interface does not handle application program data byte ordering differences. Application program writers must handle byte order differences themselves.

Addressing within the AF_INET domain

A socket address in the Internet address family comprises the following fields: the address family (AF_INET), an Internet address, the length of that Internet address, a port, and a character array. The structure of the Internet socket address is defined by the following `sockaddr_in` structure, which is found in the `netinet/in.h` include file:

```
struct in_addr {
    ip_addr_t s_addr;

    struct sockaddr_in {
        unsigned char    sin_len;
        unsigned char    sin_family;
        unsigned short   sin_port;
        struct in_addr    sin_addr;
        unsigned char    sin_zero[8];
    };
};
```

sin_len

set to the length of the `sockaddr_in` structure

sin_family

set to AF_INET

sin_port

port used by the application program, in network byte order

sin_zero

field should be set to all zeros

Addressing within the AF_INET6 domain

A socket address in the Internet address family comprises the following fields: the address family (AF_INET6), an Internet address, the length of that Internet address, a port, flow information, and scope information. The structure of the Internet socket address is defined by the following `sockaddr_in6` structure, which is found in the `netinet/in.h` include file:

```
struct in6_addr {
    union {
        uint8_t  _S6_u8[16];
        uint32_t _S6_u32[4];
    } _S6_un;
};
#define s6_addr _S6_un._S6_u8

struct sockaddr_in6 {
    uint8_t          sin6_len;
    sa_family_t      sin6_family;
    in_port_t        sin6_port;
    uint32_t          sin6_flowinfo;
    struct in6_addr   sin6_addr;
    uint32_t          sin6_scope_id;
};
```

sin6_len

Set to the length of the `sockaddr_in6` structure

sin6_family

Set to AF_INET

sin6_port

Port used by the application program, in network byte order

sin6_flowinfo

32-bit field that contains the traffic class and the flow label

sin6_addr

a single *in6_addr* structure, which holds one 128-bit IPv6 address stored in network byte order

sin6_scope_id

a 32 bit integer that identifies a set of interfaces as appropriate for the scope of the address carried in the *sin6_addr* field.

Note: IPv6 structures are exposed by defining the `_OPEN_SYS_SOCKET_IPV6` feature test macro.

Addressing within the AF_UNIX domain

A socket address in the AF_UNIX address family is comprised of three fields: the length of the following pathname, the address family (AF_UNIX), and the pathname itself. The structure of an AF_UNIX socket address is defined as follows:

```
struct sockaddr_un {
    unsigned char  sun_len;
    unsigned char  sun_family;
    char  sun_path[108];      /* pathname */
};
```

This structure is defined in the `sockaddr_un` structure found in `sys/un.h` include file. The *sun_len* contains the length of the pathname in *sun_path*; *sun_family* field is set to *AF_UNIX*; and *sun_path* contains the null-terminated pathname.

The conversation

The client and server exchange data using a number of functions. They can send data using `send()`, `sendto()`, `sendmsg()`, `write()`, or `writew()`. They can receive data using `recv()`, `recvfrom()`, `recvmsg()`, `read()`, or `readv()`. The following is an example of the `send()` and `recv()` call:

```
send(s, addr_of_data, len_of_data, 0);
recv(s, addr_of_buffer, len_of_buffer, 0);
```

The `send()` and `recv()` function calls specify the sockets on which to communicate, the address in memory of the buffer that contains, or will contain, the data (*addr_of_data*, *addr_of_buffer*), the size of this buffer (*len_of_data*, *len_of_buffer*), and a flag that tells how the data is to be sent. Using the flag `0` tells TCP/IP to transfer the data normally. The server uses the socket that is returned from the `accept()` call.

These functions return the amount of data that was either sent or received. Because stream sockets send and receive information in streams of data, it can take more than one call to `send()` or `recv()` to transfer all the data. It is up to the client and server to agree on some mechanism of signaling that all the data has been transferred.

When the conversation is over, both the client and server call the `close()` function to end the connection. The `close()` function also deallocates the socket, freeing its space in the table of connections. To end a connection with a specific client, the server closes the socket returned by `accept()`. If the server closes its original socket, it can no longer accept new connections, but it can still converse with the clients it is connected to. The following is an example of the `close()` call:

```
close(s);
```

The server perspective

Before the server can accept any connections with clients, it must register itself with TCP/IP and "listen" for client requests on a specific port.

Allocation with socket()

The server must first allocate a socket. This socket provides an endpoint that clients connect to.

A socket is actually an index into a table of connections, so socket numbers are usually assigned in ascending order. In the C language, the programmer calls the `socket()` function to allocate a new socket, as shown in the following example:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

The `socket()` function requires the address family (`AF_INET`), the type of socket (`SOCK_STREAM`), and the particular networking protocol to use (when 0 is specified, the system automatically uses the appropriate protocol for the specified socket type). A new socket is allocated and returned.

bind()

At this point, an entry in the table of communications has been reserved for your application program. However, the socket has no port or IP address associated with it until you use the `bind()` function, which requires the following:

- The socket the server was just given
- The number of the port on which the server wishes to provide its service
- The IP address of the network connection on which the server is listening (to understand what is meant by "listening", see ["listen\(\)" on page 300](#))

In C language, the server puts the port number and IP address into a `sockaddr_in` structure, passing it and the socket to the `bind()` function. For example:

```
bind(s, (struct sockaddr *)&server, sizeof(struct sockaddr_in));
```

listen()

After the `bind`, the server has specified a particular IP address and port. Now it must notify the system that it intends to listen for connections on this socket. In C, the `listen()` function puts the socket into passive open mode and allocates a backlog queue of pending connections. In passive open mode, the socket is open for clients to contact. For example:

```
listen(s, backlog_number);
```

The server gives the socket on which it will be listening and the number of requests that can be queued (known as the *backlog_number*). If a connection request arrives before the server can process it, the request is queued until the server is ready.

accept()

Up to this point, the server has allocated a socket, bound the socket to an IP address and port, and issued a passive open. The next step is for the server actually to establish a connection with a client. The `accept()` call blocks the server until a connection request arrives, or, if there are connection requests in the backlog queue, until a connection is established with the first client in the queue. The following is an example of the `accept()` call:

```
client_sock = accept(s, &clientaddr, &addrlen);
```

The server passes its socket to the `accept()` call. When the connection is established, the `accept()` call returns a new socket representing the connection with the client. When the server wishes to communicate with the client or end the connection, it uses this new socket, `client_sock`. The original socket `s` is now ready to accept connections with other clients. The original socket is still allocated, bound, and opened passively. To accept another connection, the server calls `accept()` again. By repeatedly calling `accept()`, the server can establish almost any number of connections at once.

select()

The server is now ready to start handling requests on this port from any client with the server's IP address and port number. Up to this point, it has been assumed that the server will be handling only one socket.

However, an application program is not limited to one socket. Typically, a server listens for clients on a particular socket but allocates a new socket for each client it handles. For maximum performance, a server should operate only on those sockets that are ready for communication. The `select()` call allows an application program to test for activity on a group of sockets.

Note: The `select()` function can also be used with other descriptors, such as file descriptors, pipes, or character special files.

To allow you to test any number of sockets with just a single call to `select()`, place the sockets to test into a bit set, passing the bit set to the `select()` call. A *bit set* is a string of bits where each possible member of the set is represented by a 0 or a 1. If the member's bit is 0, the member is not in the set. If the member's bit is 1, the member is in the set. Sockets are actually small integers. If socket 3 is a member of a bit set, then the bit that represents it is set to 1 (on).

In C, the functions to manipulate the bit sets are the following:

FD_SET

Sets the bit corresponding to a socket

FD_ISSET

Tests whether the bit corresponding to a socket is set or cleared

FD_ZERO

Clears the whole bit set

FD_CLR

Clears a bit within the bit set

To be active, a socket is ready for reading data or for writing data, or an exceptional condition may have occurred. Therefore, the server can specify three bit sets of sockets in its call to the `select()` function: one bit set for sockets on which to receive data; another for sockets on which to write data; and any sockets with exception conditions. The `select()` call tests each socket in each bit set for activity and returns only those sockets that are active.

A server that processes many clients at the same time can easily be written so that it processes only those clients that are ready for activity.

The client perspective

The client first issues the `socket()` function call to allocate a socket on which to communicate:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

To connect to the server, the client places the port number and the IP address of the server into a `sockaddr_in` structure. If the client does not know the server's IP address, but does know the server's host name, the `gethostbyname()` function or the `getaddrinfo()` function is called to translate the host name into its IP address. The client then calls `connect()`. The following is an example of the `connect()` call:

```
connect(s, (struct sockaddr *)&server, sizeof(struct sockaddr_in));
```

When the connection is established, the client uses its socket to communicate with the server.

A typical TCP socket session

You can use TCP sockets for both passive (server) and active (client) processes. Whereas some functions are necessary for both types, some are role-specific. After you make a connection, it exists until one of the following has occurred:

- The socket is closed by client or server
- A shutdown is performed by client or server for both read and write
- The socket is *unconnected* using a blank `sockaddr` structure with another `connect()` call to the socket

During the connection, data is either delivered or an error code is returned by TCP/IP.

See Figure 87 on page 302 for the general sequence of calls to be followed for most socket routines using TCP, or stream sockets.

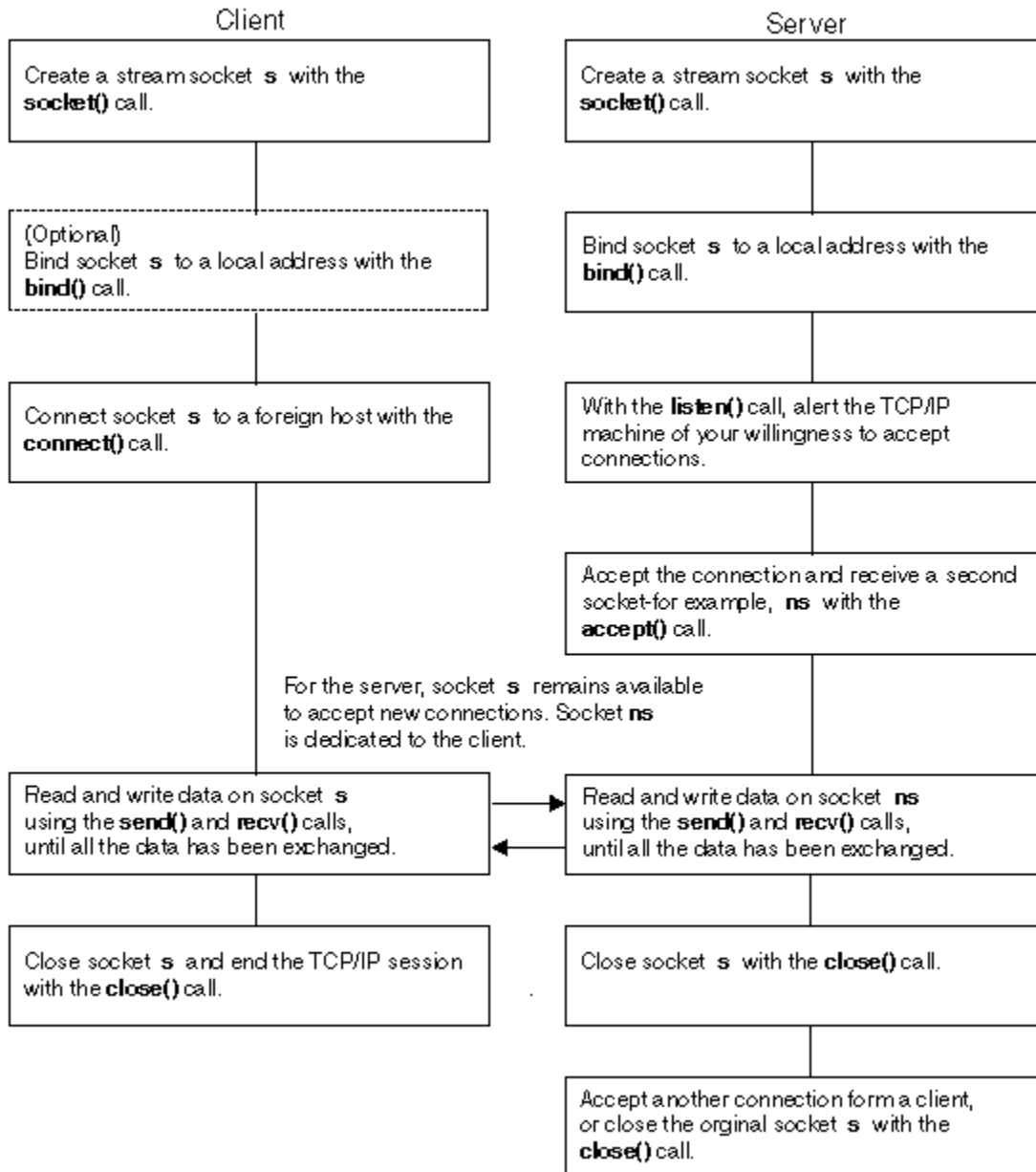


Figure 87. A typical stream socket session

A typical UDP socket session

User Datagram Protocol (UDP) socket processes, unlike TCP socket processes, are not clearly distinguished by server and client roles. The distinction is between connected and unconnected sockets. An unconnected socket can be used to communicate with any host; but a connected socket, because it has a dedicated destination, can send data to, and receive data from, only one host.

Both connected and unconnected sockets send their data over the network without verification. Consequently, after a packet has been accepted by the UDP interface, the arrival and integrity of the packet cannot be guaranteed.

See Figure 88 on page 303 for the general sequence of calls to be followed for most socket routines using UDP, or datagram, sockets.

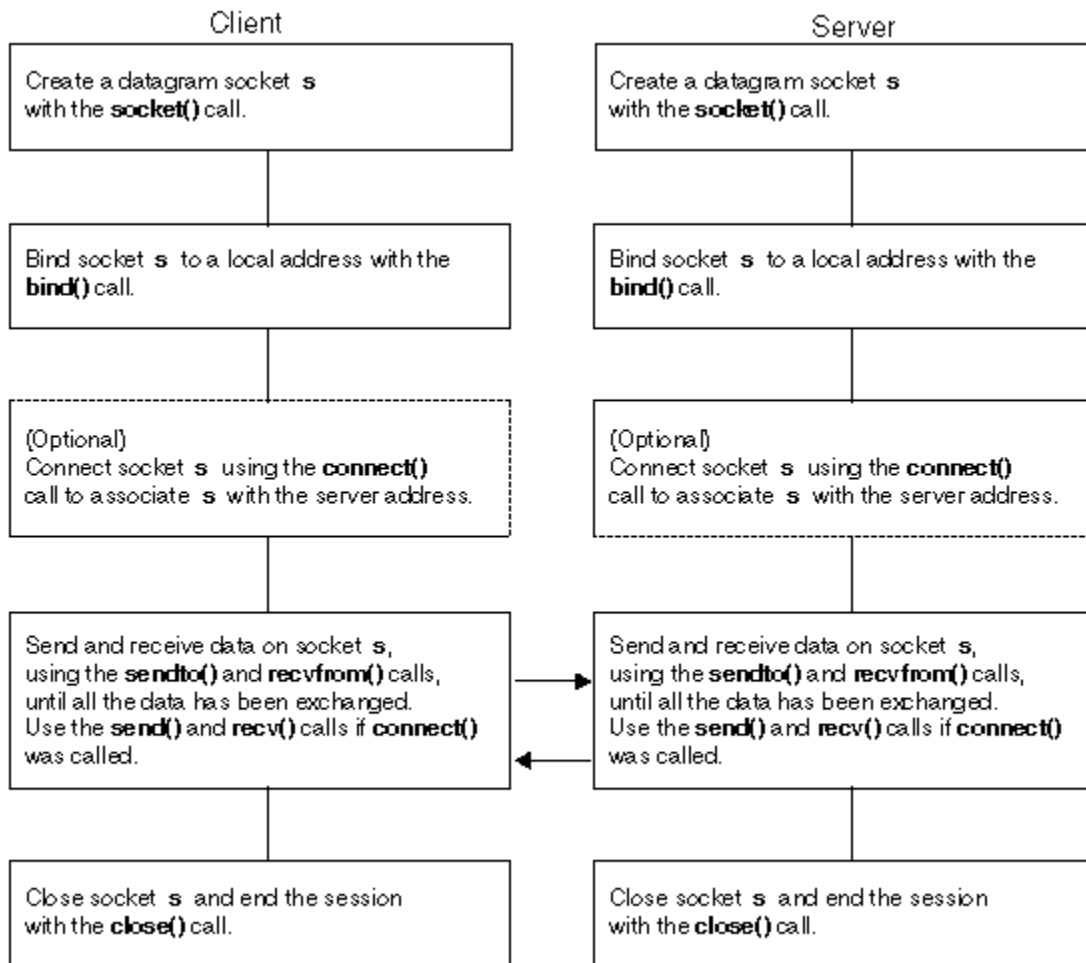


Figure 88. A typical datagram socket session

Locating the server's port

In the client/server model, the server provides a resource by listening for clients on a particular port. Such application programs as FTP, SMTP, and Telnet listen on a *well-known port*—a port assigned for use to a specific application program or protocol. However, for your own client/server application programs, you need a method of assigning port numbers to represent the services you intend to provide. An easy method of defining services and their ports is to enter them into the `/etc/services` file or the `tcpip.ETC.SERVICES` data set. In C, the programmer uses the `getservbyname()` function or `getaddrinfo()` function to determine the port for a particular service. If the port number for a particular service changes, only the `/etc/services` file or the `tcpip.ETC.SERVICES` data set must be modified.

Note: TCP/IP is shipped with a `tcpip.ETC.SERVICES` file containing such well-known services as FTP, SMTP, and Telnet.

Network application example

The following example illustrates using socket functions in a network application program. The steps are written using many of the basic socket functions, C socket syntax, and conventions described in this book.

1. First, an application program must get a socket descriptor using the `socket()` call, as in the example listed in [Figure 89 on page 304](#). For a complete description, see [z/OS C/C++ Runtime Library Reference](#).

```
#include <sys/socket.h>
...
int s;
...
s = socket(AF_INET, SOCK_STREAM, 0);
```

Figure 89. An application using `socket()`

The code fragment in [Figure 89](#) on page 304 allocates a socket descriptor `s` in the Internet address family. The *domain* parameter is a constant that specifies the domain where the communication is taking place. A *domain* is the collection of application programs using the same addressing convention. z/OS UNIX supports three domains: `AF_INET`, `AF_INET6`, and `AF_UNIX`. The *type* parameter is a constant that specifies the type of socket, which can be `SOCK_STREAM`, or `SOCK_DGRAM`.

The *protocol* parameter is a constant that specifies the protocol to use. For `AF_INET`, it can be set to `IPPROTO_UDP` for `SOCK_DGRAM` and `IPPROTO_TCP` for `SOCK_STREAM`. Passing 0 chooses the default protocol. If successful, the `socket()` call returns a positive integer socket descriptor. For `AF_UNIX`, the protocol parameter *must* be 0. These values are defined in the `netinet/in.h` include file.

2. After an application program has a socket descriptor, it can explicitly bind a unique address to the socket, as in the example listed in [Figure 90](#) on page 304. For a complete description, see [z/OS C/C++ Runtime Library Reference](#).
-

```
int bind(int s, struct sockaddr *name, int namelen);
...
int rc;
int s;
struct sockaddr_in myname;

/* clear the structure to be sure that the sin_zero field is clear */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1");
/* specific interface */
myname.sin_port = htons(1024);
:
rc = bind(s, (struct sockaddr *) &myname,
sizeof(myname));
```

Figure 90. An application using `bind()`

This example binds socket descriptor `s` to the address 129.5.24.1 and port 1024 in the Internet domain. Servers must bind to an address and port to become accessible to the network. The example in [Figure 90](#) on page 304 shows two useful utility routines:

- `inet_addr()` takes an IPv4 Internet address in dotted-decimal form and returns it in network byte order. Note that the `inet_pton()` function can take either an IPv4 or IPv6 Internet address in its standard text presentation form and return it in its numeric binary form. For a complete description, see [z/OS C/C++ Runtime Library Reference](#).
- `htons()` takes a port number in host byte order and returns the port in network byte order. For a complete description, see [z/OS C/C++ Runtime Library Reference](#).

[Figure 91](#) on page 305 shows another example of the `bind()` call. It uses the utility routine `gethostbyname()` to find the Internet address of the host, rather than using `inet_addr()` with a specific address.

```

int bind(int s, struct sockaddr_in name, int namelen);
:
int rc;
int s;
char *hostname = "myhost";
struct sockaddr_in myname;
struct hostent *hp;

    hp = gethostbyname(hostname);

    /*clear the structure to be sure that
the sin_zero field is clear*/
    memset(&myname,0,sizeof(myname));
    myname.sin_family = AF_INET;
    myname.sin_addr.s_addr = *((ip_addr_t
*)hp->h_addr);
    myname.sin_port = htons(1024);
:
rc = bind(s,(struct
sockaddr *) &myname, sizeof(myname));

```

Figure 91. A bind() function using gethostbyname()

3. After binding to a socket, a server that uses stream sockets must indicate its readiness to accept connections from clients. The server does this with the `listen()` call, as illustrated in the example in [Figure 92 on page 305](#).
-

```

int listen(int s, int backlog);
:
int s;
int rc;
:
rc = listen(s, 5);

```

Figure 92. An application using listen()

The `listen()` call tells the TCP/IP address space that the server is ready to begin accepting connections, and that a maximum of five connection requests can be queued for the server. Additional requests are ignored. For a complete description, see [z/OS C/C++ Runtime Library Reference](#).

4. Clients using stream sockets begin a connection request by calling `connect()`, as shown in [Figure 93 on page 305](#).
-

```

int connect(int s, struct sockaddr *name, int namelen);
:
int s;
struct sockaddr_in servername;
int rc;
:
memset(&servername, 0,sizeof(servername));
servername.sin_family = AF_INET;
servername.sin_addr = inet_addr("129.5.24.1");
servername.sin_port = htons(1024);
:
rc = connect(s, (struct sockaddr *) &servername,
sizeof(servername));

```

Figure 93. An application using connect()

The `connect()` call attempts to connect socket descriptor `s` to the server with an address `servername`. This could be the server that was used in the previous `bind()` example. The `connect` request is completed immediately and returns control to the caller, regardless of the server accepting the connection. After a successful return, the socket descriptor `s` is associated with the connection to the server. For a complete description, see [z/OS C/C++ Runtime Library Reference](#).

5. Servers using stream sockets accept a connection request with the `accept()` call, as shown in the example listed in [Figure 94 on page 306](#).

```

int accept(int s, struct sockaddr *addr, int *addrlen);
:
int clientsocket;
int s;
struct sockaddr clientaddress;
int addrlen;
:
addrlen = sizeof(clientaddress);
:
clientsocket = accept(s, &clientaddress, &addrlen);

```

Figure 94. An application using accept()

When a connection request is accepted on socket descriptor *s*, the name of the client and length of the client name are returned, along with a new socket descriptor. The new socket descriptor is associated with the client that began the connection, and *s* is again available to accept new connections. For a complete description, see [z/OS C/C++ Runtime Library Reference](#).

6. Clients and servers have many calls from which to choose for data transfer. The `read()` and `write()`, `readv()` and `writv()`, and `send()` and `recv()` calls can be used only on sockets that are in the connected state. The `sendto()` and `recvfrom()`, and `sendmsg()` and `recvmsg()` calls can be used at any time on datagram sockets. The example listed in [Figure 95 on page 306](#) illustrates the use of `send()` and `recv()`.
-

```

int send(int socket, char *buf, int buflen, int flags);
int recv(int socket, char *buf, int buflen, int flags);
:
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
int s;
:
bytes_sent = send(s, data_sent,
sizeof(data_sent), 0);
:
bytes_received = recv(s,
data_received, sizeof(data_received), 0);

```

Figure 95. An application using send() and recv()

The example in [Figure 95 on page 306](#) shows an application program sending data on a connected socket and receiving data in response. The `flags` field can be used to specify additional options to `send()` or `recv()`, such as sending out-of-band data. For more information see [z/OS C/C++ Runtime Library Reference](#).

7. If the socket is not in a connected state, additional address information must be passed to `sendto()` and can be optionally returned from `recvfrom()`. An example of the use of the `sendto()` and `recvfrom()` calls is listed in [Figure 96 on page 307](#).

```

int sendto(int socket, char *buf, int buflen, int flags,
           struct sockaddr *addr, int addrlen);
int recvfrom(int socket, char *buf, int buflen, int flags,
             struct sockaddr *addr, int *addrlen);
:
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
struct sockaddr_in to;
struct sockaddr from;
int addrlen;
int s;
:
memset(&to, 0, sizeof(to));
to.sin_family = AF_INET;
to.sin_addr = inet_addr("129.5.24.1");
to.sin_port = htons(1024);
:
bytes_sent = sendto(s, data_sent,
                   sizeof(data_sent), 0, &to, sizeof(to));
:
addrlen = sizeof(from); /* must be initialized */
bytes_received = recvfrom(s, data_received,
                          sizeof(data_received), 0, &from, &addrlen);

```

Figure 96. An application using `sendto()` and `recvfrom()`

The `sendto()` and `recvfrom()` calls take additional parameters that allow the caller to specify the recipient of the data or to be notified of the sender of the data. For more information see [z/OS C/C++ Runtime Library Reference](#). Usually, `sendto()` and `recvfrom()` are used for datagram sockets, and `send()` and `recv()` are used for stream sockets.

8. The `writev()`, `readv()`, `sendmsg()`, and `recvmsg()` calls provide the additional features of *scatter and gather data*—two related operations where data is received and stored in multiple buffers (scatter data), and then taken from multiple buffers and transmitted (gather data). Scattered data can reside in multiple data buffers. The `writev()` and `sendmsg()` calls gather the scattered data and send it. The `readv()` and `recvmsg()` calls receive data and scatter it into multiple buffers.
 9. Applications can handle multiple descriptors. In such situations, use the `select()` call to determine the descriptors that have data to be read, those that are ready for data to be written, and those that have pending exceptional conditions. An example of how the `select()` call is used is listed in [Figure 97 on page 307](#).
-

```

fd_set readsocks;
fd_set writesocks;
fd_set exceptsocks;
struct timeval timeout;
int number_of_sockets;
int number_found;
:
/* number_of_sockets previously set to the socket number of largest
 * integer value.
 * Clear masks out.
 */
FD_ZERO(&readsocks); FD_ZERO(&writesocks); FD_ZERO(&exceptsocks);
/* Set masks for socket s only */
FD_SET(s, &readsocks)
FD_SET(s, &writesocks)
FD_SET(s, &exceptsocks)
:
/* go into select wait for 5 minutes waiting for socket s to become
ready or the timer has popped*/
rc = select(number_of_sockets+1,
           &readsocks, &writesocks, &exceptsocks, &timeout);
:
/* Check rc for condition set upon exiting select */
number_found = select(number_of_sockets,
                     &readsocks, &writesocks, &exceptsocks, &timeout);

```

Figure 97. An application using `select()`

In this example, the application program uses bit sets to indicate that the sockets are being tested for certain conditions and also indicates a timeout. If the timeout parameter is NULL, the `select()` call blocks until a socket becomes ready. If the timeout parameter is nonzero, `select()` waits up to this amount of time for at least one socket to become ready on the indicated conditions. This is useful for application programs servicing multiple connections that cannot afford to block, waiting for data on one connection. For a complete description, see [z/OS C/C++ Runtime Library Reference](#).

10. In addition to `select()`, application programs can use the `ioctl()` or `fcntl()` calls to help perform asynchronous (nonblocking) socket operations. An example of the use of the `ioctl()` call is listed in [Figure 98 on page 308](#).

```
int ioctl(int s, unsigned long command, char *command_data);
...
int s;
int dontblock;
char buf[256];
int rc;
...
dontblock = 1;
...
rc = ioctl(s, FIONBIO, (char *) &dontblock);
...
if (((rc=recv(s, buf, sizeof(buf),
0)) < 0)&&(errno == EWOULDBLOCK))
/* no data available */
else
/* either got data or some other error occurred */
```

Figure 98. An Application Using `ioctl()`

This example causes the socket descriptor `s` to be placed into nonblocking mode. When this socket is passed as a parameter to calls that would block, such as `recv()` when data is not present, it causes the call to return with an error code, and the global `errno` value is set to `EWOULDBLOCK`. Setting the mode of the socket to be nonblocking allows an application program to continue processing without becoming blocked. For a complete description, see [z/OS C/C++ Runtime Library Reference](#).

11. A socket descriptor, `s`, is deallocated with the `close()` call. [Figure 99 on page 308](#) shows an example. For a complete description, see [z/OS C/C++ Runtime Library Reference](#).

```
int close(int s);
...
int rc;
int s;
rc = close(s);
```

Figure 99. An application using `close()`

Using common INET

With Common INET (CINET), you have the capability to define up to 32 `AF_INET` or dual `AF_INET/AF_INET6` transport providers or stacks. The stacks can all be active at the same time. The information for modifying `BPXPRMxx` and bringing up Common INET is in [z/OS UNIX System Services Planning](#).

For a server that you want to be able to listen to all of the available stacks at the same time, specify `INADDR_ANY` and it will be listening to all at once. Note that for an IPv6 server, `IN6ADDR_ANY` can be specified allowing the server to listen for IPv4 and IPv6 connections from all stacks.

The z/OS UNIX Common INET layer performs a multiplexing/demultiplexing function when more than one stack is activated under z/OS UNIX. Each stack has its own home IP addresses and when a program binds to a specific IP address that socket becomes associated with the one stack that is that IP address. When a program binds to `NADDR_ANY (0.0.0.0)` or `IN6ADDR_ANY (: :)`, the socket remains available to all the stacks.

There are three ways that an `INADDR_ANY` or `IN6ADDR_ANY` program can associate itself with a single stack:

- Call `setibmopt(IBM_TCP_IMAGE)` - This sets a process so all future `socket()` calls create sockets with only the one specified stack.
- The `_BPXK_SETIBMOPT_TRANSPORT` environment variable can be used in the `PARM=` parameter of an MVS started proc to effectively issue a `SETIBMOPT` outside of the program.
- Call `ioctl(SIOCSETRTTD)` - This associates an existing socket with the one specified stack, removing the others.

Also, you should be able to set up things so `gethostbyname()` or `getaddrinfo()` returns the home IP address of the local TCP/IP you are interested. With that, you can issue a specific `bind()` to that IP address. This may not be useful though, if that stack has multiple IP addresses and you really want to use `INADDR_ANY` to service all of them. Applications can bind to `IN6ADDR_ANY` to service both IPv4 and IPv6 clients when TCP/IP is enabled for IPv6.

Compiling and binding

This section describes how to bind, load, and run z/OS XL C programs containing z/OS UNIX sockets. This information is specific to the z/OS UNIX application program interface and assumes that you are familiar with the information on compiling and binding z/OS UNIX application programs in [z/OS XL C/C++ Programming Guide](#) and [z/OS Language Environment Programming Guide](#). C++ programs can also use z/OS UNIX sockets, but C++ programs cannot use Berkley Sockets, they must always use X/Open Sockets.

You compile and bind your sockets application program in the same way as for any other C language program. The process is shown conceptually in [Figure 100 on page 309](#). You must make sure that the z/OS UNIX socket application programs have access to the files they need to compile and bind.

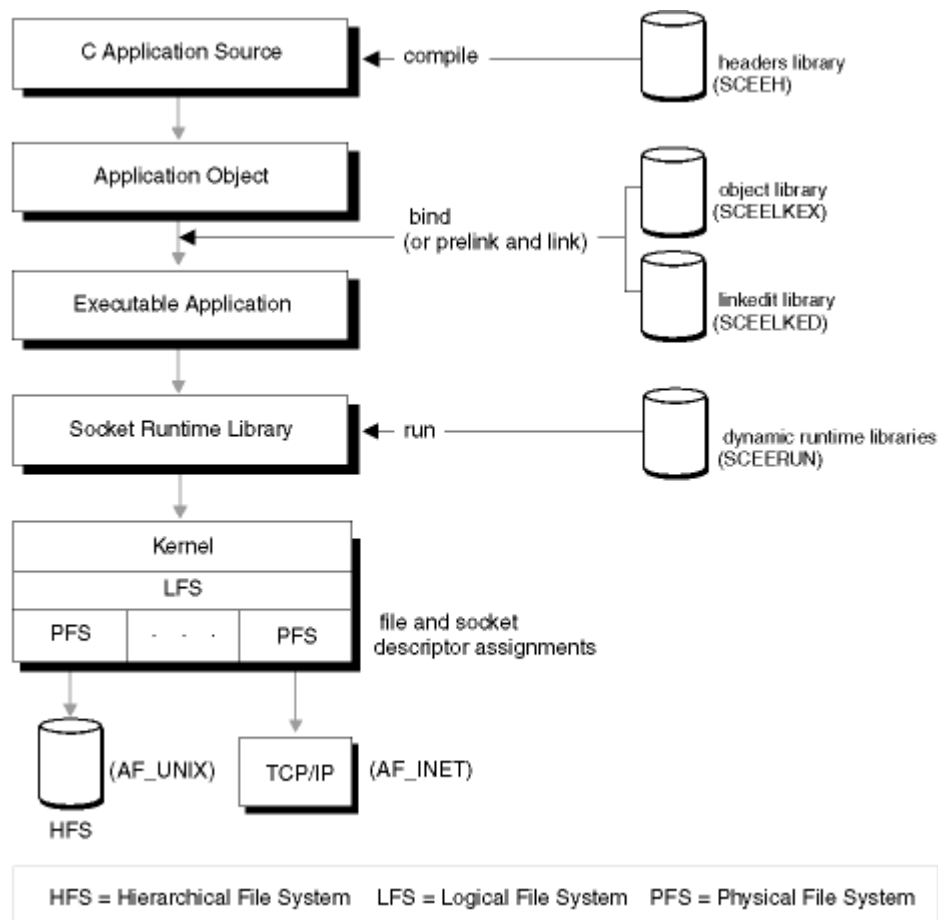


Figure 100. A conceptual overview of the compile, bind, and run steps

As shown, whether an application program's I/O request is targeted at the network (TCP/IP) or at a file, the z/OS UNIX logical file system (LFS) will route the request to the appropriate physical file system (PFS).

If your C language statements contain information, such as sequence numbers, which are not part of the input for the z/OS XL C compiler, you must include the following pragma directive in your program:

```
#pragma margins(1,72)
```

Note: In order to use AF_INET sockets, you must have release 3.1 or a later level of TCP/IP installed on your system. In order to use AF_INET6 sockets, you must have release z/OS V1R4 or later of TCP/IP installed on your system.

Using TCP/IP APIs

If you will be using the TCP/IP socket API, also called non-Berkeley sockets, you will need to read and understand this section.

When a XL C/C++ application program running under z/OS UNIX needs to communicate with another program that is running simultaneously, it needs to exploit, from within itself, both z/OS UNIX POSIX.1 and one or more of the following application programming interfaces (APIs) provided with the IBM product TCP/IP:

- Socket APIs
 - C sockets
 - Inter-User Communication Vehicle (IUCV) sockets
- X Window System interface
- remote procedure call (RPC)

With the exception of described restrictions, you can code z/OS UNIX XL C/C++ application programs to take advantage of the documented APIs available as part of the Communications Server IP.

z/OS UNIX application programs can use socket API calls from the TCP/IP product to access UNIX file system files or MVS data sets, communicate with other systems running TCP/IP, or establish communication with and request services from a workstation system acting as an X Windows server.

Note: For UNIX file system file access to TCP/IP, the TCP/IP socket API calls must be used instead of the POSIX file access functions to preserve the uniqueness of file descriptors in the UNIX file system.

Before you attempt to code your application program to use TCP/IP APIs, you should understand the X Windows protocol running on the workstations that will be used as application clients. You will also need to know how to invoke X Windows to create a connection to the server on the workstation or z/OS system.

Restrictions for using z/OS TCP/IP API with z/OS UNIX

The restrictions can be grouped into categories:

• Header Files

- *Header file conflicts between TCP/IP and z/OS XL C/C++.* z/OS XL C/C++ and TCP/IP have header files with the same name and overlapping function. For example, both have a `types.h` file. If you use TCP/IP API functions in your application but the z/OS XL C/C++ header file is searched for and used, the TCP/IP function does not work as intended.

You can circumvent this problem by developing your application program with separate compilation source files for TCP/IP function and normal z/OS XL C/C++ function. You can then compile the TCP/IP source files separately from the normal z/OS XL C/C++ source files. Use the `c89 -I` option to point to the MVS data sets to search for the TCP/IP header files. Finally, you can bind all the application object files together to produce the application executable file. For the bind step, use the `c89 -l` option to point to the correct TCP/IP libraries on MVS. For example:

```
c89 -I "///'tcpip.sezacmac'" pgm.c -l "///'tcpip.sezarnt1'" ...
```

- **TCP/IP socket API.** Both z/OS UNIX POSIX.1-defined support and the TCP/IP for z/OS socket API use a small subset of common function calls that cannot be resolved correctly between them:

- close()
- fcntl()
- read()
- write()

Use of these calls should be reserved for one or the other, but not both, of these programming interfaces. For example, if an application program is written to use the open(), close(), read(), and write() functions for z/OS TCP/IP socket communication, it cannot use them for UNIX file system file access. z/OS XL C/C++ stream I/O functions (fopen(), fclose(), fread(), and fwrite()) must be used for UNIX file system file access. See *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference* for more information.

- **Creating child processes.** Generally speaking, an application program cannot have a parent process open resources—in this case sockets—and then support those resources for a child process created through a fork() function or in a process following use of an exec function. The new child process does not inherit sockets from the parent process if forked. If the child process needs sockets, it must request TCP/IP for z/OS socket support independently of the parent process. In fact, if a child process is to be forked by an application program using TCP/IP sockets under z/OS UNIX, all MVS resources to be opened *should* be opened by the child process rather than by the parent process.
- **TCP/IP configuration file access.** An application executable file that uses TCP/IP APIs and was bound with the c89 utility cannot locate the necessary TCP/IP configuration files, because they reside in MVS sequential data sets rather than in UNIX file system files.

To circumvent this problem, have the system programmer copy the TCP/IP configuration data sets into the root directory exactly as shown:

```
OPUT 'tcpip.tcpip.data' 'etc/resolv.conf' text
```

Copy the address of the name server, the name, and the domain name from *tcpip.HOST.LOCAL* to *|etc|hosts*. You should not copy the entire file directly because you only need the address and name. The entry in the *|etc|hosts* file follows the BSD format. The case of the filenames and the use of the quote characters as part of the name are *significant*. Use the TSO/E OPUT command to copy the MVS sequential data sets to the root directory. (Placing files in the root file system requires superuser authority.)

- **Program reentrancy.** The TCP/IP sockets and X Windows reentrant libraries must have a special C370LIB-directory member created for them before an application program using TCP/IP functions can be bound. The system administrator must run the C370LIB DIR function against the reentrant libraries to create it. The system administrator must do this once per library for an MVS system.

Specify the TCP/IP libraries to search on the c89 utility when binding the application program. For example:

```
c89 -I"//'tcpip.sezacmac'" pgm.c -l "//'tcpip.sezarnt1'" ...
```

For information on C370LIB, see *z/OS XL C/C++ User's Guide*.

Using z/OS UNIX sockets

To compile, each z/OS UNIX socket application program must have access to the following **z/OS** C include files in an MVS PDS or in the UNIX file system directory:

CEE.SCEEH.H	/usr/include
CEE.SCEEH.ARPA.H	/usr/include/arpa
CEE.SCEEH.NET.H	/usr/include/net
CEE.SCEEH.NETINET.H	/usr/include/netinet
CEE.SCEEH.SYS.H	/usr/include/sys

Note: The data set prefix for each of the previous files must match the name used at your installation. CEE is the default for z/OS Language Environment.

For **Berkeley SOCKETS** or **X/OPEN SOCKETS**, all you need are the z/OS C include files.

Note: The data set prefix for each of these files must match the name used at your installation. CEE is the default for the z/OS XL C library.

You must compile your application program using *all* include files in order to access the entire z/OS UNIX socket API. To compile a program written using a particular API, you must include certain files specific to that API even though your program may not require all of them.

See *z/OS C/C++ Runtime Library Reference*, which lists the header files that must be included for each type API. They may be different for **Berkeley Sockets** and **X/Open sockets**.

The following list describes the files that each z/OS UNIX socket application program must have access to in order to bind:

- CEE.SCEELKED contains stub routines in the link library that are used to resolve external references to z/OS XL C and z/OS UNIX socket APIs.
- CEE.SCEELKEX contains LONGNAME stub routine object modules for a large portion of the Language Environment function library, including the z/OS C and z/OS UNIX socket APIs. When you IPA Link your application program, place the SCEELKEX library ahead of the SCEELKED Load Module library in the search order. This preserves long runtime function names in the object module and listings generated by IPA Link. When you bind your application program, place the SCEELKEX library ahead of the SCEELKED Load Module library in the search order. This preserves long runtime function names in the executable module and listings generated by the binder.
- CEE.SCEERUN contains the z/OS XL C and z/OS UNIX socket runtime libraries.

Compiling under MVS batch for Berkeley sockets

You can use several methods to compile, bind, and run your sockets program. This section describes one way to compile and bind your C source program, under MVS batch, using the IBM-supplied EDCCB cataloged procedure.

Note: If you are planning on developing your application as a C++ application and use sockets, you must use XOpen Sockets for your application. See section [“Compiling under MVS batch for X/Open sockets”](#) on page 313 for more information.

Sample cataloged procedure additions and changes

The following steps describe how to compile, and bind your program. For more information about the z/OS XL C/C++ cataloged procedures refer to the *z/OS XL C/C++ User's Guide*.

You must make changes to the cataloged procedure, which is supplied with z/OS XL C/C++ Compiler. After you select the procedure you want to use from those available in the XL C/C++ supplied data set, CBC.SCCNPRC, you modify it. For example, if you choose EDCC then you modify it as follows:

1. Change the CPARM parameters to:

```
CPARM=' DEF (MVS, _OE_SOCKETS, _POSIX1_SOURCE=1),RENT,LO',
```

RENT is the reentrant option and LO is the long name option. You must specify these options to use POSIX functions `read()`, `write()`, `fcntl()`, and `close()` that are all included in z/OS XL C.

You must specify the feature test macro, `_POSIX1_SOURCE=1` to access the `read()`, `write()`, `fcntl()`, and `close()` functions in the z/OS XL C include files. Or, if you choose to access all z/OS UNIX POSIX functions supported by z/OS XL C, you can specify the `_OPEN_SYS` feature test macro. The `_OE_Sockets` feature test macro exposes the socket-related definitions in all of the include files. For information on binding C code compiled with the RENT and LONGNAME options, see *z/OS XL C/C++ User's Guide*.

2. To run your program under TSO/E, type the following:

```
CALL 'USER.MYPROG.LOAD(PROGRAM1)' 'POSIX(ON)'
```

This loads the runtime library from CEE.SCEERUN and/or SCEERUN2.

To use the POSIX z/OS XL C functions, you *must* either specify the runtime option POSIX(ON), or include the following statement in your C source program:

```
#pragma runopts(POSIX(ON))
```

The *z/OS C/C++ Runtime Library Reference* identifies the POSIX z/OS XL C functions, in the standards information at the beginning of each function description.

Compiling under MVS batch with X windows for Berkeley sockets

If you are using z/OS UNIX sockets with the latest announced release level of TCP/IP X Windows, and compiling and binding under MVS batch, you *must* bind your application program with the latest announced release level of TCP/IP X Windows libraries that are enabled for use with z/OS UNIX sockets.

For a complete discussion of compiling and binding z/OS UNIX sockets with TCP/IP, see [z/OS Communications Server: IP Programmer's Guide and Reference](#).

Compiling using the c89 utility for Berkeley sockets

If you want to use the c89 utility to compile and bind your program, you must use the following define options on the c89 command:

```
-D MVS  
-D _OE_SOCKETS
```

For more information about compiling and binding, see [z/OS XL C/C++ User's Guide](#).

Compiling using c89 with X Windows

See [z/OS Communications Server: IP Programmer's Guide and Reference](#) for a complete discussion of compiling and binding with X Windows.

Compiling under MVS batch for X/Open sockets

You can use several methods to compile, bind, and run your sockets program. This section describes one way to compile and link-edit your C source program, under MVS batch, using the IBM-supplied EDCCB cataloged procedure.

Sample cataloged procedure additions and changes

The following steps describe how to compile, bind, and run your program. For more information about the z/OS XL C/C++ cataloged procedures refer to the [z/OS XL C/C++ User's Guide](#).

You must make changes to the cataloged procedure, which is supplied with z/OS XL C/C++ Compiler. After you select the procedure you want to use from those available in the XL C/C++ supplied data set, CBC.SCCNPRC, you modify it. For example, if you choose EDCCB then you modify it as follows:

1. Change the CPARM parameters to:

```
CPARM=' DEF(MVS,_XOPEN_SOURCE_EXTENDED=1,_POSIX1_SOURCE=1),  
RENT,LO',
```

RENT is the reentrant option and LO is the long name option. You must specify these options to use POSIX functions read(), write(), fcntl(), and close() that are all included in z/OS XL C.

You must specify the feature test macro, _POSIX1_SOURCE=1 to access the read(), write(), fcntl(), and close() functions in the z/OS XL C include files. Or, if you choose to access all z/OS UNIX POSIX functions supported by z/OS XL C, you can specify the _OPEN_SYS feature test macro. The _XOPEN_SOURCE_EXTENDED feature test macro exposes the socket-related definitions in all of the include files.

Note: Because you are now required to compile with the RENT and LONGNAME options, you must bind your sockets application with the z/OS binder.

2. To run your program under TSO/E, type the following:

```
CALL 'USER.MYPROG.LOAD(PROGRAM1)' 'POSIX(ON)'
```

To use the POSIX z/OS XL C functions, you *must* either specify the runtime option `POSIX(ON)`, or include the following statement in your C source program:

```
#pragma runopts(POSIX(ON))
```

Using API data sets and files for sockets

- CEE.SCEELKED contains stub routines in the link library that are used to resolve external references to z/OS XL C and z/OS UNIX socket APIs.
- CEE.SCEELKEX contains LONGNAME stub routine object modules for a large portion of the Language Environment function library, including the z/OS C and z/OS UNIX socket APIs. When you IPA Link or bind your application program, place the SCEELKEX library ahead of the SCEELKED Load Module library in the search order. This preserves long runtime function names in the object module and listings generated by IPA Link or the binder.
- CEE.SCEERUN contains the z/OS XL C and z/OS UNIX socket runtime libraries.

Notes:

1. The data set prefix for each the previous files must match the name used at your installation. CEE is the default for z/OS Language Environment.
2. Applications developed for Open Sockets can continue to use the linkage editor but cannot be compiled.

Understanding the X/Open Transport Interface (XTI)

The X/Open Transport Interface (XTI) specification defines an independent transport-service interface that allows multiple users to communicate at the transport level of the OSI reference model. Transport-layer protocols support the following characteristics:

- connection establishment
- state change support
- event handling
- data transfer
- option manipulation

Although all transport-layer protocols support these characteristics, they vary in their level of support and their interpretation of format.

Transport endpoints

A `transport` endpoint specifies a communication path between a transport user and a specific transport provider, which is identified by a local file descriptor (`fd`). When a user opens a transport endpoint, a local file descriptor `fd` is returned which identifies the endpoint. A transport provider is defined to be the transport protocol that provides the services of the transport layer. All requests to the transport provider must pass through a transport endpoint. The file descriptor `fd` is returned by the function `t_open()` and is used as an argument to the subsequent functions to identify the transport endpoint. A transport endpoint can support only one established transport connection at a time.

To be active, a transport endpoint must have a transport address associated with it by the `t_bind()` function. A transport connection is characterized by the association of two active endpoints, made by using the transport connection establishment functions `t_listen()`, `t_accept()`, `t_connect()`, and `t_rcvconnect()`.

Transport providers for X/Open Transport Interface

The transport layer may comprise one or more transport providers at the same time. The identifier parameter of the transport provider passed to the `t_open()` function determines the required transport provider. To keep the applications portable, the identifier parameter of the transport provider should not be hard-coded into the application source code.

Currently, the only valid value for the *identifier* parameter for the `t_open()` function is `/dev/tcp`, indicating the TCP transport provider. Even though no device with this pathname actually exists, the library uses this value to determine which transport provider to use.

General restrictions for z/OS UNIX

The following restrictions apply when you use XTI under z/OS UNIX.

- The file descriptor number must not exceed the limit of 65535 for XTI endpoints.
- If an endpoint is being shared among multiple processes, events such as, `T_LISTEN`, `T_DATA`, and `T_EXDATA`, can be consumed by another process in the time between calls to `t_look()` and `t_rcv()` or `t_accept()`. In order to avoid processes not being aware of events occurring on endpoints, you should provide explicit synchronization mechanisms between processes
- If an endpoint is shared:
 - The process that issues the `t_listen()` should also issue for the pending connection `t_accept()`.
 - If any other process accesses the endpoint in the time between the listen and the accept, the behavior is undefined. In order to avoid this, you should provide explicit synchronization between processes.
- If a process dies while an endpoint it was accessing is in `T_INCON` state, it is impossible for any other sharing endpoints to bring it out of that state.
- If access to endpoints is shared, the participating processes are responsible for serialization of access to the endpoints. If no synchronization is performed, the behavior is undefined.
- Functions are thread-safed; therefore, no two threads in a process can manipulate an endpoint at the same time. Serialization of access to endpoints beyond this level is the responsibility of the threads sharing the endpoint.

Chapter 26. Interprocess communication using z/OS UNIX

z/OS UNIX offers software vendors and customers several ways for programming processes to communicate:

- Message queues
- Semaphores
- Shared memory
- Memory mapping
- Issuing TSO commands from a shell

These forms of interprocess communication extend the possibilities provided by the simpler forms of communication: pipes, named pipes or FIFOs, signals, and sockets. Like these forms, message queues, semaphores, and shared memory are used for communication between processes. (Sockets are the most common form of interprocess communication across different systems.) For more information on these communication forms, see [z/OS UNIX System Services Planning](#).

Message queues

XPG4 provides a set of C functions that allow processes to communicate through one or more message queues in an operating system's kernel. A process can create, read from, or write to a message queue. Each message is identified with a "type" number, a length value, and data (if the length is greater than 0).

A message can be read from a queue based on its type rather than on its order of arrival. Multiple processes can share the same queue. For example, a server process can handle messages from a number of client processes and associate a particular message type with a particular client process. Or the message type can be used to assign a priority in which a message should be dequeued and handled.

A common client/server implementation on the same system uses two message queues for communication between client and server. An inbound message queue allows group write access and limits read access to the server. An outbound message queue allows universal read access and limits write access to the server. This implementation allows users to place invalid messages on the inbound queue or remove messages belonging to another process from the outbound queue. To solve this problem, you can use two new z/OS message queue types, `ipc_SndTypePID` and `ipc_RcvTypePID` to enforce source and destination process identification.

Create the inbound queue to the server with `ipc_SndTypePID` and the outbound queue from the server with `ipc_RcvTypePID`. This arrangement guarantees that the server knows the process ID of the client, and that the client is the only process that can receive the server's returned message. The server can also issue `msgrcv()` with `TYPE=0` to see if any messages belong to process IDs that have gone away. Security checks on clients are not needed, since clients are unable to receive messages intended for another process.

The `ipc_PL0` constants provide possible message queue performance improvements based on workload. For information on the `ipc_PL0` constants, see the `msgget()` function in the [z/OS C/C++ Runtime Library Reference](#).

Semaphores

Semaphores, unlike message queues and pipes, are not used for exchanging data, but as a means of synchronizing operations among processes. A semaphore value is stored in the kernel and then set, read, and reset by sharing processes according to some defined scheme. A semaphore is created or an existing one is located with the `semget()` function. Typical uses include resource counting, file locking, and the serialization of shared memory.

A semaphore can have a single value or a set of values; each value can be binary (0 or 1) or a larger value, depending on the implementation. For each value in a set, the kernel keeps track of the process ID that did the last operation on that value, the number of processes waiting for the value to increase, and the number of processes waiting for the value to become 0.

If you define a semaphore set without any special flags, `semop()` processing obtains a kernel latch to serialize the semaphore set for each `semop()` or `semctl()` call. The more semaphores you define in the semaphore set, the higher the probability that you will experience contention on the semaphore latch. One alternative is to define multiple semaphore sets with fewer semaphores in each set. To get the least amount of latch contention, define a single semaphore in each semaphore set.

z/OS has added the `__IPC_BINSEM` option to `semget()`. The `__IPC_BINSEM` option provides significant performance improvement on `semop()` processing. `__IPC_BINSEM` can only be specified if you use the semaphore as a binary semaphore and do not specify `UNDO` on any `semop()` calls. `__IPC_BINSEM` also allows `semop()` to use special hardware instructions to further reduce contention. With `__IPC_BINSEM`, you can define many semaphores in a semaphore set without impacting performance.

Shared memory

Shared memory provides an efficient way for multiple processes to share data (for example, control information that all processes require access to). Commonly, the processes use semaphores to take turns getting access to the shared memory. For example, a server process can use a semaphore to lock a shared memory area, then update the area with new control information, use a semaphore to unlock the shared memory area, and then notify sharing processes. Each client process sharing the information can then use a semaphore to lock the area, read it, and then unlock it again for access by other sharing processes.

Processes can also use shared mutexes and shared read-write locks to communicate. For more information on mutexes and read-write locks see [“Synchronization primitives” on page 248](#).

Memory mapping

In z/OS, a programmer can arrange to transparently map into a UNIX file system file process storage.

The use of memory mapping can reduce the number of disk accesses required when randomly accessing a file.

The related `mmap()`, `mprotect()`, `msync()`, and `munmap()` functions that provide memory mapping are part of the X/OPEN CAE Specification.

TSO commands from a shell

In z/OS UNIX, users of the z/OS UNIX shells can issue TSO/E commands. The user simply enters the shell command `tso`, followed by a TSO command string. The user can specify whether the TSO command is to be run through the shell (in which case the output will be displayed on the screen) or through a TSO environment (in which case the command output will be written to the defined standard output). For more information about running the command through the shell or through a TSO environment, see [z/OS UNIX System Services Command Reference](#).

Chapter 27. Using templates in C++ programs

In C++, you can use a template to declare and define a set of related:

- Classes (including structs)
- Functions
- Static data members of template classes

Within an application, you can instantiate the same template multiple times with the same arguments or with different arguments. If you use the same arguments, the repeated instantiations are redundant. These redundant instantiations increase compilation time, increase the size of the executable, and deliver no benefit.

There are several basic approaches to the problem of redundant instantiations:

Control implicit instantiation in the source code

To use this approach, you can use either of the following methods:

- Concentrate implicit instantiations of a specialization

Organize your source code so that object files contain fewer instances of each required instantiation and fewer unused instantiations. This is the least usable approach, because you must know where each template is defined and where each template instantiation is required.

- Use explicit instantiation declarations (C++11 only)

Support for explicit instantiation declarations can be enabled by setting the `LANGVL (EXTENDED)` or `LANGVL (EXTENDED0X)` compiler group suboptions. Explicit instantiation declarations give you the ability to suppress implicit instantiation of templates. This helps reduce the collective size of the object files. It may also reduce the size of the final executable if the suppressed symbol definitions are meant to be found in a shared library, or if the system linker is unable to always remove additional definitions of a symbol. This approach is described in [“Using explicit instantiation declarations \(C++11 only\)”](#) on page 324.

Store instantiations in an include directory

Use the `TEMPINC` compiler option. If the template header and the template definition file have the required structure (described in [“Using the TEMPINC compiler option”](#) on page 319), each template instantiation is stored in a template include directory. If the compiler is asked to instantiate the same template again with the same arguments, it uses the stored version instead. This is the default.

Store instantiation information in a registry

Use the `TEMPLATEREGISTRY` compiler option. Information about each template instantiation is stored in a template registry. If the compiler is asked to instantiate the same template again with the same arguments, it points to the instantiation in the first object file instead.

The `TEMPLATEREGISTRY` compiler option provides the benefits of the `TEMPINC` compiler option but does not require a specific structure for the template header and the template definition file.

Note: The `NOTEMPINC` and `TEMPLATEREGISTRY` compiler options are mutually exclusive.

Using the TEMPINC compiler option

To use `TEMPINC`, you must structure your application as follows:

- Declare your class templates and function templates in template declaration files. This file must have the same file name as the template definition file and an extension or LLQ of `.h`. In the following example, the template declaration file is named `stack.h`.

You can identify a template declaration file in either of the following ways:

- In the UNIX file system: `/usr/src/stack.h`
- In a PDS: `MYUSERID.USER.H(STACK)`
- For each template declaration file, create a template definition file. This file must have the same file name as the template declaration file and an extension or LLQ of `.c`. For a class template, this file defines all of the member functions and static data members. For a function template, this file defines the function.

You can identify a template definition file in either of the following ways:

- In the UNIX file system: `/usr/src/stack.c`
- In a PDS: `MYUSERID.USER.C(STACK)`
- In your source program, specify an `#include` statement for each template declaration file.
- In each template declaration file, conditionally include the corresponding template definition file if the `__TEMPINC__` macro is *not* defined.

This produces the following results:

- Whenever you compile with `NOTEMPINC`, the template definition file is included.
- Whenever you compile with `TEMPINC`, the compiler does not include the template definition file. Instead, the compiler looks for a file with the same name as the template declaration file and extension `.c` the first time it needs a particular instantiation. If the compiler subsequently needs the same instantiation, it uses the copy stored in the template include directory.

TEMPINC example

This section contains example files and compilation code examples that show how to use the `TEMPINC` compiler option. The following types of files are shown:

- Two source files: `stackadd.cpp` and `stackops.cpp`
- A template declaration file: `stack.h`
- The corresponding template definition file: `stack.c`
- A function prototype: `stackops.h`

In this example section, note that:

1. Both source files include the template declaration file `stack.h`
2. Both source files include the function prototype `stackops.h`
3. The template declaration file conditionally includes the template definition file `stack.c` if it is compiled with `NOTEMPINC`.

[Figure 101 on page 321](#) shows the first source file, `stackadd.cpp`.

```

#include "stack.h"           // 1
#include "stackops.h"        // 2
#include <iostream>
using namespace std;

main() {
    Stack<int, 50> s;         // create a stack of ints
    int left=10, right=20;
    int sum;

    s.push(left);            // push 10 on the stack
    s.push(right);           // push 20 on the stack
    add(s);                  // pop the 2 numbers off the stack
                             // and push the sum onto the stack
    sum = s.pop();           // pop the sum off the stack

    cout << "The sum of: " << left << " and: " << right << " is: " << sum << endl;

    return(0);
}

```

Figure 101. *stackadd.cpp* file (*ccntmp3.cpp*)

Figure 102 on page 321 is the source file, *stackops.cpp*.

```

#include "stack.h"           // 1
#include "stackops.h"        // 2

void add(Stack<int, 50>& s) {
    int tot = s.pop() + s.pop();
    s.push(tot);
    return;
}

```

Figure 102. *stackops.cpp* file (*ccntmp4.cpp*)

Figure 103 on page 321 shows *stack.h*, which is the template declaration file.

```

#ifndef STACK_H
#define STACK_H

template <class Item, int size> class Stack {
public:
    void push(Item item); // Push operator
    Item pop();           // Pop operator
    int isEmpty(){
        return (top==0); // Returns true if empty, otherwise false
    }
    Stack() { top = 0; } // Constructor defined inline
private:
    Item stack[size];    // The stack of items
    int top;              // Index to top of stack
};

#ifdef _TEMPINC_
#include "stack.c"
#endif

```

Figure 103. *stack.h* file (*ccntmp2.h*)

Figure 104 on page 322 shows *stack.c*, which is the template definition file.

```
//stack.c
template <class Item, int size>
void Stack<Item,size>::push(Item item) {
    if (top >= size) throw size;
    stack[top++] = item;
}
template <class Item, int size>
Item Stack<Item,size>::pop() {
    if (top <= 0) throw size;
    Item item = stack[--top];
    return(item);
}
}
```

Figure 104. stack.c file (ccntmp1.c)

The stackops.h file (Figure 105 on page 322) contains the prototype for the add function, which is used in both stackadd.cpp and stackops.cpp.

```
void add(Stack<int, 50>& s);
```

Figure 105. stackops.h File (ccntmp5.h)

Figure 106 on page 322 contains the JCL to compile the source files; this JCL does the following:

1. Compiles both compilation units and creates the TEMPINC destination, which is a sequential file with the following data set name MYUSERID . TEMPINC
2. Compiles the template instantiation file in the TEMPINC destination.

```
//CC EXEC CBCC,
// INFILE='MYUSERID.USER.CPP(STACKADD)',
// OUTFILE='MYUSERID.USER.OBJ(STACKADD)', DISP=SHR',
// CPARM='LSEARCH(USER.+)'
//*-----
//CC EXEC CBCC,
// INFILE='MYUSERID.USER.CPP(STACKOPS)',
// OUTFILE='MYUSERID.USER.OBJ(STACKOPS)', DISP=SHR',
// CPARM='LSEARCH(USER.+)'
//*-----
//CC EXEC CBCC,
// INFILE='MYUSERID.TEMPINC',
// OUTFILE='MYUSERID.USER.OBJ', DISP=SHR',
// CPARM='LSEARCH(USER.+)'
//*-----
//BIND EXEC CBCBG,
// INFILE='MYUSERID.USER.OBJ(STACKADD)',
// OUTFILE='MYUSERID.USER.LOAD(STACKADD)', DISP=SHR'
//BIND.OBJ DD DSN=MYUSERID.USER.OBJ, DISP=SHR
//BIND.SYSIN DD *
// INCLUDE OBJ(STACKOPS)
// INCLUDE OBJ(STACK)
/*
```

Figure 106. JCL to compile source Files and TEMPINC destination

Figure 107 on page 322 shows the syntax of how to compile the program within the z/OS shell.

```
export _CXX_CXXSUFFIX=cpp
c++ stackadd.cpp stackops.cpp
```

Figure 107. z/OS UNIX Syntax

Regenerating the template instantiation file

The compiler builds a template instantiation file, in the UNIX file system tempinc directory or the TEMPINC PDS, corresponding to each template declaration file. With each compilation, the compiler may add information to the file but it never removes information from the file.

As you develop your program, you may remove template function references or reorganize your program so that the template instantiation files become obsolete. You can periodically delete the TEMPINC destination and recompile your program.

TEMPINC considerations for shared libraries

In a traditional application development environment, different applications can share both source files and compiled files. When you use templates, applications can share source files but cannot share compiled files.

If you use TEMPINC:

- Each application must have its own tempinc destination.
- You must compile all of the files for the application, even if some of the files have already been compiled for another application.

Under MVS or z/OS UNIX System Services, you can easily assign a separate tempinc PDS or directory for each application.

Using the **TEMPLATEDEPTH** compiler option

To instantiate a large number of nested templates, you can specify the instantiation depth of recursively instantiated templates with the TEMPLATEDEPTH compiler option. By specifying a value between 1 and INT_MAX as the suboption of the TEMPLATEDEPTH compiler option, you can control the maximum number of recursively instantiated template specializations that are processed by the compiler.

The default of the option is TEMPLATEDEPTH(300).

Using the **TEMPLATEREGISTRY** compiler option

Unlike TEMPINC, the TEMPLATEREGISTRY compiler option does not impose specific requirements on the organization of your source code. Any program that compiles successfully with NOTEMPINC will compile with TEMPLATEREGISTRY.

The template registry uses "first come first served" algorithm:

- When a program references a new instantiation for the first time, it is instantiated in the compilation unit in which it occurs.
- When another compilation unit references the same instantiation, it is not instantiated. Thus, only one copy is generated for the entire program.

The instantiation information is stored in a template registry file. You must use the same template registry file for the entire program. Two programs cannot share a template registry file.

The default file name for the template registry file is `templereg` in the UNIX file system and `TEMPLREG` in batch (a sequential file), but you can specify any other valid file name to override this default. When cleaning your program build environment before starting a fresh or scratch build, you must delete the registry file along with the old object files.

Recompiling related compilation units

If two compilation units, A and B, reference the same instantiation, the TEMPLATEREGISTRY compiler option has the following effect:

- If you compile A first, the object file A contains the code for the instantiation.

- When you later compile B, the object file for B contains a reference to the object file A.
- If you later change A so that it no longer references this instantiation, the reference in object B would produce an unresolved symbol error. When you recompile A, the compiler detects this problem and handles it as follows:
 - If the `TEMPLATERECOMPILE` compiler option is in effect, the compiler automatically recompiles B using the same compiler options that were specified for A.
 - If the `NOTEMPLATERECOMPILE` compiler option is in effect, the compiler issues a warning and you must manually recompile B.

Switching from `TEMPINC` to `TEMPLATEREGISTRY`

Because the `TEMPLATEREGISTRY` compiler option does not impose any restrictions on the file structure of your application, it has less administrative overhead than `TEMPINC`. You can make the switch as follows:

- If your application compiles successfully with both `TEMPINC` and `NOTEMPINC`, you do not need to make any changes.
- If your application compiles successfully with `TEMPINC` but not with `NOTEMPINC`, you must change it so that it will compile successfully with `NOTEMPINC`. In each template declaration file, conditionally include the corresponding template definition file if the `__TEMPINC__` macro is *not* defined. This is illustrated in [“TEMPINC example”](#) on page 320.

Using explicit instantiation declarations (C++11 only)

Syntactically, an explicit instantiation declaration is an explicit instantiation definition preceded by the `extern` keyword. This C++11 feature is controlled by the `LANGLVL (EXTENDED)` or `LANGLVL (EXTENDED0X)` compiler group suboptions, or by individual suboptions `LANGLVL (EXTERNTEMPLATE)` and `LANGLVL (NOEXTERNTEMPLATE)`. When multiple `LANGLVL` suboptions are applied, the last one wins. For example, the support for explicit instantiation declaration is disabled when `LANGLVL (NOEXTERNTEMPLATE)` is set. The default settings for `LANGLVL (EXTERNTEMPLATE)` are as follows:

<code>compat366</code>	<code>strict98</code>	<code>extended</code>	<code>extended0x</code>
N	N	Y	Y

There are several things to be considered when using explicit instantiation declarations:

1. (IBM extension) An explicit instantiation declaration of a class template specialization does not cause implicit instantiation of said specialization.
2. If, in a translation unit, a user-defined inline function is subject to an explicit instantiation declaration and not subject to an explicit instantiation definition:
 - implicit instantiation of said function will still occur regardless of whether it will be inlined or not.
 - (IBM extension) no out-of-line copy of the function will be generated in that translation unit regardless of whether compiler option `KEEPINLINES` is enabled or not.

Note: This does not limit the behavior for functions implicitly generated by the compiler. Implicitly declared special members such as the default constructor, copy constructor, destructor and copy assignment operator are inline and the compiler may instantiate them. In particular, out-of-line copies may be generated.

3. Degradation of the amount of inlining achieved on functions that are not "inline" and are subject to explicit instantiation declarations may occur.
4. When a non-pure virtual member function is subject to an explicit instantiation declaration, either directly or through its class, the virtual member function must be subject to an explicit instantiation definition somewhere in the entire program or an unresolved symbol error may result at link time.

- When implicit instantiation of a class template specialization is allowed, the user program must be written as if a use requiring the implicit instantiation of all virtual member functions of that class specialization occurs or an unresolved symbol error for a virtual member function may result at link time.
- When implicit instantiation of a class template specialization is allowed and the specialization is subject to explicit instantiation declaration, the class template specialization must be subject to an explicit instantiation definition somewhere in the user program or an unresolved symbol error may result at link time.

The following compiler options interact with explicit instantiation declarations:

INLINE	All functions, subject to explicit instantiation declaration or not, will be considered for inlining.
TEMPINC, TEMPLATeregistry	Explicit instantiation declarations are honored. Referenced specializations that are subject to explicit instantiation declaration but not subject to explicit instantiation definition in a translation unit will not be instantiated because of that translation unit.

The following IBM language extensions interact with explicit instantiation declarations:

#pragma instantiate	Semantically the same as an explicit instantiation definition.
#pragma do_not_instantiate	This pragma provides a subset of the functionality of standard C++ explicit instantiation declarations. It is provided for backwards compatibility purposes only. New applications should use standard C++11 explicit instantiation declarations.
#pragma ishome #pragma hashome	#pragma ishome will cause generation of the virtual function table (VFT) for a class template specialization irrespective of explicit instantiation declarations of the specialization.

Examples of explicit instantiation declarations

Figure 108 on page 325 shows a simple and typical use of explicit instantiation declarations. The header (sample1.h) contains explicit instantiation declaration. One translation unit (sample1a.C) contains explicit instantiation definition. Another translation unit (sample1b.C) can use the specialization without having the specialization instantiated multiple times.

```

sample1.h:
template <typename T, T val>
union A {
    T foo();
};

extern template union A<int, 55>;

template <class T, T val>
T A<T, val>::foo(void) {
    return val;
}

sample1a.C:
#include "sample1.h"

template union A<int, 55>;

sample1b.C:
#include "sample1.h"

int main(void) {
    return A<int, 55>().foo();
}

```

Figure 108. Simple and typical use of explicit instantiation declarations

Figure 109 on page 326 shows an erroneous case. That is, an explicit instantiation declaration of virtual member function `foo()` is present; but, the explicit instantiation definition of the function is not found.

```
sample2.C:
template <typename T, T val>
struct A {
    virtual T foo();
    virtual T bar();
};

extern template int A<int, 55>::foo();

template <class T, T val>
T A<T, val>::foo(void) {
    return val;
}

template <class T, T val>
T A<T, val>::bar(void) {
    return val;
}

int main(void) {
    return A<int, 55>().bar();
}
```

Figure 109. Erroneous use of explicit instantiation declarations

Chapter 28. Using environment variables

This chapter describes environment variables that affect the z/OS XL C/C++ environment. You can use environment variables to define the characteristics of a specific environment. They may be set, retrieved, and used during the execution of a z/OS XL C/C++ program.

The following environment variables affect the z/OS XL C/C++ environment if they are on when an application program runs. The variables that begin with `_EDC_` and `_CEE_` are described in detail in “Environment variables specific to the z/OS XL C/C++ library” on page 335. See “Locale source files” on page 653 for more information on the locale-related environment variables.

Note: The settings of these variables affect your environment even if you are using the C++ I/O stream classes. For more detailed information on I/O streaming and the Standard C++ I/O stream classes, see *Standard C++ Library Reference*. For information on environment variables used in z/OS UNIX System Services, see *z/OS UNIX System Services Command Reference* and *z/OS UNIX System Services User's Guide*.

`_BIDIATTR`

Used to specify the attributes which will determine the way the bidirectional layout transformation takes place, as shown in the following example. If `_BIDIATTR` is not specified or contains erroneous values, the default values will be used. For a detailed description of the bidirectional layout transformation, see Chapter 60, “Bidirectional language support,” on page 747.

```
export _BIDIATTR="@ls typeoftext=visual:implicit, orientation=ltr:ltr,
numerals=nominal:national"
```

`_BIDION`

Used to specify if `iconv` will perform bidirectional layout transformation beside the basic main function (code page conversion). The value of this variable is either set to `TRUE` to activate the bidirectional layout transformation, or `FALSE` to prevent the bidirectional layout transformation. If this variable is not defined in the environment it defaults to `FALSE`.

`_BPXK_AUTOCVT`

Activates or deactivates automatic text conversion of tagged UNIX file system files. The value of this environment variable is interrogated during initialization of the C `main()`, and at each pthread initialization in order to set the autoconversion state for the thread. The autoconversion state for the thread is looked at by the logical file system (LFS) when determining if automatic text conversion should be performed during read/write operations to tagged UNIX file system files.

The default autoconversion state is unset, meaning that the LFS must look to the `BPXPRMxx AUTOCVT` parameter, which is `ON`, `OFF`, or `ALL`. When set to a valid value, this environment variable overrides the `BPXPRMxx AUTOCVT` parameter.

Restriction: When `_BPXK_AUTOCVT` is `ON`, automatic conversion can only take place between IBM-1047 and ISO8859-1 code sets. Other CCSID pairs are not supported for automatic text conversion. To request automatic conversion for any CCSID pairs that Unicode service supports, set `_BPXK_AUTOCVT` to `ALL`.

During `main()` initialization, the following behavior is defined for this environment variable:

Setting

Autoconversion State for the Thread

ON

Activates the automatic conversion of tagged files for Enhanced ASCII. This affects conversion for I/O for regular, pipe, and character special files that are tagged.

OFF

Deactivates the automatic conversion of tagged files.

ALL

Activates the automatic conversion of tagged files that are supported by z/OS UNICODE. This affects conversion for I/O for regular and pipe files that are tagged. If the conversion is between EBCDIC and ASCII, it also affects conversion for I/O for character special files.

<other>

Treated as unset. Autoconversion defers to BPXPRMxx AUTOCVT parameter.

Changing the value of this environment variable using `setenv()`, `putenv()`, or `clearenv()` during execution of the application will behave in the following manner:

- Ignored after the first pthread create, although `getenv()` might show otherwise. The autoconversion state will remain unchanged.
- Deleting or clearing the environment variable, or setting the value to an invalid value before the first pthread create will change the autoconversion state to unset.
- Has no effect on initially untagged UNIX file system files that have already been opened using `fopen()` or `freopen()` on the current thread and FILETAG(AUTOCVT,) is in effect. These files were specifically marked, or not marked, for automatic text conversion, at the file descriptor level, at the time they were opened. The text conversion state for the already opened file descriptors depended on whether or not autoconversion for the thread was activated or deactivated at the time of the open.
- The standard streams may have already been setup for automatic text conversion, before the `main()` begins execution, using EBCDIC CCSID 1047 as the File CCSID. Therefore, changing the autoconversion state using one of these methods will not affect the standard streams. Specifically, an application running with ASCII CCSID 819 as the Program CCSID will continue to have text conversion with the standard streams.

Changing the value of this environment variable using any other mechanism is ignored, although `getenv()` might show otherwise. You can use `setenv()` with a value of NULL to delete an environment variable.

_BPXK_CCSIDS

Defines the EBCDIC<->ASCII pair of coded character set IDs (CCSIDS) to be used when converting text data, and for automatic tagging new or empty UNIX file system files. The syntax of the environment variable value is as follows, where *e* is the EBCDIC CCSID and *a* is the ASCII CCSID.

```
_BPXK_CCSIDS=(e,a)
```

Language Environment C/C++ applications will initialize with the default IBM-1047<->ISO8859-1 pair. This is equivalent to specifying: `_BPXK_CCSIDS=(1047,819)` before running the application.

The value of this environment variable is interrogated during initialization of the C `main()`, and at each pthread initialization in order to set the Program CCSID for the thread. For the `main()`, the Program CCSID is set to the ASCII value of the pair when the `main()` is part of an ASCII compile unit, otherwise it is set to the EBCDIC value of the pair. The Program CCSID for a thread is set based on the compiled codeset of the thread start routine. When ASCII, the ASCII value of the CCSID pair is used, else the EBCDIC value.

Note: Starting from z/OS V2R1, environment variable `_BPXK_PCCSID` is introduced to represent Program CCSID. The behavior of `_BPXK_CCSIDS` of the existing programs will not be affected unless `_BPXK_PCCSID` is set. If both `_BPXK_CCSIDS` and `_BPXK_PCCSID` are set before a program runs, `_BPXK_PCCSID` is used as the initial value of program CCSID. When a program is running, the CCSID of a thread can be affected by calling `setenv()` or `putenv()` with either of the two environment variables.

Changing the value of this environment variable using `setenv()`, `putenv()`, or `clearenv()` during execution of the application will behave in the following manner:

- Ignored after the first pthread create, although `getenv()` might show otherwise. The current CCSID pair used for conversion & tagging purposes will remain unchanged.

- Deleting or clearing the environment variable before the first pthread create will result in the default CCSID pair (1047,819) being used for conversion and tagging purposes.
- Using improper syntax before the first pthread create will result in the CCSID pair being set to (0,0). This will prevent any further conversion.
- Has no effect on initially untagged new or empty UNIX file system files that have already been opened using fopen(), fropen(), or popen() on the current thread and FILETAG(AUTOTAG) is in effect. These files were setup for tagging upon first write at the time they were opened. The File CCSID was set to what the Program CCSID was at the time of the open.
- The standard streams may have already been setup for automatic text conversion, before the main() begins execution, using EBCDIC CCSID 1047 as the File CCSID, therefore changing the CCSID pair using one of these methods will not affect the standard streams.

Note: Changing the value of this environment variable using any other mechanism is ignored, although getenv() might show otherwise. You can use setenv() with a value of NULL to delete an environment variable.

_BPXK_PCCSID

Identifies the program CCSID for the running thread or user. It can be used to override the internal default of 1047 (EBCDIC). Any value between 0 and 65535 can be assigned. However, to avoid any subsequent errors, only values that are supported by Unicode Services can be used. Setting or unsetting this variable has no effect when translation for a file has started. When unset, the internal value of the program CCSID reverts back to the default of 1047.

_BPXK_SIGDANGER

Set to either YES or NO, this variable modifies the process termination mechanism used during UNIX System Services Shutdown. During Shutdown the kernel sends a signal to each non-permanent non-blocking process. If _BPXK_SIGDANGER is not in the environment, or if its value is not YES, then SIGTERM is sent to these processes. If _BPXK_SIGDANGER is present in the environment and has the value YES then signal SIGDANGER will be sent instead of SIGTERM. The default action for SIGTERM is to terminate the process, but the default action for SIGDANGER is to ignore the signal. The application may register a SIGDANGER signal catcher function to handle shutdowns. If the process does not end in a short while after being sent the first signal, the kernel will send SIGKILL to the process. If the process does not end in a short while after the second signal is sent, the process will be brought down using CALLRTM ABTERM=YES.

Note: The program should not use the environ external variable to put this or any other "_BPXK_" environment variable into its own environment. The Kernel will not be told about the environment variable setting when it is added to the environment this way. The program should use an environ pointer to put this variable into the environment of a new process created with spawn() or exec(). In this case the kernel will notice _BPXK_ environment variables being created for a new program image. In addition, the kernel will correctly detect _BPXK_ environment variables generated into child processes created via fork() and spawn().

_CEE_DLLLOAD_XPCOMPAT

Used to indicate whether certain 31 bit XPLINK DLL application initialization compatibility behaviors should be disabled.

_CEE_DMPTARG

Used to specify the directory in which Language Environment dumps (CEEDUMPs) are written for applications that are running as the result of a fork, exec, or spawn. This environment variable is ignored if the application is not run as a result of a fork, exec, or spawn. Additionally _CEE_DMPTARG can be used to direct the CEEDUMPs output to a specific sysout class.

_CEE_ENVFILE

Used to specify a file from which to read environment variables.

_CEE_ENVFILE_COMMENT

Used to define the comment character to be checked for when z/OS XL C/C++ reads subsequent records from the file.

_CEE_ENVFILE_CONTINUATION

Used to define the continuation character to be checked for when z/OS XL C/C++ reads subsequent records from the file.

_CEE_ENVFILE_S

Used to specify a file from which to read environment variables, stripping trailing white space from each NAME=VALUE line read.

_CEE_HEAP_MANAGER

Used to specify the DLL name for the Vendor Heap Manager to be used during execution of the application.

_CEE_REALLOC_CONTROL

Used to specify the lower bound for the tolerance percentage to be applied and specify the percentage that the storage request will be increased if the request is greater than or equal to the lower bound specified.

_CEE_RUNOPTS

Used to specify Language Environment runtime options to a program invoked by using one of the exec functions, such as a program which is invoked from one of the z/OS UNIX shells.

_EDC_ADD_ERRNO2

Appends errno2 information to the output of perror() and strerror().

_EDC_ANSI_OPEN_DEFAULT

Affects the characteristics of MVS text files opened with the default attributes.

_EDC_AUTOCVT_BINARY

If automatic file conversion is enabled (_BPXK_AUTOCVT=ON and running with FILETAG(AUTOCVT) runtime option), this environment variable activates or deactivates automatic conversion of untagged UNIX file system files opened in binary mode and not opened for record I/O.

_EDC_BYTE_SEEK

Specifies that fseek() and ftell() should use relative byte offsets.

_EDC_CLEAR_SCREEN

Affects the behavior of output text terminal files.

_EDC_COMPAT

Specifies that C/C++ should use specific functional behavior from previous releases of C/370.

_EDC_CONTEXT_GUARD

Allows the user to control the method used to handle the guard page for AMODE 64 user context stacks.

_EDC_C99_NAN

Sets the binary floating-point representation behavior of infinite value and Not a Number for the printf family of functions

_EDC_DLL_DIAG

Indicates if additional DLL diagnostic information should be generated upon failure for the following DLL functions: dllload(), dlopen(), dllqueryfn(), dllqueryvar(), dlsym(), dllfree(), and dlclose(). _EDC_DLL_DIAG has no effect on implicit DLLs. If _EDC_DLL_DIAG is not set by the user, it will default to QUIET.

_EDC_EOVERFLOW

Sets the behavior of the ftell(), fseek(), fstat(), lstat(), stat(), and mmap() functions. By default these functions will not check for the EOVERFLOW error condition. Setting _EDC_EOVERFLOW to YES enables testing for this condition, and, if overflow is detected, setting errno to EOVERFLOW and returning an error.

_EDC_ERRNO_DIAG

Indicates if additional diagnostic information should be generated, when the perror() or strerror() functions are called to produce an error message.

_EDC_FLUSH_STDOUT_PIPE

Flushes the stdout stream when the stdin stream is being read. Both stdin and stdout must be pipes.

_EDC_FLUSH_STDOUT_SOCKET

Flushes the stdout stream when the stdin stream is being read. Both stdin and stdout must be sockets.

_EDC_GLOBAL_STREAMS

Allows the C standard streams stdin, stdout and stderr to have global behavior.

`_EDC_GLOBAL_STREAMS` is not supported in AMODE 64.

_EDC_IEEEV1_COMPATIBILITY_ENV

Used to access original versions of the fdlibm functions when the value of

`_EDC_IEEEV1_COMPATIBILITY_ENV` is set to ON.

_EDC_IO_ABEND

Controls if the runtime library should attempt to recover from an abend issued during OS I/O processing.

_EDC_IO_TRACE

Indicates which files to perform file I/O tracing on, the level of detail to provide for file I/O tracing, and the trace buffer size to use for each file.

_EDC_POPEN

Specifies that `popen()` uses `spawn()` instead of `fork()`.

_EDC_PTHREAD_YIELD

Used to control when `pthread_yield()` and `sched_yield()` will allow a thread to give up control of a processor so that another thread may have the opportunity to run.

_EDC_PTHREAD_YIELD_MAX

Allows a user program to define the max yield (wait) time for a particular thread.

_EDC_PUTENV_COPY

Copies the `putenv()` string into storage owned by Language Environment.

_EDC_RRDS_HIDE_KEY

Relevant for VSAM RRDS files opened in record mode. Enables calls to `fread()` that specify a pointer to a character string and do not append the Relative Record Number to the beginning of the string.

_EDC_SIG_DFLT

To be compliant with the SUSV3 standard, when `_EDC_SIG_DFLT` is set to 1, no message is output to standard error when the signal is handled with default action.

_EDC_STOR_INCREMENT

Sets the size of increments to the internal library storage subpool acquired above the 16M line.

`_EDC_STOR_INCREMENT` is not supported in AMODE 64 applications. In AMODE 64 applications, this environment variable is replaced by the IOHEAP64 runtime option.

_EDC_STOR_INCREMENT_B

Sets the size of increments to the internal library storage subpool acquired below the 16M line.

`_EDC_STOR_INCREMENT_B` is not supported in AMODE 64 applications. In AMODE 64 applications, this environment variable is replaced by the IOHEAP64 runtime option.

_EDC_STOR_INITIAL

Sets the initial size of the internal library storage subpool acquired above the 16M line.

`_EDC_STOR_INITIAL` is not supported in AMODE 64 applications. In AMODE 64 applications, this environment variable is replaced by the IOHEAP64 runtime option.

_EDC_STOR_INITIAL_B

Sets the initial size of the internal library storage subpool acquired below the 16M line.

`_EDC_STOR_INITIAL_B` is not supported in AMODE 64 applications. In AMODE 64 applications, this environment variable is replaced by the IOHEAP64 runtime option.

_EDC_STKEXT_FAILURE

Controls the default behavior of stack extension failure under the POSIX(ON) environment. When the value of `_EDC_STKEXT_FAILURE` is set to ON, the default behavior of stack extension failure is to

issue CEE5203S message and terminate process with the return code of 3000. Otherwise, the default behavior of stack extension failure is to issue ABEND U4088-300x.

_EDC_STRPTM_STD

Indicates changes to `strptime()` that are provided for UNIX standard compliance.

_EDC_SUSV3

Indicates behavioral changes that are provided for SUSV3 compliance in an error path. The affected interfaces are typically setting *errno* to values that were not used before and, in some cases, returning failure for conditions that had not been tested before SUSV3. By default the affected interfaces will not check for these conditions. When the value of `_EDC_SUSV3` is set to 1, the SUSV3 behavior is enabled. When the value of `_EDC_SUSV3` is set to 2, all the behaviors protected by `_EDC_SUSV3=1` are exposed, and pole error related behaviors specified by SUSV3 will be enabled.

_EDC_UMASK_DFLT

Allows the user to control how the C library sets the default umask used when the program runs.

_EDC_ZERO_RECLEN

Enables processing of zero-length records in an MVS data set opened in variable format.

_ICONV_MODE

Selects the behavior mode for `iconv_open()`, `iconv()`, and `iconv_close()` family of functions.

_ICONV_TECHNIQUE

Determines the conversion technique used by Unicode Conversion Services. For more information regarding the Unicode conversion Services value, see *z/OS Unicode Services User's Guide and Reference*.

_ICONV_UCS2

Tells `iconv_open(Y, X)` what type of conversion method to setup when there is a choice between "direct" conversion from X to Y and "indirect" X to UCS-2 to Y. This variable is ignored when using Unicode Conversion Services.

_ICONV_UCS2_PREFIX

Tells `iconv_open()` what z/OS dataset name prefix to use to find UCS-2 tables if they cannot be found in the HFS. This variable is ignored when using Unicode Conversion Services.

LANG

Determines the locale to use for the locale categories when neither the `LC_ALL` environment variable nor the individual locale environment variables specify locale information. This environment variable does not interact with the language setting for messages.

LC_ALL

Determine the locale to be used to override any values for locale categories specified by the settings of the `LANG` environment variable or any individual locale environment variables.

LC_COLLATE

Determines the behavior of ranges, equivalence classes, and multicharacter collating elements.

LC_CTYPE

Determines the locale for the interpretation of byte sequences of text data as characters (for example, single-byte versus multibyte characters in arguments and input files).

LC_MESSAGES

Determines the language in which messages are to be written.

LC_MONETARY

Determines the locale category for monetary-related numeric formatting information.

LC_NUMERIC

Determines the locale category for numeric formatting (for example, thousands separator and radix character) information.

LC_TIME

Determines the locale category for date and time formatting information.

LC_TOD

Determines the locale category for time of day and Daylight Savings Time formatting information.

LIBPATH

Allows an absolute or relative pathname to be searched when loading a DLL. If the input filename contains a slash (/), it is used as is to locate the DLL. If the input filename does not contain a slash, then LIBPATH is used to determine the pathname to load. LIBPATH specifies a list of directories separated by colons. If the LIBPATH begins or ends with a colon, then the working directory is also searched first or last, depending on the position of the stand-alone colon. The "::<" specification can only occur at the beginning or end of the list of directories. If you are running POSIX(ON), then UNIX file system is searched first followed by MVS. If you are running POSIX(OFF), then MVS is searched first followed by UNIX file system. This double search can be avoided by using unambiguous DLL names.

LOCPATH

Tells the `setlocale()` function the name of the directory in the UNIX file system from which to load the locale object files. It specifies a colon separated list of UNIX file system directories.

If LOCPATH is defined, `setlocale()` searches UNIX file system directories in the order specified by LOCPATH for locale object files it requires. Locale object files in the UNIX file system are produced by the `localedef` utility running under z/OS UNIX.

If LOCPATH is not defined and `setlocale()` is called by a POSIX program, `setlocale()` looks in the default UNIX file system locale directory, `/usr/lib/nls/locale`, for locale object files it requires. If `setlocale()` does not find a locale object it requires in the UNIX file system, it converts the locale name to a PDS member name and searches locale PDS load libraries associated with the program calling `setlocale()`.

Note: XPLINK locales have an `.xplink` suffix added to the end of the locale name. For more information about XPLINK locale names, see [“Locale naming conventions” on page 679](#)

PATH

The set of UNIX file system directories that some z/OS XL C/C++ functions, such as `EXECVP`, use in trying to locate an executable. The directories are separated by a colon (:) delimiter. If the pathname contains a slash, the PATH environment variable will not be used.

__POSIX_SYSTEM

Determines the behavior of the `system()` function when the `POSIX(ON)` runtime option has been specified. If `__POSIX_SYSTEM=NO`, then `system()` behaves as in Language Environment/370 1.2: it creates a nested enclave within the same process as the invoker (allowing such things as sharing of memory files). Otherwise, `system()` performs a `fork()` and `exec()`, and the target program runs in a separate process (preventing such things as sharing of memory files).

Restriction: `__POSIX_SYSTEM=NO` is not supported in AMODE 64 applications.

__POSIX_TMPNAM

Determines the behavior of the `tmpnam()` function when the `POSIX(ON)` runtime option has been specified. If the `__POSIX_TMPNAM` environment variable is set to `NO`, `tmpnam()` behaves as if it was called under `POSIX(OFF)`. Otherwise, `tmpnam()` generates a unique file name in the UNIX file system.

STEPLIB

Determines the STEPLIB environment that is created for an executable file. It can be a sequence of MVS data set names separated by a colon (:), or can contain the value `CURRENT` or `NONE`. If you do not want a STEPLIB environment propagated to the environment of the executable file, specify `NONE`. The STEPLIB environment variable defaults to the value `CURRENT`, which will propagate your current environment to that of the executable file. See [z/OS UNIX System Services Command Reference](#) for more information on the use of the STEPLIB variable and changing the search order for z/OS programs.

TZ or _TZ

Time zone information. The `TZ` and `_TZ` environment variables are typically set when you start a shell session, either through `/etc/profile` or `.profile` in your home directory. For more information, see [Chapter 56, “Customizing a time zone,” on page 697](#).

Working with environment variables

The following library functions affect environment variables:

- `setenv()`
- `clearenv()`
- `getenv()`
- `__getenv()`
- `putenv()`
- `unsetenv()`

The `setenv()` function adds, changes, and deletes environment variables in the environment variable table. The `getenv()` function retrieves the values from the table. If it does not find an environment variable, `getenv()` returns NULL. The `clearenv()` function clears the environment variable table, and resets to default behavior the actions affected by z/OS XL C/C++-specific environment variables. The `unsetenv()` function deletes environment variables from the table.

The `__getenv()` function behaves almost the same as `getenv()` except `getenv()` returns the address of the environment variable value string that has been copied into a buffer, whereas `__getenv()` returns the address of the actual value string in the environment variable array. Because the value is not buffered, `__getenv()` cannot be used in a multithreaded application or in a single threaded application where the function `setenv()` changes the value of the variables.

The `putenv()` function provides a subset of the function of `setenv()` and is provided for convenience in porting UNIX applications. `putenv(env_var)` is the same as `setenv(var_name, var_value, i)` where `env_var` represents the string `var_name=var_value`.

For a complete description of these functions, refer to [z/OS C/C++ Runtime Library Reference](#).

Environment variables may be set any time in an application program or user exit. You can use the exit routine CEEBINT to set environment variables through calls to `setenv()`. For more information on the z/OS Language Environment user exit CEEBINT, refer to [“Using runtime user exits in z/OS Language Environment” on page 537](#). You can also set environment variables by using the ENVAR runtime option. The syntax for this option is as follows:

```
ENVAR("1st_var=1st_value", "2nd_var=2nd_value")
```

For more information on this runtime option, refer to [z/OS Language Environment Programming Reference](#).

Specifying the `_CEE_ENVFILE` or `_CEE_ENVFILE_S` environment variable with a filename on the ENVAR option enables you to read more environment variables from that file. See [“Environment variables specific to the z/OS XL C/C++ library” on page 335](#) for more information about `_CEE_ENVFILE` and `_CEE_ENVFILE_S`.

Environment variables set with the `setenv()` function exist only for the life of the program, and are not saved before program termination. Child programs are initialized with the environment variables of the parent. However, environment variables set by a child program are not propagated back to the parent upon termination of the child program.

Note: If you are running with POSIX(ON), environment variables are copied from a parent process to a child process when a `fork()` function is called, and are inherited by the new process image when an EXEC function is called.

When a parent process invokes a child process by using `system()`, using the ISO C/C++ form of the `system` function, the child receives its environment variables from the value of the ENVAR runtime option specified on the invocation of `system()`. For example:

```
system("PGM=CHILD, PARM=' ENVAR(ABC=5) / '");)
```

Naming conventions

Avoid the following when creating names for environment variables:

=

Not valid and will generate an error message.

CBC

Reserved for z/OS XL C/C++ specific environment variables.

CCN

Reserved for z/OS XL C/C++ specific environment variables.

EDC

Reserved for z/OS XL C/C++ specific environment variables.

CEE

Reserved for z/OS XL C/C++ specific environment variables used with z/OS Language Environment. See [“Environment variables specific to the z/OS XL C/C++ library” on page 335](#) for more information.

BPX

Reserved for z/OS XL C/C++ specific environment variables used in the kernel. See the spawn callable service in *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for more information.

DBCS characters

Multibyte and DBCS characters should not be used in environment variable names. Their use can result in unpredictable behavior. Multibyte and DBCS characters are allowed in environment variable values; however, the values are not validated, and redundant shifts are not removed.

white space

Blank spaces are valid characters and should be used carefully in environment variable names and values. For example, `setenv(" my name", " David ",1)` sets the environment variable `<space>my<space>name` to `<space><space>David`. A call to `getenv("my name");` returns NULL indicating that the variable was not found. You must specifically query `getenv(" my name")` to retrieve the value of `" David"`.

The environment variable names are case-sensitive. The empty string is a valid environment variable name.

Note: In general, it is a good idea to avoid special characters, and to use portable names containing just upper and lower case alphabetics, numerics, and underscore characters. Environment variable names containing certain special characters, such as slash (/), are not propagated by the z/OS UNIX shells. Therefore, these variable names are not available to a program called using the `POSIX system()` function.

Environment variables specific to the z/OS XL C/C++ library

The following z/OS XL C/C++ specific environment variables are supported to provide various functions. z/OS XL C/C++ variables have the prefix `_CEE_` or `_EDC_`. You should not use these prefixes to name your own variables.

- `_CEE_DLLLOAD_XPCOMPAT`
- `_CEE_DMPTARG`
- `_CEE_ENVFILE`
- `_CEE_ENVFILE_COMMENT`
- `_CEE_ENVFILE_CONTINUATION`
- `_CEE_ENVFILE_S`
- `_CEE_HEAP_MANAGER`
- `_CEE_REALLOC_CONTROL`
- `_CEE_RUNOPTS`
- `_EDC_ADD_ERRNO2`

- `_EDC_ANSI_OPEN_DEFAULT`
- `_EDC_AUTO_MAP64`
- `_EDC_AUTOCVT_BINARY`
- `_EDC_BYTE_SEEK`
- `_EDC_CLEAR_SCREEN`
- `_EDC_COMPAT`
- `_EDC_CONTEXT_GUARD`
- `_EDC_C99_NAN`
- `_EDC_DLL_DIAG`
- `_EDC_EOVERFLOW`
- `_EDC_ERRNO_DIAG`
- `_EDC_FLUSH_STDOUT_PIPE`
- `_EDC_FLUSH_STDOUT_SOCKET`
- `_EDC_GLOBAL_STREAMS`
- `_EDC_IEEEV1_COMPATIBILITY_ENV`
- `_EDC_IO_ABEND`
- `_EDC_IO_TRACE`
- `_EDC_POPEN`
- `_EDC_PTHREAD_YIELD`
- `_EDC_PTHREAD_YIELD_MAX`
- `_EDC_PUTENV_COPY`
- `_EDC_RRDS_HIDE_KEY`
- `_EDC_SIG_DFLT`
- `_EDC_STOR_INCREMENT`
- `_EDC_STOR_INCREMENT_B`
- `_EDC_STOR_INITIAL`
- `_EDC_STOR_INITIAL_B`
- `_EDC_STRPTM_STD`
- `_EDC_SUSV3`
- `_EDC_UMASK_DFLT`
- `_EDC_ZERO_RECLEN`

There are no default settings for the environment variables that begin with `_EDC_`. There are, however, default *actions* that occur if these environment variables are undefined or are set to invalid values. See the descriptions of each variable below.

The z/OS XL C/C++ specific environment variables may be set with the `setenv()` function.

`_CEE_CONDWAIT_PAUSE`

Controls the program serialization technique that is used by the C/C++ runtime library functions `pthread_cond_wait()`, `pthread_cond_timedwait()`, `pthread_cond_signal()`, and `pthread_cond_broadcast()`.

YES

The Pause and Release callable services are used. These services do not use the local lock and do not contribute to local lock contention.

NO

The WAIT and POST assembler services are used. These services use the local lock and can contribute to local lock contention.

Note: This environment variable must be set by using either of the following mechanisms:

- The ENVAR runtime option
- Inside the file that is specified by the `_CEE_ENVFILE` or `_CEE_ENVFILE_S` environment variable.

`_CEE_DLLLOAD_XPCOMPAT`

Used to indicate if certain 31-bit XPLINK DLL application initialization compatibility behaviors should be disabled.

This environment variable should only be used for applications that do not run properly when migrating from one release to another. While the correct runtime behavior is in the current release, this environment variable provides compatibility support for existing programs. The need to use these settings indicates incorrect programming within the application (for example, reliance on a particular order of C++ static construction across all DLLs that comprise the application). When possible, you should correct the application rather than use this environment variable.

0

Always the most current behavior (e.g. no compatibility behavior enabled). This is identical to the behavior when `_CEE_DLLLOAD_XPCOMPAT` is not set.

1

Disable static initialization prerequisite XPLINK DLL load ordering introduced in z/OS V1R6.

2

Disable non-XPLINK to XPLINK DLL function pointer compatibility introduced in z/OS V1R8.

3

Disable both static initialization prerequisite XPLINK DLL load ordering, and non-XPLINK to XPLINK DLL function pointer compatibility. (Disables both behaviors 1 and 2.)

z/OS Language Environment converts the specified string value to a signed integer, and interprets this value as a bit mask to determine which functions to use in compatibility mode. This allows any combination of compatibility behaviors to be specified.

Here are some examples of how you might set this environment variable:

- z/OS UNIX: `export _CEE_DLLLOAD_XPCOMPAT=1` Disable behavior 1
- Batch/TSO command line: `ENVAR("_CEE_DLLLOAD_XPCOMPAT=3")` Disable behaviors 1 and 2

Note: Any change to the `_CEE_DLLLOAD_XPCOMPAT` environment variable after the application enclave has already been initialized, will not have any effect on the current application enclave.

For information about XPLINK function pointer compatibility see [“XPLINK applications” on page 200](#).

`_CEE_DMPTARG`

You can use this variable in two ways:

1. To specify the directory in which Language Environment dumps (CEEDUMPs) are written for applications that are running as the result of a fork, exec, or spawn.

This environment variable is ignored if the application is not run as a result of a fork, exec, or spawn. When `_CEE_DMPTARG` is set in one of these environments, its value is used as the directory name in which to place CEEDUMPs.

- If in a shell, you set the environment variable as follows. Language Environment dumps will be written to directory `/u/userid/dmpdir`.

```
export _CEE_DMPTARG=/u/userid/dmpdir
```

- If in a shell, you set the environment variable as follows. In this case, Language Environment dumps will be written to directory "cwd"/dmpdir, where "cwd" is the current working directory.

```
export _CEE_DMPTARG=dmpdir
```

2. To direct the CEEDUMPS output to a specific sysout class.

When Language Environment dumps (CEEDUMPS) are produced as a result of running a job, by default they will be written to the default sysout class. You can use _CEE_DMPTARG to direct the CEEDUMPS output to a specific sysout class by using this environment variable as follows, where x is the output class.

```
_CEE_DMPTARG=SYSOUT(x)
```

You can also use the CEEDUMP runtime option to specify a sysout class for dynamically allocated Language environment dump reports. In addition to a sysout class, this runtime option allows you to specify a form-name. See [z/OS Language Environment Programming Reference](#) for further information about the CEEDUMP runtime option."

See [z/OS Language Environment Debugging Guide](#) and [z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode](#) for additional information about _CEE_DMPTARG.

_CEE_ENVFILE

Enables a list of environment variables to be set from a specified file. This environment variable only takes effect when it is set through the runtime option ENVAR on initialization of a parent program. When _CEE_ENVFILE is defined under these conditions, its value is taken as the name of the file to be used. For example, to read the ddname MYVARS, you would call your program with the ENVAR runtime option, as follows:

```
ENVAR("_CEE_ENVFILE=DD:MYVARS")
```

When you set the environment variables with a file in the UNIX file system, you need to use the absolute path to specify the file. For example, if the absolute path of the file is /u/DPGROSS/ootest/tsthello/ENV, you would call your program with the ENVAR runtime option as follows:

```
ENVAR("_CEE_ENVFILE=/u/DPGROSS/ootest/tsthello/ENV")
```

The specified file is opened as a variable length record file. For an MVS data set, the data set must be allocated with RECFM=V. RECFM=VBS must not be used because environment variables may not be contained in spanned records. RECFM=F is not suggested because RECFM=F enables padding with blanks, and the blanks are counted when calculating the size of the line. Each record consists of *NAME=VALUE*. For example, a file with the following two records:

```
_EDC_RRDS_HIDE_KEY=Y
World_Champions=New_York_Yankees
```

would set the environment variable _EDC_RRDS_HIDE_KEY to the value Y, and the environment variable World_Champions to the value New_York_Yankees.

Notes:

1. Using _CEE_ENVFILE to set environment variables through a file is not supported under CICS.
2. z/OS Language Environment searches for an equal sign to delimit the environment variable from its value. If an equal sign is not found, the environment variable is skipped and the rest of the text is treated as comments.
3. Each record of the file is processed independently from any other record in the file. Data within a record is used exactly as input with no substitution. A file containing:

```
FRED=WILMA
FRED=$FRED:BAMBAM
```

will result in the environment variable FRED being set to \$FRED:BAMBAM, rather than to WILMA:BAMBAM as would be the case if the same statements were processed by a UNIX shell.

_CEE_ENVFILE_COMMENT

Defines the comment character to be checked for when subsequent records are read from the file. `_CEE_ENVFILE_COMMENT` is defined within the file specified by the `_CEE_ENVFILE` or `_CEE_ENVFILE_S` environment variable. The comment character used is the first character after the `=` and it must not be a space character, as determined by the `isspace()` macro.

In the following example, the comment character is defined as `*`. With this in place, any subsequent line that begins with `*` in column one is treated as a comment and processing will skip to the next line.

```
_CEE_ENVFILE_COMMENT=*
* This is a comment
NAME1=VALUE1
```

Notes:

1. Comments cannot be placed within a set of continuation lines.
2. If `_CEE_ENVFILE_COMMENT` is encountered when the file is read, it is only used to define the comment character; the keyword is not added to the environment.

_CEE_ENVFILE_CONTINUATION

Defines the continuation character to be checked for when subsequent records are read from the file. `_CEE_ENVFILE_CONTINUATION` is defined within the file specified by the `_CEE_ENVFILE` or `_CEE_ENVFILE_S` environment variable. The continuation character used is the first character after the `=` and it must not be a space character, as determined by the `isspace()` macro.

When a continuation character is defined and a `name=value` sequence is found, where the last non-whitespace character matches the continuation character, the next line of the file is read and appended to `value`. If the last non-whitespace character of the updated `value` matches the continuation character, the next line of the file is read and appended to `value`. This continues until the last non-whitespace character does not match the continuation character and then the `name` and `value` pair are set into the environment.

In the following example, the continuation character is defined as `\`. With this in place, the `STEPLIB` keyword defined on the next line is defined to have a value that spans multiple lines. When queried by the application, the value of `STEPLIB` is `CEE.SCEERUN2:CEE.SCEERUN:MY.LOADLIB`.

```
_CEE_ENVFILE_CONTINUATION=\
STEPLIB=CEE.SCEERUN2:\
CEE.SCEERUN:\
MY.LOADLIB
```

Notes:

1. If `_CEE_ENVFILE_CONTINUATION` is encountered when the file is read, it is only used to define the continuation character; `_CEE_ENVFILE_CONTINUATION` is not added to the environment.
2. The `name=` portion of the `name=value` sequence cannot exceed one line.
3. The length of the value is limited only by the memory available to read, concatenate the lines, and set the variable into the environment.

_CEE_ENVFILE_S

Enables a list of environment variables to be set from a specified file, stripping trailing white space from each `NAME=VALUE` line read.. This environment variable only takes effect when it is set through the runtime option `ENVAR` on initialization of a parent program.

When `_CEE_ENVFILE_S` is defined under this condition, its value specifies the name of the file to be used. For example, to read the ddname MYVARS, you would call your program with the ENVAR runtime option as follows:

```
ENVAR("_CEE_ENVFILE_S=DD:MYVARS")
```

When you set the environment variables with a file in the UNIX file system, you need to use the absolute path to specify the file. For example, if the absolute path of the file is `/u/DPGROSS/octest/tsthello/ENV`, you would call your program with the ENVAR runtime option as follows:

```
ENVAR("_CEE_ENVFILE_S=/u/DPGROSS/octest/tsthello/ENV")
```

For an MVS data set, the data set can be allocated with any record format except RECFM=VBS, because environment variables may not be contained in spanned records. Each record consists of *NAME=VALUE*. For example, a file with the following two records:

```
_EDC_RRDS_HIDE_KEY=Y  
World_Champions=New_York_Yankees
```

would set the environment variable `_EDC_RRDS_HIDE_KEY` to the value Y, and the environment variable `World_Champions` to the value `New_York_Yankees`.

Notes:

1. Using `_CEE_ENVFILE_S` to set environment variables through a file is not supported under CICS.
2. z/OS Language Environment searches for an equal sign to delimit the environment variable from its value. If an equal sign is not found, the environment variable is skipped and the rest of the text is treated as comments.
3. Both environment variables `_CEE_ENVFILE` and `_CEE_ENVFILE_S` can be specified. `_CEE_ENVFILE_S` takes precedence, meaning it is processed second in sequence.
4. Each record of the file is processed independently from any other record in the file. Data within a record is used exactly as input with no substitution (other than trailing white space is ignored). A file containing:

```
FRED=WILMA  
FRED=$FRED:BAMBAM
```

will result in the environment variable FRED being set to `$FRED:BAMBAM`, rather than to `WILMA:BAMBAM` as would be the case if the same statements were processed by a UNIX shell.

`_CEE_HEAP_MANAGER`

Specifies the name of the Vendor Heap Manager (VHM) DLL that will be used to manage the user heap. You set the environment variable as follows:

```
_CEE_HEAP_MANAGER=dllname
```

This environment variable must be set using one of the following mechanisms:

- ENVAR runtime option
- Inside the file specified by the `_CEE_ENVFILE` or `_CEE_ENVFILE_S` environment variable.

Either of these mechanisms is before any user code gets control. This means 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.

See [z/OS Language Environment Vendor Interfaces](#) for more information on the Vendor Heap Manager support.

_CEE_REALLOC_CONTROL

`_CEE_REALLOC_CONTROL` has two parameters. The first parameter specifies the lower bound for the tolerance percentage to be applied. This variable reflects the number of bytes that will cause the `realloc()` control feature to be activated. For instance, if an application issues a `malloc()` to request storage and a subsequent `realloc()` to change the size of that storage allocation, this parameter determines whether the request will be increased - with the intent that subsequent reallocations will not require additional storage be obtained and data copied.

The second parameter specifies the percentage that the storage request will be increased if the request is greater than or equal to the lower bound specified in the first parameter.

The format of the environment variable is:

```
_CEE_REALLOC_CONTROL=bound,percentage
```

bound

a , aK , aK , in which a is an integer.

percentage

p , which is an integer between 0 and 100. The default value is 0, which means that this feature is not in use.

Attempting to reallocate storage to a new size of near 2GB while this environment variable is set may cause `realloc()` to report that a negative new size was provided as input.

Notes:

- The behavior of `realloc` request is not affected by this environment variable when the input element is allocated by `aligned_alloc()` or `posix_memalign()`.
- The following examples use storage sizes that are for illustration purposes only. The sizes are examples and do not reflect any storage rounding that might occur.

The following example shows this control feature being used within a loop and how an allocation of a new storage element can be eliminated:

```
_CEE_ REALLOC_CONTROL=100,20
```

```
/* an example in C*/
char * buffer;
size_t buffsize;
int i;
buffsize = 100;
buffer = malloc(buffsize);

for ( i = 0; i <= 2; ++i ) {
    buffsize += 10;
    buffer = realloc(buffer,buffsize);
}

for ( i = 0; i <= 2; ++i ) {
    buffsize -= 10;
    buffer = realloc(buffer,buffsize);
}
```

Because the `realloc()` request is greater than the lower bound, the first pass through the first loop results in a new buffer with the same contents as before but within a storage element of size 132 ($110 + (110 * .20)$) and `buffsize=110`. All the data in the first buffer (100 bytes) is copied to the second (new) buffer.

The second and third pass through the first loop issue the same `realloc`, but result in no action being taken because the new `buffsize` of 120 and then 130 allows the requested storage to remain within the current allocation of 132 bytes, even if both requests are greater than the lower bound. Therefore, allocation of new storage elements and copies of data are eliminated.

Additionally, the first two paths through the second loop also result in no action because they result in a `buffsize` less than the current allocation and can also fit within the current allocation.

The last path through the second loop results in a new buffer of `bufsize 100`. Although this request is also less than the current allocation and fits in the current allocation, the assumption is that no tolerance ever occurred because the requested size plus the increase have resulted in a storage allocation less than the current allocation.

In other words, if `realloc` control feature is relevant (the requested storage allocation is greater than the lower bound), the rules are:

- If the (`realloc`) request is equal to the current allocation, use the same buffer location and size (do nothing).
- If the request is greater than the current allocation, get a new buffer of size (`request + tolerance`).
- If the request is lower than the current allocation and the current allocation is greater than (`request + tolerance`), assume tolerance never applied before and get a new buffer of size requested
- If the request is lower than the current allocation and the current allocation is not greater than (`request + tolerance`), use the same buffer and size (do nothing)

_CEE_RUNOPTS

Used to specify invocation Language Environment runtime options for programs invoked using one of the `exec` family of functions. Mechanisms for setting the value of the `_CEE_RUNOPTS` environment variable include using the `export` command within the z/OS UNIX shell, or using the `setenv()` or `putenv()` functions within a C/C++ application. The runtime options set from the `_CEE_RUNOPTS` environment variable value that become active in the invoked program are known as **invocation command** runtime options.

Note: For this description, the `exec` family of functions includes the `spawn` family of functions.

The format of the environment variable is as follows, where `value` is a null-terminated character string of Language Environment runtime options.

```
_CEE_RUNOPTS=value
```

For example, you could specify the following to set the value of the environment variable within the z/OS UNIX shell.

```
export _CEE_RUNOPTS="stack(,,any,) termthdact(dump)"
```

The `_CEE_RUNOPTS` environment variable has a unique behavior. It can be unset, or modified, but will be re-created or added to across an `exec` to effect the propagation of invocation Language Environment runtime options. This behavior is designed specifically to allow runtime options such as `TRACE` to take effect for parts of an application which are not invoked directly by the user. Without this behavior, the external `TRACE` option could not be propagated to parts of an application that are executed using one of the `exec` family of functions.

At the time of the `exec`, any active invocation command runtime option settings, not already explicitly part of the `_CEE_RUNOPTS` environment variable, are added to its value. This new value for the `_CEE_RUNOPTS` environment variable is passed to the `exec` target to be used as invocation Language Environment runtime options for the invoked program. Thus, all invocation runtime options, those specified with the `_CEE_RUNOPTS` environment variable and those already active, are propagated across the `exec`.

When the `_CEE_RUNOPTS` environment variable is not defined at the time of the `exec`, but there are other active invocation command runtime options, it will be re-created with its value set to represent the active invocation command runtime option settings. This unique behavior, where the `_CEE_RUNOPTS` environment variable is added to, or re-created, across an `exec`, can cause unexpected results when the user attempts to unset (clear) the environment variable, or modify its value.

[Figure 110 on page 343](#) demonstrates this behavior. We enter the z/OS UNIX shell through OMVS, and a sub-shell is created using one of the `exec` family of functions. The propagation of the `_CEE_RUNOPTS` environment variable takes place across creation of the sub-shell.

```

/u/carbone>echo $_CEE_RUNOPTS
POSIX(ON) 1
/u/carbone>/bin/sh 2
/u/carbone>echo $_CEE_RUNOPTS 3
POSIX(ON)
/u/carbone>unset _CEE_RUNOPTS 4
/u/carbone>echo $_CEE_RUNOPTS

/u/carbone>env | grep _CEE_RUN 5
_CEE_RUNOPTS=POS(ON)
/u/carbone>echo $_CEE_RUNOPTS 6

/u/carbone>export _CEE_RUNOPTS="ABTERMENC(RETCODE)" 7
/u/carbone>echo $_CEE_RUNOPTS
ABTERMENC(RETCODE)
/u/carbone>env | grep _CEE_RUN 8
_CEE_RUNOPTS=ABTERMENC(RETCODE) POS(ON)
/u/carbone>/bin/sh 9
/u/carbone>echo $_CEE_RUNOPTS
ABTERMENC(RETCODE) POS(ON)
/u/carbone>unset _CEE_RUNOPTS
/u/carbone>echo $_CEE_RUNOPTS

/u/carbone>env | grep _CEE_RUN
_CEE_RUNOPTS=ABT(RETCODE) POS(ON)
/u/carbone>

```

Figure 110. `_CEE_RUNOPTS` behaviour

Notes:

1. The current value of the `_CEE_RUNOPTS` environment variable happens to be `POSIX(ON)`.
2. Using `/bin/sh` to create a sub-shell will go through the process where the `_CEE_RUNOPTS` environment variable is added to, or re-created, across the exec.
3. Displaying the value of the `_CEE_RUNOPTS` environment variable using `echo` in the sub-shell shows that no other invocation command runtime options were in effect at the time of the exec, since the value of the environment variable is unchanged (there were no runtime options to add).
4. Using `unset` to clear the `_CEE_RUNOPTS` environment variable does remove it from the sub-shell environment, as shown with the `echo` command, but it does not change the fact that `POSIX(ON)` is the active invocation command runtime option in the sub-shell.
5. To see this, we use the `env | grep _CEE_RUNOPTS` command. The `env` is the target of an exec. We know that the `_CEE_RUNOPTS` environment variable is re-created across the exec from the active invocation command runtime options. And as you can see, the value shows as `POS(ON)`. During re-creation, Language Environment uses the minimum abbreviations for the runtime options when re-creating or adding to the `_CEE_RUNOPTS` environment variable.
6. When the `env` returns, the `_CEE_RUNOPTS` environment variable is still unset in the sub-shell as seen using the `echo` command.
7. We now use `export` to set a different value for the `_CEE_RUNOPTS` environment variable in the sub-shell. We see the value using the `echo` command.
8. Using the `env | grep _CEE_RUNOPTS` command again, we see the behavior where the active invocation command runtime options are added to the current value of the `_CEE_RUNOPTS` environment variable.
9. The rest of the example creates a second sub-shell and shows that the `_CEE_RUNOPTS` environment variable in the sub-shell was added to across the exec of the sub-shell. And again, using `unset` does not change the active invocation command runtime options.

`_EDC_ADD_ERRNO2`

Controls whether or not `errno2` is appended to the output of `perror()`, `strerror()`, and `strerror_r()`. The `errno2` might be set by the z/OS C/C++ runtime library, z/OS UNIX callable services, or other callable services. The `errno2` is intended for diagnostic display purposes only and it is not a programming interface.

The variable `_EDC_ADD_ERRNO2` is not set by default. When the variable `_EDC_ADD_ERRNO2` is not set, `errno2` is added to `perror()` messages, but not to messages retrieved by using `strerror()` or `strerror_r()`. When `_EDC_ADD_ERRNO2` is set to 1, `errno2` is added to `perror()`, `strerror()`, and `strerror_r()` messages. For all other values of `_EDC_ADD_ERRNO2`, `errno2` is not added. For example, for `perror()`, if `errno` was 121, the default behavior might produce the following message "EDC5121I Invalid argument. (errno2=0x0C0F8402)".

`_EDC_ADD_ERRNO2` is set to zero with the command:

```
setenv("_EDC_ADD_ERRNO2","0",1);
```

It is suggested that applications run with `_EDC_ADD_ERRNO2` not being set. This causes `errno2` to be added only to `perror()` messages.

If an application is using `strerror()` or `strerror_r()` to retrieve messages associated with an error number, usually a saved `errno` value, it is suggested that `errno2` also be saved by using `__errno2()` at the time when `errno` is saved. The application can then process the retrieved message and saved `errno2` value as a pair.

Note: Not all functions set `errno2` when `errno` is set. In the cases where `errno2` is not set, the `errno2` might be a residual value. You might use the `__err2ad()` function to clear `errno2` to reduce the possibility of a residual value being returned.

`_EDC_ANSI_OPEN_DEFAULT`

Affects the characteristics of MVS text files opened with the default attributes. Issuing the following command causes text files opened with the default characteristics to be opened with a record format of FIXED and a logical record length of 254 in accordance with ISO C.

```
setenv("_EDC_ANSI_OPEN_DEFAULT","Y",1);
```

When this environment variable is not specified and a text file is created without its record format or LRECL defined, then the default is a variable record format.

`_EDC_AUTO_MAP64`

Setting `_EDC_AUTO_MAP64` to Yes causes the `__MAP_64` flag to be added for all `mmap()` calls in AMODE 64.

Value

Description

Yes

Add the `__MAP_64` flag automatically for the `mmap()` calls in AMODE 64.

<other>

No effect. This is the default.

`_EDC_AUTOCVT_BINARY`

If automatic file conversion is enabled (`_BPXK_AUTOCVT=ON` and running with FILETAG(AUTOCVT) runtime option), this environment variable activates or deactivates automatic conversion of untagged UNIX file system files opened in binary mode and not opened for record I/O.

The value of this environment variable is checked every time a UNIX file system file is opened. If automatic file conversion is enabled and `_EDC_AUTOCVT_BINARY=YES`, an untagged file opened in binary mode will trigger the file to be automatically converted from the program CCSID to the EBCDIC CCSID as specified by the `_BPXK_CCIDS` environment variable. If `_BPXK_CCIDS` is not set, a default CCSID pair is used. See `_BPXK_CCIDS` environment variable for additional details.

`_EDC_AUTOCVT_BINARY` can be set to the following values to set the conversion state for binary files.

NO (default)

If automatic file conversion is enabled, an untagged file opened in binary mode will not trigger the file to be automatically converted from the program CCSID to the EBCDIC CCSID as specified by the `_BPXK_CCIDS` environment variable. If `_BPXK_CCIDS` is not set, a default CCSID pair is used. See `_BPXK_CCIDS` environment variable for additional details. An untagged file opened in text mode will not be affected.

YES

If automatic file conversion is enabled, an untagged file opened in binary mode and not opened for record I/O will trigger the file to be automatically converted from the program CCSID to the EBCDIC CCSID as specified by the `_BPXK_CCIDS` environment variable. If `_BPXK_CCIDS` is not set, a default CCSID pair is used. See `_BPXK_CCIDS` environment variable for additional details.

Note: If this environment variable is not set, the default behavior is chosen, which is the same as `_EDC_AUTOCTV_BINARY=NO`. Because this environment variable is checked on every file open, an application can pick up the changes to this environment variable by closing and then re-opening the file at execution time. The application itself does not need to be restarted.

`_EDC_BYTE_SEEK`

Indicates to z/OS XL C/C++ that, for all binary files, `ftell()` should return relative byte offsets, and `fseek()` should use relative byte offsets as input. The default behavior is for only binary files with a fixed record format to support relative byte offsets. `_EDC_BYTE_SEEK` is set with the command:

```
setenv("_EDC_BYTE_SEEK", "Y", 1);
```

`_EDC_CLEAR_SCREEN`

Applies to output text terminal files. `_EDC_CLEAR_SCREEN` is set with the command:

```
setenv("_EDC_CLEAR_SCREEN", "Y", 1);
```

When `_EDC_CLEAR_SCREEN` is set, writing a `\f` (form feed) character to a text terminal sends all preceding unwritten data in the terminal buffer to the screen, and then clears the screen.

When `_EDC_CLEAR_SCREEN` is not set, writing a `\f` (form feed) character to a text terminal results in the character being treated as a non-control character. The character is written to the terminal buffer as `\f`.

`_EDC_COMPAT`

Indicates to z/OS XL C/C++ that it should use old functional behavior for various items in code ported from old releases of C/370. These functional items are specified by the value of the environment variable. `_EDC_COMPAT` is set with the following command, where `x` is an integer:

```
setenv("_EDC_COMPAT", "x", 1);
```

z/OS XL C/C++ converts the string `"x"` into its decimal integer equivalent, and treats this value as a bit mask to determine which functions to use in compatibility mode. The following table interprets the least significant bit as bit zero.

Bit

Function Affected

0

`ungetc()`

1

`ftell()`

2

`fclose()`

3 through 31

Unused

For this release, calls to `fseek()` with an offset of `SEEK_CUR`, `fgetpos()`, and `fflush()` take into account characters pushed back with the `ungetc()` library function. You must set the `_EDC_COMPAT` environment variable for `ungetc()` if you want these functions to ignore `ungetc()` characters as they did in old C/370 code.

For `ftell()`, z/OS XL C/C++ uses an encoding scheme that varies according to the attributes of the underlying data set. You must set the `_EDC_COMPAT` environment variable for `ftell()` if you want to use encoded `ftell()` values generated in old C/370 code.

You can set `_EDC_COMPAT` to indicate that `fclose()` should not unallocate the `SYSOUT=*` data set when it is closing "*" data sets created under batch. This is to ensure that such data sets can be concatenated with the Job Log, if their attributes are compatible.

Here are some examples of how you can set `_EDC_COMPAT`:

- `setenv("_EDC_COMPAT", "1", 1)`; invokes old `ungetc()` behavior.
- `setenv("_EDC_COMPAT", "2", 1)`; invokes old `ftell()` behavior.
- `setenv("_EDC_COMPAT", "3", 1)`; invokes both old `ungetc()` behavior and old `ftell()` behavior.
- `setenv("_EDC_COMPAT", "4", 1)`; invokes old behavior for spool data sets created by opening "*" in MVS or IMS batch.

`_EDC_CONTEXT_GUARD`

Allows the user to control the method used to handle the guard page for AMODE 64 user context stacks.

When the value of `_EDC_CONTEXT_GUARD` is set to `ACTIVE`, the guard page for a user context stack is guarded each time the context is given control and unguarded each time the context gives up control. This is the default behavior when the value of `_EDC_CONTEXT_GUARD` is not set.

When the value of `_EDC_CONTEXT_GUARD` is set to `INUSE`, the guard page for a user context stack is guarded the first time the context is given control and unguarded when the context has run to completion, that is, when the function specified on the call to `makecontext()` returns or exits. This method of handling the guard page might provide better performance but comes with the following restrictions:

- The storage for user context stacks must be allocated from the heap.
- The storage for a user context stack cannot be reused or freed until the context runs to completion.

The `_EDC_CONTEXT_GUARD` environment variable can be set with the function:

```
setenv("_EDC_CONTEXT_GUARD", "INUSE", 1);
```

Note: The setting of this environment variable is only effective if it is done before the first call to `makecontext()`.

Value

Description

ACTIVE

The user context stack is only guarded when the context is active. This is the default value.

INUSE

The user context stack is guarded the entire time that the context is in use.

`_EDC_C99_NAN`

Sets the binary floating-point representation of infinite value and Not a Number for the `printf` family of functions as follows:

- When the value of `_EDC_C99_NAN` is set to `YES`, then the `printf` family of functions use C99 compliant behavior. C99 defines the representation of infinity and Not a Number as `INF`, and `NAN` (for `E`, `F`, `G`, and `A` conversion specifiers) or `inf` and `nan` (for `e`, `f`, `g`, and `a` conversion specifiers). In C99

compliant behavior, the case of the string will be the same as the case of the conversion specifier that was used.

- When the value of `_EDC_C99_NAN` is not set, or set to a value other than YES, then the representation of infinity and Not a Number is INF and NaN.

The `_EDC_C99_NAN` environment variable can be set with the function:

```
setenv("_EDC_C99_NAN", "YES", 1);
```

_EDC_DLL_DIAG

Indicates if additional DLL diagnostic information should be generated upon failure for the following DLL functions: `dllload()`, `dlopen()`, `dllqueryfn()`, `dllqueryvar()`, `dlsym()`, `dllfree()`, and `dllclose()`. `_EDC_DLL_DIAG` has no effect on implicit DLLs. If `_EDC_DLL_DIAG` is not set by the user, it will default to QUIET.

`_EDC_DLL_DIAG` can be set with the following command, where *x* is a string.

```
setenv("_EDC_DLL_DIAG", "x", 1);
```

Acceptable values for *x* are as follows:

Value

Description

MSG

Issue DLL error messages to the Language Environment message file.

TRACE

Issue all DLL error messages to the Language Environment message file and call the `ctrace()` function to produce a traceback for each error.

SIGNAL

Issue all DLL error messages to the Language Environment message file, call the `ctrace()` function to produce a traceback for each error, and signal a condition for each error's feedback code.

QUIET

Turn off all `_EDC_DLL_DIAG` error diagnostics; this is the default setting.

The `_EDC_DLL_DIAG` values must be specified in capital letters in order to be recognized. See *z/OS C/C++ Runtime Library Reference* for details on the level of diagnostic information provided by `ctrace()`.

There is currently no way to intercept a signaled DLL condition in AMODE 64; therefore, termination is bound to happen if SIGNAL is in effect.

The following list shows examples of how to use `_EDC_DLL_DIAG`.

- Issue a DLL error message for DLL errors.

```
setenv("_EDC_DLL_DIAG", "MSG", 1);
```

- Issue a DLL error message and call the `ctrace()` function for DLL errors.

```
setenv("_EDC_DLL_DIAG", "TRACE", 1);
```

- Issue a DLL error message, call the `ctrace()` function, and signal a condition for DLL errors.

```
setenv("_EDC_DLL_DIAG", "SIGNAL", 1);
```

- Turn off `_EDC_DLL_DIAG` error diagnostics.

```
setenv("_EDC_DLL_DIAG", "QUIET", 1);
```

_EDC_EOVERFLOW

Sets the behavior of the `ftell()`, `fseek()`, `fstat()`, `lstat()`, `stat()`, and `mmap()` functions. By default these functions will not check for the EOVERFLOW error condition. Setting `_EDC_EOVERFLOW` to YES enables testing for this condition, and, if overflow is detected, setting `errno` to EOVERFLOW and returning an error. The `_EDC_EOVERFLOW` environment variable can be set with the function:

```
setenv("_EDC_EOVERFLOW", "YES", 1);
```

Value	Description
-------	-------------

YES	Check for EOVERFLOW error conditions.
------------	---------------------------------------

<other>	Ignore setting of EOVERFLOW; this is the default. Equivalent to unsetting the environment variable.
----------------------	---

_EDC_ERRNO_DIAG

Indicates if additional diagnostic information should be generated, when the `perror()` or `strerror()` functions are called to produce an error message. This environment variable also controls how much additional information is produced. `_EDC_ERRNO_DIAG` is set with the following command, where `x` is an integer and `y` is a list of integer `errno` values, for which additional diagnostic information is desired.

```
setenv("_EDC_ERRNO_DIAG", "x,y", 1);
```

The list of `errno` values must be separated by commas. If the `y` value is omitted, then additional diagnostic information is generated for all `errno` values. If a non-numeric `errno` value is found in `y`, it is treated as 0. Acceptable values for `x` are as follows:

0

No additional diagnostic information is generated (This is the default if `_EDC_ERRNO_DIAG` is not set).

1

The `ctrace()` function is called to generate additional diagnostic information.

2

The `csnap()` function is called to generate additional diagnostic information.

3

The `cdump()` function is called to generate additional diagnostic information.

See [z/OS C/C++ Runtime Library Reference](#) for details on the level of diagnostic information provided by the above functions.

The following list shows examples of how to use this environment variable.

- No additional diagnostic information is produced.

```
setenv("_EDC_ERRNO_DIAG", "0", 1);
```

- The `ctrace()` function is called for any `errno` when `perror()` or `strerror()` are called.

```
setenv("_EDC_ERRNO_DIAG", "1", 1);
```

- The `csnap()` function is called only when `errno` equals 121 when `perror()` or `strerror()` are called.

```
setenv("_EDC_ERRNO_DIAG", "2,121", 1);
```

- The `cdump()` function is called only when `errno` equals either 121 or 129 when `perror()` or `strerror()` are called.

```
setenv("_EDC_ERRNO_DIAG", "3,121,129", 1);
```

_EDC_FLUSH_STDOUT_PIPE

Instructs the C Runtime Library to flush the `stdout` stream when the `stdin` stream is being read from. Both `stdin` and `stdout` must be pipes. `_EDC_FLUSH_STDOUT_PIPE` is set with the command:

```
setenv("_EDC_FLUSH_STDOUT_PIPE", "YES", 1);
```

The purpose of this environment variable is to better facilitate communication between two processes that use pipes. The child process is using `stdin` for the read end of one pipe and `stdout` for the write end of a different pipe. The parent process has the opposite ends of the pipes.

Note the following usage notes and examples:

- The child process wants to send a prompt to the parent process and then read the response. The parent process must see the prompt, capture a response, and then send the response back to the child.
- If the child process sends the prompt without a trailing newline or without calling `fflush`, the prompt is never placed into the pipe for the parent process to read. The child process then tries to read the response from `stdin`. The application hangs, which means that the parent process cannot read the prompt that it needs to respond to and the child process cannot read the response from the parent process.
- With the environment variable set to YES, the read from `stdin` will flush the `stdout` buffer. This allows the parent process to see the prompt, respond, and then send the response back to the child process who is waiting on the read. After the data comes back from the parent process, the child process reads the response and continues.

Note: The parent process must do a read by using `read()` from the pipe so that it can receive whatever data might be there without having to wait for a specific number of bytes or a newline character. Functions like `fread()` and `fgets()` will hang unless the child process wrote enough bytes or the newline character to `stdout`.

_EDC_FLUSH_STDOUT_SOCKET

Instructs the C Runtime Library to flush the `stdout` stream when the `stdin` stream is being read from. Both `stdin` and `stdout` must be sockets. `_EDC_FLUSH_STDOUT_SOCKET` is set with the command:

```
setenv("_EDC_FLUSH_STDOUT_SOCKET", "YES", 1);
```

The purpose of this environment variable is to better facilitate communication between two processes that use sockets. The child process is using `stdin` for the read end of one socket and `stdout` for the write end of a different socket. The parent process has the opposite ends of the sockets.

Note the following usage notes and examples:

- The child process wants to send a prompt to the parent process and then read the response. The parent process must see the prompt, capture a response, and then send the response back to the child process.
- If the child sends the prompt without a trailing newline or without calling `fflush`, the prompt is never placed into the socket for the parent process to read. The child then tries to read the response from `stdin`. The application hangs, which means that the parent process cannot read the prompt that it needs to respond to and the child process cannot read the response from the parent.
- With the environment variable set to YES, the read from `stdin` will flush the `stdout` buffer. This allows the parent to see the prompt, respond, then send the response back to the child who is waiting on the read. After the data comes back from the parent, the child reads the response and continues.

Note: The parent process must do a read by using `read()` from the socket so that it can receive whatever data might be there without having to wait for a specific number of bytes or a newline character. Functions like `fread()` and `fgets()` will hang unless the child wrote enough bytes or the newline character to `stdout`.

_EDC_GLOBAL_STREAMS

Used during initialization of the first C main in the environment to allow the C standard streams `stdin`, `stdout`, and `stderr` to have global behavior. The environment variable settings and standard streams using the global behavior, are as follows:

Setting

Standard streams using global behavior

- | | |
|----------|---------------------|
| 0 | none |
| 1 | stderr |
| 2 | stdout |
| 3 | stderr,stdout |
| 4 | stdin |
| 5 | stderr,stdin |
| 6 | stdout,stdin |
| 7 | stderr,stdout,stdin |

Note: The first C main would include any Pre-Init Compatibility Interface initialization.

You can use one of the following methods to set the environment variable `_EDC_GLOBAL_STREAMS`:

- CEEBXITA assembler user exit

You can modify the sample CSECT and assemble and link with the application. The runtime options specified in the CEEBXITA assembler user exit override all other sources of runtime options except those that are specified as `NONOVR`. These options are honored only during initialization of the first enclave.

- `ENVAR(_EDC_GLOBAL_STREAMS=<setting>)`

You can call your program with the `ENVAR` runtime option. This overrides the application defaults specified using `CEEUOPT` or the `#pragma runopts` directive.

- `#pragma runopts(ENVAR(_EDC_GLOBAL_STREAMS=<setting>))`

Use the `#pragma runopts` directive in your application source code.

- `CEEUOPT` application defaults

Modify the sample CSECT and assemble and link with the application. This overrides corresponding overrideable system-level or region-level default options.

Notes:

1. `_EDC_GLOBAL_STREAMS` is not supported in AMODE 64.
2. Attempts to set this environment variable in the file specified by the `_CEE_ENVFILE` or `_CEE_ENVFILE_S` environment variable are ignored. The standard streams are initialized before that file is read.
3. You cannot use the CEEBINT user exit to set this environment variable. The CEEBINT user exit gets control after the standard streams have been initialized.

_EDC_IEEEV1_COMPATIBILITY_ENV

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 first method is through an environment variable, described here, that can be used by applications that do one of the following:

- Do not include `<math.h>`.
- Include `<math.h>` and define the `_FP_MODE_VARIABLE` feature test macro.

Either of the above will cause the application to be running in what is called "variable" mode with respect to floating-point math functions called within the compile unit.

The second method is through the `_IEEEV1_COMPATIBILITY` feature test macro and is used for applications that include `<math.h>` and do not define the `_FP_MODE_VARIABLE` feature test macro. See [z/OS C/C++ Runtime Library Reference](#) for more details.

If the application conforms to the rules of the first method, then this environment variable can be used to access the original `fdlibm` versions of the functions shown in [Table 63 on page 351](#).

<i>Table 63. Original versions of <code>fdlibm</code> functions</i>						
<code>acos()</code>	<code>acosh()</code>	<code>asin()</code>	<code>asinh()</code>	<code>atan()</code>	<code>atanh()</code>	<code>atan2()</code>
<code>cbrt()</code>	<code>cos()</code>	<code>cosh()</code>	<code>erf()</code>	<code>erfc()</code>	<code>exp()</code>	<code>expm1()</code>
<code>gamma()</code>	<code>hypot()</code>	<code>lgamma()</code>	<code>log()</code>	<code>log1p()</code>	<code>log10()</code>	<code>pow()</code>
<code>rint()</code>	<code>sin()</code>	<code>sinh()</code>	<code>tan()</code>	<code>tanh()</code>		

This environment variable will only take effect if the application is running in "variable" mode. The following list shows the acceptable values of the environment variable and the behavior for each value:

ON

Original versions of `fdlibm` functions are used.

other values

The new versions of the functions are used. This is the default.

If the application 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.

_EDC_IO_ABEND

I

When an abend condition arises during OS I/O processing, the z/OS C/C++ runtime library ignores the condition, if possible. When the abend condition cannot be ignored, the abend is issued. This environment variable controls if the runtime library should attempt to recover from an abend issued during OS I/O processing. The acceptable values for the environment variable are as follows:

ABEND

Specifies that the runtime library is to ignore abend conditions that can be ignored and that the runtime library should not attempt to recover from an abend issued during OS I/O processing. When an abend is issued during OS I/O processing, Language Environment condition handling semantics

take effect. The only methods available for the application to attempt to recover are to write a Language Environment condition handler or a SIGABND signal handler. If this environment value is not set or if a value other than ABEND or RECOVER is specified, this is the default behavior.

RECOVER

Specifies that the runtime library is to ignore abend conditions that can be ignored and also that the runtime library should attempt to recover from an abend issued during OS I/O processing. If the library can recover, then control will be returned to the application as a failing return value, with `errno` set to 92 and diagnostic information in the `__amrc` structure. If the library cannot recover, Language Environment condition handling semantics take effect.

Notes:

1. The value is not case sensitive.
2. When a stream is opened, the current setting of the environment variable defines the behavior for the life of the open stream, unless overridden by the `abend` keyword specified in the `modestring` on the `fopen()` or `freopen()` call.
3. Changes to the environment variable do not affect existing open streams.
4. This environment variable has no effect for SPC applications.

For more information about error handling during I/O operations, refer to [Chapter 8, “Performing OS I/O operations,”](#) on page 37.

The following example show one method to set this environment variable.

```
setenv("_EDC_IO_ABEND", "RECOVER", 1);
```

_EDC_IO_TRACE

Indicates which files to perform file I/O tracing on, the level of detail to provide for file I/O tracing, and the trace buffer size to use for each file. The `_EDC_IO_TRACE` format is:

```
_EDC_IO_TRACE=(Filter,Detail,Buffer Size)
```

The three values (Filter, Detail, Buffer Size) must be surrounded by parenthesis and delimited by commas. Values that are left blank use the default setting for that value.

Value

Description

Filter

Indicates which files to trace.

//DD:filter

Trace will include the DD names matching the specified filter string.

//filter

Trace will include the MVS data sets matching the specified filter string. Member names of partitioned data sets cannot be matched without the use of a wildcard.

filter

Trace will include the Unix files matching the specified filter string.

//DD:*

Trace will include all DD names.

//*

Trace will include all MVS data sets. This is the default setting.

/*

Trace will include all Unix files.

*

Trace will include all MVS data sets and Unix files.

Detail

Indicates the level of detail provided by the trace.

0

No tracing will be performed. This is the default setting.

1

First level tracing will be performed. The trace includes the following information:

- The file being traced.
- The trace detail level and buffer size.
- Details describing how the file was opened (the function called and parameters passed).
- Formatted file data returned from `fldata()`.
- The file pointer address.
- The DD name, if applicable.
- The function flow details for entry to externally documented file I/O functions.

2

Second level tracing will be performed. The trace includes everything that the first level tracing includes, except that the function flow details will be for entry to externally documented and internal slot, exit, OS, and open file I/O functions.

Buffer Size

Indicates the buffer size to use for each file's function flow details.

sizeK

Specifies the size of each file's trace buffer, where size is a number specified in kilobytes. The default buffer size is 16KB.

sizeM

Specifies the size of each file's trace buffer, where size is a number specified in megabytes.

Notes:

1. The wildcard character (*) can be used alone or as a part of a filter. If it is used as a part of a filter, it must be located at the end of the filter string.
2. The Unix file filter must either be a fully qualified name or a file name with no slashes.
3. If the trace buffer size is used up, the trace buffer will wrap, overwriting function flow details only.
4. The usage of `setenv()` with `_EDC_IO_TRACE` only has an effect on future files that are opened.

The following examples show different ways you can use this environment variable.

- The following examples demonstrate two different ways to trace the default of all MVS data sets, using level 1 tracing and the default trace table size:

```
export _EDC_IO_TRACE=(,1,)
setenv("_EDC_IO_TRACE", "(,1,)", 1);
```

- The following examples demonstrate two different ways to trace the UNIX file named `posix.data`, using level 1 tracing and the default trace table size:

```
export _EDC_IO_TRACE=(posix.data,1,)
setenv("_EDC_IO_TRACE", "(posix.data,1,)", 1);
```

- The following examples demonstrate two different ways to trace all MVS data sets with names that begin with `POSIX.DATA`, using level 1 tracing and the default trace table size:

```
export _EDC_IO_TRACE=(//POSIX.DATA*,1,)
setenv("_EDC_IO_TRACE", "(//POSIX.DATA*,1,)", 1);
```

- The following examples demonstrate two different ways to trace all MVS data sets with names that begin with POSIX.DATA, using level 1 tracing and size 24K trace tables:

```
export _EDC_IO_TRACE=(//POSIX.DATA*,1,24K)
setenv("_EDC_IO_TRACE","(//POSIX.DATA*,1,24K)",1);
```

Related information

[“Locating the file I/O trace” on page 171](#)

_EDC_OPEN_CONCAT

Enables users to control if the C library can recognize that a data set is concatenated or not on `fopen()`.

By default, a concatenated data set can only be recognized when it is opened by DDNAME, but it cannot be recognized when it is opened by the data set name. The acceptable value for the `_EDC_OPEN_CONCAT` environment variable is:

Value

Description

1

The C library is able to recognize that the data set is concatenated or not when it is opened by the data set name. With this setting, repositioning operations upon a GDG data set opened by its base name will be based on the beginning of the GDG base.

The `_EDC_OPEN_CONCAT` environment variable can be set with the following function:

```
setenv("_EDC_OPEN_CONCAT","1",1);
```

_EDC_POPEN

Sets the behavior of the `popen()` function. When the value of `_EDC_POPEN` is set to `FORK`, `popen()` uses `fork()` to create the child process. When the value of `_EDC_POPEN` is set to `SPAWN`, `popen()` uses `spawn()` to create the child process. If the value of `_EDC_POPEN` is not set, the default behavior is for `popen()` to use `fork()` to create the child process.

The `_EDC_POPEN` environment variable can be set with the function

```
setenv("_EDC_POPEN","SPAWN", 1);
```

_EDC_PTHREAD_YIELD

Used to control when `pthread_yield()` and `sched_yield()` will allow a thread to give up control of a processor so that another thread may have the opportunity to run. Possible values for `_EDC_PTHREAD_YIELD`:

0

Control of the processor is released immediately.

-1

Use an internal timing algorithm to determine if the processor should be released. This is the default.

-2

Take the machine speed into account when determining if the processor should be released.

other negative values

Invalid. The default (-1) will be used.

Notes:

1. `_EDC_PTHREAD_YIELD` may be changed by `setenv()` while the application is running. All threads will use the new value.
2. The use of `_EDC_PTHREAD_YIELD=-2` is suggested.

Examples of setting `_EDC_PTHREAD_YIELD`:

```
export _EDC_PTHREAD_YIELD=-1
set the ENVAR("_EDC_PTHREAD_YIELD=-2") runtime option
```

`_EDC_PTHREAD_YIELD_MAX`

This environment variable allows a user program to define the max yield (wait) time for a particular thread. It is used to configure the speed at which the `pthread_yield()` and `sched_yield()` functions release a processor to enable another thread to run. In some cases, such as in highly-threaded applications, improved performance may result by using this environment variable to reduce the default max wait time.

Value

Description

positive value

An integer value to set the maximum yield time allowable for a pthread to wait. This value represents microseconds (1/1000 of a millisecond).

Note: Values above 32000 (which is the current max default) and values less than or equal to zero will be ignored.

Examples of setting `_EDC_PTHREAD_YIELD_MAX`:

- This example results in wait times starting at 10 milliseconds, doubling on every successive wait (as it always has) up to a maximum of 20 milliseconds for every wait after that.

```
ENVAR("_EDC_PTHREAD_YIELD_MAX=20000")
(and _EDC_PTHREAD_YIELD=-1)
```

- This example results in wait times starting at an interval determined by the hardware processor speed, and doubling at every successive wait for a maximum wait of 1000 microseconds.

```
ENVAR("_EDC_PTHREAD_YIELD_MAX=1000")
(and _EDC_PTHREAD_YIELD=-2)
```

- This example will result in wait times starting at 1 microsecond, doubling on every successive wait up to a maximum of 1 millisecond.

```
ENVAR("_EDC_PTHREAD_YIELD_MAX=1000")
(and _EDC_PTHREAD_YIELD=4096)
```

Note: A positive integer specified in `_EDC_PTHREAD_YIELD` determines the initial Time Unit a thread will wait for; in this example, 4096 is equivalent to 1 microsecond.

- This example results in wait times that start at 10 milliseconds and remain at that wait time for every successive wait, because the starting yield time is much longer than the intended maximum wait time.

```
ENVAR("_EDC_PTHREAD_YIELD_MAX=10")
(and _EDC_PTHREAD_YIELD=-1)
```

`_EDC_PUTENV_COPY`

Sets the behavior of the `putenv()` function. When the value of `_EDC_PUTENV_COPY` is set to YES, the `putenv()` string is copied into storage owned by Language Environment. When the value of `_EDC_PUTENV_COPY` is **not** set, or set to a value other than YES, then the `putenv()` string is placed directly into the environment, so altering the string will change the environment..

The `_EDC_PUTENV_COPY` environment variable can be set with the function

```
setenv("_EDC_PUTENV_COPY", "YES", 1);
```

Notes:

1. Changes to z/OS specific environment variables beginning with `_BPXK_`, `_CEE_` or `_EDC_` may not be processed if the environment variable is updated directly rather than by using `setenv()` or `putenv()`. Results are unpredictable if these type of environment variables are updated directly.
2. For ASCII applications, the users string will be placed into the environment. However, updates should only be made with `setenv()` or `putenv()`. Results are unpredictable if the environment variable is updated directly.
3. If the user manually changes the environment, storage associated with the original environment may never be freed.
4. The `__putenv_1a()` function will always make a copy of the user string and perform as though `_EDC_PUTENV_COPY=YES` were specified.
5. `_EDC_PUTENV_COPY` may be updated during the life of the application by `setenv()`, `putenv()` or `clearenv()`. This will affect the behavior of any subsequent call to `putenv()`, however it will not change the state of existing environment variables. `putenv()` may be used to update `_EDC_PUTENV_COPY`. The behavior requested will not take effect until the next `putenv()` call.

`_EDC_RRDS_HIDE_KEY`

Applies to VSAM RRDS files opened in record mode. When this environment variable is set, you can call `fread()` with a pointer to a character string, and the Relative Record Number is not appended to the beginning of the record. The `_EDC_RRDS_HIDE_KEY` environment variable is set with the command

```
setenv("_EDC_RRDS_HIDE_KEY", "Y", 1);
```

By default, when you open a VSAM record in record mode, the `fread()` function is called with the RRDS record structure, and the record is preceded by the Relative Record Number.

`_EDC_SIG_DFLT`

To be compliant with the SUSV3 standard, when `_EDC_SIG_DFLT` is set to 1, no message is output to standard error when the signal is handled with default action.

`_EDC_STOR_INCREMENT`

Sets the size of increments to the internal library storage subpool acquired above the 16M line. By default, when the storage subpool is filled, its size is incremented by 8K. When `_EDC_STOR_INCREMENT` is set, its value string is translated to its decimal integer equivalent. This integer is then the new setting of the subpool storage increment size. The setting of this environment variable is only effective if it is done before the first I/O in the enclave.

The `_EDC_STOR_INCREMENT` value must be greater than zero, and must be a multiple of 4K. If the value is less than zero, the default setting of 8K is used. If the value is not a multiple of 4K, then it is rounded up to the next 4K interval. If `_EDC_STOR_INCREMENT` is set to an invalid value that must be modified internally to be divisible by 4K, this modification is not reflected in the character string that appears in the environment variable table.

Consider the case where `setenv()` is called as follows:

```
setenv("_EDC_STOR_INCREMENT", "9000", 1);
```

Internally, the storage subpool increment value is set to 12288 (that is, 12K). However, the following subsequent call returns "9000", as set by the call to `setenv()`.

```
getenv("_EDC_STOR_INCREMENT");
```

Note: `_EDC_STOR_INCREMENT` is not supported in AMODE 64. In AMODE 64 this environment variable is replaced by the IOHEAP64 runtime option.

`_EDC_STOR_INCREMENT_B`

Sets the increment size of an internal library storage subpool acquired below the 16M line. By default, when the below the line storage subpool is filled, its size is incremented by 4K. When `_EDC_STOR_INCREMENT_B` is set, its value string is translated to the decimal equivalent. These integers are then used as the new settings of the below subpool storage increment sizes. The setting of this environment variable is only effective if it is done before the first I/O in the enclave.

Consider the case where `setenv()` is called from CEEBINT (with the CEEBINT user exit linked to the application) as follows:

```
setenv("_EDC_STOR_INCREMENT_B", "1000", 1);
```

Internally, the storage subpool acquired from 24-bit storage will be 4096 (or 4K). However, the following subsequent call returns "1000", as set by the `setenv()` call.

```
getenv("_EDC_STOR_INCREMENT_B");
```

Note: `_EDC_STOR_INCREMENT_B` is not supported in AMODE 64. In AMODE 64, this environment variable is replaced by the IOHEAP64 runtime option.

`_EDC_STORE_INITIAL`

Sets the initial size of the internal library storage subpool acquired above the line. The default subpool storage size is 12K. When `_EDC_STORE_INITIAL` is set, its value string is translated to its decimal integer equivalent. This integer is then the new setting of the subpool storage increment size. The setting of this environment variable is only effective if it is done before the first I/O in the enclave.

The `_EDC_STORE_INITIAL` value must be greater than zero, and must be a multiple of 4K. If the value is less than zero, the default setting of 12K is used. If the value is not a multiple of 4K, then it is rounded up to the next 4K interval. If `_EDC_STORE_INITIAL` is set to an invalid value that must be modified internally to be divisible by 4K, this modification is not reflected in the character string that appears in the environment variable table.

Consider the case where `setenv()` is called from CEEBINT as follows, and the CEEBINT user exit linked to the application:

```
setenv("_EDC_STORE_INITIAL", "16000", 1);
```

Internally, the storage subpool is initialized to 16384 (that is, 16K). However, the subsequent call, shown in the following example, returns "16000", as set by the `setenv()` call:

```
getenv("_EDC_STORE_INITIAL");
```

Note: `_EDC_STORE_INITIAL` is not supported in AMODE 64. In AMODE 64, this environment variable is replaced by the IOHEAP64 runtime option.

`_EDC_STOR_INITIAL_B`

Sets the initial size of an internal library storage subpool acquired below the 16M line. The default below the line subpool storage size is 4K. When `_EDC_STOR_INITIAL_B` is set, its value string is translated to the decimal integer equivalent. This integer is then used as the new setting of the above the line subpool storage initial size. The setting of this environment variable is only effective if it is done before the first I/O in the enclave.

Consider the case where `setenv()` is called from CEEBINT as follows, and with the CEEBINT user exit linked to the application.

```
setenv("_EDC_STOR_INITIAL_B", "1000", 1);
```

Internally, the storage subpool acquired from 24-bit storage will be set to 4096 (that is, 4K). However, the subsequent call `getenv("_EDC_STOR_INITIAL_B");` returns "1000", as set by the `setenv()` call.

Note: `_EDC_STOR_INITIAL_B` is not supported in AMODE 64. In AMODE 64, this environment variable is replaced by the `IOHEAP64` runtime option.

`_EDC_STRPTM_STD`

Indicates changes to `strptime()` that are provided for UNIX standard compliance. It would affect the behavior of the following conversion specifier of the `strptime()` function:

%Y

When the value of `_EDC_STRPTM_STD` is set to 1, at most 4 digits will be consumed. When the value of `_EDC_STRPTM_STD` is set to other values or unset, more than 4 digits might be consumed, and if the generated value is greater than 9999, this function fails.

`_EDC_SUSV3`

Indicates behavioral changes that are provided for SUSV3 compliance in an error path. The affected interfaces are typically setting *errno* to values that were not used before and, in some cases, returning failure for conditions that had not been tested before SUSV3. By default the affected interfaces will not check for these conditions. When the value of `_EDC_SUSV3` is set to 1, the SUSV3 behavior is enabled. When the value of `_EDC_SUSV3` is set to 2, all the behaviors protected by `_EDC_SUSV3=1` are exposed, and pole error related behaviors specified by SUSV3 will be enabled.

For the use of `_EDC_SUSV3` in individual interface, see [z/OS C/C++ Runtime Library Reference](#). The functions that are affected by the `_EDC_SUSV3` environment variable are:

When `_EDC_SUS3 = 1` or `_EDC_SUS3 = 2`:

`setenv()`, `readdir()`, `getnameinfo()`, and `tcgetsid()`

When `_EDC_SUS3 = 2`:

`log()`, `logf()`, `logl()`, `log10()`, `log10f()`, `log10l()`, `log1p()`, `log1pf()`, `log1pl()`, `log2()`, `log2f()`, `log2l()`, `pow()`, and `powl()`

The `_EDC_SUSV3` environment variable can be set with the function:

```
setenv("_EDC_SUSV3", "1", 1);
```

1

Enables SUSV3 behavior for `setenv()`, `readdir()`, `getnameinfo()`, and `tcgetsid()`.

2

Also enables SUSV3 behavior for `log()`, `logf()`, `logl()`, `log10()`, `log10f()`, `log10l()`, `log1p()`, `log1pf()`, `log1pl()`, `log2()`, `log2f()`, `log2l()`, `pow()`, and `powl()`.

`_EDC_UMASK_DFLT`

Allows the user to control how the C library sets the default umask that is used when the program runs. If z/OS UNIX services are available, the possible values of the `_EDC_UMASK_DFLT` environment variable are as follows:

NO (case insensitive)

The C runtime library does not change the umask. The default umask value depends on the value of the system parameter `UMASK`.

A valid octal value

The library uses this octal value as the default umask value no matter how the system parameter `UMASK` is set.

Any other value

When the system parameter `UMASK` is set to a valid value, the C runtime library does not change default umask. When the system parameter `UMASK` is not set, the C runtime library set the default umask value to 022.

_EDC_ZERO_RECLLEN

Allows processing of zero-length records in an MVS Variable file opened in either record or text mode. For more information, see [Chapter 8, “Performing OS I/O operations,” on page 37](#). `_EDC_ZERO_RECLLEN` is set with the command:

```
setenv("_EDC_ZERO_RECLLEN", "Y", 1);
```

This environment variable has no effect on streams based on UNIX file system files. You can always read and write zero-byte records in UNIX file system files.

Propagating environment variables

[Figure 111 on page 359](#) shows a sample program (CCNGEV1) that sets the environment variable `_EDC_ANSI_OPEN_DEFAULT`. A child program is then initiated by a system call. This example shows that environment variables are propagated forward, but not backward.

```
/* this example shows how environment variables are propagated */
/* part 1 of 2-other file is CCNGEV2 */

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

int main(void) {
    char *x;

    /* set the environment variable _EDC_ANSI_OPEN_DEFAULT */
    setenv("_EDC_ANSI_OPEN_DEFAULT", "Y", 1);

    /* set x to the current value of _EDC_ANSI_OPEN_DEFAULT */
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("ccngev1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");

    /* call the child program */
    system("ccngev2");

    /* set x to the current value of _EDC_ANSI_OPEN_DEFAULT */
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("ccngev1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");

    return(0);
}
```

Figure 111. Environment variables example-Part 1

[Figure 112 on page 360](#) is another sample program (CCNGEV2) to show how environment variables are propagated.

```

/* this example shows how environment variables are propagated */
/* part 2 of 2-other file is CCNGEV1 */

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

int main(void) {
    char *x;

    /* set x to the current value of _EDC_ANSI_OPEN_DEFAULT */
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("ccngev2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");

    /* clear the Environment Variables Table */
    clearenv();

    /* set x to the current value of _EDC_ANSI_OPEN_DEFAULT */
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");
    printf("ccngev2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");

    return(0);
}

```

Figure 112. Environment variables example-Part 2

The preceding program produces the following output:

```

cbcgev1 _EDC_ANSI_OPEN_DEFAULT = Y
ccngev2 _EDC_ANSI_OPEN_DEFAULT = Y
ccngev2 _EDC_ANSI_OPEN_DEFAULT = undefined
ccngev1 _EDC_ANSI_OPEN_DEFAULT = Y

```


Chapter 29. Using hardware built-in functions

This section assumes the user has knowledge of assembler opcodes and assembler programming.

A built-in function is inline code that is generated in place of an actual function call. The hardware built-in functions send requests to the compiler to use instructions that are not typically generated by the compiler. Extra instructions are generated to load the parameters for the operation and to store the result. These functions require that the `LANGVL` not be set to `ANSI`. For more information about a given instruction, refer to the *z/Architecture Principles of Operation*.

Notes:

1. Using a built-in hardware instruction does not guarantee that a hardware instruction will be generated. The compiler can decide that it is not necessary to generate the code.
2. In some cases, the instruction will be generated as data before it is executed via an `EX` instruction. This occurs whenever a parameter:
 - Must be put in a mask or displacement field.
 - Is specified as a non-literal instead of a literal.

It is more efficient to execute the target instruction for a hardware built-in function without generating an `EX` instruction. If possible, some parameters of the built-in functions should be specified as literal for better performance. For examples, see [Table 73 on page 389](#).

General instructions

Hardware built-in functions are intended to provide access to instructions of which functionality is not normally provided by the C or C++ programming language. For more information on these instructions, see chapter 7 of the *z/Architecture Principles of Operation*.

Note: Before you use any of the instructions listed in [Table 65 on page 364](#) in your program, you must include the `builtins.h` header file (unless the instructions are otherwise specified) and compile the program with the **`LANGVL (EXTENDED)`** option or the **`LANGVL (LIBEXT)`** option.

Table 64. Standard general-instruction prototypes		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<code>int __cds1(void* OP1, void* OP2, void* OP3)</code> Note: The user must include <code>stdlib.h</code> to use this built-in function. It is similar to <code>cds()</code> but does not explicitly set the type to be swapped in the prototype.	<code>CDS</code> <code>OP1,OP3,OP2D(OP2B)</code>	ARCH(0)
<code>int __cdsg(void* OP1, void* OP2, void* OP3)</code> Note: The user must include <code>stdlib.h</code> to use this built-in function. It is similar to <code>cdsg()</code> but does not explicitly set the type to be swapped in the prototype.	<code>CDSG</code> <code>OP1,OP3,OP2D(OP2B)</code>	ARCH(5) with LP64
<code>int __cs1(void* OP1, void* OP2, void* OP3)</code> Note: <ul style="list-style-type: none">• <code>OP1</code> and <code>OP2</code> cannot be the same memory reference.	<code>CS</code> <code>OP1,OP3,OP2D(OP2B)</code>	ARCH(0)
<code>int __csg(void* OP1, void* OP2, void* OP3)</code> Note: The user must include <code>stdlib.h</code> to use this built-in function.	<code>CSG</code> <code>OP1,OP3,OP2D(OP2B)</code>	ARCH(5) with LP64

Table 64. Standard general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
void __dcbf(const void* OP2) It releases the cache line containing the specified address (OP2) from all accesses.	PFD M1,OP2 Note: M1 is a code generated by the compiler with a value of 7.	ARCH(8)
void __dcbst(const void* OP2) It releases the cache line containing the specified address (OP2) from store access and retains the data in the cache line for fetch access.	PFD M1,OP2 Note: M1 is a code generated by the compiler with a value of 6.	ARCH(8)
void __dcbt(const void* OP2) It prefetches the cache line containing the specified address (OP2) into the cache for fetch access.	PFD M1,OP2 Note: M1 is a code generated by the compiler with a value of 1.	ARCH(8)
void __dcbtst(const void* OP2) It prefetches the cache line containing the specified address (OP2) into the cache for store access.	PFD M1,OP2 Note: M1 is a code generated by the compiler with a value of 2.	ARCH(8)

Table 64. Standard general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>void __pack (unsigned char *OP1, unsigned char op1_len, unsigned char *OP2, unsigned char op2_len)</pre> <ul style="list-style-type: none"> • The format of OP2 is changed from zoned to signed-packed-decimal, and the result is placed at OP1 location. • OP2 is treated as having the zoned format. The numeric bits of each byte are treated as a digit. The zone bits are ignored, except the zone bits in the rightmost byte, which are treated as a sign. • The sign and digits are moved unchanged to OP1 and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field. • The result is obtained as if the operands were processed right to left. When necessary, OP2 is considered to be extended on the left with zeros. If OP1 field is too short to contain all digits of OP2, the remaining leftmost portion of OP2 is ignored. Access exceptions for the unused portion of OP2 may or may not be indicated. • op1_len specifies the zero-based length of OP1, which is the number of additional bytes to the right of OP1. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. op1_len must be in the range of 0 to 15 inclusive. • op2_len specifies the zero-based length of OP2, which is the number of additional bytes to the right of OP2. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. op2_len must be in the range of 0 to 15 inclusive. 	<pre>PACK OP1D(op1_len,OP1B), OP2D(op2_len,OP2B)</pre>	ARCH(0)

Table 64. Standard general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>void __tr (unsigned char *OP1, const unsigned char *OP2, unsigned char len)</pre> <ul style="list-style-type: none"> • The bytes of OP1 are used as eight-bit arguments to reference a list designated by the address of OP2. Each function byte selected from the list replaces the corresponding argument in OP1. • len specifies the zero-based length of OP1, which is the number of additional bytes to the right of OP1. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. len must be in the range of 0 to 255 inclusive. • The bytes of OP1 are selected one by one for translation, proceeding left to right. Each argument byte is added to the initial OP2 address. The addition is performed following the rules for address arithmetic, with the argument byte treated as an eight-bit unsigned binary integer and extended with zeros on the left. The sum is used as the address of the function byte, which then replaces the original argument byte. • The operation proceeds until the OP1 field is exhausted. The list is not altered unless an overlap occurs. • When the operands overlap, the result is obtained as if each result byte were stored immediately after fetching the corresponding function byte. 	<pre>TR OP1D(len,OP1B), OP2D(OP2B)</pre>	ARCH(0)

Table 65. Built-in general-instruction prototypes

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __clcle(char *OP1, unsigned long op1_len, unsigned char OP2, char *OP3, unsigned long op3_len)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. • op1_len specifies the length of OP1. • OP2 specifies the number of bytes to pad the shorter operand on the right. • OP3 represents the third operand in the hardware instruction. • op3_len specifies the length of OP3. <p>Note: If the LP64 compiler option is in effect, the op1_len and op3_len operands are 64-bit unsigned integers. Otherwise, op1_len and op3_len are 32-bit unsigned integers.</p> <p>The return value is the condition code set by the CLCLE instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L R2, OP1 L R3, op1_len L R4, OP3 L R5, op3_len CLCLE R2, R4, OP2D(OP2B)</pre>	ARCH(2)

Table 65. Built-in general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __clclu(unsigned short *OP1, unsigned long op1_len, unsigned short OP2, unsigned short *OP3, unsigned long op3_len)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. • op1_len specifies the length of OP1. • OP2 specifies the number of bytes to pad the first operand on the right, in the event that it is shorter than the third operand. • OP3 represents the third operand in the hardware instruction. • op3_len specifies the length of OP3. <p>Notes:</p> <ul style="list-style-type: none"> • If the LP64 compiler option is in effect, the op1_len and op3_len operands are 64-bit unsigned integers. Otherwise they are 32-bit unsigned integers. • If the operand values are odd numbers, a specification exception will be triggered by the hardware. <p>The return value is the condition code set by the CLCLU instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L R2, OP1 L R3, op1_len L R4, OP3 L R5, op3_len CLCLU R2, R4, OP2D(OP2B)</pre>	ARCH(6)
<pre>int __cu12(unsigned short *OP1, unsigned long op1_len, char *OP2, unsigned long op2_len, char **invalid_utf8, unsigned char mask)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. OP1 points to the storage location for receiving the converted UTF-16 characters. • op1_len specifies the length of OP1. • OP2 represents the second operand in the hardware instruction. OP2 points to the source UTF-8 characters. • op2_len specifies the length of OP2. • invalid_utf8 points to a pointer field for receiving the address of the invalid UTF-8 character in the source when the condition code is "2". • mask specifies the mask encoded in the machine instruction; it must be a literal value of either 0 or 1. <p>Note: If option LP64 is specified, both op1_len and op2_len are 64-bit unsigned integers. Otherwise they are 32-bit unsigned integers.</p> <p>The return value is the condition code set by the CU12 instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L R2, OP1 L R3, op1_len L R4, OP2 L R5, op2_len CU12 R2, R4, Mask</pre>	ARCH(7)

Table 65. Built-in general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<p>int __cu14 (unsigned int *OP1, unsigned long op1_len, char *OP2, unsigned long op2_len, unsigned char **invalid_utf8, unsigned char mask)</p> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. OP1 points to the storage location for receiving the converted UTF-32 characters. • op1_len specifies the length of OP1. • OP2 represents the second operand in the hardware instruction. OP2 points to the source UTF-8 characters. • op2_len specifies the length of OP2. • invalid_utf8 points to a pointer field for receiving the address of the invalid UTF-8 character in the source when the condition code is "2". • mask specifies the mask encoded in the machine instruction; it must be a literal value of either 0 or 1. <p>Note: If option LP64 is specified, both op1_len and op2_len are 64-bit unsigned integers. Otherwise they are 32-bit unsigned integers.</p> <p>The return value is the condition code set by the CU14 instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L R2, OP1 L R3, op1_len L R4, OP2 L R5, op2_len CU14 R2, R4, Mask ST *invalid_utf8,R4</pre>	<p>ARCH(7)</p>
<p>int __cu21(char *OP1, unsigned long op1_len, unsigned short *OP2, unsigned long op2_len, unsigned char mask)</p> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. OP1 points to the storage location for receiving the converted UTF-8 characters. • op1_len specifies the length of OP1. • OP2 represents the second operand in the hardware instruction. OP2 points to the source 2-byte Unicode characters. • op2_len specifies the length of OP2. • mask specifies the mask encoded in the machine instruction; it must be a literal value of either 0 or 1. <p>Note: If option LP64 is specified, both op1_len and op2_len are 64-bit unsigned integers. Otherwise they are 32-bit unsigned integers.</p> <p>The return value is the condition code set by the CU21 instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L R2, OP1 L R3, op1_len L R4, OP2 L R5, op2_len CU21 R2, R4, Mask</pre>	<p>ARCH(7)</p>

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __cu24(unsigned int *OP1, unsigned long op1_len, unsigned short *OP2, unsigned long op2_len, unsigned char mask)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. OP1 points to the storage location for receiving the converted UTF-32 characters. • op1_len specifies the length of OP1. • OP2 represents the second operand in the hardware instruction. OP2 points to the source UTF-16 characters. • op2_len specifies the length of OP2. • mask specifies the mask encoded in the machine instruction; it must be a literal value of either 0 or 1. <p>Note: If option LP64 is specified, both op1_len and op2_len are 64-bit unsigned integers. Otherwise they are 32-bit unsigned integers.</p> <p>The return value is the condition code set by the CU24 instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L R2, OP1 L R3, op1_len L R4, OP2 L R5, op2_len CU21 R2, R4, Mask</pre>	ARCH(7)
<pre>int __cu41(char *OP1, unsigned long op1_len, unsigned int *OP2, unsigned long op2_len, unsigned int **invalid_utf32)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. OP1 points to the storage location for receiving the converted UTF-8 characters. • op1_len specifies the length of OP1. • OP2 represents the second operand in the hardware instruction. OP2 points to the source UTF-32 characters. • op2_len specifies the length of OP2. • invalid_utf32 points to a pointer field for receiving the address of the invalid UTF-32 character in the source when the condition code is "2". <p>Note: If option LP64 is specified, both op1_len and op2_len are 64-bit unsigned integers. Otherwise they are 32-bit unsigned integers.</p> <p>The return value is the condition code set by the CU41 instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L R2, OP1 L R3, op1_len L R4, OP2 L R5, op2_len CU41 R2, R4 ST **invalid_utf32,R4</pre>	ARCH(7)

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __cu42(unsigned short *OP1, unsigned long op1_len, unsigned int *OP2, unsigned long op2_len, unsigned int **invalid_utf32)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. OP1 points to the storage location for receiving the converted UTF-16 characters. • op1_len specifies the length of OP1. • OP2 represents the second operand in the hardware instruction. OP2 points to the source UTF-32 characters. • op2_len specifies the length of OP2. • invalid_utf32 points to a pointer field for receiving the address of the invalid UTF-32 character in the source when the condition code is "2". <p>Note: If option LP64 is specified, both op1_len and op2_len are 64-bit unsigned integers. Otherwise they are 32-bit unsigned integers.</p> <p>The return value is the condition code set by the CU42 instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L R2, OP1 L R3, op1_len L R4, OP2 L R5, op2_len CU42 R2, R4 ST *invalid_utf32, R4</pre>	ARCH(7)
<pre>int __cvb(char *OP2)</pre> <p>OP2 points to an 8-byte storage area that contains a valid packed-decimal value. The return value is the converted 32-bit signed binary integer.</p>	CVB R1, OP2D(OP2X, OP2B)	ARCH(0)
<pre>long long __cvbg(char *OP2)</pre> <p>Operands: OP2 points to a 16-byte storage area that contains a valid packed-decimal value.</p> <p>The return value is the converted 64-bit signed binary integer.</p>	CVBG R1, OP2D(OP2X, OP2B)	ARCH(5)
<pre>void __cvd(int OP1, char *OP2)</pre> <ul style="list-style-type: none"> • OP1 is a 32-bit signed binary integer value that gets converted into a packed-decimal value. • OP2 points to an 8-byte storage area that receives the converted packed-decimal value. 	CVD OP1, OP2D(OP2X, OP2B)	ARCH(0)
<pre>void __cvdg(long long OP1, char *OP2)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 is a 64-bit signed binary integer value that gets converted into a packed-decimal value. • OP2 points to a 16-byte storage area that receives the converted packed-decimal value. 	CVDG OP1, OP2D(OP2X, OP2B)	ARCH(5)

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
int __lad(int* OP1, int OP3, int* OP2) Notes: <ul style="list-style-type: none"> Return value corresponds to the condition code set by LAA. The location pointed to by OP2 must be word aligned for __lad. Otherwise, a specification exception is recognized. 	<pre>L R3, OP3 LAA R1, R3, OP2D(OP2B) ST R1, *OP1</pre>	ARCH(9)
int __ladg(long* OP1, long OP3, long* OP2) Notes: <ul style="list-style-type: none"> Return value corresponds to the condition code set by LAAG. The location pointed to by OP2 must be double-word aligned for __ladg. Otherwise, a specification exception is recognized. 	<pre>LG R3, OP3 LAAG R1, R3, OP2D(OP2B) STG R1, *OP1</pre>	ARCH(9) with LP64
int __ladl(unsigned int* OP1, unsigned int OP3, unsigned int* OP2) Notes: <ul style="list-style-type: none"> Return value corresponds to the condition code set by LAAL. The location pointed to by OP2 must be word aligned for __ladl. Otherwise, a specification exception is recognized. 	<pre>L R3, OP3 LAAL R1, R3, OP2D(OP2B) ST R1, *OP1</pre>	ARCH(9)
int __ladlg(unsigned long* OP1, unsigned long OP3, unsigned long* OP2) Notes: <ul style="list-style-type: none"> Return value corresponds to the condition code set by LAALG. The location pointed to by OP2 must be double-word aligned for __ladlg. Otherwise, a specification exception is recognized. 	<pre>LG R3, OP3 LAALG R1, R3, Op2D(OP2B) STG R1, *OP1</pre>	ARCH(9) with LP64
int __lan(unsigned int* OP1, unsigned int OP3, unsigned int* OP2) Notes: <ul style="list-style-type: none"> Return value corresponds to the condition code set by LAN. The location pointed to by OP2 must be word aligned for __lan. Otherwise, a specification exception is recognized. 	<pre>L R3, OP3 LAN R1, R3, OP2D(OP2B) ST R1, *OP1</pre>	ARCH(9)
int __lang(unsigned long* OP1, unsigned long OP3, unsigned long* OP2) Notes: <ul style="list-style-type: none"> Return value corresponds to the condition code set by LANG. The location pointed to by OP2 must be double-word aligned for __lang. Otherwise, a specification exception is recognized. 	<pre>LG R3, OP3 LANG R1, R3, OP2D(OP2B) STG R1, *OP1</pre>	ARCH(9) with LP64

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __lao(unsigned int* OP1, unsigned int OP3, unsigned int* OP2)</pre> <p>Notes:</p> <ul style="list-style-type: none"> Return value corresponds to the condition code set by LAO. The location pointed to by OP2 must be word aligned for __lao. Otherwise, a specification exception is recognized. 	<pre>L R3, OP3 LAO R1, R3, OP2D(OP2B) ST R1, *OP1</pre>	ARCH(9)
<pre>int __laog(unsigned long* OP1, unsigned long OP3, unsigned long* OP2)</pre> <p>Notes:</p> <ul style="list-style-type: none"> Return value corresponds to the condition code set by LAOG. The location pointed to by OP2 must be double-word aligned for __laog. Otherwise, a specification exception is recognized. 	<pre>LG R3, OP3 LAOG R1, R3, OP2D(OP2B) STG R1, *OP1</pre>	ARCH(9) with LP64
<pre>int __lax(unsigned int* OP1, unsigned int OP3, unsigned int* OP2)</pre> <p>Notes:</p> <ul style="list-style-type: none"> Return value corresponds to the condition code set by LAX. The location pointed to by OP2 must be word aligned for __lax. Otherwise, a specification exception is recognized. 	<pre>L R3, OP3 LAX R1, R3, OP2D(OP2B) ST R1, *OP1</pre>	ARCH(9)
<pre>int __laxg(unsigned long* OP1, unsigned long OP3, unsigned long* OP2)</pre> <p>Notes:</p> <ul style="list-style-type: none"> Return value corresponds to the condition code set by LAXG. The location pointed to by OP2 must be double-word aligned for __laxg. Otherwise, a specification exception is recognized. 	<pre>LG R3, OP3 LAXG R1, R3, OP2D(OP2B) STG R1, *OP1</pre>	ARCH(9) with LP64
<pre>unsigned int __lcbb(const void* OP2, unsigned short boundary)</pre> <p>Operands:</p> <ul style="list-style-type: none"> OP2 represents the second operand in the hardware instruction. boundary represents the boundary to encode in the hardware instruction. It must be a literal value of either 64, 128, 256, 512, 1024, 2048, or 4096. <p>The return value is the first operand set by the LCBB instruction. The first operand contains the number of bytes to load from the second operand location without crossing the specified block boundary. If the number of bytes is greater than 16, sixteen is placed in the first operand.</p>	<pre>LCBB R1, OP2D(OP2X, OP2B), BoundaryM</pre>	ARCH(11)

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __lpd(unsigned int* OP3, unsigned int* OP4, unsigned int* OP1, unsigned int* OP2)</pre> <p>Notes:</p> <ul style="list-style-type: none"> Return value corresponds to the condition code set by LPD. The locations pointed to by OP1 and OP2 must be word aligned for __lpd. Otherwise, a specification exception is recognized. 	<pre>LPD R3, OP1D(OP1B), OP2D(OP2B) ST R3, *OP3 ST R4, *OP4</pre>	ARCH(9)
<pre>int __lpdg(unsigned long* OP3, unsigned long* OP4, unsigned long* OP1, unsigned long* OP2)</pre> <p>Notes:</p> <ul style="list-style-type: none"> Return value corresponds to the condition code set by LPDG. The locations pointed to by OP1 and OP2 must be double-word aligned for __lpdg. Otherwise, a specification exception is recognized. 	<pre>LPDG R3, OP1D(OP1B), OP2D(OP2B) STG R3, *OP3 STG R4, *OP4</pre>	ARCH(9) with LP64
<pre>unsigned int __lrv(unsigned int *OP)</pre> <p>The return value is the result.</p>	LRV R1, OPD(OPX,OPR)	ARCH(4)
<pre>unsigned long __lrvg(unsigned long *OP)</pre> <p>The return value is the result.</p>	LRVG R1, OPD(OPX,OPR)	ARCH(5) with LP64
<pre>unsigned short __lrvh(unsigned short *OP)</pre> <p>The return value is the result.</p>	LRVH R1, OPD(OPX,OPR)	ARCH(4)
<pre>int __mvcle(char *OP1, unsigned long op1_len, unsigned char OP2, char *OP3, unsigned long op3_len)</pre> <p>Operands:</p> <ul style="list-style-type: none"> OP1 represents the first operand in the hardware instruction. op1_len specifies the length of OP1. OP2 specifies the byte for padding the first operand on the right, in the event that it is shorter than the third operand. OP3 represents the third operand in the hardware instruction. op3_len specifies the length of OP3. <p>Note: If the LP64 compiler option is in effect, the op1_len and op3_len operands are 64-bit unsigned integers. Otherwise, op1_len and op3_len are 32-bit unsigned integers.</p> <p>The return value is the condition code set by the MVCLE instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L R2, OP1 L R3, op1_len L R4, OP3 L R5, op3_len MVCLE R2, R4, OP2D(OP2B)</pre>	ARCH(2)

Table 65. Built-in general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __mvclu(unsigned short *OP1, unsigned long op1_len, unsigned short OP2, unsigned short *OP3, unsigned long op3_len)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. • op1_len specifies the length of OP1. • OP2 specifies padding the first operand with 2 bytes if it is shorter than the third operand. • OP3 represents the third operand in the hardware instruction. • op3_len specifies the length of OP3. <p>Notes:</p> <ol style="list-style-type: none"> 1. If the operand values are odd numbers, a specification exception will be triggered by the hardware. 2. If the LP64 compiler option is in effect, the op1_len and op3_len operands are 64-bit unsigned integers. Otherwise they are 32-bit unsigned integers. <p>The return value is the condition code set by the MVCLU instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L R2, OP1 L R3, op1_len L R4, OP3 L R5, op3_len MVCLU R2, R4, OP2D(OP2B)</pre>	ARCH(6)
<pre>void __mvcrl(char *OP1, char *OP2, unsigned char len)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. • OP2 represents the second operand in the hardware instruction. • len specifies the zero-based length of OP1 (or OP2). A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. len+1 equals the number of bytes to copy right to left from OP2 to OP1 location. len must be in the range of 0 to 255 inclusive. <p>The content on the second operand location is placed at the first operand location by copying bytes in a right-to-left sequence, beginning with the rightmost byte of each operand.</p>	<pre>L GR0, len MVCRL OP1, OP2</pre>	ARCH(13)

Table 65. Built-in general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<p><code>int __nc (unsigned char *OP1, unsigned char *OP2, unsigned char len)</code></p> <p>Operands:</p> <ul style="list-style-type: none"> Each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes. <code>len</code> specifies the zero-based length of <code>OP1</code> (or <code>OP2</code>). A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. <code>len+1</code> equals the number of bytes to copy right to left from <code>OP2</code> to <code>OP1</code> location. <code>len</code> must be in the range of 0 to 255 inclusive. <p>The return value is the condition code.</p>	<pre>NC OP1D(len,OP1B), OP2D(OP2B)</pre>	ARCH(0)
<p><code>int __oc (unsigned char *OP1, unsigned char *OP2, unsigned char len)</code></p> <p>Operands:</p> <ul style="list-style-type: none"> Each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes. <code>len</code> specifies the zero-based length of <code>OP1</code> (or <code>OP2</code>), which is the number of additional bytes to the right of <code>OP1</code> (or <code>OP2</code>). A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. <code>len+1</code> equals the number of bytes to perform the OR operation on <code>OP1</code> and <code>OP2</code>. <code>len</code> must be in the range of 0 to 255 inclusive. <p>The return value is the condition code.</p>	<pre>OC OP1D(len,OP1B), OP2D(OP2B)</pre>	ARCH(0)

Table 65. Built-in general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>void __pack (unsigned char *OP1, unsigned char op1_len, unsigned char *OP2, unsigned char op2_len)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • The format of OP2 is changed from zoned to signed-packed-decimal, and the result is placed at OP1 location. • OP2 is treated as having the zoned format. The numeric bits of each byte are treated as a digit. The zone bits are ignored, except the zone bits in the rightmost byte, which are treated as a sign. • The sign and digits are moved unchanged to OP1 and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field. • The result is obtained as if the operands were processed right to left. When necessary, OP2 is considered to be extended on the left with zeros. If OP1 field is too short to contain all digits of OP2, the remaining leftmost portion of OP2 is ignored. Access exceptions for the unused portion of OP2 may or may not be indicated. • op1_len specifies the zero-based length of OP1, which is the number of additional bytes to the right of OP1. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. op1_len must be in the range of 0 to 15 inclusive. • op2_len specifies the zero-based length of OP2, which is the number of additional bytes to the right of OP2. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. op2_len must be in the range of 0 to 15 inclusive. 	<pre>PACK OP1D(op1_len,OP1B), OP2D(op2_len,OP2B)</pre>	ARCH(0)
<pre>void __pka(char *OP1, char *OP2, unsigned char op2_len)</pre> <ul style="list-style-type: none"> • OP1 points to a 16-byte storage area. The packed string is stored in the storage addressed by OP1. • OP2 points to an ASCII string to be packed by the operation. • op2_len specifies the zero-based length of OP2, which is the number of additional bytes to the right of OP2. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. <p>Notes:</p> <ol style="list-style-type: none"> 1. If op2_len is not between 0 and 31, a specification exception will be triggered by the hardware. 2. If op2_len is not a literal, the compiler will issue an EX instruction that executes a target PKA instruction with op2_len encoded in the register used by the EX instruction. 	<pre>PKA OP1D(OP1B), OP2D(op2_len, OP2B)</pre>	ARCH(6)

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>void __pku(char *OP1, unsigned short *OP2, unsigned char op2_len)</pre> <ul style="list-style-type: none"> OP1 points to a 16-byte storage area. The packed string is stored in the storage addressed by OP1. OP2 points to Unicode Basic Latin character string to be packed by the operation. op2_len specifies the zero-based length of OP2, which is the number of additional bytes to the right of OP2. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. <p>Notes:</p> <ol style="list-style-type: none"> If op2_len is not an odd number between 0 and 63, a specification exception will be triggered by the hardware. If op2_len is not a literal, the compiler will issue an EX instruction that executes a target PKU instruction with op2_len encoded in the register used by the EX instruction. 	<pre>PKU OP1D(OP1B), OP2D(op2_len, OP2B)</pre>	ARCH(6)
<pre>unsigned long __popcnt(unsigned long OP)</pre> <p>The population count instruction counts the number of bits set in each byte of a register. It then sets the corresponding byte of the result register to reflect the count.</p> <p>The instruction operates on all 64-bits of a register and thus clobbers the upper half of a register in 32-bit mode. Volatile registers R0, R1, and R15 are used for the instruction in 32-bit mode with NOHGPR.</p>	<pre>POPCNT R1, OP</pre>	ARCH(9)
<pre>int __popcnt4(unsigned int OP)</pre> <p>Returns the number of bits set for a 32-bit integer.</p> <p>The instruction operates on all 64-bits of a register and thus clobbers the upper half of a register in 32-bit mode. Volatile registers R0, R1, and R15 are used for the instruction in 32-bit mode with NOHGPR.</p>	<pre>POPCNT R1, OP, 1</pre>	ARCH(13)
<pre>int __popcnt8(unsigned long long OP)</pre> <p>Returns the number of bits set for a 64-bit integer.</p> <p>The instruction operates on all 64-bits of a register and thus clobbers the upper half of a register in 32-bit mode. Volatile registers R0, R1, or R15 are used for the instruction in 32-bit mode with NOHGPR.</p>	<pre>POPCNT R1, OP, 1</pre>	ARCH(13)

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __srstu (unsigned short *OP1, unsigned short *OP2, unsigned short pattern, unsigned short **found_char)</pre> <p>Operands:</p> <ul style="list-style-type: none"> OP1 represents the first operand in the hardware instruction, it points at the first 2-byte character after the end of the second operand. OP2 represents the second operand in the hardware instruction, it points at the start of a 2-byte character string. pattern is the 2-byte character to be searched for. found_char points to a pointer field for receiving the address of the 2-byte character that was found in the second operand. <p>The return value is the condition code set by the SRSTU instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L GR0, pattern L R1, OP1 SRSTU R1, OP2 ST *found_char, R1</pre>	ARCH(7)
<pre>int __stck(unsigned long long *OP1)</pre> <p>The return value is the condition code.</p>	STCK Op1D(OP1B)	ARCH(0)
<pre>int __stcke(void *OP1)</pre> <p>The return value is the condition code.</p>	STCKE Op1D(OP1B)	ARCH(4)
<pre>int __stckf(unsigned long long *OP1)</pre> <p>The return value is the condition code.</p>	STCKF Op1D(OP1B)	ARCH(7)
<pre>void __strv(unsigned int OP1, unsigned int *OP2)</pre>	STRV R1, OP2D(OP2X,OP2R)	ARCH(4)
<pre>void __strvg(unsigned long OP1, unsigned long *OP2)</pre>	STRVG R1, OP2D(OP2X,OP2R)	ARCH(5) with LP64
<pre>void __strvh(unsigned short OP1,unsigned short *OP2)</pre>	STRVH R1, OP2D(OP2X,OP2R)	ARCH(4)

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>void __tr (unsigned char *OP1, const unsigned char *OP2, unsigned char len)</pre> <p>Operands:</p> <ul style="list-style-type: none"> The bytes of OP1 are used as eight-bit arguments to reference a list designated by the address of OP2. Each function byte selected from the list replaces the corresponding argument in OP1. len specifies the zero-based length of OP1, which is the number of additional bytes to the right of OP1. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. len must be in the range of 0 to 255 inclusive. The bytes of OP1 are selected one by one for translation, proceeding left to right. Each argument byte is added to the initial OP2 address. The addition is performed following the rules for address arithmetic, with the argument byte treated as an eight-bit unsigned binary integer and extended with zeros on the left. The sum is used as the address of the function byte, which then replaces the original argument byte. The operation proceeds until the OP1 field is exhausted. The list is not altered unless an overlap occurs. When the operands overlap, the result is obtained as if each result byte were stored immediately after fetching the corresponding function byte. 	<pre>TR OP1D(len,OP1B), OP2D(OP2B)</pre>	ARCH(0)
<pre>int __tre(char *OP1, unsigned long op1_len, char *OP2, unsigned char test_char)</pre> <p>Operands:</p> <ul style="list-style-type: none"> OP1 represents the first operand in the hardware instruction. It points to the byte string that needs to be translated. op1_len specifies the length of OP1. <p>Note: If the LP64 compiler option is in effect, the op1_len operand is a 64-bit unsigned integer. Otherwise it is a 32-bit unsigned integer.</p> <ul style="list-style-type: none"> OP2 represents the second operand in the hardware instruction. It points to a 256-byte translation table. test_char specifies the terminating character in the first operand for stopping the operation. <p>The return value is the condition code set by the TRE instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	TRE R1, OP2	ARCH(5)

Table 65. Built-in general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __troo (char *OP1, char *OP2, unsigned long op2_len, char *tr_table, unsigned char test_char, unsigned char mask)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. • OP2 represents the second operand in the hardware instruction. • op2_len specifies the length of OP2. <p>Note: If option LP64 is specified, op2_len is a 64-bit unsigned integer and the length of the first operand is considered the same as that of the second operand. If option LP64 is not specified, both operands are 32-bit unsigned integers.</p> <ul style="list-style-type: none"> • tr_table points to a 256-byte translation table on a double-word boundary. <p>Note: It is the user's responsibility to provide a double-word aligned translation table.</p> <ul style="list-style-type: none"> • test_char specifies a 1-byte function character that can be coded in the translation table for stopping the operation. • mask specifies the mask encoded in the machine instruction; it must be a literal value of either 0 or 1. <p>The return value is the condition code set by the TROO instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L GR0, test_char L GR1, tr_table L R2, OP1 L R3, op2_len TROO R2, OP2, Mask</pre>	<p>ARCH(7)</p>

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __trot(unsigned short *OP1, char *OP2, unsigned long op2_len, char *tr_table, unsigned short test_char, unsigned char mask)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. • OP2 represents the second operand in the hardware instruction. • op2_len specifies the length of OP2. <p>Note: If option LP64 is specified, op2_len is a 64-bit unsigned integer and the length of the first operand is considered the same as that of the second operand. Otherwise, both operands are 32-bit unsigned integers.</p> <ul style="list-style-type: none"> • tr_table points to a 512-byte translation table on a double-word boundary. <p>Note: It is the user's responsibility to provide a double-word aligned translation table.</p> <ul style="list-style-type: none"> • test_char specifies a 2-byte function character that can be coded in the translation table for stopping the operation. • mask specifies the mask encoded in the machine instruction; it must be a literal value of either 0 or 1. <p>The return value is the condition code set by the TRTO instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L GR0, test_char L GR1, tr_table L R2, OP1 L R3, op2_len TROT R2, OP2, Mask</pre>	ARCH(7)
<pre>int __trto(char *OP1, unsigned short *OP2, unsigned long op2_len, char *tr_table, char test_char, unsigned char mask)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. • OP2 represents the second operand in the hardware instruction. • op2_len specifies the length of OP2. <p>Note: If option LP64 is specified, op2_len is an 64-bit unsigned integer. If option LP64 is not specified, op2_len is an 32-bit unsigned integer.</p> <ul style="list-style-type: none"> • tr_table points to a 64-KB translation table on a double-word boundary. <p>Note: It is the user's responsibility to provide a double-word aligned translation table.</p> <ul style="list-style-type: none"> • test_char specifies a 1-byte function character, that can be coded in the translation table, for stopping the operation. • mask specifies the mask encoded in the machine instruction; it must be a literal value of either 0 or 1. <p>The return value is the condition code set by the TROT instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L GR0, test_char L GR1, tr_table L R2, OP1 L R3, op2_len TRTO R2, OP2, Mask</pre>	ARCH(7)

Table 65. Built-in general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __trt (unsigned char *OP1, const unsigned char *OP2, unsigned char len, unsigned char *R2, unsigned char **R1)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • The bytes of OP1 are used as eight-bit arguments to select function bytes from a list designated by the address of OP2. The first nonzero function byte is inserted in general register 2, and the related argument address in general register 1. • len specifies the zero-based length of OP1, which is the number of additional bytes to the right of OP1. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. len must be in the range of 0 to 255 inclusive. • The bytes of OP1 are selected one by one for translation, proceeding left to right. OP1 remains unchanged in storage. Calculation of the address of the function byte is performed as in the __tr instruction. The function byte retrieved from the list is inspected for a value of zero. • When the function byte is zero, the operation proceeds with the next byte of OP1. • When the function byte is nonzero, the operation is completed by inserting the function byte in general register 2 and the related argument address in general register 1. Either condition code 1 or 2 is set, depending on whether the argument byte is the rightmost byte of OP1. Condition code 1 is set if one or more argument bytes remain to be translated. Condition code 2 is set if no more argument bytes remain. <p>The return value is the condition code.</p>	<pre>TRT OP1D(len,OP1B), OP2D(OP2B)</pre>	<p>ARCH(0)</p>

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __trtr (unsigned char *OP1, const unsigned char *OP2, unsigned char len, unsigned char *R2, unsigned char **R1)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction; it points to the last byte of the byte string to be translated and tested. • OP2 represents the second operand in the hardware instruction; it points to a 256-byte table. • len specifies the zero-based length of OP1, which is the number of additional bytes to the right of OP1. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. len must be in the range of 0 to 255 inclusive. <p>Note: When len is not specified as a literal, an EX instruction is generated to execute a target TRTR instruction with len encoded in the register used by the EX instruction.</p> <ul style="list-style-type: none"> • R2 points to a one-byte storage for receiving the function byte in GPR 2 when the condition code is nonzero. • R1 points to a pointer field for receiving the address in GPR 1 when the condition code is nonzero. <p>The return value is the condition code set by the TRTR instruction.</p>	<pre>TRTR OP1D(len, OP1B), OP2D(OP2B) ST *R2, GR2 ST *R1, GR1</pre>	ARCH(7)
<pre>int __trtt(unsigned short *OP1, unsigned short *OP2, unsigned long op2_len, char*tr_table, unsigned short test_char, unsigned char mask)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 represents the first operand in the hardware instruction. • OP2 represents the second operand in the hardware instruction. • op2_len specifies the length of OP2. <p>Note: If option LP64 is specified, op2_len is a 64-bit unsigned integer and the length of the first operand is considered the same as that of the second operand. Otherwise, both operands are 32-bit unsigned integers.</p> <ul style="list-style-type: none"> • tr_table points to a 128-KB translation table on a double-word boundary. <p>Note: It is the user's responsibility to provide a double-word aligned translation table.</p> <ul style="list-style-type: none"> • test_char specifies a 2-byte function character that can be coded in the translation table for stopping the operation. • mask specifies the mask encoded in the machine instruction; it must be a literal value of either 0 or 1. <p>The return value is the condition code set by the TRTT instruction.</p> <p>Note: When the condition code is 3, the condition is handled by the compiler-generated code.</p>	<pre>L GR0, test_char L GR1, tr_table L R2, OP1 L R3, op2_len TRTT R2, OP2, Mask</pre>	ARCH(7)

Table 65. Built-in general-instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>void __unpk (unsigned char *OP1, unsigned char op1_len, unsigned char *OP2, unsigned char op2_len)</pre> <p>Operands:</p> <ul style="list-style-type: none"> • The format of OP2 is changed from signed-packed-decimal to zoned, and the result is placed at OP1 location. • OP2 is treated as having the signed-packed-decimal format. Its digits and sign are placed unchanged in OP1 location, using the zoned format. Zone bits with coding of 1111 are supplied for all bytes except the rightmost byte, the zone of which receives the sign of OP2. The sign and digits are not checked for valid codes. • The result is obtained as if the operands were processed right to left. When necessary, OP2 is considered to be extended on the left with zeros. If OP1 field is too short to contain all digits of OP2, the remaining leftmost portion of OP2 is ignored. Access exceptions for the unused portion of OP2 may or may not be indicated. • op1_len specifies the zero-based length of OP1, which is the number of additional bytes to the right of OP1. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. op1_len must be in the range of 0 to 15 inclusive. • op2_len specifies the zero-based length of OP2, which is the number of additional bytes to the right of OP2. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. op2_len must be in the range of 0 to 15 inclusive. 	<pre>UNPK OP1D(op1_len,OP1B), OP2D(op2_len,OP2B)</pre>	ARCH(0)
<pre>int __unpka(char *OP1, unsigned char op1_len, char *OP2);</pre> <p>Operands:</p> <ul style="list-style-type: none"> • OP1 points to a maximum 32-byte storage area to receive the unpacked data from the second operand. • op1_len specifies the length encoded in the machine instruction (that is, the number of additional bytes to the right of the first operand). <p>Note: If op1_len is not a literal, the compiler will issue an EX instruction that executes a target UNPKA instruction with op1_len encoded in the register used by the EX instruction.</p> <ul style="list-style-type: none"> • OP2 points to a 16-byte data string that represents 31 digits and a sign. <p>The return value is the condition code set by the UNPKA instruction.</p>	<pre>UNPKA OP1D(op1_len, OP1B), OP2D(OP2B)</pre>	ARCH(6)

Table 65. Built-in general-instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __unpku(unsigned short *OP1, unsigned char op1_len, char *OP2);</pre> <p>Operands:</p> <ul style="list-style-type: none"> OP1 points to a maximum 64-byte storage area to receive the unpacked data from the second operand. op1_len specifies the zero-based length of OP1, which is the number of additional bytes to the right of OP1. A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. <p>Notes:</p> <ol style="list-style-type: none"> If op1_len is not a literal, the compiler will issue an EX instruction that executes a target UNPKU instruction with op1_len encoded in the register used by the EX instruction. If op1_len is not an odd number between 0 and 63, a specification exception will be triggered by the hardware. <ul style="list-style-type: none"> OP2 points to a 16-byte data string that represents 31 digits and a sign. <p>The return value is the condition code set by the UNPKU instruction.</p>	<pre>UNPKU OP1D(op1_len, OP1B), OP2D(OP2B)</pre>	ARCH(6)
<pre>int __xc (unsigned char *OP1, unsigned char *OP2, unsigned char len)</pre> <p>Operands:</p> <ul style="list-style-type: none"> Each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes. len specifies the zero-based length of OP1 (or OP2), which is the number of additional bytes to the right of OP1 (or OP2). A length of 1 is represented by 0, a length of 2 is represented by 1, and so forth. len+1 equals the number of bytes to perform the XOR operation on OP1 and OP2. len must be in the range of 0 to 255 inclusive. <p>The return value is the condition code.</p>	<pre>XC OP1D(len,OP1B), OP2D(OP2B)</pre>	ARCH(0)

PLO - Perform Locked Operation available in ARCH(5)

With the PLO instruction, you can perform the following atomic read-modify-write operations:

- compare and load
- compare and swap
- double compare and swap
- compare swap and store
- compare swap and double store
- compare swap and triple store

To perform a particular operation, the PLO requires an address to a lock, a function code that specifies the operation to be performed and the relevant operands. Each PLO operation has four built-in functions associated with it. The functions take 32-bit, 64-bit, and 128-bit operands. Applications written with built-ins that take 64-bit or 128-bit operands need to be compiled and linked with LP64 option. In other

words, PLO built-ins that end with the letter G, GR(64-bit operands), or X(128 bit operands) need to be compiled with LP64 option. For example, for the "compare and load" interface, the functions `__plo_CLGR`, `__plo_CLG`, and `__plo_CLX` need to be compiled with LP64 option.

The function prototypes, associated types, and helper macros will be declared in `builtins.h` header file. The argument names and order in the function prototypes attempt to reflect the description of the hardware instructions in *z/Architecture® Principles of Operation*. A brief description is provided for each function prototype, data types, and helper macros.

Note: For every 8-byte PLO operation, there are two interfaces, for example, `__plo_DSCG` and `__plo_DCSGR`. The distinction between these two prototypes is that the former uses a parameter list to receive all its arguments.

All the atomic read-modify-write operations require a memory location, which is to be used as a lock and is the first argument for all the prototypes. The return type for all the functions is a signed integer, which returns the condition code set by the PLO instruction.

Note: There is no AR mode support. You need to enable support for the unsigned long long type to use the PLO interface. You also need to enable 64 bit mode compilation when using PLO built-ins that take 64-bit or 128-bit operands.

Associated types and helper macros

For certain function codes, the PLO instruction takes the address to a parameter list that contains all the operands needed to perform the specified operation. The parameter list is a contiguous region in memory where the operands are stored. The parameter list has to be double word aligned, and for each atomic operation the layout of the operands in the list is different. All unused fields in the parameter list have to contain zeros.

To simplify the setup of the parameter list, an interface is provided to setup the parameter list prior to passing it to any of the built-in functions. The macros and the types are defined in `builtins.h` header file.

```
#define __PLO_PARAM_LIST_MAX_SIZE 18

typedef unsigned long long __plo_plist[ __PLO_PARAM_LIST_MAX_SIZE ]

//Compare and Load
#define __PLO_CL 0
#define __PLO_CLG 1
#define __PLO_CLGR 2
#define __PLO_CLX 3

//Compare and Swap
#define __PLO_CS 4
#define __PLO_CSG 5
#define __PLO_CSGR 6
#define __PLO_CSX 7

//Double Compare and Swap
#define __PLO_DCS 8
#define __PLO_DCSG 9
#define __PLO_DCSGR 10
#define __PLO_DCSX 11

//Compare Swap and Store
#define __PLO_CSST 12
#define __PLO_CSSTG 13
#define __PLO_CSSTGR 14
#define __PLO_CSSTX 15

//Compare Swap and Double Store
#define __PLO_CSDST 16
#define __PLO_CSDSTG 17
#define __PLO_CSDSTGR 18
#define __PLO_CSDSTX 19

//Compare Swap and Triple Store
#define __PLO_CSTST 20
#define __PLO_CSTSTG 21
```



```
#define __PLO_CSTSTGR 22
#define __PLO_CSTSTX 23
```

An array of fixed size is defined to correspond to the largest size the parameter list can have, and various function codes are defined.

Table 66 on page 385 describes the helper macros.

Table 66. PLO helper macros	
Helper Macros	Description
<code>__PLO_INIT_PARAM_LIST(param_list)</code>	Helper macro to initialize the parameter list and must be called prior to loading the parameter list with any values.
<code>__PLO_PUTVAL_PARAM_LIST4(function_code, param_list, uint_op1c, uint_op1r, uint_op3, uint_op4_ptr, uint_op5, uint_op6_ptr, uint_op7, uint_op8_ptr)</code>	Helper macro to setup the parameter list for read-modify write operations on 32 bit operands. The macro takes the function code for the operation that will be performed on the parameter list, a superset of the operands that might be required and the parameter list itself.
<code>__PLO_PUTVAL_PARAM_LIST8(function_code, param_list, ulonglong_op1c, ulonglong_op1r, ulonglong_op3, ulonglong_op3c, ulonglong_op3r, ulonglong_op4_ptr, ulonglong_op5, ulonglong_op6_ptr, ulonglong_op7, ulonglong_op8_ptr)</code>	Helper macro to setup the parameter list for read-modify write operations on 64-bit operands.
<code>__PLO_PUTVAL_PARAM_LIST16(function_code, param_list, ulonglong_op1c_ptr, ulonglong_op1r_ptr, ulonglong_op3_ptr, ulonglong_op3c_ptr, ulonglong_op3r_ptr, ulonglong_op4_ptr, ulonglong_op5_ptr, ulonglong_op6_ptr, ulonglong_op7_ptr, ulonglong_op8_ptr)</code>	Helper macro to setup the parameter list for 128 bit operands. Note that for 128 bit operands, pointers are taken to a 16 byte region in memory as arguments.
<code>__PLO_GETVAL_PARAM_LIST8(function_code, param_list, ulonglong_op1c, ulonglong_op3, ulonglong_op3c)</code>	Helper macro to read 64-bit values that are updated by the execution of a PLO operation from the parameter list.
<code>__PLO_GETVAL_PARAM_LIST16(function_code, param_list, ulonglong_op1c_ptr, ulonglong_op3_ptr, ulonglong_op3c_ptr)</code>	Helper macro to read 128-bit values that are updated by the execution of a PLO operation from the parameter list.

Compare and Load

Table 67 on page 386 describes the prototypes for compare and load operations.

Table 67. Compare and load prototypes		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __plo_CL(void * lock, unsigned int * op1c, unsigned int * op2, unsigned int * op3, unsigned int * op4);</pre> <pre>int __plo_CLGR(void * lock, unsigned long long * op1c, unsigned long long * op2, unsigned long long * op3, unsigned long long * op4);</pre>	<pre>L GR1, lock L GR0, function_code(0 or 2) L R1, *op1c L R3, *op3 PLO R1, R3, op2, op4 ST *op3, R3 ST *op1c, R1</pre>	ARCH(5)
<pre>int __plo_CLG (void * lock, unsigned long long* op2, void * param_list);</pre> <pre>int __plo_CLX (void * lock, void* op2, void * param_list);</pre> <p>Note: "void * param_list" is a pointer to the parameter list that is discussed in section Associated types and helper macros.</p>	<pre>L GR1, lock L GR0, function_code(1 or 3) PLO op2, param_list</pre>	ARCH(5)

Compare and Swap

Table 68 on page 386 describes the prototypes for compare and swap operations.

Table 68. Compare and swap prototypes		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __plo_CS(void * lock, unsigned int * op1c, unsigned int op1r, unsigned int * op2);</pre> <pre>int __plo_CSGR(void * lock, unsigned long long * op1c, unsigned long long op1r, unsigned long long * op2);</pre>	<pre>L GR1, lock L GR0, function_code (4 or 6) L R1, *op1c L R1+1, op1r PLO R1, op2 ST R1, *op1c</pre>	ARCH(5)
<pre>int __plo_CSG (void * lock, unsigned long long * op2, void * param_list);</pre> <pre>int __plo_CSX(void * lock, void * op2, void * param_list);</pre> <p>Note: "void * param_list" is a pointer to the parameter list that is discussed in section Associated types and helper macros.</p>	<pre>L GR1, lock L GR0, function_code (5 or 7) PLO op2, param_list</pre>	ARCH(5)

Double Compare and Swap

Table 69 on page 387 describes the prototypes for double compare and swap operations.

Table 69. Double compare and swap prototypes

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __plo_DCS(void * lock, unsigned int * op1c, unsigned int op1r, unsigned int * op2, unsigned int * op3c, unsigned int op3r, unsigned int * op4);</pre> <pre>int __plo_DCSGR(void * lock, unsigned long long * op1c, unsigned long long op1r, unsigned long long * op2, unsigned long long * op3c, unsigned long long op3r, unsigned long long * op4);</pre>	<pre>L GR1, lock L GR0, function_code (8 or 10) L R1, *op1c L R1+1, op1r L R3, *op3c L R3+1, op3r PLO R1,R3, op2,op4 ST R1, *op1c ST R3, *op3c</pre>	ARCH(5)
<pre>int __plo_DCSG(void * lock, unsigned long long * op2, void * param_list);</pre> <pre>int __plo_DCSX(void * lock, void * op2, void * param_list);</pre> <p>Note: "void * param_list" is a pointer to the parameter list that is discussed in section Associated types and helper macros.</p>	<pre>L GR1, lock L GR0, function_code (9 or 11) PLO op2, param_list</pre>	ARCH(5)

Compare and Swap and Store

Table 70 on page 387 describes the prototypes for compare and swap and store operations.

Table 70. Compare and swap and store prototypes

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __plo_CSST (void * lock, unsigned int * op1c, unsigned int op1r, unsigned int * op2, unsigned int op3, unsigned int * op4);</pre> <pre>int __plo_CSSTGR (void * lock, unsigned long long * op1c, unsigned long long op1r, unsigned long long * op2, unsigned long long op3, unsigned long long * op4);</pre>	<pre>L GR1, lock L GR0, function_code (12 or 14) L R1, *op1c L R1+1, op1r L R3, op3 PLO R1,R3,op2,op4 ST R1, *op1c</pre>	ARCH(5)
<pre>int __plo_CSSTG (void * lock, unsigned long long * op2, void * param_list);</pre> <pre>int __plo_CSSTX (void * lock, void * op2, void * param_list);</pre> <p>Note: "void * param_list" is a pointer to the parameter list that is discussed in section Associated types and helper macros.</p>	<pre>L GR1, lock L GR0, function_code (13 or 15) PLO op2, param_list</pre>	ARCH(5)

Compare Swap and Double Store

Table 71 on page 388 describes the prototypes for compare, swap, and double store operations.

Table 71. Compare swap and double store prototypes		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __plo_CSDST (void * lock, unsigned int * op1c, unsigned int op1r, unsigned int * op2, void * param_list);</pre> <pre>int __plo_CSDSTGR(void * lock, unsigned long long * op1c, unsigned long long op1r, unsigned long long * op2, void * param_list);</pre>	<pre>L GR1, lock L GR0, function_code (16 or 18) L R1, *op1c L R1+1, op1r PLO R1, op2, param_list ST R1, *op1c</pre>	ARCH(5)
<pre>int __plo_CSDSTG (void * lock, unsigned long long * op2, void * param_list);</pre> <pre>int __plo_CSDSTX(void * lock, void * op2, void * param_list);</pre> <p>Note: "void * param_list" is a pointer to the parameter list that is discussed in section Associated types and helper macros.</p>	<pre>L GR1, lock L GR0, function_code (17 or 19) PLO op2, param_list</pre>	ARCH(5)

Compare and Swap and Triple Store

Table 72 on page 388 describes the prototypes for compare and swap and triple store operations.

Table 72. Compare and swap and triple store prototypes		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __plo_CSTST (void * lock, unsigned int * op1c, unsigned int op1r, unsigned int * op2, void * param_list);</pre> <pre>int __plo_CSTSTGR(void * lock, unsigned long long * op1c, unsigned long long op1r, unsigned long long * op2, void * param_list);</pre>	<pre>L GR1, lock L GR0, function_code (20 or 22) L R1, *op1c L R1+1, op1r PLO R1, op2, param_list ST R1, *op1c</pre>	ARCH(5)
<pre>int __plo_CSTSTG (void * lock, unsigned long long * op2, void * param_list);</pre> <pre>int __plo_CSTSTX (void * lock, void * op2, void * param_list);</pre> <p>Note: "void * param_list" is a pointer to the parameter list that is discussed in section Associated types and helper macros.</p>	<pre>L GR1 lock L GR0 function_code (21 or 23) PLO op2, param_list</pre>	ARCH(5)

Decimal instructions

Hardware packed-decimal instructions are available to C/C++ programs in the form of built-in functions. These hardware built-in functions are intended to provide access to decimal instructions that are not typically generated by the compiler.

Decimal instructions of SS format carry one or two length fields. Each length field is encoded with a binary length, that is the actual length - 1. In the function prototypes, the length parameters need to be specified as the actual length - 1. Argument names and order reflects the description of the hardware instructions in *z/Architecture Principles of Operation* (that is, op1, op2, op3, etc.) Additional arguments provide required information for setting up the actual hardware instruction. For detailed description of each decimal instruction, see *Chapter 8. Decimal Instructions* of the *z/Architecture Principles of Operation*.

If you want to use any of the decimal functions, your program must include `builtins.h` and be compiled with either the `LANGLVL(EXTENDED)` option or the `LANGLVL(LIBEXT)` option.

Table 73. Decimal instruction prototypes

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __ap (unsigned char *op1, unsigned char len1, unsigned char *op2, unsigned char len2);</pre> <p>Operands:</p> <ul style="list-style-type: none">• op1 represents the first operand in the hardware instruction. It points to the first operand location. The result replaces the first operand.• len1 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the first operand). The value is between 0 and 15.• op2 represents the second operand in the hardware instruction. It points to the second operand location.• len2 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the second operand). The value is between 0 and 15. <p>Note: When either len1 or len2 is not specified as a literal, an EX instruction is generated to execute a target AP instruction with length encoded in the register used by the EX instruction.</p> <p>The return value is the condition code set by the AP instruction.</p>	<pre>AP Op1D(len1, Op1B), Op2D(len2, Op2B)</pre>	<pre>ARCH(0)</pre>

Table 73. Decimal instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __cp (unsigned char *op1, unsigned char len1, unsigned char *op2, unsigned char len2);</pre> <p>Operands:</p> <ul style="list-style-type: none"> • op1 represents the first operand in the hardware instruction. It points to the first operand location. • len1 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the first operand). The value is between 0 and 15. • op2 represents the second operand in the hardware instruction. It points to the second operand location. • len2 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the second operand). The value is between 0 and 15. <p>Note: When either len1 or len2 is not specified as a literal, an EX instruction is generated to execute a target CP instruction with length encoded in the register used by the EX instruction.</p> <p>The return value is the condition code set by the CP instruction.</p>	<pre>CP Op1D(len1, Op1B), Op2D(len2, Op2B)</pre>	ARCH(0)
<pre>void __dp (unsigned char *op1, unsigned char len1, unsigned char *op2, unsigned char len2);</pre> <p>Operands:</p> <ul style="list-style-type: none"> • op1 represents the first operand (dividend) in the hardware instruction. It points to the first operand location. The result replaces the first operand. The quotient is placed leftmost in this location. The number of bytes in the quotient field is equal to the difference between the dividend and divisor lengths (len1 - len2). The remainder is placed rightmost in this location and has a length equal to the divisor length len2. • len1 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the first operand). The value must be greater than len2 and not greater than 15. • op2 represents the second operand (divisor) in the hardware instruction. It points to the second operand location. • len2 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the second operand). The value is between 0 and 7 and must be less than the value of len1. <p>Note: When either len1 or len2 is not specified as a literal, an EX instruction is generated to execute a target DP instruction with length encoded in the register used by the EX instruction.</p>	<pre>DP Op1D(len1, Op1B), Op2D(len2, Op2B)</pre>	ARCH(0)

Table 73. Decimal instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __ed (unsigned char *OP1, unsigned char *OP2, unsigned char length);</pre> <p>Operands:</p> <ul style="list-style-type: none"> • op2 (the source), which normally contains one or more decimal numbers in the signed-packed-decimal or unsigned-packed-decimal format, is changed to the zoned format and modified under the control of op1 (the pattern). The edited result replaces op1. • len specifies the length encoded in the machine instruction (that is, the number of additional bytes to the right of the first operand). • The length of op2 is determined by the operation according to the contents of the pattern. The leftmost four bits of each source byte must specify a decimal-digit code (0000-1001); a sign code (1010-1111) is recognized as a data exception. The rightmost four bits may specify either a sign code or a decimal-digit code. Access and data exceptions are recognized only for those bytes in op2 which are actually required. <p>The return value is the condition code.</p>	<pre>ED Op1D(len,Op1B), Op2D(Op2B)</pre>	ARCH(0)
<pre>int __edmk (unsigned char *OP1, unsigned char *OP2, unsigned char length, unsigned char **R1);</pre> <p>Operands:</p> <ul style="list-style-type: none"> • op2 (the source), which normally contains one or more decimal numbers in the signed-packed-decimal or unsigned-packed-decimal format, is changed to the zoned format and modified under the control of op1 (the pattern). The edited result replaces op1. • len specifies the length encoded in the machine instruction (that is, the number of additional bytes to the right of the first operand). • __edmk is identical to __ed except for the additional function of inserting the address of the result byte in general register 1 if the result byte is a zoned source digit and the significance indicator was off before the examination. If no result byte meets the criteria, general register 1 remains unchanged; if more than one result byte meets the criteria, the address of the rightmost such result byte is inserted. • In the 24-bit addressing mode, the address replaces bits 40-63 of general register 1, and bits 0-39 of the register are not changed. In the 31-bit addressing mode, the address replaces bits 33-63 of general register 1, bit 32 of the register is set to zero, and bits 0-31 of the register remain unchanged. In the 64-bit addressing mode, the address replaces bits 0-63 of general register 1. <p>The return value is the condition code.</p>	<pre>EDMK Op1D(len,Op1B), Op2D(Op2B)</pre>	ARCH(0)

Table 73. Decimal instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>void __mp (unsigned char *op1, unsigned char len1, unsigned char *op2, unsigned char len2);</pre> <p>Operands:</p> <ul style="list-style-type: none"> • op1 represents the first operand (multiplicand) in the hardware instruction. It points to the first operand location. The result replaces the first operand. The multiplicand must have at least as many bytes of leftmost zeros as the number of bytes in the multiplier. • len1 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the first operand). The value must be greater than len2 and not greater than 15. • op2 represents the second operand (multiplier) in the hardware instruction. It points to the second operand location. • len2 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the second operand). The value is between 0 and 7 and must be less than the value of len1. <p>Note: When either len1 or len2 is not specified as a literal, an EX instruction is generated to execute a target MP instruction with length encoded in the register used by the EX instruction.</p>	<pre>MP Op1D(len1, Op1B), Op2D(len2, Op2B)</pre>	ARCH(0)
<pre>int __sp (unsigned char *op1, unsigned char len1, unsigned char *op2, unsigned char len2);</pre> <p>Operands:</p> <ul style="list-style-type: none"> • op1 represents the first operand in the hardware instruction. It points to the first operand location. The result replaces the first operand. • len1 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the first operand). The value is between 0 and 15. • op2 represents the second operand in the hardware instruction. It points to the source location. • len2 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the second operand). The value is between 0 and 15. <p>Note: When either len1 or len2 is not specified as a literal, an EX instruction is generated to execute a target SP instruction with length encoded in the register used by the EX instruction.</p> <p>The return value is the condition code set by the SP instruction.</p>	<pre>SP Op1D(len1, Op1B), Op2D(len2, Op2B)</pre>	ARCH(0)

Table 73. Decimal instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __srp (unsigned char *op1, unsigned char len1, signed char op2, unsigned char op3);</pre> <p>Operands:</p> <ul style="list-style-type: none"> • op1 represents the first operand in the hardware instruction. It points to the source location. The result replaces the first operand. • len1 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the first operand). The value is between 0 and 15. • op2 represents the second operand in the hardware instruction. It is a shift value between -32 and 31. Positive shift values specify shifting to the left. Negative shift values specify shifting to the right. • op3 represents the third operand in the hardware instruction. It is used as a decimal rounding digit. <p>Note: When either len1 or op3 is not specified as a literal, an EX instruction is generated to execute a target SRP instruction with len1 or op3 encoded in the register used by the EX instruction.</p> <p>The return value is the condition code set by the SRP instruction.</p>	<pre>SRP Op1D(len1, Op1B), Op2D(Op2B), Op3</pre>	ARCH(0)
<pre>int __tp(char *op1, unsigned char op1_len);</pre> <p>Operands:</p> <ul style="list-style-type: none"> • op1 points to a byte string to be tested for a valid packed-decimal value. • op1_len specifies the length encoded in the machine instruction (than is, the number of additional bytes to the left of the first operand used in the machine instruction). The value is between 0 and 15. <p>Note: When op1_len is not specified as a literal, an EX instruction is generated to execute a target TP instruction with op1_len encoded in the register used by the EX instruction.</p> <p>The return value is the condition code set by the TP instruction.</p>	<pre>TP Op1D(Op1_len, Op1B)</pre>	ARCH(6)

Table 73. Decimal instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __zap (unsigned char *op1, unsigned char len1, unsigned char *op2, unsigned char len2);</pre> <p>Operands:</p> <ul style="list-style-type: none"> • op1 represents the first operand in the hardware instruction. It points to the location to receive the result. • len1 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the sign byte of the first operand). The value is between 0 and 15. <p>Note: When len1 is not specified as a literal, an EX instruction is generated to execute a target ZAP instruction with len1 encoded in the register used by the EX instruction.</p> <ul style="list-style-type: none"> • op2 represents the second operand in the hardware instruction. It points to the start location. • len2 specifies the length encoded in the machine instruction (that is, the number of additional bytes to the left of the sign byte of the second operand). The value is between 0 and 15. <p>Note: When len2 is not specified as a literal, an EX instruction is generated to execute a target ZAP instruction with len2 encoded in the register used by the EX instruction.</p> <p>The return value is the condition code set by the ZAP instruction.</p>	<pre>ZAP Op1D(len1, Op1B), Op2D(len2, Op2B)</pre>	ARCH(0)

Floating-point support instructions

These functions are intended to help convert between the two floating point formats. For more information on these instructions, see chapter 9 of *z/Architecture Principles of Operation*.

If you want to use any of the following instructions, your program must include the `builtins.h` header file and be compiled with `LANGLVL(EXTENDED)` or `LANGLVL(LIBEXT)`.

Table 74. Floating-point instruction prototypes		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __tbdr(double *Op1, int M3, double Op2)</pre> <p>The return value is the condition code.</p>	<pre>TBDR F1,M3,Op2 LDR *Op1,F1</pre>	ARCH(3)
<pre>int __tbedr(double *Op1, int M3, float Op2)</pre> <p>The return value is the condition code.</p>	<pre>TBEDR F1,M3,Op2 LDR *Op1,F1</pre>	ARCH(3)
<pre>int __thder(double *Op1, float Op2)</pre> <p>The return value is the condition code.</p>	<pre>THDER F1,Op2 LDR *Op1,F1</pre>	ARCH(3)

Table 74. Floating-point instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __thdr(double *Op1, double Op2)</pre> <p>The return value is the condition code.</p>	<pre>THDR F1,Op2 LDR *Op1,F1</pre>	ARCH(3)

Decimal floating-point built-in functions

Decimal floating-point built-in functions are provided for each DFP hardware instruction. XL C/C++ developers can use the decimal floating-point built-in functions and macros, or named constants, by calling the functions with appropriate parameters. It is not necessary to include a header file before using decimal floating-point built-in functions. They will be automatically defined by the compiler when DFP is enabled.

All decimal floating-point built-in functions require a hardware level of at least ARCH(7).

Single precision support is limited, as noted in [Table 75 on page 395](#).

Table 75. Decimal floating-point instruction prototypes for IEEE operations	
PROTOTYPE and Notes	Description
<pre>_Decimal32 __d32_abs (_Decimal32); _Decimal64 __d64_abs (_Decimal64); _Decimal128 __d128_abs (_Decimal128);</pre> <p>Note: Also see functions that return the negative absolute value and functions that return the absolute value of the first parameter with the sign of the second parameter.</p>	These functions return the absolute value of the parameter.
<pre>_Decimal32 __d32_copysign (_Decimal32 exponent_and_fraction, _Decimal32 sign); _Decimal64 __d64_copysign (_Decimal64 exponent_and_fraction, _Decimal64 sign); _Decimal128 __d128_copysign (_Decimal128 exponent_and_fraction, _Decimal128 sign);</pre>	These functions return the absolute value of the first parameter, with the sign of the second parameter.
<pre>_Decimal32 __d32_sNaN (void); _Decimal64 __d64_sNaN (void); _Decimal128 __d128_sNaN (void); _Decimal32 __d32_qNaN (void); _Decimal64 __d64_qNaN (void); _Decimal128 __d128_qNaN (void);</pre>	These functions create quiet or signaling NaNs of the specified precision, with positive signs and zero payloads.
<pre>_Decimal64 __d64_integral (_Decimal64); _Decimal128 __d128_integral (_Decimal128); _Decimal64 __d64_integral_no_inexact (_Decimal64); _Decimal128 __d128_integral_no_inexact (_Decimal128);</pre> <p>The functions <code>__d64_integral</code> and <code>__d128_integral</code> allow an inexact exception. The instruction M4 bit 21 is set to 0.</p> <p>The functions <code>__d64_integral_no_inexact</code> and <code>__d128_integral_no_inexact</code> suppress any inexact exception. The instruction M4 bit 21 is set to 1.</p> <p>All these functions set the instruction M4 bit 20 to 0. If the input is a signaling NaN it is converted to a quiet NaN.</p>	These functions round a decimal floating point value to an integer value in decimal floating point format; any digits after the decimal point are discarded. The current rounding mode is used.

Table 75. Decimal floating-point instruction prototypes for IEEE operations (continued)	
PROTOTYPE and Notes	Description
<pre> _Decimal64 __d64_quantize (_Decimal64, _Decimal64, long round_mode); _Decimal128 __d128_quantize (_Decimal128, _Decimal128, long round_mode); </pre> <p>The <i>round_mode</i> parameter must be a compile-time constant expression. Use either of the following:</p> <ul style="list-style-type: none"> • DFP_ROUND_USING_CURRENT_MODE (8) to use the current rounding mode. • One of the values that can be set by <code>__dfp_set_rounding_mode</code> to temporarily override the current rounding mode. See “Definitions that support FPC register-rounding macros” on page 403. 	These functions return the arithmetic value of the first parameter, with the exponent adjusted to match the second parameter. They can temporarily override the current rounding mode and use the specified rounding mode.
<pre> bool __d64_same_quantum (_Decimal64, _Decimal64); bool __d128_same_quantum (_Decimal128, _Decimal128); </pre>	These functions compare the exponents of two parameters. If the exponents are the same, the functions return "true".
<pre> long __d64_compare_signaling (_Decimal64, _Decimal64); long __d128_compare_signaling (_Decimal128, _Decimal128); </pre> <p>These functions normally return <0, ==0 or >0 to indicate the relation. If either value is a NaN, they return "-2" (unordered).</p>	These functions compare two decimal floating-point values. Unlike a comparison using standard equality or relational operators, they also raise an Invalid Operation exception when either operand is either a quiet Nan or a signaling NaN.
<pre> long long __d64_to_long_long (_Decimal64); long long __d128_to_long_long (_Decimal128); long long __d64_to_long_long_rounding (_Decimal64, long round_mode); long long __d128_to_long_long_rounding (_Decimal128, long round_mode); </pre> <p>The functions <code>__d64_to_long_long ()</code> and <code>__d128_to_long_long ()</code> use the current decimal rounding mode, while a cast always rounds towards zero.</p> <p>The functions <code>__d64_to_long_long_rounding ()</code> and <code>__d128_to_long_long_rounding ()</code> can temporarily override the current rounding mode and use the specified rounding mode.</p> <p>The <i>round_mode</i> parameter must be a compile-time constant expression. Use either of the following:</p> <ul style="list-style-type: none"> • DFP_ROUND_USING_CURRENT_MODE (8) to use the current rounding mode. • One of the values that can be set by <code>__dfp_set_rounding_mode</code> to temporarily override the current rounding mode. See “Definitions that support FPC register-rounding macros” on page 403. 	These functions convert a decimal floating point value to a 64-bit signed binary integer with rounding mode options.
<pre> unsigned long __dfp_get_rounding_mode (void); </pre> <p>Note: See “Definitions that support FPC register-rounding macros” on page 403.</p>	This function gets the current decimal rounding mode from the z/Architecture FPC register.

Table 75. Decimal floating-point instruction prototypes for IEEE operations (continued)	
PROTOTYPE and Notes	Description
<pre>void __dfp_set_rounding_mode (unsigned long round_mode);</pre> <p>If the rounding mode is changed within a function, it must be restored before the function returns.</p> <p>Note: See “Definitions that support FPC register-rounding macros” on page 403.</p>	This function sets the specified decimal rounding mode in the z/Architecture FPC register, making it the current mode.

Table 76. Decimal floating-point instruction prototypes for IEEE . . . is operations	
PROTOTYPE and Notes	Description
<pre>bool __d32_isfinite (_Decimal32); bool __d64_isfinite (_Decimal64); bool __d128_isfinite (_Decimal128);</pre>	These functions return "true" if the parameter is not positive or negative infinity, and is not a NaN.
<pre>bool __d32_isinf (_Decimal32); bool __d64_isinf (_Decimal64); bool __d128_isinf (_Decimal128);</pre>	These functions return "true" if the parameter is positive or negative infinity.
<pre>bool __d32_isnormal (_Decimal32); bool __d64_isnormal (_Decimal64); bool __d128_isnormal (_Decimal128);</pre>	These functions return "true" if the parameter is in the normal range, not a subnormal, infinity or NaN.
<pre>bool __d32_issignaling (_Decimal32); bool __d64_issignaling (_Decimal64); bool __d128_issignaling (_Decimal128);</pre>	These functions return "true" if the parameter is positive or negative signaling NaN.
<pre>bool __d32_issigned (_Decimal32); bool __d64_issigned (_Decimal64); bool __d128_issigned (_Decimal128);</pre>	These instructions return "true" if the parameter is negative, including negative zero, negative infinity and negative NaN.
<pre>bool __d32_issubnormal (_Decimal32); bool __d64_issubnormal (_Decimal64); bool __d128_issubnormal (_Decimal128);</pre>	These functions return "true" if the parameter is a subnormal.
<pre>bool __d32_iszero (_Decimal32); bool __d64_iszero (_Decimal64); bool __d128_iszero (_Decimal128);</pre>	These functions return "true" if the parameter is positive or negative zero.

Table 77. Decimal floating-point instruction prototypes for IBM Instructions	
PROTOTYPE and Notes	Description
<pre>_Decimal32 __d32_nabs (_Decimal32); _Decimal64 __d64_nabs (_Decimal64); _Decimal128 __d128_nabs (_Decimal128);</pre> <p>Note: Also see functions that return the absolute value and functions that return the absolute value of the first parameter with the sign of the second parameter.</p>	These functions return the negative of the absolute value of the parameter.

Table 77. Decimal floating-point instruction prototypes for IBM Instructions (continued)

PROTOTYPE and Notes	Description
<pre>long long __d32_to_gpr (_Decimal32); long long __d64_to_gpr (_Decimal64); void __d128_to_gprs (_Decimal128, long long *upper, long long *lower);</pre> <p>Note: Also see functions that transfer a value from GPRs.</p>	These functions transfer a value from an FPR or FPR pair to a GPR, GPR pair, or four GPRs.
<pre>_Decimal32 __gpr_to_d32 (long long); _Decimal64 __gpr_to_d64 (long long); _Decimal128 __gprs_to_d128 (long long upper, long long lower);</pre> <p>Note: Also see functions that transfer a value to GPRs.</p>	These functions transfer a value from a GPR, GPR pair, or four GPRs to an FPR or FPR pair.
<pre>_Decimal32 __d64_round_to_d32 (_Decimal64, unsigned long round_mode, bool suppress_invalid); _Decimal64 __d128_round_to_d64 (_Decimal128, unsigned long round_mode, bool suppress_invalid);</pre> <p>The <i>round_mode</i> parameter must be a compile-time constant expression. Use either of the following:</p> <ul style="list-style-type: none"> • DFP_ROUND_USING_CURRENT_MODE (8) to use the current rounding mode. • One of the values that can be set by <code>__dfp_set_rounding_mode</code> to temporarily override the current rounding mode. See “Definitions that support FPC register-rounding macros” on page 403. <p>If the input value is a signaling NaN, and:</p> <ul style="list-style-type: none"> • If <code>suppress_invalid</code> is false, the result will be a quiet NaN and an invalid operation exception will be raised. • If <code>suppress_invalid</code> is true, the result will be a signaling NaN and no exception will be raised. 	These functions convert a value to a narrower format, with rounding control and invalid exception control that is unavailable when using a cast.
<pre>long long __d64_to_signed_BCD (_Decimal64, bool CF); void __d128_to_signed_BCD (_Decimal128, bool CorF, unsigned long long *upper, unsigned long long *lower);</pre> <p><code>__d64_to_signed_BCD</code> produces 15 decimal digits followed by a decimal sign in a 64-bit result.</p> <p><code>__d128_to_signed_BCD</code> produces 31 decimal digits followed by a decimal sign in a 128-bit result.</p> <p>Negative values will be given the sign 0xD.</p> <p>If <code>CorF</code> is false, positive values will be given the sign 0xC.</p> <p>If <code>CorF</code> is true, positive values will be given the sign 0xF.</p>	These functions convert the lower digits of the parameter to signed packed format.

Table 77. Decimal floating-point instruction prototypes for IBM Instructions (continued)

PROTOTYPE and Notes	Description
<pre> _Decimal64 __signed_BCD_to_d64 (signed long long); _Decimal128 __signed_BCD_to_d128 (signed long long upper, signed long long lower); </pre> <p>The signs 0xA, 0xC, 0xE, and 0xF will be treated as positive, and 0xB and 0xD as negative.</p> <p><code>__signed_BCD_to_d64</code> converts 15 decimal digits followed by a decimal sign in a 64-bit input.</p> <p><code>__signed_BCD_to_d128</code> converts 31 decimal digits followed by a decimal sign in a 128-bit input.</p>	<p>These functions convert signed packed decimal to decimal floating point.</p>
<pre> _Decimal64 __unsigned_BCD_to_d64 (unsigned long long); _Decimal128 __unsigned_BCD_to_d128 (unsigned long long upper, unsigned long long lower); </pre> <p><code>__unsigned_BCD_to_d64</code> converts 16 decimal digits with no sign in a 64-bit input.</p> <p><code>__unsigned_BCD_to_d128</code> converts 32 decimal digits with no sign in a 128-bit input.</p>	<p>These functions convert the lower digits of the parameter to unsigned packed format.</p>
<pre> _Decimal64 __d64_reround (_Decimal64, unsigned long number_of_digits, unsigned long round_mode); _Decimal128 __d128_reround (_Decimal128, unsigned long number_of_digits, unsigned long round_mode); </pre> <p>The <code>round_mode</code> parameter must be a compile-time constant expression. Use either of the following:</p> <ul style="list-style-type: none"> • <code>DFP_ROUND_USING_CURRENT_MODE</code> (8) to use the current rounding mode. • One of the values that can be set by <code>__dfp_set_rounding_mode</code> to temporarily override the current rounding mode. See “Definitions that support FPC register-rounding macros” on page 403. 	<p>These functions round a value to fewer digits. They can temporarily override the current rounding mode. For correct rounding, the input value must have been calculated using</p> <p><code>ROUND_TO_PREPARE_FOR_SHORTER_PRECISION</code></p> <p>For details, see “Definitions that support FPC register-rounding macros” on page 403.</p>
<pre> _Decimal64 __d64_insert_biased_exponent (_Decimal64, long biased_exponent); _Decimal128 __d128_insert_biased_exponent (_Decimal128, long biased_exponent); </pre> <p>Notes:</p> <ol style="list-style-type: none"> 1. Also see the functions that return the exponent of a specified parameter. 2. For the type definitions of infinity, quiet NaN, or signaling NaN, see “Biased exponent definitions” on page 404. 	<p>These functions return the digits and sign of the first parameter with the biased exponent of the second parameter, with special values for infinity, quiet NaN, or signaling NaN.</p>

Table 77. Decimal floating-point instruction prototypes for IBM Instructions (continued)

PROTOTYPE and Notes	Description
<pre> _Decimal64 __d64_shift_left (_Decimal64, unsigned long number_of_digits); _Decimal128 __d128_shift_left (_Decimal128, unsigned long number_of_digits); </pre> <p>Note: Also see functions that return the parameter with the coefficient shifted to the right.</p>	<p>These functions return the parameter with the coefficient shifted to the left. The sign and exponent are unchanged. The shift count must be in the 0-to-63 range; otherwise the result is undefined.</p>
<pre> _Decimal64 __d64_shift_right (_Decimal64, unsigned long number_of_digits); _Decimal128 __d128_shift_right (_Decimal128, unsigned long number_of_digits); </pre> <p>Note: Also see functions that return the parameter with the coefficient shifted to the left.</p>	<p>These functions return the parameter with the coefficient shifted to the right. The sign and exponent are unchanged. The shift count must be in the 0-to-63 range; otherwise the result is undefined.</p>
<pre> long __d32_test_data_class (_Decimal32, unsigned long data_class_mask); long __d64_test_data_class (_Decimal64, unsigned long data_class_mask); long __d128_test_data_class (_Decimal128, unsigned long data_class_mask); </pre> <p>These functions:</p> <ul style="list-style-type: none"> • Determine the exponent type (zero, subnormal, normal, infinity, quiet NaN or signaling NaN) and sign of the value. • Return a long integer that indicates whether the exponent matches the mask specifications. If there is a match, the function returns "1"; if there is no match, the function returns "0". <p>Note: The mask must be a constant expression at compile time. See “Test Data Class masks” on page 404.</p> <p>Also see test data group functions.</p>	<p>These functions determine whether a parameter is in a defined data class or a set of data classes, by testing its exponent and sign.</p>

Table 77. Decimal floating-point instruction prototypes for IBM Instructions (continued)

PROTOTYPE and Notes	Description
<pre>long __d32_test_data_group (_Decimal32, unsigned long data_group_mask); long __d64_test_data_group (_Decimal64, unsigned long data_group_mask); long __d128_test_data_group (_Decimal128, unsigned long data_group_mask);</pre> <p>These functions:</p> <ul style="list-style-type: none"> Determine the exponent type (safe zero, subnormal, normal with no leading zero, or an infinity or NaN), as well as the sign and first digit of the parameter. <p>Notes:</p> <ol style="list-style-type: none"> A "safe zero" has leading zero digits and a non-extreme exponent. A "subnormal" can appear as either an extreme non-zero or a safe non-zero. <ul style="list-style-type: none"> Return a long integer that indicates whether the exponent matches the mask specifications. If there is a match, the function returns "1"; if there is no match, the function returns "0". <p>Note: The mask must be a constant expression at compile time. For the statements that define these masks, see “Test Data Group masks” on page 405.</p> <p>Also see the test data class functions.</p>	<p>These functions determine whether a parameter is in a defined data group or set of data groups, by testing its exponent, sign and first digit.</p>
<pre>long __d64_biased_exponent (_Decimal64); long __d128_biased_exponent (_Decimal128);</pre> <p>Notes:</p> <ol style="list-style-type: none"> Also see functions that return the digits and sign of the first parameter with the biased exponent of the second parameter. See “Biased exponent definitions” on page 404. 	<p>These functions return the exponent of the parameter as an integer.</p>
<pre>unsigned long long __d64_to_unsigned_BCD (_Decimal64); void __d128_to_unsigned_BCD (_Decimal128, bool CorF, unsigned long long *upper, unsigned long long *lower);</pre> <p>Positive values will be given the sign 0xC if CorF is false or 0xF if it is true. Negative values will be given the sign 0xD.</p> <p><code>__d64_to_unsigned_BCD</code> produces 16 decimal digits with no sign in a 64-bit result.</p> <p><code>__d128_to_unsigned_BCD</code> produces 32 decimal digits with no sign in a 128-bit result.</p> <p>Note: Any digits to the left of those are ignored. To access the ignored digits, use the appropriate __d#_shift_right function.</p>	<p>These functions convert the lower digits of the parameter to unsigned packed format.</p>

Table 77. Decimal floating-point instruction prototypes for IBM Instructions (continued)	
PROTOTYPE and Notes	Description
<pre>long __d64_compare_exponents (_Decimal64, _Decimal64); long __d128_compare_exponents (_Decimal128, _Decimal128);</pre> <p>If both exponents are finite, these return "<0", "==0" or ">0" to indicate the relation between the exponents.</p> <p>If both exponents are infinite, they return "0".</p> <p>If one exponent is infinite and the other is finite, they return "-2" (unordered).</p>	<p>These functions compare exponents to one another.</p>
<pre>long __d64_extract_significance (_Decimal64); long __d128_extract_significance (_Decimal128);</pre>	<p>These functions normally return the number of significant digits in the input value.</p> <p>Exceptions:</p> <ul style="list-style-type: none"> • When the input is a zero, the return value is "0". • When the input is an infinity, the return value is "-1". • When the input is a quiet NaN, the return value is "-2". • When the input is a signaling NaN, the return value is "-3".
<pre>void __SFASR (unsigned long);</pre> <p>Note: See “Definitions that support FPC register-rounding macros” on page 403.</p>	<p>This function modifies the Floating Point Control (FCP) register, and could raise an exception.</p>

IBM zEnterprise® EC12 provides hardware instructions for conversions between decimal floating-point and zoned types. When both the DFP and ARCH(10) compiler options are in effect, the following hardware built-in functions are available:

Table 78. Instruction prototypes for conversions between decimal floating-point and zoned types	
PROTOTYPE and Notes	Description
<pre> _Decimal64 __cdzt(void* source, unsigned char length, const unsigned char mask); _Decimal128 __cxzt(void* source, unsigned char length, const unsigned char mask); </pre> <p>The source points to the memory location that contains data in valid zoned format.</p> <p>The length specifies the length of the source field encoded in the machine instruction. The range of the length value can be "0-15" for __cdzt and "0-33" for __cxzt. When the length value is not a literal, an EX instruction is generated to execute a target CDZT or CXZT instruction.</p> <p>The mask value provides the M3 value encoded in the machine instruction. The mask value must be provided as a literal.</p> <p>The return value is the converted decimal floating-point value.</p>	<p>These functions convert zoned type to decimal floating-point type.</p>
<pre> int __czdt(_Decimal64 source, void* result, unsigned char length, const unsigned char mask); int __czxt(_Decimal128 source, void* result, unsigned char length, const unsigned char mask); </pre> <p>The source contains the decimal floating-point value to be converted.</p> <p>The result points to the memory location that receives the converted data in zoned format.</p> <p>The length specifies the number of rightmost digits of the decimal floating-point value to be converted. The length value specifies the length of the result field encoded in the machine instruction in bytes. The range of the length value can be "0-15" for __czdt and "0-33" for __czxt. When the length value is not a literal, an EX instruction is generated to execute a target CZDT or CZXT instruction.</p> <p>The mask value provides the M3 value encoded in the machine instruction. The mask value must be provided as a literal.</p> <p>The return value is the condition code set by the instruction.</p>	<p>These functions convert decimal floating-point type to zoned type.</p>

Macros for use with decimal floating-point built-in functions

You do not need to include a header file before you can use these macros.

Definitions that support FPC register-rounding macros

Hardware built-in functions described in [Table 75 on page 395](#) use the specified decimal rounding mode in the z/Architecture FPC register, making it the current mode. [Figure 113 on page 404](#) lists the statements that define those decimal rounding modes.

```

#define DFP_ROUND_TO_NEAREST_WITH_TIES_TO_EVEN      0
#define DFP_ROUND_TOWARD_ZERO                       1
#define DFP_ROUND_TOWARD_POSITIVE_INFINITY          2
#define DFP_ROUND_TOWARD_NEGATIVE_INFINITY          3
#define DFP_ROUND_TO_NEAREST_WITH_TIES_AWAY_FROM_ZERO 4
#define DFP_ROUND_TO_NEAREST_WITH_TIES_TOWARD_ZERO  5
#define DFP_ROUND_AWAY_FROM_ZERO                   6
#define DFP_ROUND_TO_PREPARE_FOR_SHORTER_PRECISION 7
#define DFP_ROUND_USING_CURRENT_MODE                8

```

Figure 113. z/Architecture FPC register-rounding mode definitions

Note: DFP_ROUND_USING_CURRENT_MODE applies the rounding mode currently set in the z/Architecture FPC register. It is for use in functions that can temporarily override the specified rounding mode.

Biased exponent definitions

Hardware built-in functions that return biased exponents use special values to identify or specify the exponent type. An exponent type could be integer, infinity, quiet NaN, or signaling NaN. These functions are described in [Table 77 on page 397](#).

[Figure 114 on page 404](#) lists the statements that define those special values.

```

#define DFP_BIASED_EXPONENT_FINITE      0      /* integer type exponent plus bias */
#define DFP_BIASED_EXPONENT_INFINITY   -1
#define DFP_BIASED_EXPONENT_QNAN      -2
#define DFP_BIASED_EXPONENT_SNAN      -3

```

Figure 114. Biased exponent type definitions

Test Data Class masks

Test Data Class functions, which test the exponent of a parameter, must be used with at least one Test Data Class mask. Test Data Class functions are described in [Table 77 on page 397](#).

The following supplied masks look for an exponent with a specific type and sign:

```

DFP_Z_DATA_CLASS_POSITIVE_ZERO
DFP_Z_DATA_CLASS_NEGATIVE_ZERO
DFP_Z_DATA_CLASS_POSITIVE_SUBNORMAL
DFP_Z_DATA_CLASS_NEGATIVE_SUBNORMAL
DFP_Z_DATA_CLASS_POSITIVE_NORMAL
DFP_Z_DATA_CLASS_NEGATIVE_NORMAL
DFP_Z_DATA_CLASS_POSITIVE_INFINITY
DFP_Z_DATA_CLASS_NEGATIVE_INFINITY
DFP_Z_DATA_CLASS_POSITIVE_QUIET_NAN
DFP_Z_DATA_CLASS_NEGATIVE_QUIET_NAN
DFP_Z_DATA_CLASS_POSITIVE_SIGNALING_NAN
DFP_Z_DATA_CLASS_NEGATIVE_SIGNALING_NAN

```

Figure 115. Test Data Class masks

Note: A subnormal is also known as *denorm*.

To get a Test Data Class function to perform a desired test more efficiently, OR several of supplied masks together. For example, to determine whether the exponent of a parameter is any positive value, OR the following masks together:

- DFP_Z_DATA_CLASS_POSITIVE_ZERO
- DFP_Z_DATA_CLASS_POSITIVE_SUBNORMAL
- DFP_Z_DATA_CLASS_POSITIVE_NORMAL
- DFP_Z_DATA_CLASS_POSITIVE_INFINITY
- DFP_Z_DATA_CLASS_POSITIVE_QUIET_NAN
- DFP_Z_DATA_CLASS_POSITIVE_SIGNALING_NAN

Test Data Group masks

Test Data Group functions, which test the exponent of a parameter, must be used with at least one Test Data Group mask. Test Data Group functions are described in [Table 77 on page 397](#).

The following supplied Test Data Group masks look for an exponent that matches a specific type, sign, and first digit:

```
DFP_Z_DATA_GROUP_POSITIVE_ZERO_WITH_NONEXTREME_EXPONENT
DFP_Z_DATA_GROUP_NEGATIVE_ZERO_WITH_NONEXTREME_EXPONENT
DFP_Z_DATA_GROUP_POSITIVE_ZERO_WITH_EXTREME_EXPONENT
DFP_Z_DATA_GROUP_NEGATIVE_ZERO_WITH_EXTREME_EXPONENT
DFP_Z_DATA_GROUP_POSITIVE_SUBNORMAL_OR_EXTREME_EXPONENT
DFP_Z_DATA_GROUP_NEGATIVE_SUBNORMAL_OR_EXTREME_EXPONENT
DFP_Z_DATA_GROUP_POSITIVE_NONEXTREME_EXP_LEFTMOST_ZERO
DFP_Z_DATA_GROUP_NEGATIVE_NONEXTREME_EXP_LEFTMOST_ZERO
DFP_Z_DATA_GROUP_POSITIVE_NONEXTREME_EXP_LEFTMOST_NONZERO
DFP_Z_DATA_GROUP_NEGATIVE_NONEXTREME_EXP_LEFTMOST_NONZERO
DFP_Z_DATA_GROUP_POSITIVE_INFINITY_OR_NAN
DFP_Z_DATA_GROUP_NEGATIVE_INFINITY_OR_NAN
```

Figure 116. Test Data Group masks

To get a Test Group Class function to perform a desired test more efficiently, OR several of supplied masks together. For example, to determine whether a parameter is an extreme exponent, OR the following masks together:

- DFP_Z_DATA_GROUP_POSITIVE_ZERO_WITH_EXTREME_EXPONENT
- DFP_Z_DATA_GROUP_NEGATIVE_ZERO_WITH_EXTREME_EXPONENT

Hexadecimal floating-point instructions

If the FLOAT(HEX) option is in effect, these functions are intended to generate hexadecimal floating-point instructions. For more information about the instructions themselves, see chapter 18 of *z/Architecture Principles of Operation*.

If you want to use any of the following functions, your program must include `builtins.h` and be compiled with either the LONGLVL(EXTENDED) option or the LONGLVL(LIBEXT) and FLOAT(HEX) options.

Note: Some of these instructions also require that the ARCH option is set to a minimum level.

Table 79. Hexadecimal floating-point instruction prototypes		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __cfer(int *Op1, int M3, float Op2)</pre> <p>The return value is the condition code.</p>	<pre>CFER R2,M3,Op2 LR *Op3,R2</pre>	ARCH(3)
<pre>int __cfdr(int *Op1, int M3, double Op2)</pre> <p>The return value is the condition code.</p>	<pre>CFDR R2,M3,Op2 LR *Op3,R2</pre>	ARCH(3)
<pre>int __cfxr(int *Op1, int M3, long double Op2)</pre> <p>The return value is the condition code.</p>	<pre>CFXR R2,M3,Op2 LR *Op3,R2</pre>	ARCH(3)
<pre>float __fier(float Op2)</pre> <p>The return value is the result.</p>	<pre>FIER F1,Op2</pre>	ARCH(3)

Table 79. Hexadecimal floating-point instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
double __fidr(double Op2) The return value is the result.	FIDR F1,Op2	ARCH(3)
long double __fixr(long double Op2) The return value is the result.	FIXR F1,Op2	ARCH(3)
int __lnxr(long double *Op1, long double Op2) The return value is the condition code.	LDR F1,*Op1 LNXR F1,Op2 LDR *Op1,F1	ARCH(3)
int __lndr(double *Op1, double Op2) The return value is the condition code.	LDR F1,*Op1 LNDR F1,Op2 LDR *Op1,F1	ARCH(0)
int __lpdr(double *Op1, double Op2) The return value is the condition code.	LDR F1,*Op1 LPDR F1,Op2 LDR *Op1,F1	ARCH(0)
int __lpxr(long double *Op1, long double Op2) The return value is the condition code.	LDR F1,*Op1 LPXR F1,Op2 LDR *Op1,F1	ARCH(3)
int __lner(float *Op1, float Op2) The return value is the condition code.	LER F1,*Op1 LNER F1,Op2 LER *Op1,F1	ARCH(0)
int __lper(float *Op1, float Op2) The return value is the condition code.	LER F1,*Op1 LPER F1,Op2 LER *Op1,F1	ARCH(0)
float __sqer(float Op2) The return value is the square root.	SQER F1,Op2	ARCH(0) or above
double __sqdr(double Op2) The return value is the square root.	SQDR F1,Op2	ARCH(0) or above
long double __sqxr(long double Op2) The return value is the square root.	SQXR F1,Op2	ARCH(3)

Binary floating-Point instructions

These functions are intended to generate binary floating-point instructions. These instructions will only be generated if the FLOAT(IEEE) option is in effect. For more information about the instructions themselves, see chapter 19 of *z/Architecture Principles of Operation*.

If you want to use any of the following functions, your program must include `builtins.h` and be compiled with either the LANGLVL(EXTENDED) option or the LANGLVL(LIBEXT) and FLOAT(IEEE) options.

Table 80. Binary floating-point instruction prototypes

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __cfdbr(int *Op1, int M3, double Op2)</pre> <p>The return value is the condition code.</p>	<pre>CFDDBR R2,M3,Op2 LR *Op3,R2</pre>	ARCH(3)
<pre>int __cfebr(int *Op1, int M3, float Op2)</pre> <p>The return value is the condition code.</p>	<pre>CFEBR R2,M3,Op2 LR *Op3,R2</pre>	ARCH(3)
<pre>int __cfxbr(int *Op1, int M3, long double Op2)</pre> <p>The return value is the condition code.</p>	<pre>CFXBR R2,M3,Op2 LR *Op3,R2</pre>	ARCH(3)
<pre>int __efpc(void)</pre> <p>The return value is the z/Architecture FPC.</p> <p>Note: Also see functions that get and set FPC rounding modes and “Definitions that support FPC register-rounding macros” on page 403.</p>	<pre>EFPC R1</pre>	ARCH(3)
<pre>float __fiebr(int M3, float Op2)</pre> <p>The return value is the result.</p>	<pre>FIEBR F1,M3,Op2</pre>	ARCH(3)
<pre>float __fmadds(float Op1, float Op2, float Op3)</pre> <p>The return value is the result.</p>	<pre>FMADDS Op1,Op3,Op2</pre>	ARCH(3)
<pre>double __fidbr(int M3, double Op2)</pre> <p>The return value is the result.</p>	<pre>FIDBR F1,M3,Op2</pre>	ARCH(3)
<pre>long double __fixbr(int M3, long double Op2)</pre> <p>The return value is the result.</p>	<pre>FIXBR F1,M3,Op2</pre>	ARCH(3)
<pre>int __didbr(double *rem, double *quotient, double Op3, double Op4 int M4)</pre> <p>The return value is the condition code.</p>	<pre>LDR F1,Op3 DIDBR F1,F3,Op4,M4 LDR *rem,F1 LDR *quotient,F3</pre>	ARCH(3)
<pre>int __lndbr(double *Op1, double Op2)</pre> <p>The return value is the condition code.</p>	<pre>LDR F1,*Op1 LNDBR F1,Op2 LDR *Op1,F1</pre>	ARCH(3)
<pre>int __lnxbr(long double *Op1, long double Op2)</pre> <p>The return value is the condition code.</p>	<pre>LDR F1,*Op1 LNxDBR F1,Op2 LDR *Op1,F1</pre>	ARCH(3)
<pre>int __lpdbr(double *Op1, double Op2)</pre> <p>The return value is the condition code.</p>	<pre>LDR F1,*Op1 LPDBR F1,Op2 LDR *Op1,F1</pre>	ARCH(3)

Table 80. Binary floating-point instruction prototypes (continued)

PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>int __lpxbr(long double *Op1, long double Op2)</pre> <p>The return value is the condition code.</p>	<pre>LDR F1,*Op1 LPXBR F1,Op2 LDR *Op1,F1</pre>	ARCH(3)
<pre>int __diebr(float *rem, float *quotient, float Op3, float Op4, int M4)</pre> <p>The return value is the condition code.</p>	<pre>LER F1,Op3 DIEBR F1,F3,Op4,M4 LER *rem,F1 LER *quotient,F3</pre>	ARCH(3)
<pre>int __lnebr(float *Op1, float Op2)</pre> <p>The return value is the condition code.</p>	<pre>LER F1,*Op1 LNEBR F1,Op2 LER *Op1,F1</pre>	ARCH(3)
<pre>int __lpebr(float *Op1, float Op2)</pre> <p>The return value is the condition code.</p>	<pre>LER F1,*Op1 LPEBR F1,Op2 LER *Op1,F1</pre>	ARCH(3)
<pre>double __madbr(double Op1, double Op2, double Op3)</pre> <p>The return value is the result.</p>	<pre>MADBR Op1,Op3,Op2</pre>	ARCH(3)
<pre>float __maebr(float Op1, float Op2, float Op3)</pre> <p>The return value is the result.</p>	<pre>MAEBR Op1,Op3,Op2</pre>	ARCH(3)
<pre>double __msdbr(double Op1, double Op2, double Op3)</pre> <p>The return value is the condition code.</p>	<pre>MSDBR Op1,Op3,Op2</pre>	ARCH(3)
<pre>float __msebr(float Op1, float Op2, float Op3)</pre> <p>The return value is the result.</p>	<pre>MSEBR Op1,Op3,Op2</pre>	ARCH(3)
<pre>void __sfpc(int Op1)</pre> <p>Only a constant literal can be passed to this built-in function. Note: See “Definitions that support FPC register-rounding macros” on page 403.</p>	<pre>SFPC Op1</pre>	ARCH(3)
<pre>float __sqebr(float Op2)</pre> <p>The return value is the square root.</p>	<pre>SQEBR F1,Op2</pre>	ARCH(3)
<pre>double __sqdbr(double Op2)</pre> <p>The return value is the square root.</p>	<pre>SQDBR F1,Op2</pre>	ARCH(3)

Table 80. Binary floating-point instruction prototypes (continued)		
PROTOTYPE and Notes	Sample Pseudo Assembly	MIN ARCH
<pre>long double __sqrbr(long double Op2)</pre> <p>The return value is the square root.</p>	SQXBR F1,Op2	ARCH(3)
<pre>void __srnm(int Op1)</pre>	SRNM Op1	ARCH(3)
<pre>int __tceb(float Op1, int Op2)</pre> <p>The return value is the condition code.</p>	TCEB Op1,Op2(0,0)	ARCH(3)
<pre>int __tcdb(double Op1, int Op2)</pre> <p>The return value is the condition code.</p>	TCDB Op1,Op2(0,0)	ARCH(3)
<pre>int __tcxb(long double Op1, int Op2)</pre> <p>The return value is the condition code.</p>	TCXB Op1,Op2(0,0)	ARCH(3)

Built-in functions for transaction execution

Transactional memory is a model for controlling concurrent memory accesses in the scope of parallel programming. In this model, you can designate a block of instructions or statements to be treated atomically. Such a block is called a transaction.

When a thread of execution executes a transaction, it is seen by other threads as an atomic block. That is, all of the memory operations within the transaction are seen to occur simultaneously. At the end of the transaction, a decision is made to commit or abort the transaction. If the transaction is committed, results are written and the execution moves on. If it is aborted, the execution branches back to the beginning of the transaction and the original register state is restored, except for a single bit to indicate that a failure has occurred. The program can decide to retry or skip the transaction.

For parallel programming, a transaction implementation can be tremendously more efficient than other implementation methods. The following built-in functions provide the ability to mark the beginning and end of transactions, and to diagnose the reasons for failure. All the functions and types here will be defined in `builtins.h`.

Table 81. Built-in functions for transactional memory	
PROTOTYPE and Notes	Description
<pre>long __TM_simple_begin(); long __TM_begin(void* const TM_buff);</pre> <p>The <code>__TM_simple_begin()</code> function leads to the cheap kind of transaction. Cheap transactions have higher performance, but if the transaction fails, the failure information available is only the two-bit cc returned by the tbegin hardware instruction.</p> <p>The <code>__TM_begin(const void* TM_buff)</code> function leads to the expensive kind of transaction. The address provided (TM_buff) must be valid. It points to a 256-byte transaction diagnostic block (TDB) containing various debug information. If the transaction fails, the TDB is populated with lots of information.</p> <p>Note: Extensive control is provided over what state is saved and restored. These built-ins save all gpr's, and no fpr's or ar's. Using float values or ar-mode is not supported, and the facility of filtering some interrupts is not provided.</p>	<p>These functions start a transaction, a cheap transaction or an expensive one, corresponding to only some debugging capability or full debugging capability.</p> <p>The return value is "0" or non-zero indicating success or failure of the transaction.</p>
<pre>long __TM_end();</pre>	<p>This function marks the end of a transaction.</p> <p>The return value is non-zero if the thread was not in transactional state before the instruction started, and "0" otherwise.</p>
<pre>void __TM_abort(); void __TM_named_abort(unsigned char const code);</pre>	<p>These functions abort a transaction with a failure code, or "0" if none is provided.</p> <p>Providing a code enables the named abort functionality. The parameter code is constrained to be $0 \leq \text{code} \leq 255$.</p>
<pre>void __TM_abort_assist(unsigned int op1);</pre>	<p>This function requests assistance from the processor in performing a transaction-abort assist function.</p> <p>"op1" specifies the number of times the transaction has aborted.</p>
<pre>long __TM_is_user_abort(void* const TM_buff); long __TM_is_named_user_abort(void* const TM_buff, unsigned char* code);</pre>	<p>These functions return "1" if the transaction failed due to a user abort instruction, otherwise "0". The value returned in the parameter code is the code that was passed to the tabort instruction, or "0" if none was passed.</p> <p>If the TDB specified by TM_buff is valid and the transaction abort code is ≥ 256, return "true" and set code to the transaction abort code minus 256, otherwise return "0".</p>

Table 81. Built-in functions for transactional memory (continued)

PROTOTYPE and Notes	Description
<code>long __TM_is_illegal(void* const TM_buff);</code>	<p>Returns "1" if the TDB specified by TM_buff is valid and the transaction aborted due to trying to do something illegal (an instruction not permitted in transactional mode or some other kind of illegal access.)</p> <p>Returns "1" if the TDB specified by TM_buff is valid and the transaction abort code is "11" or "4".</p>
<code>long __TM_is_footprint_exceeded(void* const TM_buff);</code>	<p>Returns "1" if the TDB specified by TM_buff is valid and the transaction aborted due to reaching the maximum number of cache lines, otherwise "0".</p> <p>Returns "1" if the TDB pointed at by TM_buff is valid and the transaction abort code is "7" or "8", otherwise returns "0".</p>
<code>long __TM_nesting_depth(void* const TM_buff);</code>	Returns the current nesting depth. If the thread is not in transactional mode, returns the depth at which the most recent transaction aborted, or "0" if it is completed successfully.
<code>long __TM_is_nested_too_deep(void* const TM_buff);</code>	<p>Returns "1" if the transaction aborted due to reaching the maximum nesting depth.</p> <p>Returns "1" if the TDB pointed at by TM_buff is valid and the transaction abort code is "13".</p>
<code>long __TM_is_conflict(void* const TM_buff);</code>	<p>Returns "1" if the transaction aborted due to a conflict.</p> <p>Returns "1" if the TDB pointed at by TM_buff is valid and the transaction abort code is "9" or "10", otherwise returns "0".</p>
<code>long __TM_is_failure_persistent(long const result);</code>	<p>Returns "1" if the transaction aborted due to a reason that is persistent.</p> <p>Returns "1" if result==3 (result should be the CC value returned by tbegin, which is stored in the int return value of tbegin), otherwise returns "0".</p>
<code>long __TM_failure_address(void* const TM_buff);</code>	<p>Returns the code address at which the most recent transaction aborted.</p> <p>For 64 bit: returns the fourth doubleword of the TDB.</p> <p>For 31 bit or 24 bit: returns the rightmost word of the TDB.</p>

Table 81. Built-in functions for transactional memory (continued)

PROTOTYPE and Notes	Description
<pre>long long __TM_failure_code(void* const TM_buff);</pre>	<p>Returns the raw failure code.</p> <p>Returns bytes 8-15 of the TDB.</p>
<pre>void __TM_non_transactional_store(void* const addr, long long const value);</pre>	<p>This function indicates that 8 bytes provided by value are stored at the address pointed at by addr. The store is non-transactional.</p>

Chapter 30. Using runtime check library

The runtime check library provides a library routine that identifies the target hardware model to the application during its execution. This runtime check library also provides built-in functions that are associated with individual architectures to check for safety of the **arch_section** directive during application execution.

If you have code sections that are customized for specific hardware, you can use these built-in functions to perform a fast and efficient safety check before entering the customized sections. For more information about the **arch_section** directive, see [#pragma arch_section](#) in *z/OS XL C/C++ Language Reference*.

Check the type of CPU

```
void __builtin_cpu_init (void);
```

This built-in function runs the CPU detection code to check the type of CPU. This function must be invoked along with the `__builtin_cpu_is` and `__builtin_cpu_supports` built-in functions to check CPU type and features. This function must be run first so that CPU type is available to `__builtin_cpu_is` and `__builtin_cpu_supports`.

Check features the CPU supports

```
int __builtin_cpu_supports(const char* feature);
```

This built-in function returns a positive integer if the runtime CPU supports the specified feature; otherwise, it returns 0. The following features are supported:

- "5" through "13"
- "dfp"
- "dfpzoned"
- "etf3"
- "htm"
- "interlocked"
- "loadstoreoncond"
- "loadstoreoncond2"
- "longdisplacemnt"
- "miscinsnfacility2"
- "popcount"
- "prefetch"
- "storeclockfast"
- "vector128"
- "vectorenhfacility1"
- "vectorpacked"

Check the CPU model

```
int __builtin_cpu_is(const char* cpumodel);
```

This built-in function returns a positive integer if the runtime CPU is of type `cpumodel`; otherwise, it returns 0. Supported CPU is one of the models that are associated with "5", "6", "7", "8", "9", "10", "11", "12", and "13", which correspond to the ARCH levels.

Example

The following example shows the usage of the runtime check routine and built-in functions:

```
int main() {
    builtin_cpu_init();
    if (__builtin_cpu_supports("dfp"))
        #pragma arch_section(7)
        // ....
    return SUCCESS;
}
```

Chapter 31. XL C++ 98 applications and C99

The z/OS XL C Compiler and z/OS Language Environment C runtime library are designed to support the *Programming languages - C (ISO/IEC 9899:1999)* standard and its amendments. This standard is commonly referred to as C99. The z/OS XL C++ compiler is also designed to support the latest ISO C++ 2003 (*Programming languages - C++ (ISO/IEC 14882:2003)*) standard and the latest technical corrigendum.

The compiler language additions for C99 do not apply to C++ applications because the C++ standard does not mandate support for C99.

Obtaining C99 behavior with XL C

You obtain compiler behavior defined by the C99 language standard when you do either of the following:

- Use the **c89** command with LANGLVL(STDC99) option.
- Use the **c99** command (supported by the **xlc** utility) with the default language level.

When you want to obtain C99 extended support, do either of the following:

- Use the **c89** utility with the LANGLVL(EXTC99) option.
- Use the **c99** command with the LANGLVL(EXTENDED) option.

Using C99 functions in XL C++ applications

IBM has made some C99 functions accessible to the XL C++ compiler. These C99 functions are enabled by the individual LANGLVL or KEYWORD suboptions.

You can also obtain C99 behavior with the XL C++ compiler by using the following KEYWORD and LANGLVL suboptions:

- KEYWORD(RESTRICT) or LANGLVL(EXTENDED), which enable the `restrict` qualifier for improved aliasing information.
- LANGLVL(UCS), which enables support for valid universal character name ranges.
- LANGLVL(C99__FUNC__) or LANGLVL(EXTENDED), which enable the `__func__` identifier for debugging assistance.

C++ applications can also access C99 runtime library functions by using feature test macros. See [“Feature test macros that control C99 interfaces in XL C++ applications”](#) on page 415.

Feature test macros that control C99 interfaces in XL C++ applications

The following C99 headers are not available to C++ applications:

- `<tgmath.h>`
- `<complex.h>` — If `<complex.h>` is included, the USL Complex Class Library version of this header file will be used.
- `<stdbool.h>`

To expose C99 interfaces, C++ applications can define the appropriate feature test macros before including the identified header:

`_ISOC99_SOURCE`

Used to control exposure to new C99 interfaces that do not require a C99-compliant compiler. The application programmer defines this feature test macro to inform the compile time library that new C99 interfaces are desired. This feature test macro must be defined prior to inclusion of the first header in order to expose the new C99 interfaces that do not require a C99-compliant compiler. This

feature test macro requires a minimum of the z/OS V1R2 C/C++ compiler and TARGET(zOSV1R5) in order to expose the new C99 interfaces.

__STDC_LIMIT_MACROS

Required by C++ applications wishing to expose limits of fixed-width integer types and limits of other integer types as documented in `<stdint.h>`. The `_ISOC99_SOURCE` feature test macro must be defined before this feature test macro.

__STDC_CONSTANT_MACROS

Required by C++ applications wishing to expose macros for integer constants as documented in `<stdint.h>`. The `_ISOC99_SOURCE` feature test macro must be defined before this feature test macro.

__STDC_FORMAT_MACROS

Required by C++ applications wishing to expose macros for format specifiers as documented in `<inttypes.h>`. The `_ISOC99_SOURCE` feature test macro must be defined before this feature test macro.

_TR1_C99

Used to control exposure to the C++ TR1 C99 name space as described in Chapter 8 of *ISO/IEC DTR 19768: Draft Technical Report on C++ Library Extensions*.

Using C99 functions in C++ applications when ambiguous definitions exist

The C++ standard namespace does not include any C99 functions. Therefore, when ambiguous definitions exist, C++ applications must access these functions through the global namespace. The syntax of the global namespace is `::function()`.

XL C++ applications that need C99 interfaces must use the required feature macros or, when ambiguous definitions exist, global namespace syntax (when ambiguous definitions exist).

Figure 117 on page 416 is an example of code that requires the global namespace syntax. In this example, `std::` is not allowed for C99 interfaces.

```
#include <cstdio>
namespace FRED {
    int snprintf(char *b, size_t x, const char *f, ...) { return(x); }
};
using namespace FRED;
main() {
    char buf[512];
    int rc;
    /*rc = snprintf(buf,32,"hello\n");      AMBIGUOUS */
    rc = ::snprintf(buf,32,"hello\n");
    rc = FRED::snprintf(buf,32,"hello\n");
    /*rc = std::snprintf(buf,32,"hello\n"); NOT ALLOWED */
}
```

Figure 117. Example of code that requires the global namespace syntax

Chapter 32. Writing applications for Single UNIX Specification, Version 3

As of z/OS V1R9, the most current UNIX standard, Single UNIX Specification, Version 3 (SUSv3), is defined by IEEE Std 1003.1-2001, updated in 2004 with the integration of the two corrigenda issued subsequent to the 2001 release. The standard, also known as The Open Group Base Specifications Issue 6, is aligned with the ISO/IEC 9899:1999 Programming Languages - C standard. The new UNIX standard draws from the POSIX.1 and POSIX.2 specifications and their amendments, as well as The Open Group Base Specifications Issue 5 and The Open Group Technical Standard, January 2000, Networking Services, Issue 5.2.

For complete details about the normative references and base documents that underlie Single UNIX Specification, Version 3, see the introduction to the Base Definitions volume of IEEE Std 1003.1-2001.

In z/OS V1R9, the z/OS C/C++ runtime library provides a high degree of support for the Single UNIX Specification, Version 3. With the exception of the list of known non-compliances, library users may expect most conforming SUSv3 applications to compile and run on the z/OS platform in conformance with the standard.

Announcing your intentions

When compiling with system headers, there is a standard announcement mechanism for applications to notify the compiler that they are written for SUSv3. Notification is accomplished through the use of feature test macros `_POSIX_C_SOURCE` and `_XOPEN_SOURCE`. Applications insert either of the following definitions before any system header `#include` directives:

- `#define _POSIX_C_SOURCE 200112L` - Exposes the base POSIX namespace of SUSv3.
- `#define _XOPEN_SOURCE 600` - Exposes the X/Open System Interface (XSI) extension of SUSv3.

Because XSI is a superset of the POSIX base, `_POSIX_C_SOURCE 200112L` is implicit in the definition of `_XOPEN_SOURCE 600` and does not need to be defined when `_XOPEN_SOURCE 600` is defined.

Several other implementation-specific feature test macros impact support of SUSv3. An application may request SUSv3 threads support by defining `_UNIX03_THREADS` or as part of XSI. Because the Threads Option is a required component of XSI, you do not need to define `_UNIX03_THREADS` when `_XOPEN_SOURCE 600` is defined. Another new macro, `_UNIX03_WITHDRAWN`, preserves symbols withdrawn from the UNIX standard, making them visible in the SUSv3 namespace.

Before z/OS V1R9, other feature test macros enabled subsets of SUSv3 functionality. The `_UNIX03_SOURCE` macro has been used to expose a number of SUSv3 functions and constants missing from the z/OS C/C++ runtime library, before the more general implementation of SUSv3 in z/OS V1R9. The macro is additive, in that for a given target release, the compiler exposes all symbols exposed by `_UNIX03_SOURCE` for the target and any prior release. Given that any symbol it exposes is part of XSI, the `_UNIX03_SOURCE` macro is subsumed by `_XOPEN_SOURCE 600`, and is redundant when the latter is defined.

Like `_UNIX03_SOURCE`, the definition of `_OPEN_THREADS 2` or `_OPEN_THREADS 3` enables a specific subset of SUSv3 threads interfaces. Similarly, these interfaces are also exposed by defining `_XOPEN_SOURCE 600` (or `_UNIX03_THREADS`). However, there are behavioral differences and some differences in the naming of constants. With `_OPEN_THREADS`, other than the new functions, which are SUSv3 compliant, the remainder of the thread-related functions behave according to the POSIX.4a, draft 6 specification, which differs from SUSv3.

Program developers have the option of writing SUSv3 applications that still use the old threads behavior. An application may override the implicit XSI threads behavior by defining both `_OPEN_THREADS` and `_XOPEN_SOURCE 600`, if there is a reason to maintain the previous POSIX.4a, draft 6 behavior. On the

other hand, concurrent definition of the `_UNIX03_THREADS` and `_OPEN_THREADS` macros is not allowed and will generate a compile-time error message.

The intent of the z/OS C/C++ runtime library SUSv3 implementation is to maintain a great deal of flexibility and choice, supporting as many customer environments as possible. As in the case of `_OPEN_THREADS`, you can use most feature test macros that provide implementation-specific extensions with the SUSv3 macros, with the understanding that the resulting namespace does not conform to SUSv3. For more specific details about requirements or restrictions, see the individual feature test macros in *Feature test macros in z/OS C/C++ Runtime Library Reference*.

In addition to the feature test macros controlling that symbols are exposed in the system headers, two new environment variables determine behaviors at run time. In both cases, these variables impact error handling, enabling paths to provide `errno` information and fail a function for errors that were not previously detected. Default behavior for affected functions is unchanged. The new behavior must be explicitly enabled through a new setting of the environment variable.

- **`_EDC_SUSV3`** - Default is unset. For parts of SUSV3 behavior, set to 1. For additional pole error related SUSV3 behavior, set to 2.
- **`_EDC_EOVERFLOW`** - Default is NO. For new behavior, set to YES.

When `_EDC_SUSV3=1`, SUSV3 error handling occurs in `setenv()`, `readdir()`, `getnameinfo()`, and `tcgetsid()`. When `_EDC_SUSV3=2`, in addition to behaviors introduced by `_EDC_SUSV3=1`, error handling compliant with SUSV3 occurs in `log()`, `logf()`, `logl()`, `log10()`, `log10f()`, `log10l()`, `log1p()`, `log1pf()`, `log1pl()`, `log2()`, `log2f()`, `log2l()`, `pow()`, and `powl()`. With `_EDC_EOVERFLOW=YES`, overflow detection takes place in `ftell()`, `fseek()`, `fstat()`, `lstat()`, `stat()`, and `mmap()`. The `Eoverflow` error checking is for 31-bit applications only. For more specific information about behavior affected by these environment variables, see the individual function descriptions in *z/OS C/C++ Runtime Library Reference*.

Testing the environment

Many symbols have been added in `<limits.h>` and `<unistd.h>` to support SUSv3. Some of these symbols are tested at compile time by the preprocessor to allow differential compiles based on the values found, while others are intended for use by `sysconf()`, `pathconf()`, `fpathconf()`, and `confstr()` to test the version of the standard and availability of options at run time. Table 82 on page 418 lists the SUSv3 options and option groups supported by the z/OS C/C++ runtime library:

Table 82. SUSv3 options and option groups

Description	Symbol Name
File Synchronization	<code>_POSIX_FSYNC</code>
Memory Mapped Files	<code>_POSIX_MAPPED_FILES</code>
Memory Protection	<code>_POSIX_MEMORY_PROTECTION</code>
Realtime Signals Extension	<code>_POSIX_REALTIME_SIGNALS</code>
Thread Stack Address Attribute	<code>_POSIX_THREAD_ATTR_STACKADDR</code>
Thread Stack Size Attribute	<code>_POSIX_THREAD_ATTR_STACKSIZE</code>
Thread Process-Shared Synchronization	<code>_POSIX_THREAD_PROCESS_SHARED</code>
Thread-Safe Functions	<code>_POSIX_THREAD_SAFE_FUNCTIONS</code>
Threads	<code>_POSIX_THREADS</code>
Encryption Option Group	<code>_XOPEN_CRYPT</code>
Legacy Option Group	<code>_XOPEN_LEGACY</code>
XSI Streams Option Group	<code>_XOPEN_STREAMS</code>
XSI Extension	<code>_XOPEN_UNIX</code>

What is different in SUSv3

The SUSv3 implementation adds a number of new functions to the z/OS C/C++ runtime library, while applying modifications to the signatures of some existing functions in SUSv3 as result of the following behaviors:

- Addition or removal of arguments.
- Use of const declarator.
- Specialization of argument types.
- Use of restrict keyword.

The SUSv3 namespace excludes all withdrawn headers, functions, external variables, and constants. The specification further targets additional symbols for removal in a future version of the standard.

As noted earlier, Single UNIX Specification, Version 3 aligns with ISO/IEC 9899:1999, is commonly referred to as the C99 language standard. In some cases, SUSv3 extends the C99 definition, although in the case of a conflict, it always defers to the C99 standard. For this reason, applications compiled for SUSv3 are also implicitly C99, and you do not need to define feature test macro `_ISOC99_SOURCE` when `_XOPEN_SOURCE 600` or `_POSIX_C_SOURCE 200112L` is defined.

Beyond these obvious differences in the library, there are other behavioral differences. One of the more notable differences is the change in the return value of most threads functions. The POSIX.4a, draft 6 threads behavior indicates a return of -1 on failure with the error code set in `errno`. In SUSv3, the majority of these functions now return the error code on failure rather than a value of -1. With the exception of `pthread_getspecific()`, the z/OS implementation will continue to set `errno` in addition to returning the error code.

For complete details about syntactical differences and special behavior of functions, see the individual function descriptions in [z/OS C/C++ Runtime Library Reference](#).

Symbols withdrawn in SUSv3

Functions, headers, and external variables that comprised the Legacy Feature Group in Single UNIX Specification, Version 2 are removed and not part of SUSv3. Also, any symbols that were marked obsolescent in Version 2 have been removed.

For a complete list of the withdrawn symbols, see the description of `_UNIX03_WITHDRAWN` in [Feature test macros](#) in [z/OS C/C++ Runtime Library Reference](#). In the case of headers that have been withdrawn, there is no special mechanism employed. An application simply includes them, like any other non-standard header. The withdrawn headers are `<re_comp.h>`, `<regex.h>`, and `<varargs.h>`. Dependencies on symbols that are not part of the SUSv3 namespace affect portability and, if this is an issue, make such applications nonconforming.

Candidates for removal in a future version

In addition to symbols withdrawn from the SUSv3 namespace, a number of symbols have been marked obsolete in the POSIX base or added to the Legacy Option for XSI that may be removed from the Single UNIX Specification in a future version.

Obsolescent: `bsd_signal()`, `gethostbyaddr()`, `gethostbyname()`, `h_errno`, `pthread_attr_getstackaddr()`, `pthread_attr_setstackaddr()`, `scalb()`, `ualarm()`, `usleep()`, `vfork()`

Legacy Option `bcmp()`, `bcopy()`, `bzero()`, `ecvt()`, `fcvt()`, `ftime()`, `gcvt()`, `getwd()`, `index()`, `mktemp()`, `rindex()`, `utimes()`, `wcswcs()`

Implementation compliance

IBM makes no claim that the z/OS platform complies with Single UNIX Specification, Version 3 or that the z/OS C/C++ runtime library contains a complete implementation of the C programming interfaces and headers found in the Base Definitions of this standard. While the library implementation comes close, a

number of known discrepancies remain in z/OS R9. The following summary lists those discrepancies from SUSv3 behavior that have been identified.

The z/OS C/C++ runtime library does not support:

- `accept()` – `ECONNABORTED` from `accept()` when a connection has been aborted.
- `alarm()` and `setitimer()` deliver `SIGALRM` to the thread that invoked the service rather than at the process level.
- `exec()` family of functions in a multi-threaded environment.
- `fgetc()`, `fgetwc()`, `read()` – `E_OVERFLOW` in `fgetc()`, `fgetwc()`, or `read()` when reading at or beyond a file offset maximum.
- `getsockopt()`, `setsockopt()` – Options `SO_DONTROUTE`, `SO_RCVLOWAT`, `SO_RCVTIMEO`, `SO_SNDLOWAT`, `SO_SNDTIMEO` in `getsockopt()` or `setsockopt()`.

`EDOM` in `setsockopt()` when send/receive timeout values are too big to fit into the timeout fields in the socket structure.

`EISCONN` in `setsockopt()` when socket is already connected and a specified option cannot be set while the socket is connected.

- `fpathconf()` and `pathconf()` do not support the symbols shown in [Table 83 on page 420](#).

Table 83. Symbols not supported

_PC_FILESIZEBITS	(<code>FILESIZEBITS</code>)
_PC_2_SYMLINKS	(<code>POSIX2_SYMLINKS</code>)
_PC_ALLOC_SIZE_MIN	(<code>POSIX_ALLOC_SIZE_MIN</code>)
_PC_REC_INCR_XFER_SIZE	(<code>POSIX_REC_INCR_XFER_SIZE</code>)
_PC_REC_MAX_XFER_SIZE	(<code>POSIX_REC_MAX_XFER_SIZE</code>)
_PC_REC_MIN_XFER_SIZE	(<code>POSIX_REC_MIN_XFER_SIZE</code>)
_PC_REC_XFER_ALIGN	(<code>POSIX_REC_XFER_ALIGN</code>)
_PC_SYMLINK_MAX	(<code>SYMLINK_MAX</code>)
_PC_ASYNC_IO	(<code>_POSIX_ASYNC_IO</code>)
_PC_PRIO_IO	(<code>_POSIX_PRIO_IO</code>)
_PC_SYNC_IO	(<code>_POSIX_SYNC_IO</code>)

- `mmap()` – `ENXIO` in `mmap()` when `MAP_FIXED` is specified in flags and the combination of `addr`, `len`, and `off` is not valid for the `fd` object.
- `msync()` – `EBUSY` during `msync()` when some or all addresses in the range starting at `addr` and continuing for `len` bytes are locked, and `MS_INVALIDATE` is specified.
- `recvmsg()` – `EMSGSIZE` in `recvmsg()` when `msg_iovlen` in the `msg_hdr` pointed to by `message` is less than or equal to 0, or is greater than `{IOV_MAX}`.
- `regexexec()` - Use of the `restrict` keyword is not supported.
- `socket()`, `socketpair()` – `SOCK_SEQPACKET` type in `socket()` and `socketpair()`.
- `stderr` open for reading.

Chapter 33. Saved compile-time options information

Following a successful compilation, the application executable file will always include compile-time options information in a compact form. This information supports determination of runtime problems. Saved compile-time options information includes:

- Fixed subset of compilation options for each source file compiled.
- Source file name for each source file compiled.
 - The source file name of the compilation unit for which the options are saved is the first entry in the source file and component version information block.
 - For names longer than 252 characters, only the last 252 bytes of the source file name are provided.
 - The name does not include path information for UNIX files and only the member name is provided for partitioned data sets.
 - A dummy name "IPA Link" is provided for programs optimized with IPA.
- Version information for each compiler component that is active during the compilation.
 - You can use the version information to determine the compile-time maintenance level. If the maintenance level is not the most current, there might be an update available that solves the problem.
 - This version information will be the same as the information produced using the PHASEID compiler option. For further information about the PHASEID compiler option, see [z/OS XL C/C++ User's Guide](#).
 - Although information about the debug writer is not included because the debug writer runs after the code generation, that information is available inside the .dbg file.

A compilation flag in the Program Prolog Area-2 (PPA2) indicates the presence of saved options information. If the service string is specified, the saved option string follows it. Otherwise the saved options information follows the time stamp string. For more information about PPA2, see [z/OS Language Environment Vendor Interfaces](#).

Saved options information layout

The layout of the saved options information is show as below.

```
#define USHRT unsigned short
#define UINT unsigned int
typedef struct SOS_s {
    UINT  sos_words      : 8, /* 0xFF000000 sizeof(SOS)/4 */
    UINT  sos_version    : 8, /* 0x00FF0000 version number */
    /*
     * 1 - as of XL C/C++ 1.10
     * 2 - as of XL C/C++ 1.11
     * 3 - as of XL C/C++ 1.13
     * 4 - as of XL C/C++ 2.1
     * 5 - as of XL C/C++ 2.1 PTF
     * 6 - as of XL C/C++ 2.1.1
     * 7 - as of XL C/C++ 2.2
     * 8 - as of XL C/C++ 2.3
     */
    sos_arch      : 8, /* 0x0000FF00 ARCHITECTURE */
    sos_tune      : 8; /* 0x000000FF TUNE */
    /* offset: 4 */
    UINT  sos_csect      : 1, /* 0x80000000 0 - NOCSECT */
    /*
     * 1 - CSECT
     */
    sos_version_info :15, /* 0x7FFF0000 version list (PHASEIDs location -
    /*
     * offset/4 from SOS start)
     */
    sos_locale_ccsid :16; /* 0x0000FFFF CCSID from the LOCALE */
    /* offset: 8 */
    USHRT sos_lit_ccsid; /* 0xFFFF0000 CONVLIT(xx) */
    USHRT sos_wlit_ccsid; /* 0x0000FFFF CONVLIT(,WCHAR|UNICODE) */
}
```

```

/* offset: 12 */
UINT  sos_target_rel;          /* 0xFFFFFFFF TARGET(release value in HEX) */
/*                               As given by __TARGET_LIB__ */

/* offset: 16 */
UINT  sos_initauto_val;       /* 0xFFFFFFFF INITAUTO value */

/* offset: 24 */
UINT  sos_ansialias           : 1, /* 0x80000000 ANSIALIAS */
      sos_argparse            : 1, /* 0x40000000 ARGPARSE */
      sos_compress            : 1, /* 0x20000000 COMPRESS */
      sos_compact              : 1, /* 0x10000000 COMPACT */
      sos_execops              : 1, /* 0x08000000 EXECOPS */
      sos_goff                 : 1, /* 0x04000000 GOFF */
      sos_hot                  : 1, /* 0x02000000 HOT */
      sos_ignerrno             : 1, /* 0x01000000 IGNERRNO */
      sos_libansi              : 1, /* 0x00800000 LIBANSI */
      sos_upconv               : 1, /* 0x00400000 UPCONV */
      sos_longname             : 1, /* 0x00200000 LONGNAME */
      sos_lp64                 : 1, /* 0x00100000 LP64 */
      sos_rent                 : 1, /* 0x00080000 0 - NORENT */
                                /*                               1 - RENT */
      sos_wsizeof              : 1, /* 0x00040000 WSIZEOF */
      sos_roconst              : 1, /* 0x00020000 ROCONST */
      sos_rostring             : 1, /* 0x00010000 ROSTRING */
      sos_start                : 1, /* 0x00008000 START */
      sos_strict               : 1, /* 0x00004000 STRICT */
      sos_strictind            : 1, /* 0x00002000 STRICT_INDUCATION */
      sos_xpl_bkchn            : 1, /* 0x00001000 XPLINK(BACKCHAIN) */
      sos_xpl_callbk           : 1, /* 0x00000800 XPLINK(CALLBACK) */
      sos_xpl_grd              : 1, /* 0x00000400 XPLINK(GUARD) */
      sos_xpl_oscalls          : 2, /* 0x00000300 0 - XPLINK(OSCALL(NOSTACK)) */
                                /*                               1 - XPLINK(OSCALL(DOWNSTACK)) */
                                /*                               2 - XPLINK(OSCALL(UPSTACK)) */
      sos_hook_line            : 1, /* 0x00000080 DEBUG(HOOK(LINE)) */
      sos_hook_block           : 1, /* 0x00000040 DEBUG(HOOK(BLOCK)) */
      sos_hook_path            : 1, /* 0x00000020 DEBUG(HOOK(PATH)) */
      sos_hook_func            : 1, /* 0x00000010 DEBUG(HOOK(FUNC)) */
      sos_hook_call            : 1, /* 0x00000008 DEBUG(HOOK(CALL)) */
      sos_debug_sym            : 1, /* 0x00000004 DEBUG(SYMBOL) */
      sos_debug_fmt            : 2, /* 0x00000003 0 - NODEBUG */
                                /*                               1 - DEBUG(FORMAT(ISD)) */
                                /*                               2 - DEBUG(FORMAT(DWARF)) */

/* offset: 28 */
UINT  sos_gonumber            : 1, /* 0x80000000 GONUMBER */
      sos_target               : 2, /* 0x60000000 0 - TARGET(LE) */
                                /*                               1 - TARGET(IMS) */
      sos_plist                 : 1, /* 0x10000000 0 - PLIST(HOST) */
                                /*                               1 - PLIST(OS) */
      sos_optlevel              : 4, /* 0x0F000000 OPTIMIZE */
      sos_redir                 : 1, /* 0x00800000 REDIR */
      sos_cvft                  : 1, /* 0x00400000 CVFT */
      sos_objmodel              : 2, /* 0x00300000 0 - OBJECTMODEL(CLASSIC) */
                                /*                               1 - OBJECTMODEL(IBM) */
      sos_exh                   : 1, /* 0x00080000 EXH */
      sos_rtti                  : 1, /* 0x00040000 RTTI */
      sos_namemangling          : 6, /* 0x0003F000 0 - NAMEMANGLING(zOSV1R2) and */
                                /*                               NAMEMANGLING(ZOSV1R5_DEFAULT) */
                                /*                               1 - NAMEMANGLING(ANSI) */
                                /*                               2 - NAMEMANGLING(COMPAT) and */
                                /*                               NAMEMANGLING(OSV2R10) */
                                /*                               3 - NAMEMANGLING(ZOSV1R5_ANSI) */
                                /*                               4 - NAMEMANGLING(ZOSV1R7_ANSI) */
                                /*                               5 - NAMEMANGLING(ZOSV1R8_ANSI) */
                                /*                               6 - NAMEMANGLING(ZOSV1R9_ANSI) */
                                /*                               7 - NAMEMANGLING(ZOSV1R10_ANSI) */
                                /*                               8 - NAMEMANGLING(ZOSV1R11_ANSI) */
                                /*                               9 - NAMEMANGLING(ZOSV1R12_ANSI) */
                                /*                               10 - NAMEMANGLING(ZOSV2R1_ANSI) */
                                /*                               11 - NAMEMANGLING(ZOSV2R1M1_ANSI) */
      sos_ansisinit             : 1, /* 0x00000800 LANGLVL(ANSISINIT) */
      sos_newexcp               : 1, /* 0x00000400 LANGLVL(NEWEXCP) */
      sos_oldmath               : 1, /* 0x00000200 LANGLVL(OLDMATH) */
      sos_oldstr                : 1, /* 0x00000100 LANGLVL(OLDSTR) */
      sos_oldtmplalign          : 1, /* 0x00000080 LANGLVL(OLDTMPLALIGN) */
      sos_restrict              : 1, /* 0x00000040 ASSERT(RESTRICT) */
      sos_prefetch              : 1, /* 0x00000020 PREFETCH */

```

```

        sos_rtc          : 1, /* 0x00000010 Applicable to zOSV1R11 and later */
        /*              /* RTCHECK */
        sos_rtc_bounds   : 1, /* 0x00000008 Applicable to zOSV1R11 and later */
        /*              /* RTCHECK(BOUNDS) */
        sos_rtc_divzero   : 1, /* 0x00000004 Applicable to zOSV1R11 and later */
        /*              /* RTCHECK(DIVZERO) */
        sos_rtc_nullptr   : 1, /* 0x00000002 Applicable to zOSV1R11 and later */
        /*              /* RTCHECK(NULLPTR) */
        sos_restrict_param : 1; /* 0x00000001 Applicable to zOSV1R11 and later */
        /*              /* RESTRICT */
        /*              /* Applicable to zOSV1R12 and later */

/* offset: 32 */
    UINT  sos_rvaluerefs   : 1 /* 0x80000000 LANGLVL(RVALUEREFERENCES) */
    ,      sos_refcollapsing : 1 /* 0x40000000 LANGLVL(REFERENCECOLLAPSING) */
    ,      sos_rightanglebkt : 1 /* 0x20000000 LANGLVL(RIGHTANGLEBRACKET) */
    ,      sos_scopedenum    : 1 /* 0x10000000 LANGLVL(SCOPEENUM) */
    ,      sos_debug_level   : 4 /* 0x0F000000 DEBUG(LEVEL(0-9)) */
    ,      sos_subscript_wrap : 1 /* 0x00800000 STRICT(SUBSTRICTWRAP) */
    ,      sos_tempsaslocals : 1 /* 0x00400000 LANGLVL(TEMPSASLOCALS) */
    ,      sos_smp           : 1 /* 0x00200000 SMP */
    /*      /* Applicable to zOSV2R1 and later */
    /*      /* SMP(OPT) */
    /*      /* Applicable to zOSV2R1 and later */
    /*      /* SMP(EXPLICIT) */
    /*      /* Applicable to zOSV2R1 and later */
    /*      /* THREADED */
    /*      /* Applicable to zOSV2R1 and later */
    /*      /* FUNCEVENT */
    /*      /* Applicable to zOSV2R1M1 and later */
    /*      /* VECTOR */
    /*      /* Applicable to zOSV2R1M1 and later */
    /*      /* ASM */
    /*      /* Applicable to zOSV2R1M1 and later */
    /*      /* UNROLL(N) where 0 <= N <= 255 */
    /*      /* Applicable to zOSV2R1M1 and later */
    /*      /* LANGLVL(CHECKPLACEMENTNEW) */
    /*      /* Applicable to zOSV2R1M1 and later */
    /*      /* CHECKNEW */
    /*      /* Applicable to zOSV2R2 and later */
    /*      /* VECTOR(TYPE) */
    /*      /* Applicable to zOSV2R2 and later */
    /*      /* VECTOR(AUTOSIMD) */
    /*      /* Applicable to zOSV2R2 and later */
    /*      /* STACKPROTECT */
    /*      /* Applicable to zOSV2R3 and later */
    /*      /*
    ,      sos_padding      : 2
    ;

} SOS_t;

```

Part 5. Performance optimization

This part describes guidelines for improving the performance of your XL C/C++ application. Performance improvement can be achieved through coding, compiling, and the runtime environment. The following chapters discuss guidelines for these three areas:

- [Chapter 34, “Improving program performance,” on page 427](#)
- [Chapter 35, “Using built-in functions to improve performance,” on page 443](#)
- [Chapter 36, “I/O Performance considerations,” on page 447](#)
- [Chapter 37, “Improving performance with compiler options,” on page 451](#)
- [Chapter 39, “Optimizing the system and Language Environment,” on page 481](#)
- [Chapter 40, “Balancing compilation time and application performance,” on page 485](#)
- [Chapter 41, “Stepping through optimized code using the dbx debugger utility,” on page 489](#)

Chapter 34. Improving program performance

This information discusses coding guidelines that improve the performance of a C or C++ application. While they are most effective when creating new code, these guidelines can also provide a gradual performance improvement when they are consistently used when porting or fixing areas of the code. The guidelines cover the following topics:

- [“Writing code for performance” on page 427](#)
- [“Using C++ constructs in performance-critical code” on page 427](#)
- [“Using explicit instantiation declarations \(C++11 only\)” on page 429](#)
- [“ANSI aliasing rules” on page 429](#)
- [“Using ANSI aliasing rules” on page 431](#)
- [“Using variables” on page 432](#)
- [“Passing function arguments” on page 433](#)
- [“Coding expressions” on page 434](#)
- [“Coding conversions” on page 434](#)
- [“Arithmetical considerations” on page 435](#)
- [“Using loops and control constructs” on page 435](#)
- [“Choosing a data type” on page 436](#)
- [“Using library extensions” on page 437](#)
- [“Using #pragmas” on page 438](#)
- [“Using rvalue references \(C++11\)” on page 439](#)

Writing code for performance

When you write code, it is a good practice to write it so that you can understand it when you simply read it on a printed page or on a screen, without having to refer to anything else. If the code is simple and concise, both the programmer and the compiler can understand it easily. Code that is easy for the compiler to understand is also easy for it to optimize. If you follow this practice you might not only create code that performs well on execution, you might also create code that compiles more quickly.

If you follow the guidelines in this information, you will create code that performs well on execution and can be compiled efficiently.

Using C++ constructs in performance-critical code

Note: The discussion in this information applies to high-level language constructs that might seriously degrade the performance of C++ programs. All other coding discussions in this information apply to both C and C++ programs.

Be aware that in C++, more than in C, certain coding constructs can lead to n-to-1, m-to-1 or even z-to-1 code expansion. You can create well-performing code with these constructs, but you must use them carefully and appropriately, especially when you are writing critical-path or high-frequency code.

When writing performance-critical C++ programs, ensure that you understand why problems might occur and what you can do about them if you use any of the following high-level language constructs:

Virtual

The `virtual` construct is an important part of object-oriented coding and can be very useful in removing the `if` and `switch` logic from an application. Programmers often use `virtual` and neglect to remove the `switch` logic. Note the following:

- The use of a `virtual` construct (like the use of a pointer and unlike the use of `if` statements) prevents the compiler from knowing how that construct is defined, which would provide the compiler with an optimization opportunity. In other words, when you use a `virtual` construct instead of `if` or `switch` statements, you limit optimization opportunities.
- In a non-XPLINK module, because of function overhead, `virtual` functions are costlier to execute than straight-line code with `if` or `switch` statements.

Exception handling

When exception handling is available (that is, when you are using the EXH compiler option), opportunities for both normal optimizations and for inlining are limited. This is because the compiler must generate extra code to keep track of execution events and to ensure that all required objects are caught by the correct routines.

When you use the C++ `try` and `catch` blocks, the compiler creates obstacles to optimization. The compiler cannot pull common code out of a `try` block because it might trigger an exception that would need to be caught. Similarly, code cannot be pulled out of a `catch` block because:

- The code in a `catch` block is triggered far down the call chain, after the exception has occurred
- After a `catch` has occurred, the compiler must ensure that all requested tasks have been executed

You might improve compiler performance by:

- Removing dependencies on C++ exception handling from your code
- Compiling with the NOEXH compiler option

Dynamic casts/Runtime type identification (RTTI)

A dynamic cast (also known as RTTI) is a coding construct that delays, until run time, the determination of which code is to be executed. This limits the potential for optimization. In addition, the process of actually doing the dynamic cast involves multiple function calls and large amounts of code.

Note: We strongly recommend that RTTI/dynamic casts not be used in performance-critical code. You can often avoid the use of RTTI through careful application design.

iostream

As discussed in [Chapter 4, “Using the Standard C++ Library I/O Stream Classes,”](#) on page 21, `iostream` is often built upon the standard C I/O library (`fprintf`, `fopen`, `fclose`, `fread`, `fwrite`). For I/O performance-critical portions of your application, it is often faster to use the C I/O functions explicitly instead of `iostream`.

Note: You must be careful if you are mixing the C++ stream classes with the C library. For more information, see [Chapter 4, “Using the Standard C++ Library I/O Stream Classes,”](#) on page 21.

Standard Template Library and other class libraries

These libraries are very convenient and are often well coded, but you must remember that each use of a class can involve one or more function calls. If you keep this in mind when coding, you can design applications that use these libraries efficiently. For example, you would not initialize all local string variables to the NULL string and then redefine the string on first reference.

new/delete

New C++ applications on z/OS often depend heavily on `new` and `delete` operators because they are commonly one of the first things taught in a C++ introductory course, and many courses never explicitly teach that classes can also be automatic (default for local) or global variables.

You should be aware that the `new` and `delete` operators are costlier to use than variables. Also, before using `new`, you should carefully consider:

- The scope/usage pattern of the variable
- Whether an automatic (local) or global variable is more appropriate

Note: You can ensure that all memory and storage requests are properly optimized by following the instructions given in [Chapter 37, “Improving performance with compiler options,”](#) on page 451 and [“Optimizing memory and storage”](#) on page 481.

Using explicit instantiation declarations (C++11 only)

Use template explicit instantiation declarations to suppress implicit template instantiations. This helps reduce the collective size of the object files and shorten compiler time. This technique is described in [“Using explicit instantiation declarations \(C++11 only\)”](#) on page 324.

ANSI aliasing rules

You must indicate whether your source code conforms to the ANSI aliasing rules when you use the IPA or the OPT(2) (or above) z/OS XL C/C++ compiler options. If the code does not conform to the rules, it must be compiled with NOANSIALIAS. Incorrect use of these options might generate bad code.

Note: The compiler expects that the source code conforms to the ANSI aliasing rules when the ANSIALIAS option is used. This option is on by default.

The ANSI aliasing rules are part of the ISO C Standard, and state that a pointer can be dereferenced only to an object of the same type or compatible type. Because the z/OS XL C/C++ compiler follows these rules during optimization, the developer must create code that conforms to the rules.

Note: The common coding practice of casting a pointer to an incompatible type and then dereferencing it violates ANSI aliasing rules.

When you are using ANSI aliasing, you can cast an `int` pointer only to the types described in [Table 84](#) on page 429 then dereference it to access the object it points to. Corresponding casts apply to other types.

Table 84. Examples of acceptable alias types

Type	Reason for acceptance
<code>int</code>	This is the declared type of the object.
<code>const int</code> , or <code>volatile int</code> , or <code>restrict int</code> , or any combination of these qualifiers	These types are the qualified version of the declared type of the object.
<code>signed int</code> or <code>unsigned int</code>	This is a signed or unsigned type corresponding to the declared type of the object.
<code>const unsigned int</code> or <code>volatile unsigned int</code>	These types are the signed or unsigned types corresponding to a qualified version of the declared type of the object.
<pre>struct foo { unsigned int bar; };</pre>	This is an aggregate or union type that includes one of the aforementioned types among its members. This can include, recursively, a member of a subaggregator-contained union.
<code>char</code> , or <code>unsigned char</code> , or (for C only) <code>signed char</code>	The <code>char</code> pointers are an exception to the rules, as a <code>char</code> pointer can be used to point to and dereferenced to access a variable of any type. For example, the address passed to <code>memcpy</code> may be any pointer type.

Conversely, your code breaks the aliasing rules if it casts a `float` to an `int` and then assigns it to the `int` pointer and dereferences that.

In C/C++ 11, the typeless memory returned by `malloc` etc. receives the "effective type" of the first access to it. For example, if the address returned by `malloc` is cast to an `int*` pointer and that is dereferenced to store an initial value, then that memory's effective type becomes `int`, and only pointers compatible with `int` can be used to access it.

For more information, see type-based aliasing in [z/OS XL C/C++ Language Reference](#) and [ANSIALIAS | NOANSIALIAS](#) in [z/OS XL C/C++ User's Guide](#).

You can cast and mix data types as long as you are careful how you intermix values and their pointers in your code. The compiler follows the ANSI aliasing rules to determine:

- Which variables must be stored into memory before you read a value through a pointer

- Which variables must be updated from memory after you have updated a value through a pointer

When you use the NOANSIALIAS option, the compiler generates code to accommodate worst-case assumptions (for example, that any variable could have been updated by the store through a pointer). This means that every variable (local and global) must be stored in memory to ensure that any value can be read through a pointer. This severely limits the potential for optimization.

```
int ei1;
float ef1;
int *eip1;
float *efp1;

float exmp1 ()
{
    ef1 = 3.0;
    ei1=5;
    *efp1 = ef1;
    *eip1 = ei1;
    return *efp1;
}
```

Table 85 on page 430 shows the difference between code generated with, and without, ANSI aliasing.

Table 85. Comparison of code generated with the ANSIALIAS and NOANSIALIAS options

ANSIALIAS RENT and OPT(2)	NOANSIALIAS RENT and OPT(2)
<pre>* { * ef1 = 3.0; L r4,=A(@CONSTANT_AREA)(,r3,94) L r2,=Q(EF1)(,r3,98) LD f0,+CONSTANT_AREA(,r4,0) L r14,_CEECAA_(,r12,500) L r15,=Q(EFP1)(,r3,102) L r4,=Q(EIP1)(,r3,106) L r1,#retvalptr_1(,r1,0) STE f0,ef1(r2,r14,0) L r15,efp1(r15,r14,0)</pre>	<pre>* { * ef1 = 3.0; L r2,=A(@CONSTANT_AREA)(,r3,110) L r14,_CEECAA_(,r12,500) L r4,=Q(EF1)(,r3,114) L r15,=Q(EFP1)(,r3,118) LD f0,+CONSTANT_AREA(,r2,0)</pre>
<pre>* ei1=5; L r2,=Q(EI1)(,r3,110) LA r0, L r4,eip1(r4,r14,0)</pre>	<pre>* ei1=5; L r2,=Q(EI1)(,r3,122) STE f0,ef1(r4,r14,0)</pre>
<pre>* *efp1 = ef1; STE f0,(*)float(,r15,0) ST r0,ei1(r2,r14,0)</pre>	<pre>* *efp1 = ef1; L r4,efp1(r15,r14,0)</pre>
<pre>* *eip1 = ei1; ST r0,(*)int(,r4,0)</pre>	<pre>* *eip1 = ei1; L r5,=Q(EIP1)(,r3,126) LA r0,5 ST r0,ei1(r2,r14,0) STE f0,(*)float(,r4,0) L r4,eip1(r5,r14,0) L r0,ei1(r2,r14,0)</pre>
<pre>* return *efp1; STD f0,#retval_1(,r1,0) * }</pre>	<pre>* return *efp1; L r1,#retvalptr_1(,r1,0) ST r0,(*)int(,r4,0) L r14,efp1(r15,r14,0) SDR f0,f0 LE f0,(*)float(,r14,0) STD f0,#retval_1(,r1,0) * }</pre>

Table 85. Comparison of code generated with the ANSIALIAS and NOANSIALIAS options (continued)

ANSIALIAS RENT and OPT(2)	NOANSIALIAS RENT and OPT(2)
<ul style="list-style-type: none"> • In the ANSIALIAS case: <ul style="list-style-type: none"> – f0, loaded with 3.0, is used whenever referring to ef1 or efp1 – r0 is loaded with the value of 5, which is used for ei and eip • In the NOANSIALIAS case, the loads and stores are always done. This removes opportunities for optimizations. For example, if a + b + c were used instead of 3.0 and ef1, saving through the pointer might have updated a, b, or c, and therefore you cannot common at all, and many more reloads. • ANSIALIAS would not help if all the floats were also integers • There is a group of problems that occurs when the ANSIALIAS option is used to compile code that does not conform to ANSI-aliasing rules (for example, when it casts a variable to a non-ANSI-aliasing type and then assigns the address of the value to a pointer for later use). If the ANSIALIAS option is in effect (it is the default) when a value is used through a pointer, the compiler might not reload the pointer value when the original value is updated, and the value might be stale when it is read. 	

Using ANSI aliasing rules

Your programs are likely to perform better if you follow these guidelines:

- Use ANSI aliasing whenever possible.
- Declare constant variables with `const`. This is particularly helpful when using the C++ compiler because if something is qualified as `const`, the compiler will not be forced to perform unnecessary reloads to see if the value has changed. This can generate significantly faster code.

```
ggPoint3 operator*(const ggAffineMatrix3 &m
, const ggPoint3 &p)
{
    return ggPoint3(
        m.e[0][0] * p.x() + m.e[0][1] * p.y() + m.e[0][2] * p.z() + m.e[0][3],
        m.e[1][0] * p.x() + m.e[1][1] * p.y() + m.e[1][2] * p.z() + m.e[1][3],
        m.e[2][0] * p.x() + m.e[2][1] * p.y() + m.e[2][2] * p.z() + m.e[2][3]
    );
}
```

- Whenever their values cannot change, qualify pointers and their targets as constants, ensuring that you mark the appropriate part as `const`.
 - If only the pointer is constant, you can use a statement that is similar to the following:

```
int * const i = p /* a constant pointer to an integer that may vary */
```

- If only the target is constant, use a statement similar to either of the following:

```
int const * i = p /* a variable pointer to a constant integer */
const int * i = p /* a variable pointer to a constant integer */
```

- If both the target integer and the pointer are constants, use a statement similar to either of the following:

```
const int * const i = &p; /* a constant pointer to a constant integer */
int const * const i = &p; /* a constant pointer to a constant integer */
```

- Use the `ROCONST` compiler option. The `ROCONST` option works with both C and C++. This option causes the compiler to treat variables that are defined as `const` as if they are read-only. In some cases, these variables will be stored in read-only memory. For more information, see [“ROCONST” on page 467](#).
- *For global variables initialized to large read-only arrays or strings:* Use a `#pragma` variable to ensure that they are implemented as read-only csects. This prevents them from being initialized at load time.

Example: For large initialized arrays

```
# pragma variable (arrayname, noent)
```

- *In a read-only situation:* If you are using the value through a pointer, use a temporary automatic variable. The difference in the source code is significant, as shown in the following table:

ANSIALIAS RENT and OPT(2)	NOANSIALIAS RENT and OPT(2)
<pre> ... while (hot_loop < hot_loop_end) { hot_loop = hot_loop + foo->increment; fun[x] = hot_loop*foo->expansion; } </pre>	<pre> { ... increment = foo->increment; expansion = foo->expansion; while (hot_loop < hot_loop_end) { hot_loop = hot_loop + increment; fun[x] = hot_loop*expansion; } </pre>

Using variables

When choosing variables and data structures for your application, keep the following guidelines in mind:

- Use local variables, preferably automatic variables, as often as possible.

The compiler can accurately analyze the use of local variables, while it has to make several worst-case assumptions about global variables, which hinders optimizations. For example, if you code a function that uses external variables, and calls several external functions, the compiler assumes that every call to an external function could change the value of every external variable.

- If none of the function calls affect the global variables being used and you have to read them frequently with function calls interspersed, copy the global variables to local variables and use these local variables to help the compiler perform optimizations that otherwise would not be done.

Using IPA can improve the performance of code written using global variables, because it coalesces global variables. IPA puts global variables into one or more structures and accesses them using offsets from the beginning of the structures. For more information, see [“Using the IPA option” on page 458](#).

- If you need to share variables only between functions within the same compilation unit, use static variables instead of external variables. Because static variables are visible only in the current source file, they might not have to be reloaded if a call is made to a function in another source file.

Organize your source code so references to a given set of externally defined variables occur only in one source file, and then use static variables instead of external variables.

In a file with several related functions and static variables, the compiler can group the variables and functions together to improve locality of reference.

Use a local static variable instead of an external variable or a variable defined outside the scope of a function.

The **#pragma isolated_call** preprocessor directive can improve the runtime performance of optimized code by allowing the compiler to make fewer assumptions about the references to external and static variables. For more information, see [#pragma isolated_call](#) in [z/OS XL C/C++ Language Reference](#).

Coalescing global variables causes variables that are frequently used together to be mapped close together in memory. This strategy improves performance in the same way that changing external variables to static variables does.

- Group external data into structures (all elements of an external structure use the same base address) or arrays wherever it makes sense to do so.

Before it can access an external variable, the compiler has to make an extra memory access to obtain the variable’s address. The compiler removes extraneous address loads, but this means that the compiler has to use a register to keep the address.

Using many external variables simultaneously requires many registers, thereby causing spilling of registers to storage. If you group variables into structures then it can use a single variable to keep the

base address of the structure and use offsets to access individual items. This reduces register pressure and improves overall performance, especially in programs compiled with the RENT option.

The compiler treats register variables the same way it treats automatic variables that do not have their addresses taken.

- Minimize the use of pointers.

Use of pointers inhibits most memory optimizations such as dead store elimination in C and C++.

You can improve the runtime performance of optimized code by using the z/OS C **#pragma disjoint** directive to list identifiers that do not share the same physical storage. A similar mechanism that can be used to improve runtime performance of optimized code includes using the C99 restrict qualifier for pointers feature. The restrict type qualifier indicates for the lifetime of the pointer, and only it or a value directly derived from it will be used to access the object to which it points. For more information, see [#pragma disjoint](#) and [The restrict type qualifier in z/OS XL C/C++ Language Reference](#).

Passing function arguments

When writing code for optimization, it is usually better to pass a value as an argument to a function than to let the function take the value from a global variable. Global variables might have to be stored before a value is read from a pointer or before a function call is made. Global variables might have to be reloaded after function calls, or stored through a pointer. For more information, see [“Using ANSI aliasing rules” on page 431](#) and [“Using variables” on page 432](#).

The **#pragma isolated_call** preprocessor directive lists functions that do not modify global storage. You can use it to improve the runtime performance of optimized code. For more information, see [#pragma isolated_call in z/OS XL C/C++ Language Reference](#).

Linkage convention or how arguments are passed is not specified in the C language, but is defined by the platform. Compilers in general follow the calling convention as described by the Application Binary Interface (ABI). An ABI can define more than one linkage due to performance considerations; for example, the XPLINK and non-XPLINK linkages on the z/OS platform. To correctly invoke a function, the arguments passed must match the parameters as defined in the function definition. For example, if you pass a pointer argument to a function expecting an integer, the code generated by the compiler for the call and for the function definition may not match (see the note at the end of this topic).

As the following example shows, you can declare a function without providing information about the number and types of its parameters.

```
int func();
...
int a;
func(a);
...
int func(p)
    void *p;
{
    ...
}
```

Because the function declaration has no parameter information, the compiler is not required to diagnose parameter mismatch. You can call this function, passing it any number of arguments of any type, but the compilation will not be guaranteed to work if the function is not defined to receive the arguments as passed, due to differences in linkage conventions. In the worse case, when the z/OS XL C/C++ compiler attempts inlining of such ill-formed function calls, it may get into an unrecoverable condition and the compilation is halted.

To correct the situation, use the CHECKOUT(GEN) option to identify missing function declarations and non-prototype function declarators. Add or change the declarations to prototyped declarations, and proceed with compilation again. Should you receive diagnostic messages regarding incorrect function argument assignment, change the function call to pass the expected parameter type.

Note: Such a mismatch may sometimes turn out not to be an issue, depending on the ABI; for example, if the ABI happens to allow both pointers and integers passed using general purpose registers. Even

in this case, there is no guarantee that the optimized code would work as expected due to ambiguous information received by the compiler.

Coding expressions

When coding expressions, consider the following recommendations:

- When components of an expression are duplicate expressions, code them either at the left end of the expression or within parentheses, as shown in the following example.

```
a = b*(x*y*z);          /* Duplicates recognized */
c = x*y*z*d;
e = f + (x + y);
g = x + y + h;

a = b*x*y*z;           /* No duplicates recognized */
c = x*y*z*d;
e = f + x + y;
g = x + y + h;
```

The compiler can recognize $x*y*z$ and $x + y$ as duplicate expressions when they are coded in parentheses or coded at the left end of the expression.

It is the best practice to avoid using pointers as much as possible within high-usage or other performance-critical code.

Note: The compiler might not be able to optimize duplicate expressions if either of the following are true:

- The address of any of the variables is already taken
- Pointers are involved in the computation
- When components of an expression in a loop are constant, code the constant expressions either at the left end of the expression or within parentheses.

The following example shows the difference in evaluation when c , d , and e are constant and v , w , and x are variable.

```
v*w*x*(c*d*e);          /* Constant expressions recognized */
c + d + e + v + w + x;

v*w*x*c*d*e;           /* Constant expressions not recognized */
v + w + x + c + d + e;
```

Coding conversions

Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. Conversions require several instructions, including some double-precision floating-point arithmetic. When you must use mixed-mode arithmetic, code the integral, floating-point, and decimal arithmetic in separate computations wherever possible. [Figure 118 on page 435](#) shows an example.

```

/* this example shows how numeric conversions are done */
int main(void)
{
    int i;
    float array[10]={1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0}
    float x = 1.0;
    for (i = 0; i < 10; i++)
    {
        array[i] = array[i]*x; /* No conversions needed */
        x = x + 1.0;
    }

    for (i = 1; i <= 9; i++)
        array[i] = array[i]*i; /* Conversions may be needed */

    return(0);
}

```

Figure 118. Numeric conversions example

Arithmetical considerations

Wherever possible, use multiplication rather than division. For example,

```
x*(1.0/3.0); /* 1.0/3.0 is evaluated at compile time */
```

produces faster code than:

```
x/3.0;
```

If you divide many values by the same number in your code: Assign the divisor's reciprocal to a temporary variable and then multiply by that variable.

Using loops and control constructs

For the for-loop index variable:

- Use a long type variable whenever possible. Under ILP32, long and int are equivalent, but long is better for portability to an LP64 environment.
- Use the auto or register storage class over the extern or static storage class.
- If you use an enum variable, expand the variable to be a fullword by using the ENUMSIZE compiler option or by placing a large defined value at the end of your enum variable, as follows:

```

enum animals {
    ant
    cat,
    dog,
    robin,
    last_animal = INT_MAX;
};

```

- Do not use the address operator (&) on the index.
- The index should not be a member of a union.

For if statements:

- Order the if conditions efficiently; put the most decisive tests first and the most expensive tests last.

By performing the most common tests first, you increase the efficiency of your code; fewer tests are required to meet the test conditions.

```
if (command.is_classg &&
    command.len == 6 &&
    !strcmp (command.str, "LOGON")) /* call to strcmp() most expensive */
    logon ();
```

Choosing a data type

Use the `int` data type instead of `char` when performing arithmetic operations.

```
char_var += '0';
int_var += '0';          /* better */
```

A `char` type variable is efficient when you are:

- Assigning a literal to a `char` variable
- Comparing the variable with a `char` literal

For example:

```
char_var = 27;
if (char_var == 'D')
```

Table 87 on page 436 lists analogous data types and shows which data types are more expensive to reference.

Table 87. Referencing data types	
More Expensive	Less Expensive
unsigned short	signed short (Although unsigned short is less expensive on many systems, the z/OS implementation of signed short is less expensive.)
signed char	unsigned char
long double	double
Longer decimal	Shorter decimal

For storage efficiency, the compiler packs enumeration variables in 1, 2 or 4 bytes, depending on the largest value of a constant. When performance is critical, expand the size to a fullword either by adding an enumeration constant with a large value or by specifying the `ENUMSIZE` compiler option. For example:

```
enum byte { land, sea, air, space };
enum word { low, medium, high, expand_to_fullword = INT_MAX };
```

Example that is equivalent to using the `ENUMSIZE(INT)` compiler option:

```
enum word { low, medium, high };
```

Fullword enumeration variables are preferred as function parameters.

For efficient use of extern variables:

- Place scalars ahead of arrays in extern `struct`.
- Copy heavily referenced scalars to auto or register variables (especially in a loop).

When using `float`:

- When passing variables of type `float` to a function, an implicit widening to `double` occurs (which takes time).

- On some machines divisions of type float are faster than those of type double.

When using bit fields, be aware that:

- Even though the compiler supports a bit field spanning more than 4 bytes, the cost of referencing it is higher.
- An unsigned bit field is preferred over a signed bit field.
- A bit field used to store integer values should have a length of 8, 16, or 24 bits and be on a byte boundary.

```

struct {      unsigned   xval   :8,
                      xbool   :1,
                      xmany   :6,
                      xset    :1;
} b;

if (b.xval == 3)
:
if (b.xmany + 5 == x)    /* inefficient because it does not */
                      /* fall on a byte boundary          */
:
if (b.xbool)
:

```

Using library extensions

Effective use of DLLs could improve the performance of your application if either of the following is true:

- The application relies on a `fetch()` or `system()` function to call programs in other modules.
- The application is overly large and there are some low-use or special-purpose routines that you can move to a DLL.

If you are using C, consider calling other C modules with `fetch()` or DLLs instead of `system()`. A `system()` call does full environment initialization and termination, but a fetched module and a DLL share the environment of the calling routine. If you are using C++, consider using DLLs.

Use of DLLs requires more overhead than use of statically-bound function calls. You can test your code to determine whether you can afford this extra overhead. First, write the code so that it can be built to implement either a single module or a DLL. Next build your application both ways, and time both applications to see if you can handle the difference in execution time. For best DLL performance, structure the code so that once a function in the DLL is called, it does all it needs to do in the DLL before it returns control to the caller.

You can also choose how to implement DLLs. If you are using C, you can choose between:

- The XPLINK compiler option
- The DLL compiler option (which is used with the NOXPLINK option)

Note: In C++, DLL is not an option, but a default. When you use the XPLINK option, the compiler loads and accesses DLLs faster than it would if you used the DLL option.

The following suggestions could improve the performance of the application:

- If you are using a particular DLL frequently across multiple address spaces, you can install the DLL in either the LPA/ELPA or the DLPA to avoid load overhead. When the DLL resides in a PDSE, the DLPA services should be used.
- When you are binding your code, specify both the RENT and the REUSE options. Otherwise, each load of a DLL results in a separately loaded DLL with its own writable static area.
- Group external variables into one external structure.
- When you are using z/OS UNIX, avoid unnecessary load attempts.

z/OS Language Environment supports loading a DLL that resides in the UNIX file system or in a data set. However, the location from which it first tries to load the DLL varies, depending whether your application runs with the runtime option `POSIX(ON)` or `POSIX(OFF)`.

- If your application runs with POSIX(ON), z/OS Language Environment tries to load the DLL from the UNIX file system first. If you are doing an explicit DLL load using the `dllload()` function, you can avoid searching the UNIX file system directories. You can direct a DLL search to a data set by prefixing the DLL name with two slashes (`//`), as follows:

```
//MYDLL
```

- If your application runs with POSIX(OFF), z/OS Language Environment tries to load your DLL from a data set. Similarly, if you are loading your DLL with the `dllload()` function and your DLL is loading in UNIX file system, you can avoid the search of the data set by directing a DLL search to the UNIX file system. You can do so by prefixing the DLL name with a period and slash (`./`), as follows:

```
./mydll
```

Note: DLL names are case sensitive in the UNIX file system.

- When you are using IPA, export only those subprograms (functions and C++ methods) or variables that you need for the interface to the final DLL.

If you export subprograms or variables unnecessarily (for example, by using the `EXPORTALL` option), you severely limit IPA optimization. In this case, global variable coalescing and pruning of unreachable or 100% inlined code does not occur. Before it can be processed by IPA, DLLs must contain at least one subprogram. Any attempt to process a data-only DLL will result in a compilation error.

- The suboption `NOCALLBACKANY` of the compiler option `DLL` is more efficient than the `CALLBACKANY` suboption.

The `CALLBACKANY` option calls a Language Environment routine at run time. This runtime service enables a C or C++ `NOXPLINK` DLL routine to call a C `NOXPLINK NODLL` routine, which use function pointers that point to actual function entry points rather than function descriptors.

Note: Compiling source with the `DLL` option will often cause a degradation in performance when compared against a statically bound application compiled without that option.

Using #pragmas

Table 88 on page 438 describes `#pragmas` that can affect performance. For information about using each pragma, see [z/OS XL C/C++ Language Reference](#).

Table 88. *Pragmas that affect performance*

Name	Description
<code>#pragma disjoint</code>	Lists identifiers that do not share the same physical storage, which provides more opportunities for optimizations.
<code>#pragma execution_frequency</code>	Marks program source code that you expect will be either very frequently or very infrequently executed.
<code>#pragma export</code>	Selectively exports functions or variables from a DLL module. The <code>EXPORTALL</code> compiler option exports <i>all</i> functions or variables, which often results in larger modules and significantly increased WSA requirements.
<code>#pragma inline (C only)</code>	Together with the <code>INLINE</code> compiler option, ensures that frequently used functions are inlined. This directive is only supported in C; however, you can use the <code>inline</code> keyword in C++.
<code>#pragma isolated_call</code>	Lists functions that have no side effects (that do not modify global storage). This directive can improve the runtime performance of variables and storage by allowing the compiler to make fewer assumptions about whether external and static variables could be updated.
<code>#pragma leaves</code>	Specifies that a function never returns to the instruction following a call to that function. This directive provides information to the compiler that enables it to explore additional opportunities for optimization.

Table 88. Pragmas that affect performance (continued)

Name	Description
#pragma noline	This directive can improve pipeline usage and allow more of the used routines to be inlined.
#pragma option_override	<p>Allows you to specify optimization options on a per-routine basis rather than on only a per-compilation basis. It enables you to specify which functions you do not want to optimize while compiling the rest of the program optimized. This directive helps you to isolate which function is causing problems under optimization.</p> <p>The option_override pragma can be also used to change the spill size for a function. If the compiler requests that you to increase the spill size for a specific function, you should use the option_override pragma instead of the SPILL compiler option, which increases the spill size for all functions in the compile unit and can have a negative performance impact on the generated code.</p> <p>Note: The spill size should not be increased unless requested by a compiler message.</p>
#pragma reachable	Declares that the point in the program after the specified function can be the target of a branch from some unknown location. That is, you can reach the instruction after the specified function from a point in your program other than the return statement in the named function. This directive provides information to the compiler that enables it to explore additional opportunities for optimization.
#pragma strings	Indicates if strings should be placed in read-only memory or read/write memory. You can reduce the memory requirements for DLLs by specifying #pragma strings(readonly) , so that string literals are not placed in the writable static area. Alternatively, you can also use the ROSTRING compiler option (the default), which informs the compiler that string literals are read-only.
#pragma unroll	Informs the compiler how to perform loop unrolling on the loop body that immediately follows it. The directive works in conjunction with the UNROLL compiler option to provide you with some control over the application of this optimization technique. The pragma directive overrides the “UNROLL” on page 468 or NOUNROLL compiler option in effect for the designated loop.
#pragma variable	<p>Indicates if a named external object is used in reentrant or non-reentrant fashion. If an object is qualified as RENT, its references or its definition will be in the writable static area, which is in modifiable storage. If an object is qualified as NORENT, its references or its definition will be in the code area.</p> <p>You can reduce the memory requirements for DLLs by specifying #pragma variable(var_name, NORENT), so that constant variables are not placed in the writable static area.</p> <p>Alternatively, you can use the ROCONST compiler option to inform the compiler that constant variables are not to be placed in the writable static area.</p>

Using rvalue references (C++11)

Note: C++11 is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++11 standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++11 standard and therefore they should not be relied on as a stable programming interface.

In C++11, you can overload functions based on the value categories of arguments and similarly have lvalueness detected by template argument deduction. You can also have an rvalue bound to an rvalue reference and modify the rvalue through the reference. This enables a programming technique with which you can reuse the resources of expiring objects and therefore improve the performance of your libraries, especially if you use generic code with class types, for example, template data structures. Additionally, the value category can be considered when writing a forwarding function.

When you want to optimize the use of temporary values, you can use a move operation in what is known as destructive copying. Consider the following string concatenation and assignment:

```
std::string a, b, c;  
c = a + b;
```

In this program, the compiler first stores the result of `a + b` in an internal temporary variable, that is, an rvalue.

The signature of a normal copy assignment operator is as follows:

```
string& operator = (const string&)
```

With this copy assignment operator, the assignment consists of the following steps:

1. Copy the temporary variable into C using a deep-copy operation.
2. Discard the temporary variable.

Deep copying the temporary variable into C is not efficient because the temporary variable is discarded at the next step.

To avoid the needless duplication of the temporary variable, you can implement an assignment operator that moves the variable instead of copying the variable. That is, the argument of the operator is modified by the operation. A move operation is faster because it is done through pointer manipulation, but it requires a reference through which the source variable can be manipulated. However, `a + b` is a temporary value, which is not easily differentiated from a const-qualified value in C++ before C++11 for the purposes of overload resolution.

With rvalue references, you can create a move assignment operator as follows:

```
string& operator= (string&&)
```

With this move assignment operator, the memory allocated for the underlying C-style string in the result of `a + b` is assigned to C. Therefore, it is not necessary to allocate new memory to hold the underlying string in C and to copy the contents to the new memory.

The following code can be an implementation of the `string` move assignment operator:

```
string& string::operator=(string&& str)  
{  
    // The named rvalue reference str acts like an lvalue  
    std::swap(_capacity, str._capacity);  
    std::swap(_length, str._length);  
  
    // char* _str points to a character array and is a  
    // member variable of the string class  
    std::swap(_str, str._str);  
    return *this;  
}
```

However, in this implementation, the memory originally held by the string being assigned to is not freed until `str` is destroyed. The following implementation that uses a local variable is more memory efficient:

```
string& string::operator=(string&& parm_str)  
{  
    // The named rvalue reference parm_str acts like an lvalue  
    string sink_str;  
    std::swap(sink_str, parm_str);  
    std::swap(*this, sink_str);  
    return *this;  
}
```

In a similar manner, the following program is a possible implementation of a `string` concatenation operator:

```
string operator+(string&& a, const string& b)  
{
```



```

    return std::move(a+=b);
}

```

Note: The `std::move`¹ function only casts the result of `a+=b` to an rvalue reference, without moving anything. The return value is constructed using a move constructor because the expression `std::move(a+=b)` is an rvalue. The relationship between a move constructor and a copy constructor is analogous to the relationship between a move assignment operator and a copy assignment operator.

The `std::forward`¹ function is a helper template, much like `std::move`. It returns a reference to its function argument, with the resulting value category determined by the template type argument. In an instantiation of a forwarding function template, the value category of an argument is encoded as part of the deduced type for the related template type parameter. The deduced type is passed to the `std::forward` function.

The `wrapper` function in the following example is a forwarding function template that forwards to the `do_work` function. Use `std::forward` in forwarding functions on the calls to the target functions. The following example also uses the `decltype` and trailing return type features to produce a forwarding function that forwards to one of the `do_work` functions. Calling the `wrapper` function with any argument results in a call to a `do_work` function if a suitable overload function exists. Extra temporaries are not created and overload resolution on the forwarding call resolves to the same overload as it would if the `do_work` function were called directly.

```

struct s1 *do_work(const int&);           // #1
struct s2 *do_work(const double&);       // #2
struct s3 *do_work(int&&);                // #3
struct s4 *do_work(double&&);            // #4
template <typename T> auto wrapper(T && a)->
    decltype(do_work(std::forward<T>(*static_cast<typename std
    ::remove_reference<T>::type*>(0))))
{
    return do_work(std::forward<T>(a));
}
template <typename T> void tPtr(T *t);
int main()
{
    int x;
    double y;
    tPtr<s1>(wrapper(x));                 // calls #1
    tPtr<s2>(wrapper(y));                 // calls #2
    tPtr<s3>(wrapper(0));                 // calls #3
    tPtr<s4>(wrapper(1.0));               // calls #4
}

```

Note:

1. The following sample implements functionality similar to `std::move` and `std::forward`:

```

namespace MyStd {
    template <typename T> struct remove_reference {
        typedef T type;
    };
    template <typename T> struct remove_reference<T&> {
        typedef T type;
    };
    template <typename T> struct remove_reference<T&&> {
        typedef T type;
    };
}

namespace Impl {
    template <typename T> struct NotAnLvalueReference {
        enum { value = 1 };
    };
    template <typename T> struct NotAnLvalueReference<T&> {
        enum { value = 0 };
    };
}

template <typename T> inline
T &&forward(typename remove_reference<T>::type &t) {
    return static_cast<T &&>(t);
}

template <typename T> inline
T &&forward(typename remove_reference<T>::type &&t) {

```

```

        static_assert(Impl::NotAnLvalueReference<T>::value,
                      "T cannot be an lvalue reference type when
                      calling this overload.");
        return static_cast<T &&>(t);
    }

    template <typename T> inline
    typename remove_reference<T>::type &&move(T &&t) {
        return static_cast<typename remove_reference<T>::type &&>(t);
    }
}

```

Using shared-memory parallelism (SMP)

You can compile your program with the SMP option to generate threaded code that exploits shared-memory parallelism. The SMP option implies the HOT option and an optimization level of OPTIMIZE(2).

The following table lists the suboptions of the SMP option. For descriptions and syntax of the suboptions, see the SMP option in [z/OS XL C/C++ User's Guide](#).

Table 89. SMP suboptions	
suboption	Behavior
EXPLICIT	Enables directives that control explicit parallelization of loops.
NOEXPLICIT	Disables the directives that control explicit parallelization of loops.
OPT	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to OPTIMIZE(2) and HOT in the absence of other optimization options.
NOOPT	Instructs the compiler to do the smallest amount of optimization that is required to parallelize the code. During development, it can be useful to turn off optimization to facilitate debugging.

Tips for using the SMP option

Here are some suggestions for using the SMP option:

- Before using the SMP option, test your programs using OPTIMIZATION and HOT in a single-threaded manner.
- Use the thread-safe version of system library routines inside the parallel regions.
- By default, the runtime environment uses all available processors. Do not set the `OMP_NUM_THREADS` environment variable unless you want to use fewer than the number of available processors. You might want to set the number of executing threads to a small number or to 1 to ease debugging.
- When debugging an OpenMP program, try using SMP(NOOPT) (without OPTIMIZE) to make the debugging information produced by the compiler more precise.

Chapter 35. Using built-in functions to improve performance

A built-in function is inline code that is generated in place of an actual function call. The compiler will generate inline code for built-in functions, if the appropriate header files are included in the source code. For a list of the built-in functions, see [Chapter 29, “Using hardware built-in functions,” on page 361](#).

If you have included the header files but you want to call either the library version of the function or your own version, enclose the function name in parentheses when you make the call. For example, if you wanted to call only `memcpy` from the header file and use the built-in functions for other memory-related functions, code the function call as follows:

```
(memcpy)(buf1, buf2, len)
```

Note: When **NOOPT** or **COMPACT** is specified, the compiler might not expand all built-in functions.

The compiler can also generate inline code for some of the C library functions, if the appropriate header files are included in the source code. The inline code behaves exactly the same as these C library functions. For more information, see [Using hardware built-in functions in z/OS C/C++ Runtime Library Reference](#).

The following table lists the C library built-in functions and the header files that they belong to.

Table 90. C-library built-in functions

Built-In Function	Header File
<code>abs()</code>	<code>stdlib.h</code>
<code>alloca()</code>	<code>stdlib.h</code>
<code>ceil()</code> “1” on page 444	<code>math.h</code>
<code>ceilf()</code> “1” on page 444	<code>math.h</code>
<code>ceill()</code> “1” on page 444	<code>math.h</code>
<code>decabs()</code>	<code>decimal.h</code>
<code>decchk()</code>	<code>decimal.h</code>
<code>decfix()</code>	<code>decimal.h</code>
<code>fabs()</code> “1” on page 444	<code>math.h</code>
<code>floor()</code> “1” on page 444	<code>math.h</code>
<code>floorf()</code> “1” on page 444	<code>math.h</code>
<code>floorl()</code> “1” on page 444	<code>math.h</code>
<code>fortrc()</code>	<code>stdlib.h</code>
<code>memchr()</code>	<code>string.h</code>
<code>memcpy()</code>	<code>string.h</code>
<code>memcmp()</code>	<code>string.h</code>
<code>memset()</code>	<code>string.h</code>
<code>strcat()</code>	<code>string.h</code>
<code>strchr()</code>	<code>string.h</code>
<code>strcmp()</code>	<code>string.h</code>

Table 90. C-library built-in functions (continued)

Built-In Function	Header File
strcpy()	string.h
strlen()	string.h
strncat()	string.h
strncmp()	string.h
strncpy()	string.h
strrchr()	string.h
wmemchr() “2” on page 444	wchar.h
wmemcmp() “2” on page 444	wchar.h
wmemcpy() “2” on page 444	wchar.h
wmemset() “2” on page 444	wchar.h

Notes:

1. The compiler only attempts to generate inline code for this built-in function when the **OPTIMIZE(2)** compiler option is in effect.
2. The compiler only attempts to generate inline code for this built-in function when the **ARCH(7)** compiler option is in effect. LP64 compiles will not generate inline code.

__builtin_expect

You can use the `__builtin_expect` built-in function to indicate that an expression is likely to evaluate to a specified value. The compiler can use this knowledge to direct optimizations. This built-in function is portable with the GNU C/C++ `__builtin_expect` function.

The prototype of this built-in function is as follows:

```
long __builtin_expect (long exp, long c);
```

where `exp` is the integral-type expression to be evaluated and `c` is the expected value of the expression.

If `exp` does not actually evaluate at run time to the predicted value `c`, performance might suffer. Therefore, you must use this built-in function with caution.

Platform-specific functions

The built-in functions in this section are related to C-library functions that are z/OS specific. The full description of each function can be found in *z/OS C/C++ Runtime Library Reference*.

Table 91. Platform-specific built-in functions

Built-In Function	Header File
cds()	stdlib.h
cs()	stdlib.h

Note: `cds()` and `cs()` are masking macros. The system header expands them to the `__cds` and `__cs`. It is advisable to use the hardware functions instead of the library functions whenever possible. For more information, see [Table 64 on page 361](#).

Examples

- You can use the following macros rather than their equivalent functions, if you include the `ctype.h` header file.

<code>isalpha()</code>	<code>isalnum()</code>	<code>islower()</code>	<code>isprint()</code>	<code>isupper()</code>	<code>isxdigit()</code>
<code>iscntrl()</code>		<code>ispunct()</code>		<code>toupper()</code>	
<code>isdigit()</code>		<code>isspace()</code>		<code>tolower()</code>	
<code>isgraph()</code>					

- If you are using the `__cs1` or `__c1s1` function with arguments other than the ones declared in the prototypes in `stdlib.h`, the compiler might not be able to generate correct code at OPT. In this case, use the `NOANSIALIAS` option.

Note: As of z/OS V1R2, the new forms for `cs()` and `c1s()` are `__cs1` and `__c1s1`, respectively. For more information, see [Chapter 29, “Using hardware built-in functions,”](#) on page 361.

- Typically, arrays are compared element-by-element, using a loop. When you compare two arrays for equality, replace the loop with the `memcmp()` library function. This could result in the execution of many machine instructions being replaced by the execution of only a few machine instructions.

More efficient comparison with <code>memcmp()</code> library function	Less efficient comparison in a loop
<pre>if (!memcmp (a, b, sizeof(a))) /* arrays are equal */</pre>	<pre>int a[1000], b[1000]; for (i = 0; i < 1000; ++i) if (a[i] != b[i]) break; if (i == 1000) /* arrays are equal */</pre>

- Neither the C nor the C++ language allows structure comparison, because structures might contain padding bytes with undefined values. In cases where you know that no padding bytes exist, use `memcmp()` to compare structures. The z/OS AGGREGATE compiler option for C is used to obtain a structure and union map.
- The `memset()` library function should be used to initialize a character buffer and to initialize an array to a repetitive byte pattern (such as zeros).
- Use `memset()` to clear structs, unions, arrays or character buffers as follows:

<pre>char c[10]; for (i = 0; i < 10; i++) c[i] = ' '; memset (c, ' ', sizeof (c));</pre>	<pre>/* do not use */ /* better */</pre>
---	--

- Use the `alloca()` function to automatically allocate memory from the stack. This function frees memory at the end of a function call when z/OS XL C/C++ collapses the stack. For more information, see [`alloca\(\)` in z/OS C/C++ Runtime Library Reference](#).
- When using `strlen()`, do not hide size information. Less code is needed for `strlen()` when the upper bound is known at compile time.

```

char    small_str_array[100];
char    *small_str_ptr;
:
x = strlen(small_str_ptr);    /* unknown upper bound */
x = strlen(small_str_array); /* better */

```

- When concatenating strings, use `strcat()`.
- When performing character-to-integer conversions, use `atoi()` rather than `sscanf()`.
- Whenever possible, replace `strxxx()` functions with their corresponding `memxxx()` functions, because `memxxx()` functions are more efficient. You can minimize the execution cost of a `strxxx()` function by using fixed-length character buffers to save the length of incoming strings (including null terminators) for subsequent calls to `memcpy()` and `memcmp()`.

```

total_len = strlen (s) + 1;
:
for (i = 0; i < 10; i++)
    if (memcmp (s, t[i], total_len) == 0) /* total_len ≤ sizeof(t) */
:
memcpy (a, s, total_len);

```

If you try to replace all `strcmp()` calls with a `memcmp()` call taking a `strlen()` value of one of the strings, the result might be an attempt to access protected storage which follows the shorter string. Such an attempt could cause an exception because `memcmp()` does not stop comparing strings when it encounters a null in one of the strings.

- Whenever possible, replace `wcsxxx()` functions with their corresponding `wmemxxx()` functions, because `wmemxxx()` functions are more efficient. You can minimize the execution cost of a `wcsxxx()` function by using fixed-length wide character buffers to save the length of incoming wide character strings (including null terminators) for subsequent calls to `wmemcpy()` and `wmemcmp()`.

Chapter 36. I/O Performance considerations

This chapter discusses the most efficient use of the available XL C/C++ input and output methods. This includes:

- [“Accessing MVS data sets” on page 447](#)
- [“Accessing UNIX file system files” on page 448](#)
- [“Using memory files” on page 449](#)
- [“Using the C++ I/O stream libraries” on page 449](#)

Accessing MVS data sets

- Consider the use of the file when choosing DCB parameters:
 - Specify largest possible BLKSIZE (blocked files).
 - Use `recfm = FBS` or `F` over `FB` unless dealing with a PDS. The use of standard (`S`) blocks optimizes the sequential processing of a file on a direct-access device.
 - `fseek()` on sequential files is most efficient when using `recfm = F` or `recfm = FBS`.
 - If you are accessing an existing sequential file created as `FB`, and you know that there are no short blocks in the file, specify `FBS` on the call to `fopen()` or `freopen()` to enable the library to perform faster repositions.

The proper choice of file attributes is important for efficient I/O.

- When you do not need to reposition within a file, take advantage of `NOSSEEK` for more efficient reading and writing to a data set. You can also specify `NCP` or `BUFNO` on the DD statement for MVS DASD data sets, thereby reducing the clock time of the application. See [“Multiple buffering” on page 53](#) for more information.
- If possible, read or write a block at a time to minimize the I/O overhead and elapsed time.
- Using text I/O for writing can be slower than using binary or record I/O. When you use binary or record I/O, the application must ensure that the data is written to the file in the correct format.
- If you are using `FB` or `FBS` files, use binary I/O instead of record I/O. This way, you can read or write more than one record at a time.
- Use `fread()` instead of `fgets()`, and `fwrite()` in place of `fputs()`, wherever possible.
- Use `putc()` instead of `fputc()`, and `getc()` instead of `fgetc()`, if you must read or write a character.

The `fputc()` function, as defined by ANSI, puts a single character to the text stream. Special action occurs when writing a control character. On the other hand, the `putc()` macro buffers characters in storage and invokes `fputc()` only when encountering a control character. This reduces call overhead when you are writing one character at a time.

- If you are using hiperspace memory files, you can use `setvbuf()` to set the buffer size.

The default buffer size for memory files in hiperspace is 16K. You can override this by calling `setvbuf()` after `fopen()`, but before performing any I/O operations on the file. The minimum buffer size is 4K. If you specify a smaller size, it is ignored, and the default is used instead.

If your file will be large, you can improve execution time by increasing the buffer size. This will result in less frequent flushing of the buffer to the hiperspace, but will cost you memory in the user address space for the larger buffers. For example,

```
rc = setvbuf(fp, NULL, _IOFBF, 32768);
```

Alternatively, if your memory is constrained, you can reduce requirements for memory in the user address space by reducing the buffer size. This will result in more frequent flushing of the buffer to the hiperspace. For example,

```
rc = setvbuf(fp, NULL, _IOFBF, 4096);
```

For more information on hiperspace memory files, refer to [Chapter 12, “Performing memory file and hiperspace I/O operations,”](#) on page 141.

- When writing to text files that do not use DBCS characters, ensure that MB_CUR_MAX is set to 1 for the current locale. This will prevent internal I/O checks for DBCS strings.
- Avoid using `fscanf()` or `fprintf()` if you can use other I/O routines instead. For example, use `fwrite()` rather than `fprintf()` to write out a format string with no substitution variables.
- When using `fflush()` beware of NULL file pointers; `fflush(NULL)` flushes all open streams.
- Specify DCB parameters on `fopen()` only when you are creating the file. When you are appending, updating or reading a file, these attributes are retrieved from the existing file.

Many file attributes (DCB parameters) are possible when you open a file with z/OS XL C/C++. DCB parameters specified on `fopen()` must be compatible with those of the file or the ddname. This checking may cause unwanted overhead.

- Use `fgetpos()` and `fsetpos()` instead of `ftell()` and `fseek()` when you are saving a position you will return to later. `fgetpos()` saves more information about the position than `ftell()`.
- Where possible, use striped data sets. These data sets improve overall I/O throughput.
- For temporary files, use memory files rather than files created with `tmpfile()`.

You can use MVS memory files from z/OS UNIX C++ application programs. However, use of the `fork()` function from the program clears a memory file and removes access from a hiperspace memory file for the child process. Use of an `exec` function from the program clears a memory file when the process address space is cleared.

- For large memory files (1MB or larger) in which you perform random seeking, use hiperspace memory files, if they are available.
- When your library is below the 16MB line, use hiperspace memory files.

The non-hiperspace files use up your storage from below the line. Hiperspace memory files do not reside in user virtual storage. Changing a memory file to a hiperspace memory file saves user virtual storage only if the file is larger than one hiperspace memory file buffer.

- For VSAM I/O use VSAM buffers appropriately and use `flocate()` instead of `ftell()` and `fseek()`.

Accessing UNIX file system files

When accessing UNIX file system files, you should review the following considerations:

- Use `fread()` instead of `fgets()`, and `fwrite()` in place of `fputs()`, wherever possible.
- Use `putc()` instead of `fputc()`, and `getc()` instead of `fgetc()`, if you must write or read a character.
- When using `fflush()`, beware of NULL file pointers; `fflush(NULL)` flushes all open streams.
- Changing the buffer size for access to UNIX file system may provide advantages. Rather than using the default of 4K, you may want to use the `setvbuf()` function to set the buffer size to be the length of the read or write operation that you normally do.

For example, in applications that access files in a z/OS File System (zFS) file system, the recommended `setvbuf()` buffer size is 64K.

When you include the header file `stdio.h`, macros are defined for `getc()`, `putc()`, `getchar()`, and `putchar()`. To use the function calls instead of the macro calls, use `#undef` after the `stdio.h` header file is included. If you are working with a threaded application, these macros are automatically undefined forcing the application to use function calls, which are thread safe. The feature test macro

`_ALL_SOURCE` causes these four macros to be undefined. However, if you require `_ALL_SOURCE`, and want these macros to be used in a non multi-threaded application, you can use feature test macro `_ALL_SOURCE_NOTHREADS`.

Using memory files

Use memory files as efficient temporary files by specifying the `type=memory` attribute in `fopen()` before creating the temporary file. Some applications use temporary files to pass data between program modules.

When using one of the z/OS UNIX shells, an MVS memory file may or may not make an efficient temporary file. This depends on whether your z/OS UNIX XL C/C++ application program uses `fork()` and `exec()` functions to call another program to run in a child process. The child process does not inherit MVS memory files after an `exec()` function. For more information, see [“Accessing MVS data sets” on page 447](#).

Using the C++ I/O stream libraries

The following information applies to the USL I/O Stream Class Library and to the Standard C++ I/O stream classes.

- Unit-buffering incurs a significant performance penalty. Unit-buffering can be enabled by setting the `ios::unitbuf` flag. It is enabled for the `cerr` object by default.
- The `sync_with_stdio()` function enables unit-buffering of standard streams, to ensure their synchronization with C standard streams. However, a runtime performance penalty is incurred to ensure this synchronization. For more information about `sync_with_stdio()`, see [Chapter 4, “Using the Standard C++ Library I/O Stream Classes,” on page 21](#).
- In most cases, calls to functions in the USL or ANSI C++ I/O stream libraries are mapped to calls to the I/O functions of the C standard library. For this reason, direct calls to the C I/O functions are recommended for applications that must have the best possible performance. This does not mean that these types of applications cannot or should not contain any `iostream.h` calls. However, you might want to ensure that `iostream.h` I/O calls do not appear on the critical path; it is safe to keep them for unused debugging code.

Note: If you access the same file with both C and C++ I/O stream classes, undefined results will occur.

Chapter 37. Improving performance with compiler options

This information discusses and lists the z/OS XL C/C++ compiler options that you can use to improve application performance.

Using the OPTIMIZE option

During optimization, the compiler changes the unoptimized code sequences, derived from the source code, into equivalent code sequences that execute faster and usually require less memory space. It is also possible for an expression that would normally cause an exception to be removed by optimization, thus preventing the exception.

Note: You can optimize code by specifying either OPTIMIZE(2) or OPTIMIZE(3). Optimized code takes significantly more time to compile than unoptimized code, but will likely result in faster-running code. There is no guarantee that the compile time at OPTIMIZE(3) will remain similar from release to release.

Because the optimization is achieved by transforming the code using knowledge obtained from a larger program context, the direct correspondence between source and object code is often lost. Optimized code is also more sensitive to subtle coding errors.

One example of a subtle coding error is to type cast a pointer variable incorrectly. The compiler assumes ISO conformance when doing optimization. If your program does not conform, you may receive undefined results. For more information, see [“ANSI aliasing rules” on page 429](#) and [“Using ANSI aliasing rules” on page 431](#).

Optimizations performed by the compiler

The compiler performs several optimizations, including:

Inlining

Inlining replaces certain function calls with the actual code of the function being performed. For more information on inlining, see [“Inlining” on page 455](#).

For z/OS XL C/C++, automatic inlining is performed by default when you specify OPTIMIZE. You can override this inlining by using the NOINLINE option. For more information, see [INLINE | NOINLINE in z/OS XL C/C++ User's Guide](#).

Value numbering

Value numbering involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

Straightening

Straightening is rearranging the program code to minimize branching logic and to combine physically separate blocks of code.

Common expression elimination

Common expressions recalculate the same value in a subsequent expression. The duplicate expression can be eliminated by using the previous value. This is done even for intermediate expressions within expressions.

If your program contains the following statements, the common expression `c + d` is saved from its first evaluation and is used in the subsequent statement to determine the value of `f`.

```
a = c + d;  
.  
.  
.  
f = c + d + e;
```

Code motion

If variables used in a computation within a loop are not altered within the loop, it may be possible to perform the calculation outside of the loop and use the results within the loop.

Strength reduction

Less efficient instructions are replaced with more efficient ones. For example, in array addressing, an add instruction replaces a multiply.

Constant propagation

Constants used in an expression are combined and new ones generated. Some mode conversions are done, and compile-time evaluation of some intrinsic functions takes place.

Instruction scheduling

Instructions are reordered to minimize execution time.

Dead store elimination

The compiler eliminates stores when the value stored is never referred to again. For example, if two stores to the same location have no intervening load, the first store is unnecessary, and is therefore removed.

Dead code elimination

The compiler may eliminate code for calculations that are not required. Other optimization techniques may cause code to become dead.

Graph coloring register allocation

The compiler uses a global register allocation for the whole function, thereby allowing variables to be kept in registers rather than in memory.

These optimization techniques may be performed both locally and globally. Increases in storage and compile time requirements over NOOPT will occur. Higher levels of optimization may perform the same options more rigorously as well as adding additional options.

Aggressive optimizations with OPTIMIZE(3)

The compiler optimizes more aggressively with OPTIMIZE(3) than with OPTIMIZE(2). Code may be moved, and computations may be scheduled, even if this could potentially raise an exception.

OPTIMIZE(3) may place instructions onto execution paths where they will be executed when they may not have been according to the actual semantics of the program. For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved using OPTIMIZE(2) because the computation may cause an exception. For OPTIMIZE(3), the compiler will move the computation because it is not certain to cause an exception.

The same is true for moving loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable using OPTIMIZE(3). Loads in general are not considered to be absolutely safe using OPTIMIZE(2) because a program can contain a declaration of a static array of 10 elements and load a[60000000003], which could cause a segmentation violation.

The same concepts apply to scheduling. In [Figure 119 on page 452](#), using OPTIMIZE(2), the computation of b+c is not moved out of the loop for two reasons:

- It is considered dangerous because it is a floating-point operation
- It does not occur on every path through the loop

```

:
int i;
float a[100], b, c;
for (i=0; i < 100; i++)
{
    if (a[i] < a[i+11])
        a[i] = b + c;
}
:

```

Figure 119. Example of using OPTIMIZE(2)

At OPTIMIZE(3), the computation $b + c$ is moved out of the loop.

Some general differences with OPTIMIZE(2) are as follows:

- Increased optimization scope, typically to encompass a whole procedure
- Specialized optimizations that might not help all programs
- Optimizations that require large amounts of compile time or space
- Elimination of implicit memory usage
- Activation of NOSTRICT, which allows some reordering of floating-point computations and potential exceptions

Because OPTIMIZE(3) implies the NOSTRICT option, certain floating-point semantics of your application can be altered to gain execution speed. These typically involve precision trade-offs such as the following operations:

- Reordering of floating-point computations
- Reordering or elimination of possible exceptions (for example, division by zero or overflow)
- Combining multiple floating-point operations into single machine instructions; for example, replacing an add then multiply with a single more accurate and faster float-multiply-and-add instruction

You can still gain most of the benefits of OPTIMIZE(3) while preserving precise floating-point semantics by specifying STRICT. This is only necessary if a particular level of floating-point computational accuracy, as compared with NOOPT or OPTIMIZE(2) results, is important. You can also specify STRICT if your application is sensitive to floating-point exceptions, or if the order and manner in which floating-point arithmetic is evaluated is important. Largely, without STRICT, the difference in computed values on any one source-level operation is very small compared to lower optimization levels. However, the difference can compound if the operation involved is in a loop structure, and the difference becomes additive.

The xlc utility optimization option levels

You can use the xlc utility options to specify five base optimization levels, which map to the z/OS batch options as follows:

- **-00** or NOOPT, almost no optimization, best for getting the most debugging information
- **-02** or OPTIMIZE(2), strong low-level optimization that benefits most programs
- **-03** or OPTIMIZE(3), intense low-level optimization analysis
- **-04** or OPTIMIZE(3), HOT, IPA(LEVEL(1)), all of **-03** plus detailed loop analysis and basic whole-program analysis at link time
- **-05** or OPTIMIZE(3), HOT, IPA(LEVEL(2)), all of **-04** and detailed whole-program analysis at link time

Note: **-01** level is not supported.

Optimization progression

Table 92 on page 453 details options you should use with each level and some useful additional options.

Table 92. Optimization levels and options		
The xlc utility optimization option level	Additional batch options implied by optimization level	Additional recommended batch options
-00	None	ARCH(n)
-02	None	ARCH (n) INLINE (to tune inlining) TUNE(n)

Table 92. Optimization levels and options (continued)		
The xlc utility optimization option level	Additional batch options implied by optimization level	Additional recommended batch options
-03	NOSTRICT	ARCH(n) TUNE(n)
-04	All of OPTIMIZE(3) plus: HOT IPA(LEVEL(1))	ARCH(n) TUNE(n) PDF
-05	All of OPTIMIZE(3) plus: HOT IPA(LEVEL(2))	ARCH(n) TUNE(n) PDF

While Table 92 on page 453 provides a list of the most common compiler options for optimization, the z/OS XL C/C++ compiler offers optimization facilities for almost any application. For more information, see [“Additional options that affect performance”](#) on page 465.

Using the ARCHITECTURE and TUNE options

ARCHITECTURE option

The ARCHITECTURE option specifies the architectural level for which the executable program's instructions will be generated.

The ARCHITECTURE option instructs the compiler to structure your application to execute on a particular set of machines that support the specified instruction set and later. The choice of processor gives you the flexibility of compiling your application to execute optimally on a particular machine or on any higher-level architecture machines but still have as much architecture-specific optimization applied as possible.

Using the correct ARCHITECTURE option is the most important step in influencing chip-level optimization. The compiler uses the ARCHITECTURE option to make both high and low-level optimization decisions and trade-offs. The ARCHITECTURE option allows the compiler to access the full range of processor hardware instructions and capabilities when making code generation decisions. Even at low optimization levels, specifying the correct target architecture can have a positive impact on performance.

For example, to compile applications with the z/OS XL C/C++ compiler to produce code that uses instructions available on the z13[®] models, use ARCHITECTURE(11). To compile z/OS C/C++ applications that will only run on 64-bit mode capable hardware, use ARCHITECTURE(5) to select the entire 64-bit z/Architecture family of processors.

TUNE option

The TUNE option specifies for which architectural level the executable program will be optimized. The TUNE option allows the compiler to take advantage of differences (such as scheduling of instructions) in architectural levels.

Use the TUNE option to direct the optimizer to bias optimization decisions for executing the application on a particular architecture but not preventing the application from running on other architectures. The default TUNE setting depends on the setting of the ARCHITECTURE option. If the ARCHITECTURE option selects a particular machine architecture, the range of TUNE suboptions that are supported is limited by the chosen architecture and all architectures above that level. Using TUNE allows the optimizer to perform transformations, such as instruction scheduling, so that resulting code executes most efficiently on your chosen TUNE architecture.

Use TUNE to specify the most common or important processor where your application executes. For example, if your application usually executes on zEC12 models but sometimes executes on z13 models, use TUNE(10). The code generated executes more efficiently on zEC12 models but can run correctly on z13 models.

Inlining

Inlining replaces certain function calls with the actual code of the function and is performed before all other optimizations. Not only does inlining eliminate the linkage overhead, it also exposes the entire called function to the caller, which enables the compiler to better optimize your code.

Note: See “Inlining under IPA” on page 457 for information on differences in inlining under IPA.

The following types of calls are not inlined:

- A call where the number of parameters on the call does not match that on the function definition. An example of this is a variable argument function call.
- A call that is directly recursive; the routine calls itself.
- K&R style var_arg functions.

Consider the C examples CCNGOP1 and CCNGOP2, shown in Table 93 on page 455. CCNGOP1 specifies the **#pragma inline** directive for the function which_group(). If you use the OPTIMIZE option when you compile CCNGOP1, the compiler determines that CCNGOP1 is equivalent to CCNGOP2.

Table 93. Examples of optimization

Sample program CCNGOP1

```
/* this example demonstrates optimization */
#include <stdio.h>
#pragma inline (which_group)
int which_group (int a) {
    if (a < 0) {
        printf("first group\n");
        return(99);
    }
    else if (a == 0) {
        printf("second group\n");
        return(88);
    }
    else {
        printf("third group\n");
        return(77);
    }
}

int main (void) {
    int j;

    j = which_group (7);

    return(j);
}
```

Sample program CCNGOP2

```
/* this example also demonstrates optimization */
#include <stdio.h>

int main(void) {

    printf("third group\n");    /* a lot
less code

generation */

    return(77);
}
```

Selectively marking code to inline

The z/OS XL C/C++ inliner supports two modes of running: selective and automatic.

Selective mode enables you to specify, in your source code, the functions that you do, and do not, want inlined.

If you know which functions are frequently invoked from within a compilation unit, you can mark them for inlining:

- For a C program, add the appropriate **#pragma inline** directives in your source and compile with `INLINE (NOAUTO,REPORT,,)`.
- For a C++ program, add `inline` keywords to your source and compile with `INLINE (NOAUTO,REPORT,,)`.
- You can also use the `always_inline` function attribute to inline a function, regardless of whether optimization was specified at compile time.

If your code contains complex macros, the macros can be made into static routines that are marked to be inlined at no execution-time cost. All static routines that are interfaces to a data object can be placed in a header file.

Automatically choosing functions to inline

Automatic mode assists you with starting to optimize your code. It allows the compiler to choose potential functions to inline. The compiler will inline all routines that are less than the *threshold* in abstract code units (ACUs) until the function that the functions are inlined into is greater than *limit* abstract code units. The *threshold* and *limit* parameters are defined as follows:

threshold

Maximum relative size of a function to inline. The default value is 100 Abstract Code Units (ACUs), both for C and C++. ACUs are proportional in size to the executable code in the function; your code is translated into ACUs by the compiler. Specifying a threshold of 0 is equivalent to specifying `NOAUTO`. Note that the proportion of ACUs to executable code in a function is different under IPA.

limit

Maximum relative size a function can grow before auto-inlining stops. The default is 1000 ACUs for the specific function. Specifying a limit of 0 is equivalent to specifying `NOAUTO`.

Note: When functions become too large, runtime performance can degrade.

Under the z/OS UNIX shell, to provide assistance in choosing which routines to inline, use the **c89 -W** command to pass the `INLRPT` option to the z/OS XL C/C++ compiler. At `NOOPT`, you will also need to specify the `INLINE` option. The default at `NOOPT` is `NOINLINE`.

For example, at `NOOPT`, to get `INLINE(AUTO,REPORT,100,1000)` for a C program, use one of the following `c89` commands:

```
c89 -W "0,inline(,REPORT,,)" example.c
c89 -W "0,inline,inlrpt" example.c
```

You can get the same value at `OPT` for a C program passing the `INLRPT` option to the z/OS XL C/C++ compiler as follows:

```
c89 -2 -W "0,inlrpt"
```

Note: Inlining debugging functions or functions that are rarely invoked can degrade performance. Use the **#pragma noline** directive to instruct the automatic inliner to not inline these types of functions. The **#pragma inline** and the **#pragma noline** directives and the `inline` keyword are honored by automatic inlining regardless of the *limit* and *threshold* you have specified. For more information, see [#pragma inline \(C only\) / noline in z/OS XL C/C++ Language Reference](#).

Modifying automatic inlining choices

While automatic inlining is the best choice the compiler can make for you, you can further improve your performance. Use **#pragma inline** and **#pragma noline** to reduce the need to modify your inlining choices when you change your application. You may want to wait until you have a stable application before you do the following steps.

1. Compile with the `OPTIMIZE` option and ask for a report from the inliner by specifying the compiler options `INLINE(,REPORT,,)` or `INLRPT` and `OPTIMIZE`.

2. Look at the report to see if anything was inlined that should not have been; for example, routines for debugging or handling exceptions. Add **#pragma noline** to your source to insure that these functions do not get inlined.
3. Add the **inline** keyword (for C++) or the **#pragma inline** directive (for C) to any frequently used routines to ensure that it gets inlined.
4. Recompile with **OPTIMIZE** then, regenerate the inline report and reanalyze for functions that should and should not be inlined.
5. You should also vary the limit and threshold values.
 - The inline report tells you the abstract code units (ACUs) for each function. These should help you determine an appropriate *threshold* to start from. In general, your initial *threshold* should be as small as possible, and your initial limit should be in the 1000 to 2000 range.
 - Increase the *threshold* by an increment small enough to catch a few more routines each time.
 - Change the limit when you wish. Because performance will improve as a function of both the *limit* and the *threshold* values, it is not recommended that you change both *limit* and *threshold* at the same time.
6. Repeat the process until you feel that you have found the best performance parameters. You should run your application to determine if the tuning has found the best performance parameters.
7. When you are satisfied with the selection of inlined routines, add the appropriate **#pragma inline** directives or **inline** keywords to the source. That is, when the selected routines are forced with these directives, you can then compile the program in selective mode. This way, you do not need to be affected by changes made to the heuristics used in the automatic inliner.

Overriding inlining defaults

Automatic and selective inlining are performed when the **OPTIMIZE** compiler option is specified. You can override this by specifying the **NOINLINE** option when you specify your optimization level. You can also override this by specifying the **#pragma noline** directive for a particular function. For more information, see [#pragma inline \(C only\) / noline](#) in *z/OS XL C/C++ Language Reference*.

Inlining under IPA

The IPA Inliner functions differently from the regular inliner:

- It performs inlining across compilation units, rather than within a compilation unit.
- It handles inlining of functions with variable argument lists.
- It inlines calls from recursive cycles (for example, where function A calls function B calls function C calls function A). However, it avoids making the functions too large.

For more information about IPA, see [“Using the IPA option”](#) on page 458.

Using the XPLINK option

Applications that make many calls to small functions get the most benefit from using XPLINK. Many C++ applications are structured this way, because of the object oriented programming model. C applications that make many function calls may also be suitable for XPLINK.

When you should not use XPLINK

Functions compiled XPLINK and NOXPLINK cannot be combined in the same program object.

XPLINK provides a significant performance enhancement to some applications, but can degrade the performance of applications that are not suitable for XPLINK.

Another way to call an XPLINK function from a non-XPLINK program object is to use the DLL call mechanism. There is an overhead cost associated with calls made from non-XPLINK to XPLINK, and from XPLINK to non-XPLINK. This overhead includes the need to swap from one stack type to another and to

convert the passed parameters to the style accepted by the callee. Applications that make a large number of these "cross-linkage" calls may lose any benefit obtained from the parts that have been compiled XPLINK. In fact, performance could degrade from the pure non-XPLINK case. If the number of pure XPLINK function calls is significantly greater than the number of "cross-linkage" calls, the cost saved on XPLINK calls will offset the costs associated with calls that involve stack swapping.

When you introduce an XPLINK program object into your application (for example, an XPLINK version of a vendor-DLL which your application uses), your application must run in an XPLINK environment (this is controlled by the XPLINK runtime option). In an XPLINK environment, an XPLINK version of the C/C++ Runtime Library (RTL) is used. You cannot have both the non-XPLINK and XPLINK versions of the C/C++ RTL active at the same time, so non-XPLINK callers of the C/C++ RTL will also incur this stack swapping overhead in an XPLINK environment.

The maximum performance improvement can be achieved by recompiling an entire application XPLINK. The further the application gets from pure XPLINK, the less the performance improvement. At some point, you may actually see a performance degradation.

The only compiler that currently supports the XPLINK compiler option is the z/OS C/C++ compiler. All COBOL and PL/I programs are non-XPLINK. Calls between COBOL or PL/I and XPLINK-compiled C/C++ are cross-linkage calls and will incur the stack swapping overhead.

For more information on making ILC calls with XPLINK, refer to *z/OS Language Environment Writing Interlanguage Communication Applications*.

Applications that use Language Environment facilities that are not supported in an XPLINK environment, or that use products that are not supported in an XPLINK environment (for example, CICS), can not be recompiled as XPLINK applications.

For more information about XPLINK, see *z/OS Language Environment Programming Guide*.

Using the HOT option

The HOT option enables the compiler to request high-order transformations on loops during optimization, which gives you the ability to generate more highly optimized code.

Loops typically account for the majority of the execution time of most applications and the HOT optimizer performs in-depth analysis of loops to minimize their execution time. Loop optimization techniques include: interchange, fusion, unrolling of loop nests, and reducing the use of temporary arrays. There are three goals in these optimizations:

- Reducing the costs of memory access through the effective use of caches and translation look-aside buffers (TLBs). Increasing memory locality reduces cache/TLB misses.
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware.
- Improving the utilization of processor resources through reordering and balancing the usage of instructions with complementary resource requirements. Loop computation balance typically involves load/store operations balanced against floating-point computations.

Using the IPA option

Interprocedural Analysis (IPA), through the IPA option, can also improve the execution time of your z/OS XL C/C++ application. IPA is a mechanism for performing optimizations across compilation unit boundaries. It also performs optimizations not otherwise available with the z/OS XL C/C++ compiler, such as:

- Inlining across compilation units
- Program partitioning
- Coalescing of global variables
- Code straightening
- Unreachable code elimination

- Call graph pruning of unreachable functions

IPA also supports Program-directed feedback (PDF). The PDF suboptions allow the compiler to use information from training runs when optimizing the code. The compiler can then focus its optimizations on the most executed parts of the code and move low-priority code out of the critical path.

This information provides an overview of the Interprocedural Analysis (IPA) processing that is available through the IPA compiler option. For more information, see:

- For the effects of IPA on compiling, compiler options, and compiler listings: [IPA considerations in z/OS XL C/C++ User's Guide](#)
- For the effects of IPA on pragmas: [IPA effects in z/OS XL C/C++ Language Reference](#)

Types of procedural analysis

The z/OS XL C/C++ compiler performs both intraprocedural and interprocedural analysis.

Intraprocedural analysis is a mechanism for performing optimization for each function in a compilation unit, using only the information available for that function and compilation unit.

Interprocedural analysis is a mechanism for performing optimization across function and compilation unit boundaries. When inlining is in effect, the C/C++ compiler performs a limited form of interprocedural analysis, where it only applies within a compilation unit.

Interprocedural analysis through the IPA compiler option improves upon the limited interprocedural analysis described above. When you invoke interprocedural analysis through the IPA option, the compiler performs optimizations across the entire program. It also performs optimizations not otherwise available with the C/C++ compiler. The types of optimizations performed include:

Inlining across compilation units

Inlining replaces certain function calls with the actual code of the function. Inlining not only eliminates the linkage overhead but also exposes the entire function to the caller and thus enables the compiler to better optimize your code.

Program partitioning

Program partitioning improves performance by reordering functions to exploit locality of reference. Functions that call each other frequently will be closer together in memory.

Coalescing of global variables

The compiler puts global variables into one or more structures and accesses the variables by calculating the offsets from the beginning of the structures. This lowers the cost of variable access and exploits data locality.

Code straightening

Code straightening streamlines the flow of your program.

Unreachable code elimination

Unreachable code elimination removes unreachable code within a function.

Call graph pruning of unreachable functions

Call graph pruning of unreachable functions removes code that is 100% inlined or never referenced.

Intraprocedural constant and set propagation

IPA propagates floating point and integer constants to their uses and computes constant expressions at compile time. Also, variable uses that are known to be one of several constants can result in the folding of conditionals and switches.

Intraprocedural pointer alias analysis

IPA tracks pointer definitions to their uses, resulting in more refined information about memory locations that a pointer dereference may use or define. This enables other parts of the compiler to better optimize code around such dereferences. IPA tracks data and function pointer definitions. When a pointer dereference can only refer to a single memory location or function, the dereference is rewritten to be an explicit reference to the memory location or function.

Intraprocedural copy propagation

IPA propagates expressions defining some variables to the uses of the variable. This creates additional opportunities for constant expression folding. It also eliminates redundant variable copies.

Intraprocedural unreachable code and store elimination

IPA removes definitions of variables that cannot be reached, along with the computation feeding the definition.

Conversion of reference (address) arguments to value arguments

IPA converts reference (address) arguments to value arguments when the formal parameter is not written in the called procedure.

Conversion of static variables to automatic (stack) variables

IPA converts static variables to automatic (stack) variables when their use is limited to a single procedure invocation.

The execution time for code optimized using interprocedural analysis (IPA compile and link) is normally faster than for code optimized using intraprocedural analysis (IPA compile only) or the OPT compiler option. Please note that not all applications are suited for IPA optimization and the performance gains realized from using IPA will vary.

Note: For additional information about using the IPA(LINK) option, see [“IPA\(LINK\) option and exploitation of 64-bit virtual memory”](#) on page 227.

Program-directed feedback

IPA uses program-directed feedback (PDF) to organize the code and to focus optimization on the frequently-used portions of the code. This can result in significant performance gains.

PDF is an optimization the compiler applies to your application in two stages. The first stage collects information about your program as you run it with typical input data. The second stage applies transformations to your application based on that information. The compiler uses PDF to get information such as the locations of heavily used or infrequently used blocks of code. Knowing the relative execution frequency of code provides opportunities to bias execution paths in favor of heavily used code. PDF can perform program restructuring in order to ensure that infrequently-executed blocks of code are less likely to affect program path length or participate in instruction cache fetching.

It is important that the data sets PDF uses to collect information be characteristic of data your application will typically see. Using atypical data or insufficient data can lead to a faulty analysis of the program and suboptimal program transformation. If you do not have sufficient data, PDF optimization is not recommended.

The first step in PDF optimization is to compile and link your application with the IPA(PDF1) option. Doing so instruments your code with calls to a PDF runtime library that will link with your program. Then execute your application with typical input data as many times as you wish with as many data sets as you have. Each run records information in data files.

After you collect sufficient PDF data, recompile or simply relink your application with the IPA(PDF2) option. The compiler reads the PDF data files and makes the information available to all levels of optimization that are active. PDF optimization can be combined with other optimization techniques such as the standard OPT(2) or OPT(3) compilation unit optimizations, the HOT optimization, or the IPA optimization.

Stale profiling data can be used in the second stage of the PDF process, if minor changes are made to the source file in the first PDF stage. As a result of this change to the behavior of the PDF process, it will be possible to run both stages of PDF under different compilation options. The compiler will issue a list of warnings but will not terminate. During the link step, the compiler will print a message identifying any functions that have been detected to change from the first stage of PDF process.

PDF optimization is most effective when you apply it to applications that contain blocks of code that are infrequently and conditionally executed. Typical examples of this coding style include blocks of error-handling code and code that has been instrumented to conditionally collect debugging or statistical information.

Compiler processing flow

IPA changes the flow of compiler processing. The following sections explain the differences.

Regular compiler execution

If you specify the NOIPA compiler option (the default), the compiler processes source files, as shown in [Figure 120 on page 461](#). The output is an object module for each source file processed. You can then bind the object modules to produce an executable module.

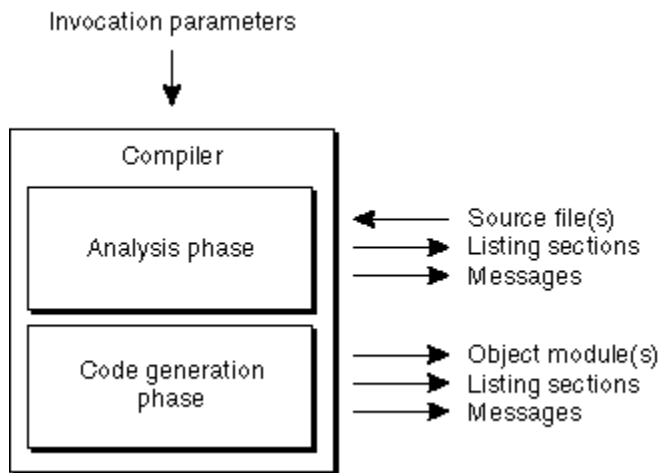


Figure 120. Flow of regular compiler processing

Compiler execution with IPA

IPA processing consists of two steps: IPA Compile and IPA Link. You run the IPA Compile step once for each compilation unit, and run the IPA Link step once for the program as a whole. The final output is a single IPA-optimized object module which you must bind with the binder to produce an executable load module.

Notes:

- If you want to get the maximum benefit from IPA, run both the IPA Compile and IPA Link steps.
- In z/OS UNIX shell environments, the c89 utility is required during the IPA Link step.

You can invoke the IPA Compile step in the same environments that you use for a regular compilation. You can invoke the IPA Link step only in MVS batch mode or in one of the z/OS UNIX shell environments through the c89 utility.

This information describes the flow of IPA processing under MVS batch. The flow of processing with the c89 utility is the same, but there are differences in how you invoke IPA.

IPA Compile step processing

You invoke the IPA Compile step by specifying the IPA(NOLINK) compiler option, as shown in [Figure 121 on page 462](#). (NOLINK is the default suboption). During the IPA Compile step, the compiler creates optimized objects. These objects contain information that the IPA Link step can use for further optimization.

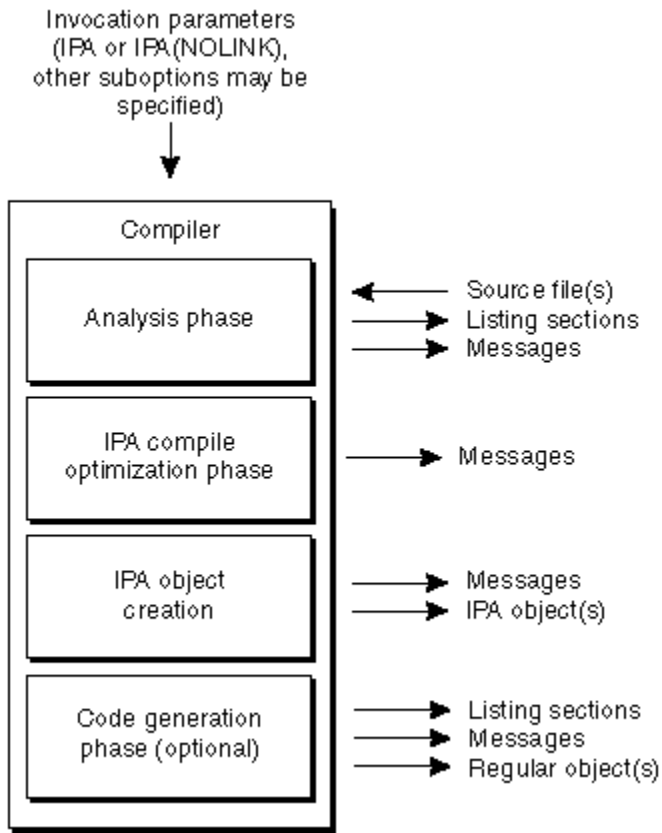


Figure 121. IPA compile step processing

The following processing takes place for each compilation unit that you specify for the IPA Compile step:

1. The compiler determines the final suboptions for the IPA option, based upon the compiler options and IPA suboptions that you specified. This is necessary because the compiler does not support some combinations of compiler options and IPA suboptions. The compiler issues a warning message if it finds unsupported combinations.
2. The compiler promotes some IPA suboptions based upon the presence of related compiler options and issues informational messages if it does so. For more information, see [IPA considerations in z/OS XL C/C++ User's Guide](#).
3. The compiler generates an IPA object file. This object file contains control information for a compilation unit required for the IPA Link step.

The IPA object module produced by IPA (NOLINK,NOOBJECT) has the same structure as a regular object module. It should not be used as input to the prelinker, linker, or binder.

Each IPA object contains a CSECT that includes the ESD name @@IPA0BJ.

4. If you specify the OBJECT suboption of the IPA option, the compiler produces a combined IPA and conventional object file. While the conventional object file is not required by the IPA Link step, creating it permits you to bind this file to create an executable module, without doing the IPA Link step, because it is more difficult to debug coded optimized by the interprocedural analysis.

During the IPA Compile step, the compiler generates information that allows you to create object libraries with the **C370LIB** utility or to create z/OS UNIX archives with the **ar** utility. The information consists of XSD and ESD records for the external symbols that were defined in the compilation units of your program. You can use the object libraries and z/OS UNIX archives for autocall searching in the IPA Link step. During autocall searching, the IPA Link step searches these libraries and archives for external references from your program.

IPA Link step processing

You invoke the IPA Link step by specifying the IPA(LINK) compiler option, as shown in Figure 122 on page 463. During this step, the compiler links the IPA objects that were produced by the IPA Compile step (along with non-IPA object files and load modules, if specified), does partitioning, performs optimizations, and generates the final object code.

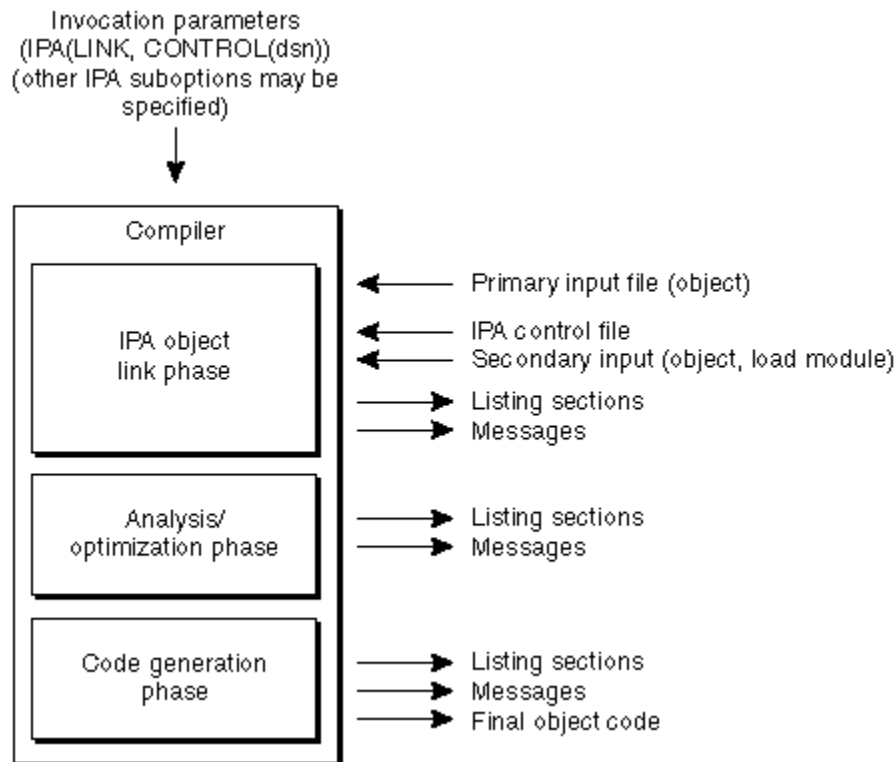


Figure 122. IPA link step processing

The following processing takes place:

1. The compiler determines the final suboptions for the IPA option, based upon the compiler options and IPA suboptions you specify. This is necessary because some combinations of compiler options and IPA suboptions are unsupported. The compiler issues informational and warning messages for unsupported combinations.
2. The compiler links IPA object files, as well as non-IPA object files and load modules (if specified). The compiler also merges information from the IPA Compile step.

Input for the Link step comes from one of three sources:

- The primary input file (specified by the SYSIN ddname). This can be either:
 - A set of IPA Link control statements that you create
These may be INCLUDE and LIBRARY IPA Link control statements that explicitly identify secondary input files. IPA uses the same control statement format (with some exceptions) used by the binder.
 - The IPA object file from the compilation unit that contains the main function or fetchable entry point. If you specify this file, the compiler searches for all other IPA files using the SYSLIB ddname.

- One or more secondary input files

The secondary input file may contain:

- IPA object files or PDS libraries
- Conventional object files or PDS libraries

- Load module libraries
- z/OS UNIX archive libraries
- IPA Link control statements

These secondary input files are to be used for autocall searches. You can specify these files through the SYSLIB ddname or explicitly include them through INCLUDE or LIBRARY IPA Link control statements on the IPA Link step.

Load module libraries are used to support library interface routines (such as CICS and Language Environment) that are implemented as load module libraries. Since IPA must resolve all parts of your application program before beginning optimization, make all of these libraries as well as your application object modules available to the IPA Link step.

The IPA Link step resolves external references using explicit and autocall resolution. This allows IPA to identify the static and global data and the external references for the whole program.

Ensure that you do not accidentally specify FB, LRECL 80 source files as input to the IPA Link step. The IPA Link step will assume that records from these files contain valid object information, and will retain them in the object file. When the linkage editor processes the object file, it will determine the records to be invalid, and will issue diagnostic messages.

- The IPA Link step control file. This file contains additional IPA control directives. The CONTROL suboption of the IPA compiler option identifies this file. For more information, see [IPA Link step control file](#) in *z/OS XL C/C++ User's Guide*.
3. As objects are processed, IPA Link Step builds the program call graph, merging the IPA object code according to its place in the call graph. If necessary, IPA Link Step stores non-IPA object code for inclusion in the final object file, and converts load module library members into object format for inclusion in the final object file.
 4. The compiler performs optimizations across the call graph. You specify the type and extent of optimizations using the LEVEL suboption of the IPA compiler option.
 5. IPA Link Step divides the program call graph into separate units called partitions. Partitioning of the call graph is controlled by:
 - The partition size limit that is specified in the IPA control file.
 - The connectivity of your program. IPA places code that is isolated from the rest of the program into a separate partition.
 - Resolution of conflicting effects between the compiler options and pragmas specified for compilation units processed during the IPA Compile step. These are the compiler options and pragmas that generate information during the analysis phase of the compiler for input to the code-generation phase.

IPA Link Step produces a final single object module for the program from these partitions.

You must bind the IPA single object module to produce the executable module.

Note: IPA Compile and IPA Link, as follows:

- An object file produced by an IPA Compile that contains IPA Object or combined IPA and conventional object information can be used as input to the z/OS XL C/C++ IPA Link of the same or later Version/Release.
- An object file produced by an IPA Compile that contains IPA Object or combined IPA and conventional object information cannot be used as input by the z/OS XL C/C++ IPA Link of an earlier Version/Release. If this is attempted, the IPA Link will issue an error diagnostic message.
- If the IPA object is recompiled by a later z/OS XL C/C++ IPA Compile, additional optimizations may be performed and the resulting application program may perform better.

An exception to this is the IPA object files produced by the OS/390 Release 2 C IPA Compile. These must be recompiled from the program source using a compiler that is version OS/390 Release 3 or later before attempting to process them with the current IPA Link.

Additional options that affect performance

The following topics describe compiler options that affect performance. For more information, see [Compiler options in z/OS XL C/C++ User's Guide](#).

ANSIALIAS

The ANSIALIAS option specifies whether type-based aliasing is to be used during optimization. Type-based aliasing will improve optimization. For more information about ANSI aliasing, see [“ANSI aliasing rules”](#) on page 429 and [“Using ANSI aliasing rules”](#) on page 431.

AGGRCOPY

The AGGRCOPY option specifies whether aggregate assignments might have overlapping source and target locations. AGGRCOPY(NOOVERL), which is the default, asserts to the compiler that the source and destination for structure and union assignments do not overlap. This assumption enables destructive copy operations for structures and unions, which can improve performance.

ASSERT(RESTRICT)

The ASSERT(RESTRICT) option enables optimizations for restrict qualified pointers.

COMPRESS

Use the COMPRESS option to suppress the generation of function names in the function control block to reduce the size of your application's load module. The amount of reduction depends on the average function size in the application, as compared to the length of the function name.

COMPACT

When the COMPACT option is active, the compiler favors optimizations that tend to limit the growth of the code. Depending on your specific program, the object size may increase or decrease and the execution time may increase or decrease. Any time you change your program, or change the release of the compiler, you should re-evaluate your use of the COMPACT option.

CVFT (C++ only)

Use the NOCVFT option to reduce the size of the writable static area for constructors that call virtual functions within the class hierarchy where virtual inheritance is used.

EXH (C++ only)

You might improve the run time of your C++ code by using NOEXH. The resultant code will run faster, but it will not be ISO-compliant if the program uses exception handling.

EXPORTALL

Use the EXPORTALL option only if you want to export all external functions and variables in the source file so that a DLL application can use them. If you only need to export some externally defined functions and variables, use the **#pragma export** directive or the `_Export` C++ keyword instead of EXPORTALL.

If you use EXPORTALL, you can severely limit IPA optimization, and can cause your modules and WSA to be larger than necessary.

FLOAT

Some of the FLOAT suboption provide precise control over the handling of floating-point calculations with binary floating-point numbers. Some of the most frequently used suboptions that affect performance when used with FLOAT(IEEE) are listed as follows:

FOLD

Enables compile time evaluation of floating-point calculations. You should disable folding only if your application must handle certain floating-point exceptions such as overflow or inexact. `FLOAT(FOLD)` is the default.

MAF

Makes floating-point calculations faster and more accurate by using floating-point multiply-add instructions where appropriate. However, the results might not be equivalent to those from similar calculations performed at compile time or on other types of computers.

NORRM

Allows the compiler to assume that the rounding mode is always round-to-nearest. `FLOAT(NORRM)` is the default.

HGPR

The HGPR option enables the compiler to exploit 64-bit General Purpose Registers (GPRs) in 32-bit programs targeting z/Architecture hardware.

IGNERRNO

The IGNERRNO option informs the compiler that the program is not using `errno`. This allows the compiler more freedom to explore optimization opportunities for certain library functions (for example, `sqrt`). You need to include the system header files to get the full benefit of the IGNERRNO option.

LANGLVL(NOCHECKPLACEMENTNEW)

The `LANGLVL(NOCHECKPLACEMENTNEW)` option instructs the compiler not to perform the null pointer check on the pointer that is returned by an invocation of the reserved forms of the placement operator `new` and operator `new[]`. This option is especially beneficial if the calls to placement operator `new` and operator `new[]` are inside loops or in functions which are called frequently.

LIBANSI

The LIBANSI option specifies whether or not all functions with the name of an ISO C library function are in fact the ISO C functions. This allows the compiler to generate code based on existing knowledge concerning the behavior of the function. For example, the compiler will determine whether any side effects are associated with a particular library function. LIBANSI can provide additional benefits when used in conjunction with IGNERRNO.

NOCHECKNEW

The NOCHECKNEW option can be used to remove the null pointer check performed on the pointer that is returned by an invocation of the throwing versions of operator `new` and operator `new[]`.

OBJECTMODEL (C++ only)

You can compile your programs using two different object models. They differ in the following areas:

- Layout for the virtual function table
- Name mangling scheme

The OBJECTMODEL compiler option has the following suboptions to set the type of object model:

CLASSIC

uses the original object model that was available on all previous releases of C++ compiler.

IBM

uses the new object model and should be selected if you want improved performance. This is especially true for class hierarchies with many virtual base classes. The size of the derived class is considerably smaller and access to the virtual function table is faster.

All classes in the same inheritance hierarchy must have the same object model.

Use the **#pragma object_model** directive to specify an object model in your source. For more information, see `object_model` in [z/OS XL C/C++ Language Reference](#).

PREFETCH

The PREFETCH option inserts prefetch instructions automatically where there are opportunities to improve code performance.

RESTRICT

The RESTRICT option indicates to the compiler that all pointer parameters in some or all functions are disjoint.

ROCONST

The ROCONST option specifies that the `const` qualifier is respected by the program. Variables that are defined with the `const` keyword are not overridden by a casting operation. When you use this option in C with the DLL option, you must ensure that `const` global variables (static or external) are initialized with the address of an entity from the compile unit that defines the variables.

ROSTRING

The ROSTRING option specifies that strings are placed in read-only memory. It has the same effect as the **#pragma strings(readonly)** directive.

RTTI

If you are not using RTTI/dynamic casts in your program, compile with the NORTTI option.

SPILL

When you specify a very large spill size, you can force the compiler to generate less than optimal code. For this reason, you might not want to specify the large spill size for an entire application. For example, either you can specify the large spill size for only the specific compilation unit that needs it or you can use the **#pragma option_override** directive.

STRICT

The STRICT option prevents optimizations done by default at optimization levels OPT(3), and optionally at OPT(2), from re-ordering instructions that could introduce rounding errors.

STRICT_INDUCTION

With strict induction, induction (loop counter) variables are not optimized. This guards against problems that can occur if an optimized induction variable overflows.

If it is certain that the induction variables will not overflow, use the NOSTRICT_INDUCTION option. This option can improve the performance of induction variables that are smaller than the register size on the processor.

THREADED

The THREADED option indicates to the compiler whether it must generate thread-safe code.

UNROLL

The UNROLL option gives the user the ability to control the amount of loop unrolling done by the compiler. Loop unrolling exposes instruction level parallelism for instruction scheduling and software pipelining and thus can improve a program's performance. It should be used in conjunction with [#pragma unroll](#).

VECTOR

The VECTOR option controls whether the compiler enables the vector programming support and automatically takes advantage of vector/SIMD instructions. The VECTOR option provides potential performance improvements in the following cases:

Binary floating-point data types: float and long double

When the VECTOR option is specified with ARCH(12) and FLOAT(IEEE), the long double and float data types can be processed in the vector registers.

Binary floating-point data type: double

When the VECTOR option is specified with ARCH(11) or higher and FLOAT(IEEE), the double data types can be processed in the vector registers.

Built-in library functions

When the VECTOR option is specified with ARCH(11) or higher, certain built-in library functions can take advantage of vector string instructions to accelerate the processing of strings of character data.

Fixed-point decimal operations

When the VECTOR option is specified with ARCH(12), the compiler can take advantage of vector decimal instructions that perform the operations in register operands, which can improve the performance of such applications.

SIMD instructions

When the AUTOSIMD suboption is in effect, the compiler generates code, when possible, using the SIMD instructions. SIMD instructions calculate several results at one time, which is faster than calculating each result sequentially.

Related information

- [VECTOR](#) in *z/OS XL C/C++ User's Guide*.

Chapter 38. Parallelizing your programs

The z/OS XL C/C++ compiler offers you the following method of implementing shared memory program parallelization:

- Explicit parallelization of C and C++ program code using pragma directives compliant to the OpenMP Application Program Interface specification. An overview of the OpenMP directives is provided in [“Using OpenMP directives”](#) on page 469.

Program parallelization is enabled when the **SMP** compiler option is in effect. The thread-safe version of system library routines should be used inside the parallel regions.

Parallel regions of program code are executed by multiple threads, possibly running on multiple processors. The number of threads created is determined by environment variables and calls to library functions. Work is distributed among available threads according to scheduling algorithms specified by the environment variables. If you are using OpenMP constructs, you can use the OpenMP environment variables to control thread scheduling.

For information about OpenMP runtime functions, see [“OpenMP runtime functions for parallel processing”](#) on page 471.

For detailed description of the OpenMP directives, see [Pragma directives for parallel processing in z/OS XL C/C++ Language Reference](#).

For information about OpenMP environment variables, see [Environment variables for OpenMP in z/OS XL C/C++ User's Guide](#).

For details about the OpenMP constructs, environment variables, and runtime routines, refer to the *OpenMP Application Program Interface Specification* at [OpenMP \(www.openmp.org\)](http://www.openmp.org).

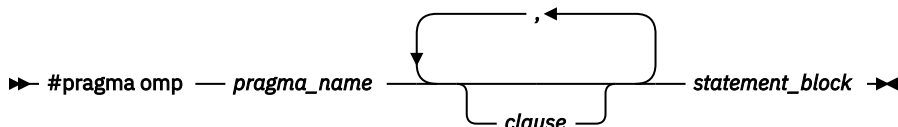
Using OpenMP directives

OpenMP directives exploit shared memory parallelism by defining various types of parallel regions. Parallel regions can include both iterative and non-iterative segments of program code.

The **#pragma omp** pragmas fall into the following general categories:

1. The **#pragma omp** pragmas for defining parallel regions in which work is done by threads in parallel (**#pragma omp parallel**). Most of the OpenMP directives either statically or dynamically bind to an enclosing parallel region.
2. The **#pragma omp** pragmas for defining how work is distributed or shared across the threads in a parallel region (**#pragma omp sections**, **#pragma omp for**, **#pragma omp single**, **#pragma omp task**).
3. The **#pragma omp** pragmas for controlling synchronization among threads (**#pragma omp atomic**, **#pragma omp master**, **#pragma omp barrier**, **#pragma omp critical**, **#pragma omp flush**, **#pragma omp ordered**).
4. The **#pragma omp** pragmas for defining the scope of data visibility across parallel regions within the same thread (**#pragma omp threadprivate**).
5. The **#pragma omp** pragmas for synchronization (**#pragma omp taskwait**, **#pragma omp barrier**).

OpenMP directive syntax



Including clauses in the **#pragma omp** pragmas can fine tune the behavior of the parallel or work-sharing regions. For example, a `num_threads` clause can be used to control a parallel region pragma.

The **#pragma omp** pragmas generally appear immediately before the section of code to which they apply. The following code defines a parallel region in which iterations of a for loop can run in parallel:

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<n; i++)
        ...
}
```

The following example defines a parallel region in which two or more non-iterative sections of program code can run in parallel:

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        structured_block_1
        ...
        #pragma omp section
        structured_block_2
        ...
    }
}
```

For detailed description of the OpenMP directives, see [Pragma directives for parallel processing in z/OS XL C/C++ Language Reference](#).

Shared and private variables in a parallel environment

Variables can have either shared or private context in a parallel environment. Variables in shared context are visible to all threads running in associated parallel regions. Variables in private context are hidden from other threads. Each thread has its own private copy of the variable, and modifications made by a thread to its copy are not visible to other threads.

The default context of a variable is determined by the following rules:

- Variables with static storage duration are shared.
- Dynamically allocated objects are shared.
- Variables with automatic storage duration that are declared in a parallel region are private.
- Variables in heap allocated memory are shared. There can be only one shared heap.
- All variables defined outside a parallel construct become shared when the parallel region is encountered.
- Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
- Memory allocated within a parallel loop by the `alloca` function persists only for the duration of one iteration of that loop, and is private for each thread.

The following code segments show examples of these default rules:

```
int E1;                                /* shared static */

void main (argc,...) {                 /* argc is shared */
    int i;                             /* shared automatic */

    void *p = malloc(...);             /* memory allocated by malloc */
                                        /* is accessible by all threads */
                                        /* and cannot be privatized */

    #pragma omp parallel firstprivate (p)
    {
        int b;                         /* private automatic */
    }
```

```

static int s;                /* shared static */

#pragma omp for
for (i =0;...) {
    b = 1;                   /* b is still private here ! */
    foo (i);                 /* i is private here because it */
                             /* is an iteration variable */
}

#pragma omp parallel
{
    b = 1;                   /* b is shared here because it */
                             /* is another parallel region */
}

int E2;                      /*shared static */

void foo (int x) {           /* x is private for the parallel */
                             /* region it was called from */

int c;                       /* the same */
... }

```

Some OpenMP clauses enable you to specify visibility context for selected data variables. A brief summary of data scope attribute clauses are listed below:

Data scope attribute clause	Description
private	The private clause declares the variables in the list to be private to each thread in a team.
firstprivate	The firstprivate clause provides a superset of the functionality provided by the private clause. The private variable is initialized by the original value of the variable when the parallel construct is encountered.
lastprivate	The lastprivate clause provides a superset of the functionality provided by the private clause. The private variable is updated after the end of the parallel construct.
shared	The shared clause declares the variables in the list to be shared among all the threads in a team. All threads within a team access the same storage area for shared variables.
reduction	The reduction clause performs a reduction on the scalar variables that appear in the list, with a specified operator.
default	The default clause allows the user to affect the data-sharing attribute of the variables appeared in the parallel construct.

For more information, you can also refer to the *OpenMP Application Program Interface Language Specification*, which is available at [OpenMP \(www.openmp.org\)](http://www.openmp.org).

OpenMP runtime functions for parallel processing

Function definitions for the `omp_` functions can be found in the `omp.h` header file.

For complete information about OpenMP runtime library functions, refer to the OpenMP Application Program Interface specification at [OpenMP \(www.openmp.org\)](http://www.openmp.org).

omp_destroy_lock, omp_destroy_nest_lock

Purpose

Ensures that the specified lock variable *lock* is uninitialized.

Prototype

```
void omp_destroy_lock (omp_lock_t *lock);  
void omp_destroy_nest_lock (omp_nest_lock_t *lock);
```

Parameter

lock

Must be a variable of type `omp_lock_t` that is initialized with `omp_init_lock` or `omp_init_nest_lock`.

omp_get_active_level

Purpose

Returns the number of nested, active parallel regions enclosing the task that contains the call. The routine always returns a nonnegative integer, and returns 0 if it is called from the sequential part of the program.

Prototype

```
int omp_get_active_level(void);
```

omp_get_ancestor_thread_num

Purpose

Returns the thread number of the ancestor of the current thread at a given nested level. Returns -1 if the nested level is not within the range of 0 and the current thread's nested level as returned by `omp_get_level`.

Prototype

```
int omp_get_ancestor_thread_num(int level);
```

Parameter

level

Specifies a given nested level of the current thread.

omp_get_dynamic

Purpose

Returns non-zero if dynamic thread adjustment is enabled and returns 0 otherwise.

Prototype

```
int omp_get_dynamic (void);
```


omp_get_level

Purpose

Returns the number of active and inactive nested parallel regions that the generating task is executing in. This does not include the implicit parallel region. Returns 0 if it is called from the sequential part of the program. Otherwise, returns a nonnegative integer.

Prototype

```
int omp_get_level(void);
```

omp_get_max_active_levels

Purpose

Returns the value of the *max-active-levels-var* internal control variable that determines the maximum number of nested active parallel regions. *max-active-levels-var* can be set with the `OMP_MAX_ACTIVE_LEVELS` environment variable or the **omp_set_max_active_levels** runtime routine.

Prototype

```
int omp_get_max_active_levels(void);
```

omp_get_max_threads

Purpose

Returns the first value of *num_list* for the `OMP_NUM_THREADS` environment variable. This value is the maximum number of threads that can be used to form a new team if a parallel region without a **num_threads** clause is encountered.

Prototype

```
int omp_get_max_threads (void);
```

omp_get_nested

Purpose

Returns non-zero if nested parallelism is enabled and 0 if it is disabled.

Prototype

```
int omp_get_nested (void);
```

omp_get_num_procs

Purpose

Returns the maximum number of processors that could be assigned to the program.

Prototype

```
int omp_get_num_procs (void);
```

omp_get_num_threads

Purpose

Returns the number of threads currently in the team executing the parallel region from which it is called.

Prototype

```
int omp_get_num_threads (void);
```

omp_get_schedule

Purpose

Returns the *run-sched-var* internal control variable of the team that is processing the parallel region. The argument *kind* returns the type of schedule that will be used. *modifier* represents the chunk size that is set for applicable schedule types. *run-sched-var* can be set with the *OMP_SCHEDULE* environment variable or the **omp_set_schedule** function.

Prototype

```
int omp_get_schedule(omp_sched_t  
* kind, int * modifier);
```

Parameters

kind

The value returned for *kind* is one of the schedule types affinity, auto, dynamic, guided, runtime, or static.

modifier

For the schedule type dynamic, guided, or static, *modifier* is the chunk size that is set. For the schedule type auto, *modifier* has no meaning.

omp_get_team_size

Purpose

Returns the thread team size that the ancestor or the current thread belongs to. **omp_get_team_size** returns -1 if the nested level is not within the range of 0 and the current thread's nested level as returned by **omp_get_level**.

Prototype

```
int omp_get_team_size(int level);
```

Parameter

level

Specifies a given nested level of the current thread.

omp_get_thread_limit

Purpose

Returns the maximum number of OpenMP threads available to the program. The value is stored in the *thread-limit-var* internal control variable. *thread-limit-var* can be set with the *OMP_THREAD_LIMIT* environment variable.

Prototype

```
int omp_get_thread_limit(void);
```

omp_get_thread_num

Purpose

Returns the thread number, within its team, of the thread executing the function.

Prototype

```
int omp_get_thread_num (void);
```

Return value

The thread number lies between 0 and `omp_get_num_threads()-1`, inclusive. The primary thread of the team is thread 0.

omp_get_wtick

Purpose

Returns the number of seconds between clock ticks.

Prototype

```
double omp_get_wtick (void);
```

Usage

The value of the fixed starting time is determined at the start of the current program, and remains constant throughout program execution.

omp_get_wtime

Purpose

Returns the time elapsed from a fixed starting time.

Prototype

```
double omp_get_wtime (void);
```

Usage

The value of the fixed starting time is determined at the start of the current program, and remains constant throughout program execution.

omp_in_final

Purpose

Returns a nonzero integer value if the function is called in a final task region; otherwise, it returns 0.

Prototype

```
int omp_in_final(void);
```

omp_in_parallel

Purpose

Returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, returns 0.

Prototype

```
int omp_in_parallel (void);
```

omp_init_lock, omp_init_nest_lock

Purpose

Initializes the lock associated with the parameter *lock* for use in subsequent calls.

Prototype

```
void omp_init_lock (omp_lock_t *lock);  
void omp_init_nest_lock (omp_nest_lock_t *lock);
```

Parameter

lock

Must be a variable of type `omp_lock_t`.

omp_set_dynamic

Purpose

Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.

Prototype

```
void omp_set_dynamic (int dynamic_threads);
```

Parameter

dynamic_threads

Indicates whether the number of threads available in subsequent parallel region can be adjusted by the runtime. If *dynamic_threads* is nonzero, the runtime can adjust the number of threads. If *dynamic_threads* is zero, the runtime cannot dynamically adjust the number of threads.

omp_set_lock, omp_set_nest_lock

Purpose

Blocks the thread executing the function until the specified lock is available and then sets the lock.

Prototype

```
void omp_set_lock (omp_lock_t * lock);  
void omp_set_nest_lock (omp_nest_lock_t * lock);
```

Parameter

lock

Must be a variable of type `omp_lock_t` that is initialized with `omp_init_lock` or `omp_init_nest_lock`.

Usage

A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function.

omp_set_max_active_levels

Purpose

Sets the value of the *max-active-levels-var* internal control variable to the value in the argument. If the number of parallel levels requested exceeds the number of the supported levels of parallelism, the value of *max-active-levels-var* is set to the number of parallel levels supported by the runtime. If the number of parallel levels requested is not a positive integer, this routine call is ignored.

When nested parallelism is turned off, this routine has no effect and the value of *max-active-levels-var* remains 1. *max-active-levels-var* can also be set with the `OMP_MAX_ACTIVE_LEVELS` environment variable. To retrieve the value for *max-active-levels-var*, use the **omp_get_max_active_levels** function.

Use `omp_set_max_active_levels` only in serial regions of a program. This routine has no effect in parallel regions of a program.

Prototype

```
void omp_set_max_active_levels(int max_levels);
```

Parameter

max_levels

An integer that specifies the maximum number of nested, active parallel regions.

omp_set_nested

Purpose

Enables or disables nested parallelism.

Prototype

```
void omp_set_nested (int nested);
```

Usage

If the argument to `omp_set_nested` evaluates to true, nested parallelism is enabled for the current task; otherwise, nested parallelism is disabled for the current task. The setting of `omp_set_nested` overrides the setting of the `OMP_NESTED` environment variable.

Note: If the number of threads from all regions exceeds the number of available processors, your program might suffer performance degradation.

omp_set_num_threads

Purpose

Overrides the setting of the `OMP_NUM_THREADS` environment variable, and specifies the number of threads to use for a subsequent parallel region by setting the first value of *num_list* for `OMP_NUM_THREADS`.

Prototype

```
void omp_set_num_threads (int num_threads);
```

Parameters

num_threads

Must be a positive integer.

Usage

If the `num_threads` clause is present, then for the parallel region it is applied to, it supersedes the number of threads requested by this function or the `OMP_NUM_THREADS` environment variable. Subsequent parallel regions are not affected by it.

omp_set_schedule

Purpose

Sets the value of the *run-sched-var* internal control variable. Use **omp_set_schedule** if you want to set the schedule type separately from the `OMP_SCHEDULE` environment variable.

Prototype

```
void omp_set_schedule (omp_sched_t kind,  
int modifier);
```

Parameters

kind

Must be one of the schedule types affinity, auto, dynamic, guided, runtime, or static.

modifier

For the schedule type dynamic, guided, or static, *modifier* is the chunk size that you want to set.

Generally it is a positive integer. If the value is less than one, the default will be used. For the schedule type auto, *modifier* has no meaning.

Related reference

[“omp_get_schedule” on page 474](#)

omp_test_lock, omp_test_nest_lock

Purpose

Attempts to set a lock but does not block execution of the thread.

Prototype

```
int omp_test_lock (omp_lock_t * lock);  
int omp_test_nest_lock (omp_nest_lock_t * lock);
```

Parameter

lock

Must be a variable of type `omp_lock_t` that is initialized with `omp_init_lock` or `omp_init_nest_lock`.

omp_unset_lock, omp_unset_nest_lock

Purpose

Releases ownership of a lock.

Prototype

```
void omp_unset_lock (omp_lock_t * lock);  
void omp_unset_nest_lock (omp_nest_lock_t * lock);
```

Parameter

lock

Must be a variable of type `omp_lock_t` that is initialized with `omp_init_lock` or `omp_init_nest_lock`.

Chapter 39. Optimizing the system and Language Environment

This chapter gives some basic tips for tuning Language Environment for optimal C/C++ performance, and some basic system setup tips for efficient program execution.

Improving the performance of the Language Environment

This section discusses how to increase the performance of an application by:

- [“Storing libraries and modules in system memory” on page 481](#)
- [“Optimizing memory and storage” on page 481](#)
- [“Optimizing runtime options” on page 482](#)

Storing libraries and modules in system memory

One way to boost performance is to load common or reusable modules into memory. For example, placing the Language Environment Library in a link pack area (LPA) can increase the performance of your entire system. This is recommended if your z/OS system contains many applications that use the Language Environment Library, or is a heavy user of z/OS UNIX. LPAs store reentrant routines from system libraries. This saves loading time when a reentrant routine is needed. Individual modules can also be loaded into a single LIBPACK, in order to reduce the time that would otherwise be needed to load the individual load modules. For instructions for placing Language Environment Modules in Link Pack and LIBPACK, see [z/OS Language Environment Customization](#).

If LPAs or LIBPACKS do not have enough space for the Language Environment Library, then you can place it into a library lookaside (LLA). This reduces library I/O activity by keeping selected directory entries in storage.

Similarly, if your application uses C++ class libraries, then application performance may be increased by placing specific libraries in the LPA or the dynamic link pack area (DLPA). For example:

- If the application is a heavy user of the ISO C++ Standard Libraries, then place the 31-bit CEE.SCEERUN2(C128) or 64-bit CEE.SCEERUN2(C64) Language Environment runtime library in the DLPA.
- If the application is using the non-XPLINK C++ standard library, then place the CEE.SCEERUN(C128N) Language Environment runtime library in an LPA.
- If the application is a heavy user of the USL IOSTREAM libraries, then place the CBC.SCLBDLL Language Environment runtime library in an LPA (for non-XPLINK applications) or the CBC.SCLBDLL2 Language Environment runtime library in a DLPA (for XPLINK applications).

Optimizing memory and storage

Memory allocations can significantly affect the performance of your application. You can optimize your runtime space requirements by using the following Language Environment runtime options:

ANYHEAP	BELOWHEAP	HEAP	HEAPPOOLS
LIBSTACK	STACK	STORAGE	THREADSTACK
HEAP64	HEAPPOOLS64	STACK64	THREADSTACK64

Stack extensions can also cause significant performance hits. For this reason:

- The STACK/STACK64 specified should be large enough to ensure that a stack extension never occurs during the execution of the program.

- The HEAP/HEAP64 should be large enough for an average application execution run, and the increment size should be a reasonable portion of the difference between the typical heap used and the maximum amount of heap that may be used.
- Use the RPTSTG(ON) Language Environment runtime option or the `__heaprpt()` function to determine the storage usage and the option settings for the given run of your application. The generated report will show if the ANYHEAP, BELOWHEAP, LIBSTACK, and THREADSTACK/THREADSTACK64 are set to the recommended values. The STACK/STACK64 and HEAP/HEAP64 defaults should be as specified above.

The RPTSTG(ON) option should not be used in the final build or run because it is resource-intensive, which adversely affects the performance of the application. The `__heaprpt()` function, which does not require the RPTSTG(ON) option, obtains a summary heap storage report while your application is running. For more information, see `__heaprpt()` in *z/OS C/C++ Runtime Library Reference*.

You can also tune I/O storage by using the `_EDC_STOR_INITIAL` and `_EDC_STOR_INCREMENT` environment variables. The I/O storage usage is not in the storage report.

For more information about runtime storage, see Stack and heap storage in *z/OS Language Environment Programming Guide*.

Optimizing runtime options

In addition to the memory options, the ALL31 and HEAPP00LS runtime options can improve the performance of your application. ALL31 indicates that a Language Environment application has a 31-bit addressing mode. The Language Environment default is ALL31(ON). If your application has some AMODE 24 components, you will need to run the application with ALL31(OFF), but will lose some performance.

The HEAPP00LS runtime option might increase storage use, but will improve the performance of the application. This option is effective if:

- The application is multi-threaded
- The application often uses:
 - `new()`
 - `delete()`
 - `new[]()`
 - `delete[]()`
 - `malloc()`
 - `__malloc31()`
 - `realloc()`
 - `calloc()`
 - `free()`
 - `aligned_alloc()`

Note: If you are not sure which settings of ALL31 and HEAPP00LS are in effect, use the Language Environment runtime option RPTOPTS. RPTOPTS(ON) generates a report of runtime options and their settings that are in use by the currently-running application. Because this option diminishes the performance of the application, it should be used for diagnosis purposes only.

Tuning the system for efficient execution

This section is a quick overview of a ways to preload modules, DLLs, files, and directories into z/OS. In general, preloading reduces overhead and memory cost. For more detailed information, see the following documents:

- *z/OS UNIX System Services Planning*
- *z/OS MVS Initialization and Tuning Guide*

Link pack areas

It is recommended that you preload items that are either critical or frequently used into the link pack area (LPA). For batch and z/OS UNIX tasks, use LPA for modules or dynamic LPA for program objects. If LPA is not an option due to system requirements, then consider putting the module into LLA.

IMS and CICS both have similar methods to allow you to preload a frequently used module.

Library lookasides

The library lookaside facility (LLA) reduces the amount of I/O activity necessary to locate and fetch modules and program objects from storage. In addition, LLA can work with virtual lookasides to quickly fetch modules from virtual storage instead of from a direct access storage device (DASD).

Virtual lookasides

The virtual lookaside facility (VLF) is used to cache various items to reduce I/O, reduce CPU time, and increase response time. For example, you can cache the user IDs (UIDs) and group IDs (GIDs), which will reduce the DASD I/O overhead for Resource Access Control Facility (RACF) calls.

Chapter 40. Balancing compilation time and application performance

Compilation time increases as the level of optimization increases. An end user requires that an application run as fast as possible, and therefore will compile with the maximum optimization possible. Conversely, a developer rebuilds an application many times while debugging a problem, and therefore will compile with the minimum optimization possible. In addition, a developer might need to implement debugging tools, or activate extra debugging code, both of which would affect the performance of the application. This information discusses how to determine the proper balance between compilation time and application performance.

General tips

The following list contains suggestions to support your efforts to debug programs, and reduce compilation time, and improve application performance.

- All builds for testing or production should be compiled with the optimization level at which you intend to ship the final product.
- Even if you compile with `opt (0)` and debug on a regular basis, you should also do some testing at higher optimization levels to ensure that no aliasing rules or ISO C/C++ rules have been broken, which would cause the code to be optimized incorrectly.
- You can ensure the cleanest possible optimized compilations, as well as reduce the number of bugs that occur only at high optimization levels, by reviewing every warning issued by the compiler.

Note: Warnings are often a sign that the compiler is not sure how to interpret the code. If the compiler is not sure how to interpret code at `Opt (0)`, the code could cause an error at higher optimization levels or contribute to longer compilation times.

- The simpler the code is, the more easily the compiler can understand it and the faster it will compile. For more information, see [Chapter 34, “Improving program performance,”](#) on page 427.
- The `CHECKOUT` (for C) or `INFO` (for C++) option can be used to look for certain common problems (such as unprototyped functions and uninitialized variables) that can increase both compilation time and execution time.
- Generate production builds each week throughout the project cycle. This makes it easier to determine when problems entered the code base. Waiting until the end of a cycle to generate a build with high optimization can make it more difficult to find errors caused by coding that does not conform to ANSI aliasing rules.
- Set up a build so that you can customize options for any source file, if necessary. For example, use a makefile for a UNIX System Services-based build with a default rule for compilation. You can then customize targets for source files that require different options. Similarly, use the `OPTFILE` compiler option for a JCL-based build. A build script can then use a project-level option file for all source files in a module or DLL. You can specify either of the following:
 - Both a project-level option file and additional specific options for a source file
 - A source-specific option file in the option list that follows the options file name
- Set up build scripts so that they can be used for both development and production builds to:
 - Eliminate a common source of errors (because it is necessary to update only one build environment)
 - Make it easier to reproduce and debug problems that occur only in the development build
 - Minimize occurrences of bugs that are reproducible only in the development build

Programmer tips

- You can add code to the beginning and end of a header file to ensure that it is not processed unnecessarily during compilation.

The following example contains code that is included in a header file called myheader.

```
#ifndef __myheader
#ifdef __COMPILER_VER
#pragma filetag ("IBM-1047")
#endif
#define __myheader 1
/* header file contents */
#endif
```

You must ensure that the `filetag` statement, if used, appears before the first statement or directive (except for any conditional compilation directives). The `ifndef` statement is the first non-comment statement in the header file (the actual token used after the `ifndef` statement is your choice). The `define` statement must follow; it cannot appear before the `filetag` statement, but it must appear before any other preprocessor statement (other than comments).

Note that the header can contain comment statements in any location. Using this format of header-file blocking will improve compilation time for programs where a header file is included more than once.

- Use the system header files from UNIX file system instead of partitioned data sets to improve compilation time. Specify the following compiler options to do this:

For C++

```
NOSEARCH SEARCH('/usr/include/', '/usr/lpp/cbclib/include/')
```

For C

```
NOSEARCH SEARCH('/usr/include/')
```

- With the MEMORY compiler option (the default), the compiler uses a hiperspace or memory file in place of a work file (if possible). This option increases compilation speed, but you might require additional memory to use it. If the compilation fails because of a storage error, either increase your storage size or recompile your program using the NOMEMORY option.
- If your file has many recursive template definitions and you want to use the TEMPINC option, the FASTTEMPINC compiler option might reduce the compilation time.

Note: This option defers generating object code until the final versions of all template definitions have been determined. Then, a single compilation pass generates the final object code. Time is not wasted generating object code that will be discarded and generated again.

If your application has very few recursive template definitions, NOFASTTEMPINC might be faster than FASTTEMPINC.

- If you want to achieve a good balance of compilation time and small modules that execute quickly, consider using the TEMPLATEREGISTRY option instead of TEMPINC or NOTEMPINC.
- If a source file does not have try/catch blocks or does not throw objects, then the NOEXH C++ compiler option may improve the compilation time. The resultant code will not be ISO C/C++-compliant if the program uses exception handling.
- If you want to improve your OPT compilation time at the expense of runtime performance, you can specify:

MAXMEM

Limits the amount of memory used for local tables of specific memory intensive optimizations. If this amount of memory is insufficient for a particular optimization, the compiler performs somewhat poorer optimization and issues a warning message. Reducing the MAXMEM value from 2G to 10M may disable some optimizations, which may cause some decrease in execution performance.

NOINLINE

Disables inlining, which might decrease the compilation time. There might also be a corresponding increase in execution time.

System programmer tips

- If you do a lot of application development on your machine, put the compiler and runtime library in the LPA. Similarly, if you are working in z/OS UNIX System Services also put the c89/cxx/cc utilities in the dynamic LPA, LPA or linklist.
- Use packs that are cached with DASD fast write.

If you are working in z/OS UNIX System Services, give each user a separate mountable file system to avoid I/O contention.

If the compiler is not in LPA, tune your jobs to avoid channel and pack contention when the headers and the compiler are on the same pack and multiple compile jobs are executing.

- If you use the makedepend utility to generate dependency information, use the LIST option to generate a listing from makedepend. The summary section of this listing shows a list of the most frequently called headers and the frequency of these calls. Use this information to determine which headers should be cached.
- You can define /tmp as a RAM disk by specifying:

```
FILESYSTYPE TYPE(TFS) ENTRYPPOINT(BPXTFS)
```

This is described in more detail in [*z/OS UNIX System Services Planning*](#).

Chapter 41. Stepping through optimized code using the dbx debugger utility

Starting from z/OS V2R1, debug information can be generated for optimized code. The debugger can do normal debugging with code that is compiled with the OPTIMIZE(2) and DEBUG(LEVEL(8)) compiler options.

However, the debug information generated for code compiled with OPTIMIZE(3) and higher levels is limited. You can use the dbx debugger utility to help you determine problems. One method is to set the stop location at the point where your program detects an error situation, or detects a severe condition, which the code cannot handle. This method has the following limitations:

- The point at which you want to allow dbx to take control must be determined prior to compile time.
- You can use the dbx **stepi** subcommand to step through the code at the instruction level only.
- No source or symbolic debug information is available, which means that the debugger cannot execute any instructions that require debug information, such as relating the execution to the source file or examining the values of variables.

For more information on dbx, and the dbx **stop** and **stepi** subcommands, refer to *z/OS UNIX System Services Command Reference*. For information on the OPTIMIZE, NOOPTIMIZE, and DEBUG compiler options, see *z/OS XL C/C++ User's Guide*

Steps for setting up a stopping point for dbx in optimized code

Perform the following steps to set up a stopping point for dbx in optimized code:

1. Create a source file named `a.c`, with a function `func()` defined in it.
2. At any point in the application (in any file, in any function compiled with OPTIMIZE), insert `func()`; where you want to allow the debugger to take control.

```
main.c
main() {    /* or insert anywhere in the application, */
            /* in any file, in any function compiled at OPT. */
    ...
    func();
    ...
}
```

3. Compile `main.c` with the OPTIMIZE compiler option.
4. Compile `a.c` with the DEBUG and NOOPTIMIZE compiler options.
5. Relink the application, together with the newly created `a.o` object file.

Steps for setting up a stopping point for dbx in optimized code

Perform the following steps to use dbx to step through optimized code:

1. Load the `a.out` file into the dbx utility (`a.out` is the default name of an executable file produced by the compiler.)
2. Use the dbx **stop** subcommand to stop in `func()`.
3. Use the dbx **run** or **continue** subcommand to resume dbx so that it can hit the entry breakpoint for `func()`.
4. Use the dbx **stepi** subcommand to return to the point in the original source where the call to `func()` was inserted.

For more information on dbx, and the dbx stop and stepi subcommands, see [z/OS UNIX System Services Command Reference](#). For information on the OPTIMIZE, NOOPTIMIZE, and DEBUG compiler options, see [z/OS XL C/C++ User's Guide](#).

Part 6. z/OS XL C/C++ environments

This part describes the different z/OS XL C/C++ environments. Note that the MultiTasking Facility and the System Programming C Facilities are not available for z/OS XL C++. If you attempt to run an SPC application under z/OS XL C++, it will abend.

- [Chapter 42, “Using the system programming C facilities,” on page 493](#)
- [Chapter 43, “Library functions for system programming C,” on page 531](#)
- [Chapter 44, “Using runtime user exits,” on page 537](#)
- [Chapter 45, “Using the z/OS XL C MultiTasking Facility,” on page 555](#)

Chapter 42. Using the system programming C facilities

This chapter explains how to use the system programming C (SPC) facilities with z/OS XL C.

Notes:

1. Using the system programming C facilities, by programs which have been compiled with z/OS XL C++ is not supported.
2. IPA is not supported in an SPC environment unless there is a `main()` function present.
3. XPLINK is not supported by the SPC facilities.
4. AMODE 64 applications are not supported by the SPC facilities.

When z/OS XL C applications are compiled, many routines are needed to support the z/OS XL C environment that are not included in your executable. These routines, which are in z/OS Language Environment, are dynamically loaded at run time. This reduces the size of the program to its practical minimum and provides for the sharing of z/OS XL C library code by allowing its placement in Extended Link Pack Areas.

z/OS Language Environment provides facilities to set up the environment, handle termination, provide storage management, error handling, interlanguage calls and debugging support. Also, the C library functions are provided with z/OS Language Environment. In situations where not all of these services are needed or available, or more control over the executive environment is required, the system programming C facilities can provide a reduced customizable environment for your application.

System programming facilities enable you to run applications without z/OS Language Environment or with just the z/OS XL C library functions available. You can:

- Use a subset of the C language to develop specialized applications that do not require z/OS Language Environment on the machines where the application will run.

You can write freestanding applications that:

- Do not use the dynamic runtime library.
- Use only the C-specific library functions without any z/OS Language Environment facilities to manage the execution environment.

For example, a system programming application could use the C-specific library function `printf()` but not have the common run time initialize the environment. The system programming facilities would handle initialization. For more information on this type of application, see [“Creating freestanding applications” on page 496](#).

- Use z/OS XL C as an assembler language alternative, such as for writing exit routines for MVS, TSO, or JES.

For more information on this type of application, see [“Creating system exit routines” on page 501](#).

- Develop applications featuring a persistent C environment, where a z/OS XL C environment is created once and used repeatedly for C function execution.

For more information on this type of application, see [“Creating and using persistent C environments” on page 505](#).

- Develop co-routines using a two-stack model, as used in client-server style applications. In this style, the user application calls upon the applications server to perform services independently of the user and then returns to the user.

For more information on this type of application, see [“Developing services in the service routine environment” on page 509](#).

Note: Using the decimal data type and its related functions (`decabs()`, `decchk()`, and `decfix()`) without z/OS Language Environment is not supported.

Using functions in the system programming C environment

If you do not want to use the z/OS Language Environment runtime library and the z/OS XL C runtime component within z/OS Language Environment the following functions are available in the SPC environment:

- The following library functions are available as built-in so that they can be used without the runtime library:

Function Type	Function Name
Mathematical	<code>abs()</code> , <code>fabs()</code>
Memory manipulation	<code>memchr()</code> , <code>memcmp()</code> , <code>memcpy()</code> , <code>memset()</code> , <code>cds()</code> , <code>cs()</code>
String operations	<code>strcat()</code> , <code>strchr()</code> , <code>strcmp()</code> , <code>strcpy()</code> , <code>strlen()</code> , <code>strrchr()</code>
Wide character memory manipulation	<code>wmemchr()</code> , <code>wmemcmp()</code> , <code>wmemcpy()</code> , <code>wmemset()</code>

The built-in versions of these functions are available only if the appropriate header file (`string.h`, `wchar.h`, `math.h`, or `stdlib.h`) is included in the source file. The use of these functions is described in [z/OS C/C++ Runtime Library Reference](#).

- The memory management functions, including complete support for:
 - The `malloc()` function
 - The `calloc()` function
 - The `realloc()` function
 - The `free()` function
 - The HEAP runtime option
- The `exit()` function
- The `sprintf()` function.

Note: The use of floating point conversion specifiers (e,E,f,g or G) is not supported without the Language Environment runtime.

Additional memory management functions are available in the system programming C environment, as follows:

__4kmalc()
to allocate page-aligned storage

__24malc()
to allocate storage below the 16MB line in ESA systems (where MB is 1048576 bytes) even when HEAP (ANYWHERE) is specified.

Storage allocated by these functions is not part of the heap, so freeing it is your responsibility. You can use the `free()` function to free the storage before the environment is terminated. Storage allocated using these functions is not automatically freed when the environment is terminated.

In this environment, low-level memory management functions and contents supervision (loading and deleting executable code) are supported by low-level routines that you can replace to support non-standard environments. This is described in [“Tailoring the system programming C environment”](#) on page 524.

System programming C facility considerations and restrictions

When using any system programming C environment, consider the following:

- The long long data type is not supported for the function `sprintf()` under SPC. If you need to use the long long data type, you must use the C/C++ Runtime library version of the `sprintf()` function.).
- The `fetch()` function is not supported when you are running in a system programming C environment. You can use the EDCXLOAD routine, as described in [“Loading a module” on page 526](#), to simulate some of the functionality of the `fetch()` function.
- The IMS parameter list established by the `#pragma runopts(PLIST(IMS))` directive is not supported in any of the system programming environments. However, this does not preclude the use of IMS within these environments, because the registers upon entry are available using the `__xregs()` function and `ctdli()` is bound statically. For more information on `__xregs()`, refer to [“__xregs\(\) — Get Registers on Entry” on page 533](#).
- Interlanguage calls to COBOL and PL/I are not supported. However, an SPC program can use the `system()` function to call modules written in other languages.
- SPC is not supported under CICS or MTF.
- Library functions for use with UNIX file system I/O are not supported under SPC. Calling them causes unpredictable results.
- All runtime options are ignored except for:
 - STACK
 - HEAP
 - TRAP
- Redirection of standard streams is not supported.
- The default initial stack size is the minimum size required to start the C program. (This default is different from the non-systems programming C environments.) If a size is specified, that actual value is used, provided it is large enough. If the value specified is smaller than the requirements for the program, the required value is used.
- The default value for the HEAP runtime option is `HEAP(12K,4K,ANY,FREE)`.
- When you are running a service routine, you should with `#pragma runopts(TRAP(OFF))`.
- Exception handling is not supported in a persistent environment.
- Invoking the `system()` function from an `atexit()` function results in undefined behavior.
- When using the `atexit()` function from a persistent environment, the `atexit` list will not be run until the persistent environment has been terminated by the `__xhott()` library function. For more information about this function, see [“__xhott\(\) — Terminate a Persistent C Environment” on page 532](#).
- Calls to math library functions can be made in a system programming C environment using the dynamic library. For the most efficient use of calls to math library functions, you should enclose the function name in parentheses `()`. For example, if you make a call to `sin()`, use:

```
z = (sin)(x);
```

- You cannot call `ctrace()`, `csnap()`, `cdump()`, or `ctest()` because they rely on z/OS Language Environment callable services.
- System programming C environments are disjointed from each other; that is, memory files cannot be passed and file control is not maintained across environments. Thus, memory files cannot be passed between a C program and a callee that is written as an assembler exit.

An exception is between environments where the target environment is built with EDCXSTRL or EDCXSTRX but does not represent a server. For example, if a C program invokes a freestanding SPC application that is not a server by using `system()`, a memory file can be passed successfully between the programs.

- When developing an application with an interface with assembler, you can use the DSECT Conversion Utility to build structures mapping to the data types of your DSECTs.
- The POSIX locale features and coded character set conversion routines are supported only for system programming applications that use z/OS Language Environment. They are not available for freestanding applications.

- IEEE decimal floating-point data types are not supported for the function `sprintf()` under SPC. If you need to use IEEE decimal floating-point data types, you must use the C/C++ Run Time Library version of the `sprintf()` function.

Creating freestanding applications

Freestanding applications are C modules that run either:

- Without z/OS Language Environment and the z/OS XL C library (using `EDCXSTRT`)
- Without z/OS Language Environment but with the z/OS XL C library functions (using `EDCXSTRL`)

Three initialization routines are provided by SPC for building freestanding applications:

EDCXSTRT

For building completely freestanding applications. The applications can use no z/OS XL C runtime library functions and can have no z/OS Language Environment attachment.

EDCXSTRL

For building applications that use z/OS XL C runtime library functions but have no z/OS Language Environment attachment.

EDCXSTRX

This routine accepts a parameter to choose whether your application should behave as if it was initialized with either `EDCXSTRT` or `EDCXSTRL`. This parameter is described further in [“Setting up a C environment with preallocated stack and heap” on page 497](#).

Certain restrictions apply to freestanding applications initialized by the routines `EDCXSTRT`, `EDCXSTRL`, and `EDCXSTRX`. These restrictions are as follows:

- They cannot perform interlanguage calls, except with assembler language routines that preserve register 12 and use the IBM-supplied macros for entry and exit.
- The parameters received by the `main()` function (normally `argc` and `argv`) are undefined. `__xregs()` (described in [“__xregs\(\) — Get Registers on Entry” on page 533](#)) can be used to examine the parameters passed by the calling environment.
- They cannot do arithmetic using `long` `double` variables on pre-XA machines (that is, on machines that do not support the DXR instruction).

Creating modules without CEESTART

In many environments, the initialization normally performed by z/OS Language Environment is replaced by special-purpose routines that are tailored to the specific requirements of the type of application. This requires replacing the initialization routine (`CEESTART`) normally used by z/OS XL C.

When you do not use the System Programming C Facilities, the compiler generates a `CEESTART` CSECT (control section) whenever a `main()` or *fetchable* function is encountered in the source file. With the `NOSTART` compiler option, described in the [z/OS XL C/C++ User's Guide](#), you can suppress the generation of `CEESTART` for source files that contain a `main()` function where this is required. In a system programming C environment, you must compile using the `NOSTART` option. The object modules created will then be suitable for inclusion in applications that use the alternative initialization routines described in this section.

Including an alternative initialization routine under z/OS

When `NOSTART` is used to suppress the generation of `CEESTART`, an alternative initialization routine must be explicitly included in the executable by the user at Link Edit. Use the Linkage Editor `INCLUDE` and `ENTRY` control statements. To include the alternative initialization routines described in this chapter, allocate `CEE.SCEESPC` to the `SYSLIB DD`. For example, you can use the linkage editor statements in [Figure 123 on page 497](#) to specify `EDCXSTRT` as an alternative initialization routine:


```
//SYSLIN DD *
INCLUDE SYSLIB(EDCXSTRT)
ENTRY EDCXSTRT
INCLUDE OBJECT(main-function)
/*
```

Figure 123. Specifying alternative initialization at link edit

Another example of specifying alternative initialization under z/OS is shown in [Figure 125 on page 499](#).

Initializing a freestanding application without Language Environment.

The EDCXSTRT routine is for C applications that do not use any z/OS Language Environment facilities or z/OS XL C facilities or library functions. It must be explicitly included in the program and specified as the program entry point if it is to be used. Under this environment, only the following library routines are supported:

- Built-in compiler functions. For a list of these functions, see [“Using functions in the system programming C environment” on page 494](#).
- Memory management routines, including `malloc()`, `calloc()`, `realloc()`, and `free()`.
- The `exit()` and `sprintf()` functions.

Note: The use of floating point conversion specifiers (e, E, f, g or G) is not supported without the Language Environment runtime. Since the use of EDCXSTRT allows the application to execute without the use of the Language Environment runtime, the use of the above conversion specifiers with `sprintf()` in this environment is not supported.

- The `__4kmalc()` and `__24malc()` functions.

The value returned to the host system will be the return value from `main()`.

The RENT compiler option is supported in this environment.

Initializing a freestanding application using C functions

The EDCXSTRL routine is the analog of CEESTART for C applications that use the z/OS XL C library functions only. EDCXSTRL supports the full library of C functions except for functions such as `cdump()`, `csnap()`, `ctest()`, or `ctrace()`. EDCXSTRL must be explicitly included in the program and specified as the program entry point if it is to be used.

The value returned to the host system will be the return value from `main()`.

The RENT compiler option is supported in this environment.

Service routines (described in [“Developing services in the service routine environment” on page 509](#)) *require* this routine (or EDCXSTRT if they do not require z/OS Language Environment) for their initialization.

Applications initialized with this routine will run in any environment supported by z/OS Language Environment.

Setting up a C environment with preallocated stack and heap

The EDCXSTRX routine is the analog of CEESTART for an application where you want to have more control over contents supervision and storage management. Unlike EDCXSTRT, EDCXSTRL, and CEESTART, this routine cannot be entered directly from the operating system (that is, from JCL, REXX EXECs, CLISTs, or the TSO command line). It requires a structured parameter list (OS linkage) containing:

1. The parameter list to be passed to `main()`.

`__xregs()` can be used to examine the parameters passed by the calling environment. This list cannot be accessed by `argc` or `argv`.

2. The address of the initial storage area.

This area must be doubleword aligned with its first word containing its total length. It must be large enough to accommodate the entire stack requirements of the application.

3. The address of the complete heap allocation (or NULL if no `malloc()` family storage is required by the called routines).

This area must be doubleword aligned with its first word containing its total length. This area *must* include sufficient space for the control structures required to manage the heap (currently a minimum of 40 bytes). Applications that use the z/OS XL C library functions will always require heap space; the amount required depends on the structure of the application and may vary from run to run if external characteristics (file block sizes, for example) change.

Any heap increments that occur because the size of the initial heap is not large enough will not be freed at termination by the system programming environment. If no initial heap allocation is specified, and a heap is required (because the z/OS XL C library functions are required, for example), it will not be freed by the System Programming C Environment. If this behavior is detected, the program will run to completion, but will abend during EDCXSTRX termination with abend code 2108 and reason code 7207.

Heap increments will be freed if you explicitly free the memory (using the `free()` function) and the runtime option `HEAP(FREE)` has been specified. You should specify a heap value of at least 4K if you are running with the z/OS XL C library functions.

4. The address of the z/OS XL C runtime library or NULL. Use `CEEV003` (or `EDCZV`, if you want to maintain compatibility with previous releases of OS/390 Language Environment).

The parameters (`argc` and `argv`) passed to the `main()` function are undefined. There is no argument parsing (`argc` and `argv`) or redirection of standard streams.

If the z/OS XL C library functions are required, the routine `EDCXABRT` must be explicitly included during the link edit. This routine enables exception handling for `EDCXSTRX`. If it is not explicitly included, abend code 2107 with reason code 7206 will terminate the program.

The `RENT` compiler option is supported in this environment only if the z/OS XL C library functions are used.

Determining ISA requirements

The `EDCXISA` entry point is available to the caller of `EDCXSTRX` to determine the stack space overhead for the environment being created. Add stack space required by the application to the value returned by this routine to determine the size of the area to be passed as the second parameter to `EDCXSTRX`. If the routine is called from assembler, the value should be expected in Register 15. The routine should be declared as:

```
#pragma linkage(__xisa,05)
int __xisa(void);
```

Building freestanding applications to run under z/OS

When you are building freestanding applications under z/OS, `CEE.SCEESPC` must be included in the binder `SYSLIB` concatenation before `CEE.SCEELKD`. The routines to support this function (`EDCXSTRT`, `EDCXSTRL`, and `EDCXSTRX`) are `CEESTART` replacements (described in [“Creating modules without CEESTART”](#) on page 496) in your module. Therefore, the appropriate `EDCXSTRn` routine must be explicitly included ahead of the module at link edit. [Figure 124 on page 499](#) shows a simple freestanding routine that requires the library.

```

/* this is an example of a freestanding routine */
#include <stdio.h>

int main(void) {
    puts("Hello, World");
    return 3999;
}

```

Figure 124. Sample Freestanding z/OS Routine

This routine is compiled normally and link edited using control statements shown in [Figure 125 on page 499](#). The CEE.SCEERUN load library must be available at run time because it contains the C library function puts().

```

INCLUDE SYSLIB(EDCXSTR)
INCLUDE OBJECT
ENTRY EDCXSTR

```

Figure 125. Link edit control statements used to build a freestanding z/OS routine

[Figure 126 on page 499](#) shows how to compile and link a freestanding program using the cataloged procedure EDCCL.

```

//JOB      JOBCARD STATEMENTS
//*****
//*** COMPILE AND LINK FOR STRL ENTRY POINT
//*****
//C106001   EXEC   EDCCL,
//          INFILE='USERID.SPC.SOURCE(C106000)',
//          OUTFILE='USERID.SPC.LOAD(C106000),DISP=SHR',
//          CPARM='OPT,NOSEQ,NOMAR,NOSTART',
//          LPARM='RMODE=ANY,AMODE=31'
//COMPILE.USERLIB DD DSN=userid.HDR.FILES,DISP=SHR
//LKED.SYSLIB DD DSN=CEE.SCEESPC,DISP=SHR
//           DD DSN=CEE.SCEELKED,DISP=SHR
//LKED.SYSIN DD *
//          INCLUDE SYSLIB(EDCXSTR)
//          ENTRY EDCXSTR
/*

```

Figure 126. Compile and link using EDCCL

Special considerations for reentrant modules

A simple freestanding routine that does not require the library is shown in [Figure 127 on page 499](#). To develop a reentrant module, this routine must be compiled with both the RENT (because the module contains writable static at **2**) and NOSTART (because this is a system programming environment) compiler options. This routine uses the exit() function, which is normally part of the z/OS Language Environment library. Like sprintf(), it is available to freestanding routines without requiring the dynamic library.

```

/* this is an example of a reentrant freestanding routine */
#include <stdlib.h> 1
int main() {
    static int i[5]={0,1,2,3,4}; 2
    exit(320+i[1]);
}

```

Figure 127. Sample reentrant freestanding z/OS routine

Figure 128 on page 500 shows the JCL required to build and execute the routine in Figure 127 on page 499.

```
//PLKED EXEC PGM=EDCPRLK,PARM='MAP,NCAL' 1
//STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
//SYMSGS DD DSN=CEE.SCEMSGP(EDCPMSG),DISP=SHR
//SYSLIB DD DUMMY
//SYSMOD DD DSNNAME=&&PLKSET,SPACE=(32000,(30,30)),UNIT=SYSDA,
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200),
// DISP=(MOD,PASS)
//SYSIN DD DSNNAME=userid.TEST.OBJECT(PROG1),DISP=SHR 2
//SYSOUT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//*
//*
//LKED EXEC PGM=HEWL,PARM='MAP,XREF,LIST' 3
//SYSLIB DD DSNNAME=CEE.SCEESPC,DISP=SHR
// DD DSNNAME=CEE.SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSMOD DD DSNNAME=&&GOSSET(GO),SPACE=(512,(50,20,1)),
// DISP=(NEW,PASS),UNIT=SYSDA
//SYSUT1 DD SPACE=(32000,(30,30)),UNIT=SYSDA
//PRELINK DD DSNNAME=&&PLKSET,DISP=(OLD,DELETE)
//SYSLIN DD *
INCLUDE SYSLIB(EDCXSTRT) 4
INCLUDE PRELINK 5
INCLUDE SYSLIB(EDCXEXIT) 6
INCLUDE SYSLIB(EDCRCINT) 7
//*
//*
//*-----
//* Go Step
//*-----
//GO EXEC PGM=*.LKED.SYSLMOD
//SYSPRINT DD SYSOUT=*
```

Figure 128. Building and running a reentrant freestanding z/OS routine

- 1** The z/OS Language Environment prelinker must be used for modules compiled with the RENT compiler option.
- 2** This is the object module created by compiling the sample module with the RENT and NOSTART compiler options.
- 3** The output from the prelinker is made available to the linkage editor.
- 4** The alternative initialization routine (EDCXSTRT in this example) must be included explicitly in the module. If this is not the first CSECT in the module, it must be explicitly named as the module entry point.
- 5** The prelinked output is included in the load module.
- 6** EDCXEXIT must be explicitly included if the `exit()` function is used in the application.
- 7** The routine EDCRCINT must be explicitly included in the module if the RENT compiler option is used. No error will be detected at load time if this routine is not explicitly included. At execution time, abend 2106, reason code 7205, will result if EDCRCINT is required but not included.

Parts used for freestanding applications

Table 94 on page 501 lists the parts used for freestanding applications and their function and location. The SYSLIB specified is CEE.SCEESPC.

Table 94. Parts used for freestanding applications				
Part Name	Function	Inclusion in Program		Location
		Notes		
EDCXSTRT	Mainline for applications that do not require the z/OS Language Environment or z/OS XL C runtime library.	1	CSECT must be the module entry point EDCXSTRT.	Member of SCEESPC
EDCXSTRL	Mainline for applications that require only the C-specific library functions.	1	CSECT must be the module entry point.	Member of SCEESPC
EDCXSTRX	Mainline for applications that receive a structured parameter list that includes preallocated storage management areas.	2		Member of SCEESPC
EDCXISA	Get ISA requirements for EDCXSTRX.	2		Member of SCEESPC
EDCXSPRT	System programming version of <code>sprintf()</code> .	3		Member of SCEESPC
EDCXEXIT	System programming version of <code>exit()</code> .	3		Member of SCEESPC
EDCXMEM	System programming version of <code>malloc()</code> , <code>calloc()</code> , <code>realloc()</code> , <code>free()</code> , <code>__4kmalc()</code> and <code>__24malc()</code> .	3		Member of SCEESPC
EDCRCINT	Must be included if the compiler option RENT is to be used.	3		Member of SCEESPC
EDCXABRT	System programming version of exception handling.	3		Member of SCEESPC
Notes:				
1. This module must be explicitly included in the program using the binder INCLUDE control statement.				
2. This module will normally be included by automatic call.				
3. This module must be explicitly included if you want to use the system programming version of the function.				
4. Including EDCXABRT requires the system programmer C environment to be library enabled				

Creating system exit routines

z/OS XL C allows the creation of routines that have no environmental requirements on entry *except*:

- Register 13 must point to a 72-byte save area
- Register 14 must contain the return address
- Register 15 must contain the entry address

There is no requirement on the name of the entry point (that is, it does not have to be `main()`), so several different entry points, with names specified by the calling environment, can be combined in the same program.

Routines that do not require the z/OS XL C environment should specify one of these two pragma forms:

- `#pragma environment(function-name)`, if the library is required, or
- `#pragma environment(function-name, nolib)`, if no library is required.

This pragma causes the compiler to generate a different prolog for the specified function. The prolog contains the instructions at the beginning of the routine that perform the housekeeping necessary for the function to run, including allocation of the function's automatic storage. This prolog will set up a C environment sufficient for both the function in which it is specified and any function that may be called. Called functions should not specify this pragma, unless they are called elsewhere without a C environment present. This new prolog will load and initialize the module containing the C library functions

if this choice is specified. For more information on the `#pragma` environment, see [z/OS XL C/C++ Language Reference](#).

The RENT compiler option is not supported in this environment; if you require reentrant system exit routines, the routine must be naturally reentrant. See [z/OS Language Environment Programming Guide](#) for more information about reentrancy.

System exit routines can be linked with their callers or dynamically loaded and invoked.

Building system exit routines under z/OS

The CEE.SCEESPC object library must be available at link-edit time. If the C library is required by the exit routines, CEE.SCEELKED must also be made available after CEE.SCEESPC. You should explicitly name the entry point with an ENTRY statement.

An example of a system exit

Table 95 on page 505 lists the parts used by exits. The C program (CCNGSP3) shown in [Figure 129 on page 503](#) is a system exit that gains control from the system when an unknown CLIST subroutine is encountered. It checks if the name is recognized as a user-specific subroutine before returning control to the system. For more information on this system exit, see [z/OS TSO/E Customization](#).

```

/* this is an example of a system exit */
#pragma environment(IKJCT44B,nolib) 1
/*
/* IKJCT44B CLIST EXIT
/*
*/
#include <stdio.h>
#include <stdlib.h>
#include <spc.h>
struct parmentry { int key;
                  int len;
                  char *pt; };
typedef struct parmentry P_ENT;

#define REVERSE 0
#define FLIPCHR 1
/* Valid commands */
static char *cmds[] =
{
    "SYSXTREV", "SYSXTFLIP" 2
};
void revstring( P_ENT *p11, P_ENT *p12 );
void flipstring( P_ENT *p11, P_ENT *p12 );
int IKJCT44B() {
    int **parme;
    struct parmentry *e7, *e10, *e11, *e12, *e13;

    /* Get registers on entry */
    parme = (void *)__xregs(1); 3

    /* Get the parameter entry values for those relevant for CLISTs */
    e7 = (struct parmentry *)parme[ 6]; /* exit return */
    e10 = (struct parmentry *)parme[ 9]; 4
    e11 = (struct parmentry *)parme[10];
    e12 = (struct parmentry *)parme[11];
    e13 = (struct parmentry *)parme[12];

    /* Is the command supported? */
    switch( cmdchk(e10) ) { 5
        case REVERSE: /* Reverse string */
            revstring( e11, e12 );
            break;

        case FLIPCHR: /* Exchange the first and last chars only */
            flipstring( e11, e12 );
            break;

        default: /* Unknown command type. Return with an error. */
            e12->pt[0] = 0x00;
            e12->len = 0;
            /* Set the return code */
            e7->key = 0x01;
            e7->len = 0x04;
            *(int *)&e7->pt = 0x06;
            return 12;
    }
}

```

System exit example (Part 1 of 2)

Figure 129. System exit example

```

    /* Return to caller - CLIST is supported. */
    e7->key = 0x01;
    e7->len = 0x04;
    *(int *)&e7->pt = 0x00;
    return 0;
}

/* cmdchk( P_ENT *pt )
/* - is the command in the list of user-specific cmds? */
int cmdchk( P_ENT *pt ) {
    int i;
    for( i=0; i<(sizeof(cmds)/sizeof(char *)); i++ ) {
        if( memcmp( pt->pt, cmds[i], pt->len ) == 0 )
            return i;
    }
    /* Not found */
    return -1;
}

/* revstring()....
/* - reverse the string
void revstring( P_ENT *p11, P_ENT *p12 ) {
    int i;

    for( i=0; i<p11->len; i++ )
        p12->pt[i] = p11->pt[p11->len-i-1];
    p12->len = p11->len;
}

/* flipstring() ...
/* - flip the first and last characters in the string
void flipstring( P_ENT *p11, P_ENT *p12 ) {
    char t;
    t = p11->pt[p11->len-1];
    memcpy( p12->pt, p11->pt, p11->len );
    p12->pt[p11->len-1] = p12->pt[0];
    p12->pt[0] = t;
    p12->len = p11->len;
}

```

System exit example (Part 2 of 2)

1

The #pragma environment directive sets up an entry point IKJCT44B other than main().

2

This is the list of user-specific subroutines that are available in this system exit.

3

The function __xregs() is used to retrieve the parameters available to the system exit in R1 from the operating system.

4

The parameters are parameter entries passed from TSO to this system exit and are used for the following reasons:

e7

Exit reason code

e10

Name of subroutine

e11

Arguments

e12

Result

5

The list of user-specific subroutines is checked and if the unknown CLIST subroutine is recognized, the subroutine is called. Otherwise, the function returns in error.

Table 95 on page 505 lists the parts used by the routines, and their function and location in MVS. The SYSLIB specified is CEE.SCEESPC.

Table 95. Parts used by exit routines			
Part Name	Function	Inclusion in Program	Location
		Notes	
EDCXENV	Extended prolog code for exits that do not require the library.	2	Member of SCEESPC
EDCXENVL	Extended prolog code for exits that require the library.	2	Member of SCEESPC
EDCXSPRT	System programming version of <code>sprintf()</code> .	3	Member of SCEESPC
EDCXEXIT	System programming version of <code>exit()</code> .	3	Member of SCEESPC
EDCXMEM	System programming version of <code>malloc()</code> , <code>calloc()</code> , <code>realloc()</code> , <code>free()</code> , <code>__4kmalc()</code> and <code>__24malc()</code> .	3	Member of SCEESPC
EDCXABRT	System programming version of exception handling.	3	Member of SCEESPC
Notes: <ol style="list-style-type: none"> 1. This module must be explicitly included in the program using the binder INCLUDE control statement. 2. This module will normally be included by automatic call. 3. This module must be explicitly included if you want to use the system programming version of the function. 			

Creating and using persistent C environments

Four routines are available to create and use a persistent C environment. These routines are used by an assembler language application that needs a C environment available to support the C functions that it calls. C main routines cannot be called in persistent C environments. The four routines are:

EDCXHOTC

Sets up a persistent C environment (no library)

EDCXHOTL

Sets up a persistent C environment (with library)

EDCXHOTU

Runs a function in a persistent C environment

EDCXHOTT

Terminates a persistent C environment

An initialization routine, EDCXHOTC or EDCXHOTL (depending upon whether the called C subroutines will need the z/OS XL C library functions), is called to create a C environment. This call returns a *handle* that can be used (through EDCXHOTU) to call C subroutines. The environment persists until it is explicitly terminated by calling EDCXHOTT.

The functions that act as entry points for these routines are `__xhotc()`, `__xhotl()`, `__xhotu()`, and `__xhott()`, respectively. For more information on these four functions, refer to [Chapter 43, "Library functions for system programming C,"](#) on page 531.

Restrictions:

1. C main routines are not supported in persistent C environments.
2. The RENT compiler option is not supported in the persistent environment described in this chapter.
3. Exception handling is not supported in persistent C environments.

As an alternative to the persistent environments, you can also create and retain a C environment using the preinitialized programming interface. This interface supports the RENT compiler option, but is less versatile in other respects. z/OS Language Environment provides a callable service for preinitialization called CEEPIPI. This is described in [z/OS Language Environment Programming Guide](#).

Building applications that use persistent C environments

There are no special restrictions for building applications that use persistent C environments. The automatic call facility will cause the correct routines from the SYSLIB to be included.

If any C library function is required by any routine called in this environment, the stub routines library CEE.SCEELKED should be made available at link time *after* CEE.SCEESPC.

An example of persistent C environments

The assembler routine shown in [Figure 131 on page 507](#) illustrates the use of this feature to call a C function shown in the sample program (CCNGSP4) in [Figure 130 on page 506](#).

```

/* this example uses a persistent C environment */
/* part 1 of 2-other file is CCNGSP5 */

#pragma linkage(crtn,OS) 1
#include <string.h>
#include <stdio.h>
#define INSIZE 300      /* the maximum length we'll tolerate */

void crtn(int p1,char *p2) {
    char    hold[2+INSIZE];
    char    *endptr;
    int     i;

    endptr=memchr(p2, '@', INSIZE);
    if (NULL==endptr)
        i=INSIZE;          /* no ender? use max */
    else
        i=endptr-p2;        /* length of stuff before it */

    memcpy(hold,p2,i);      /* copy formatting string */
    hold[i++]='\n';         /* add a new-line.. */
    hold[i]='\0';           /* ..and a null terminator */

    printf(hold,p1);        /* print it out */

    return;                 /* and return */
}

```

Figure 130. Example of function used in a persistent C environment

This C function accepts two parameters: an integer and a printf()-style formatting string. The formatting string has a maximum length of 300 bytes; it is terminated by an @ if shorter. This routine *must* use OS linkage (1). The routine scans the formatting string for the terminator, copies it to a local work area, adds a trailing newline and NULL character, and prints the integer according to the formatting string.

The structure of the assembler caller (CCNGSP5) is shown in [Figure 131 on page 507](#).

```

* this example demonstrates a persistent C environment
* part 2 of 2-other file is CCNGSP4
ENVA      CSECT
ENVA      AMODE ANY
ENVA      RMODE ANY
          STM    R14,R12,12(R13)  1
          LR     R3,R15
          USING  ENVA,R3
          GETMAIN R,LV=DSALEN
          ST     R13,4(,R1)
          LR     R13,R1
          USING  DSA,R13
          LA     R4,HANDLE  2
          LA     R5,STKSIZE
          LA     R6,STKLOC
          STM    R4,R6,PARMLIST
          OI     PARMLIST+8,X'80'
          LA     R1,PARMLIST
          L      R15,=V(EDCXHOTL)
          BALR   R14,R15
LOOP      LA     R8,10  3
          DS     0H
          ST     R8,LOOPCTR  4
          LA     R4,HANDLE
          LA     R5,USEFN
          LA     R6,LOOPCTR
          LA     R7,FMTSTR1
          STM    R4,R7,PARMLIST
          OI     PARMLIST+12,X'80'
          LA     R1,PARMLIST
          L      R15,=V(EDCXHOTU)
          BALR   R14,R15
          LA     R7,FMTSTR2  5
          STM    R4,R7,PARMLIST
          OI     PARMLIST+12,X'80'
          L      R15,=V(EDCXHOTU)
          BALR   R14,R15
          BCT    R8,LOOP
          ST     R4,PARMLIST  6
          OI     0(R1),X'80'
          LA     R1,PARMLIST
          L      R15,=V(EDCXHOTT)
          BALR   R14,R15
          LR     R1,R13  7
          L      R13,4(0,R13)
          FREEMAIN R,A=(1),LV=DSALEN
          LM     R14,R12,12(R13)
          SR     R15,R15
          BR     R14

```

Using a persistent C environment (Part 1 of 2)

Figure 131. Using a persistent C environment

```

USEFN      DC      V(CRTN)
STKSIZE    DC      A(4096)
STKLOC     DC      A(1)
FMTSTR1    DC      C'1st value of loopctr is %i@'
FMTSTR2    DC      C'value on 2nd call is %i@'
           LTORG
DSA        DSECT    ,           The dynamic storage area
SAVEAREA   DS        18A       The save area
PARMLIST   DS        4A
HANDLE     DC        A(0)
LOOPCTR    DC        A(1)
DSALEN     EQU      *-DSA
R0         EQU      0
R1         EQU      1
R2         EQU      2
R3         EQU      3
R4         EQU      4
R5         EQU      5
R6         EQU      6
R7         EQU      7
R8         EQU      8
R12        EQU      12
R13        EQU      13
R14        EQU      14
R15        EQU      15
END        ENVA

```

Using a persistent C environment (Part 2 of 2)

1

This routine is entered with standard linkage conventions. It saves the registers in the save area pointed to by register 13, acquires a dynamic storage area for its own use, and chains the save areas together.

2

A C environment that includes support for the z/OS XL C library is created by calling EDCXH0TL. The parameter list for this call is the address of the handle (for the persistent C environment created), the address of a word containing the initial stack size, and the address of a word containing the initial stack location (0 for below the 16MB line and 1 for above). This parameter list uses the normal OS linkage format.

3

The routine loops 10 times calling the C function `crtn` twice each time through the loop.

4

The parameter list for the first call is the address of the handle, the address of a word pointing to the function, and the parameters to be received by the function. EDCXH0TU is called. This causes the specified C function, `crtn()` to be given control with register 1 pointing to the remaining parameters, LOOPCTR and FMTSTR1.

5

The C function is called again, this time with FMTSTR2 as the second parameter.

6

When the loop ends, EDCXH0TT is called to terminate the environment created at **2**

7

The routine terminates by freeing its dynamic storage area and returning to its caller.

Table 96 on page 509 lists the parts used by persistent environments and their function and location. The SYSLIB is CEE.SCEESPC.

Table 96. Parts used by persistent environments

Part Name	Function	Inclusion in Program	Location
		Notes	
EDCXHOTC	Called to set up a C environment without z/OS Language Environment.	2	Member of SCEESPC
EDCXHOTL	Called to set up a C environment with the z/OS XL C library functions available.	2	Member of SCEESPC
EDCXHOTT	Called to terminate a C environment set up by EDCXHOTC or EDCXHOTL.	2	Member of SCEESPC
EDCXHOTU	Called to use a C environment set up by EDCXHOTC or EDCXHOTL.	2	Member of SCEESPC
EDCXSPRT	System programming version of <code>sprintf()</code> .	3	Member of SCEESPC
EDCXEXIT	System programming version of <code>exit()</code> .	3	Member of SCEESPC
EDCXMEM	System programming version of <code>malloc()</code> , <code>calloc()</code> , <code>realloc()</code> , <code>free()</code> , <code>__4kmalc()</code> and <code>__24malc()</code> .	3	Member of SCEESPC
Notes: <ol style="list-style-type: none"> 1. This module must be explicitly included in the program using the binder INCLUDE control statement. 2. This module will normally be included by automatic call. 3. This module must be explicitly included if you want to use the system programming version of the function. 			

Developing services in the service routine environment

The purpose of an application service routine environment is to allow the development, using z/OS XL C, of services that can be developed, tested, and packaged independently of their intended users. You can:

- Isolate the service code from its user
- Specify and enforce a clearly defined Application Programming Interface (API) between the user (another application program) and the service routine
- Share server code among more than one (perhaps different) user applications simultaneously
- Enhance or maintain the service routine code with no disruption to its various user applications

In this environment, a service application is developed as a C `main()` function together with any functions it may call, and packaged as a complete program. This program, if it is reentrant, can be freely installed in the ELPA and shared by all of its users.

To provide the service to a user application, the developer of the service must offer small assembler language stub routines that are link-edited with the user code. These stub routines use services provided by the System Programming Facilities to load or locate the server code and pass messages to it for execution. Examples of these stub routines are shown in [“Constructing user-server stub routines”](#) on page 523.

Using application service routine control flow

In this section examples are based on a service routine that manages a storage queue. This server might be used by languages that do not support dynamic memory allocation, or by applications that do not want

to concern themselves with the management of such data structures. The operations supported by this service routine are:

- Initialize
- Terminate
- Add an element to the head of the queue (last in, first out)
- Add an element to the tail of the queue (first in, first out)
- Get the element at the head of the queue

Service routine user perspective

A conversation is initiated when a user routine calls a startup routine supplied by the author of the service to establish a connection between the user and the server. This routine returns a *handle* to the user that represents the server environment. User routines may establish connections with many different services or many times with the same server as long as the needed resources, principally memory, are available in the system. Each connection has a different handle, and it is the user routine's responsibility to keep track of them.

Note: Memory files cannot be shared between the user routines and the server.

Once the user has initialized the server, it uses other server-supplied stub routines to send requests (messages) to the server for action. One of the parameters to this routine will be the handle returned by the initialize call. These request stubs would typically return a feedback code to indicate success or failure as well as any other information requested. The server defines the parameter list to be passed and the feedback codes to be given to the user.

When the user is finished with the server, it calls yet another stub routine to terminate the server. This structure is illustrated in a sample user routine (CCNGSP6) shown in [Figure 132 on page 510](#).

```
PROGRAM MAIN

C   Example User-Service Routine application

C   Define the variable that will hold the 'handle' for the server
INTEGER*4 HANDLE 1

C   Define the variable that will hold feedback codes
INTEGER*4 FEEDBACK

C   Define the variable that we'll use to get the strings back
CHARACTER*100 CH
INTEGER*4 CHLEN

C   initialize the server
CALL QMGINIT(HANDLE) 2

C   Feed some strings to the server 3
CALL QMGLIFO(HANDLE,FEEDBACK,17,'2 Sample string 1')
CALL QMGLIFO(HANDLE,FEEDBACK,23,'1 Another sample string')
CALL QMGFIFO(HANDLE,FEEDBACK,20,'3 Yet another string')

C   Get the strings back, print out length and value
DO 1 I=1,3 4
CALL QMGGET(HANDLE,FEEDBACK,CHLEN,CH)
PRINT *,CHLEN,CH(1:CHLEN) 5
1   CONTINUE

C   Terminate the server

CALL QMGTERM(HANDLE) 6

C   Go home
STOP
END
```

Figure 132. Example of user routine

- 1** The user routine sets up a variable that will be used to hold the handle returned by the server. The form taken by this handle is up to the supplier of the service, but a fullword (4 bytes) should be regarded as typical.
- 2** The user routine calls the initialize routine to set up the connection between the user routine and the server.
- 3** The user routine adds three strings to the queue. In this example, the first character of the string indicates the order in which the user expects to retrieve the strings.
- 4** The user enters a loop in which the strings are retrieved from the queue.
- 5** The user routine prints out the strings passed back by the call to the server. If there is no string remaining in the queue a null string (zero length) is returned.
- 6** Before ending, the user routine closes down the server.

This routine is linked normally with the server-supplied stub routines (described in [“Constructing user-server stub routines”](#) on page 523).

Service routine perspective

A service routine is a complete, stand alone module that runs in its own C environment. Its environment is created on demand by user application routines that call it using stub routines supplied by the server. When this happens, the server code enters at its `main()` entry point and, typically, goes into a loop that contains a function call to get the next *to-do*. One possible *to-do* is *terminate*; when this command is received the server should `exit()` or `return` from its `main()` function. The environment created when the server was started terminates and all resources held by the server are freed (except storage acquired by `__24malc()` or `__4kmalc()`, as described in [“__24malc\(\) — Allocate Storage below 16MB Line”](#) on page 535 and [“__4kmalc\(\) — Allocate Page-Aligned Storage”](#) on page 535. This structure is illustrated in a sample user routine shown in [Figure 133](#) on page 512.

```

/* this is an example of an application service routine */

#include <spc.h> 1
#include <stdlib.h>
#include <string.h>

#define LIFO 1 2
#define FIFO 2
#define GET 3
#define TERM -1

int main(void) { 3
    int retcode=0;

    /* data structures to manage the queue */
    struct queue_entry { 4
        struct queue_entry *next;
        int length;
        char val[1];
    };

    struct queue_entry *head;
    struct queue_entry *tail;

    struct { 5
        int code;
        union info *plist;
    } *req;

    union info { 6
        struct {
            int *length;
            char *string;
        } lifo;
        struct {
            int *length;
            char *string;
        } fifo;
        struct {
            int *length;
            char *string;
        } get;
    };

    /* initialize the queue pointers */
    head = NULL; 7
    tail = NULL;

```

Example of application service routine (Part 1 of 2)

Figure 133. Example of application service routine


```

/* the main processing loop goes on until a termination signal is sent */
for(;;) { 8
    union info          *info;
    int                  length;
    char                 *string;
    struct queue_entry   *ent;

    /* get a message from the user routine */
    req=__xsrvc(retcode); 9 18
    info = req->plist; 10

    switch(req->code) { 11
        case LIFO: { 12
            length=>(*info).lifo.length;
            string= (*info).lifo.string;
            ent = malloc(sizeof *ent - 1 + length); 13
            memcpy((*ent).val,string,length);
            __xsacc(0); 14
            (*ent).length=length;
            (*ent).next=head;
            head=ent;
            if (NULL==tail) tail=ent;
            break;
        }
        case FIFO: { 15
            length=>(*info).fifo.length;
            string= (*info).fifo.string;
            ent = malloc(sizeof *ent - 1 + length);
            memcpy((*ent).val,string,length);
            __xsacc(0);
            (*ent).length=length;
            (*ent).next=NULL;
            if (NULL==head) head=ent;
            else (*tail).next=ent;
            tail=ent;
            break;
        }
        case GET: { 15
            if (NULL==head) {
                *(*info).get.length=0;
                break;
            }
            length = (*head).length;
            string = (*info).get.string;
            memcpy(string,(*head).val,length);
            *(*info).get.length=length;
            __xsacc(0);
            ent=head;
            head=(*ent).next;
            free(ent);
            if (NULL==head) tail=NULL;
            break;
        }
        case TERM: 16
            return 0;
        default:
            __xsacc(666); 17
    }
} 18
return(0);
}

```

Example of application service routine (Part 2 of 2)

1

The server routine should include the appropriate header files. `spc.h` contains the function prototypes for the routines that are used to maintain the conversation between the server routine

and the user routine. `string.h` is *required* if string or memory functions are used in the code and z/OS Language Environment will not be available at run time; this header file contains the directives necessary to use these built-in functions.

2

These are the *command codes* of the requests that can be sent to this server.

3

The server begins with a `main()` function. This function gets control when the user calls `QMGINIT`.

4

This server manages an in-storage queue of unstructured elements. It does this by maintaining a linked list of elements. The structure `queue_entry` contains an individual entry; `head` and `tail` point to the first and last entries in the queue.

5

Requests come to the server in the form of a pointer to a structure containing a command code (in this case, one of `LIFO`, `FIFO`, `GET`, or `TERM`) and a pointer to a parameter list associated with the command code. The parameter list is what follows `HANDLE` and `FEEDBACK` in the calls to `QMGLIFO`, `QMGFIFO`, and `QMGGET`. Like the command codes, the structure of this parameter list is established in concert with the stub routines.

6

In this example, all the commands have exactly the same format. This may not generally be the case, so a union of the various parameter list formats is appropriate. Then the interface can be expanded without disrupting existing code.

7

Before accepting commands, required initialization is performed.

8

This server is structured as an endless loop. This loop terminates when a terminate message sends control to a `return` statement at **17**.

9

At this point, the server is ready for work. The call to `__xsrv()` causes the user routine to resume execution at the place it left off when it last called the server. The value passed as the parameter is made available to the stub routines for use as a feedback code. This function will not return until the user application sends a request (using one of the stub routines, in this example `QMGLIFO`, `QMGFIFO`, `QMGGET`, or `QMGTERM`).

10

Extract the parameters from the structure pointed to by the call to `__xsrv()`.

11

Examine the request code sent by the user application.

12

The `LIFO` request code is handled here.

13

These library functions (and many others, the complete list is given in [“Using functions in the system programming C environment”](#) on page 494) are normally available in this environment even though z/OS Language Environment is not available at run time. The amount of storage allocated is the size of the queue entry (defined at **4**) minus 1 (because the definition of the entry allowed for 1 character of value) plus the length actually required for the value.

14

This function should be used to indicate that the server has completed its use of any data structures (parameters and data areas pointed to by the parameters) belonging to the user application. The value passed to this function or the value passed by the next call to `__xsrv()` (which ever is greater in magnitude) will be passed to the stub routine for use as a feedback code.

15

The handling of `FIFO` and `GET` is similar.

16

When a terminate request is received, the server `returns`. This terminates the loop (at **8**) and the environment set up when the server was first called.

17

If the command code is not recognized the server acknowledges the request and sets a return code that can be analyzed by the stub routine or the user application.

18

The server returns to the request for another *to-do*. The value passed as a parameter here or the last value passed to `__xsacc()`, whichever has the greater magnitude, is passed to the stub routine for use as a feedback code.

The server is built as a freestanding C application, as described in [“Creating freestanding applications” on page 496](#).

You must specify `EDCXSTRT`, `QMGSERV`, `EDCXMEM` and `EDCXEXIT` when you link edit.

Understanding the stub perspective

The stub routines provide the link between the user application and the application service module. They are responsible for:

- Locating or loading the server code
- Providing the Application Programming Interface (API) seen by the user.

Many choices are available in the design of the API and how single calls in the user are mapped. For example, the initialize call could accept parameters governing the behavior of the session being established and pass them to the server as commands once the server has been initialized. In the example the interactions are straight forward, the initialize only starts up the server, and the message calls send single messages, untouched and unexamined, to the server.

There are two kinds of stubs: the initialization stub and the message stubs. Termination is a special case of a message stub. These stubs are most appropriately written in assembler so that they can run in any language environment with minimal performance cost.

The initialization stub is responsible for loading and calling the server. It can use the low-level storage management and contents supervision routines supplied in `SCEESPC`. These routines are described in [“Tailoring the system programming C environment” on page 524](#). The structure of an initialization stub is shown in [Figure 134 on page 516](#).

```

* this is an example of a server initialization stub
QMGINIT  TITLE 'SERVER supplied stub to initialize'
QMGINIT  CSECT ,
          STM R14,R12,12(R13) 1
          LR R3,R15
          USING QMGINIT,R3
          USING INPARMS,R1 2
          L R6,HANDLE@
          DROP R1
          LA R0,WALEN          length of work area, below the line 3
          L R15,=V(EDCXGET)    GETMAIN some storage
          BALR R14,R15
          USING WA,R1
          ST R13,SA+4
          LR R13,R1
          USING WA,R13          This is now our DSA
          LA R1,NAME 4
          L R15,=V(EDCXLOAD)
          BALR R14,R15          Load the server
          ST R1,PLIST 5
          MVC PLIST+4(12),PLISTINI
          L R15,=V(EDCXSRVI)
          LA R1,PLIST
          BALR R14,R15
          MVC 0(4,R15),=CL4'QMqm' eye-catcher 6
          ST R13,4(,R15) 7
          ST R15,0(,R6)      Save handle in users parameter 8
          L R13,4(,R13) 9
          LM R14,R12,12(R13)
          SR R15,R15
          BR R14
PLISTINI DS 0D
          DC A(0),V(EDCXGET,EDCXFREE)
NAME      DC CL8'QMGSERV'
INPARMS   DSECT
HANDLE@   DS F
WA         DSECT
SA         DS 18F
PLIST     DS 4F
WALEN     EQU *-WA
          YREGS
          END

```

Figure 134. Example of server initialization stub

- 1 Stub routines are presumed to have a save area available at the location pointed to by register 13.
- 2 The parameter list passed to stub routines is OS linkage; that is, register 1 points to a list of addresses. In this example, the initialization stub receives only one parameter, the handle, that gets the address of a control block representing the environment.
- 3 For efficiency, this routine gets a work area that will be used by *all* the stub routines. The low level storage management routine EDCXGET, (described in “Getting storage” on page 524) is available for this purpose. This area will be the DSA for this and all other stub routines. It begins with an 18-word save area for use by routines called by this stub. It will be freed by the "terminate" stub.
- 4 When a save area is available, EDCXLOAD (described in “Loading a module” on page 526) is called to load the server.
- 5 EDCXSRVI is called to initialize the server. When control is returned from this call, the server has built a complete environment and has asked for something to do.
- 6 The value returned by EDCXSRVI is the address of a control block that is used to manage the interface between the user application and the service application module. The first 3 words (12 bytes) of this control block are reserved for the exclusive use of the stub routines. The fields following the first 3 words may not be used by either the stub routines or the user, nor may their values be altered. In this example, an *eye-catcher* (often useful for debugging) is moved into the first word.

7

The address of the work area acquired for dynamic storage requirements is moved into the second word. The address of this control block is stored in the user's handle.

8

The address of the control block from EDCXSRVI is placed in the user routine's handle. The user routine has no knowledge of the contents or format of this field; it is simply a *token* that is passed to other stub routines to manage the conversation between the user and the service routine.

9

Having initialized the server, the stub returns to the user at **2** in [Figure 132 on page 510](#).

Message stubs are responsible for passing requests from the user application to the service application. Like the initialization stub, they are free to use the low-level storage management and contents supervision routines supplied with the system programming facilities. Example message stubs are shown in [Figure 135 on page 517](#), [Figure 136 on page 518](#), [Figure 137 on page 520](#), and [Figure 138 on page 521](#).

```

* this is an example of a server message stub
QMGLIFO  TITLE 'SERVER supplied stub for feeding strings LIFO'
QMGLIFO  CSECT
        STM R14,R12,12(R13) 1
        LR  R3,R15
        USING QMGLIFO,R3
        LR  R5,R1
        USING INPARMS,R5
        L   R6,HANDLE@
        L   R6,0(R6)          Point to the handle 2
        L   R1,4(R6)          Point to work area got by QMGINIT 3
        USING WA,R1
        ST  R13,SA+4          Keep savearea passed into us
        LR  R13,R1            WA is new savearea
        USING WA,R13
        LA  R7,LIFO 4
        LA  R8,INPARMS+8      User parms start at 3rd
        STM R6,R8,PLIST       handle, LIFO, Other parms
        LA  R1,PLIST
        L   R15,=V(EDCXSRVN) 5
        BALR R14,R15
        L   R1,FEEDBK@ 6
        ST  R15,0(R1)
        L   R13,4(R13) 7
        L   R14,12(R13)
        LM  R0,R12,20(R13)
        BR  R14
INPARMS  DSECT
HANDLE@  DS    F
FEEDBK@  DS    F
LENGTH@  DS    F
STRING@  DS    F
WA        DSECT
SA        DS    18F
PLIST     DS    4F
WALEN     EQU   *-WA
LIFO      EQU   1
FIFO      EQU   2
GET        EQU   3
TERM      EQU   -1
YREGS
END

```

Figure 135. Example of server message stub-LIFO

1

Like the initialize stub, the QMGLIFO message stub expects a standard save area pointed to by register 13. The parameters are passed with standard OS linkage (register 1 pointing to a list of addresses).

2

The *handle* contains the value that was placed there by the initialization stub at **8** in [Figure 134 on page 516](#). This is the address of the control block that is used to manage the interface between the user application and the server.

- 3** Recover the address of the stub work area for use as a Dynamic Storage Area (DSA). This value was saved here by the initialization stub at The save area back chain field is set according to usual conventions.
- 4** A parameter list consisting of the handle (as returned by EDCXSRVI at **5** in Figure 134 on page 516 in the initialization stub), code for LIFO, and the address of the remaining parameters.
- 5** Call EDCXSRVN to *re-awaken* the server. This causes the server to resume control at **9** in Figure 133 on page 512 in the server. The server has control until it asks for the next *to-do*, in this example at **9**.
- 6** The value passed to `__xsrvc()` appears as the return code from EDCXSRVN. This value is passed back to the user application in the second parameter. *This is part of the API defined by this particular server, not something inherent in the user-server relationship.*
- 7** Control is returned to the user in the usual way.

The routine in Figure 136 on page 518 uses functions supplied in SCEESPC to load or locate the server code and initialize its environment.

```
* this is an example of a server message stub
QMGFIFO TITLE 'SERVER supplied stub for feeding strings FIFO'
QMGFIFO CSECT
QMGFIFO AMODE ANY
QMGFIFO RMODE ANY
STM R14,R12,12(R13) 1
LR R3,R15
USING QMGFIFO,R3
LR R5,R1
USING INPARMS,R5
L R6,HANDLE@
L R6,0(R6) Point to the handle 2
L R1,4(R6) Point to work area got by QMGINIT 3
USING WA,R1
ST R13,SA+4 Keep savearea passed into us
LR R13,R1 WA is new savearea
USING WA,R13
LA R7,FIFO 4
LA R8,INPARMS+8 User parms start at 3rd
STM R6,R8,PLIST handle, FIFO, Other parms
LA R1,PLIST
L R15,=V(EDCXSRVN) 5
BALR R14,R15
L R1,FEEDBK@ 6
ST R15,0(R1)
L R13,4(R13) 7
L R14,12(R13)
LM R0,R12,20(R13)
BR R14
INPARMS DSECT
HANDLE@ DS F
FEEDBK@ DS F
LENGTH@ DS F
STRING@ DS F
WA DSECT
SA DS 18F
PLIST DS 4F
WALEN EQU *-WA
LIFO EQU 1
FIFO EQU 2
GET EQU 3
TERM EQU -1
YREGS
END
```

Figure 136. Example of server message stub-FIFO

- 1** Like the initialize stub, the QMGFIFO message stub expects a standard save area pointed to by register 13. The parameters are passed with standard OS linkage (register 1 pointing to a list of addresses).

2

The *handle* contains the value that was placed there by the initialization stub at **8** in [Figure 134 on page 516](#). This is the address of the control block that is used to manage the interface between the user application and the server.

3

Recover the address of the stub work area for use as a Dynamic Storage Area (DSA). This value was saved here by the initialization stub at **7** in [Figure 134 on page 516](#). The save area back chain field is set according to usual conventions.

4

A parameter list consisting of the handle (as returned by EDCXSRVI at **5** in [Figure 134 on page 516](#)), code for FIFO, and the address of the remaining parameters.

5

Call EDCXSRVN to *re-awaken* the server. This causes the server to resume control at **9** [Figure 133 on page 512](#) in the server. The server has control until it asks for the next *to-do*, in this example at **9** in [Figure 133 on page 512](#), again.

6

The value passed to `__xsrv()` appears as the return code from EDCXSRVN. This value is passed back to the user application in the second parameter. *This is part of the API defined by this particular server, not something inherent in the user-server relationship.*

7

Control is returned to the user in the usual way.

The routine in [Figure 137 on page 520](#) uses functions supplied in SCEESPC to load or locate the server code and initialize its environment.

```

* this is an example of a server message stub
QMGET  TITLE 'SERVER supplied stub for feeding strings GET'
QMGET  CSECT
QMGET  AMODE ANY
QMGET  RMODE ANY
QMGET  STM R14,R12,12(R13) 1
      LR R3,R15
      USING QMGET,R3
      LR R5,R1
      USING INPARMS,R5
      L R6,HANDLE@
      L R6,0(R6) Point to the handle 2
      L R1,4(R6) Point to work area got by QMGINIT 3
      USING WA,R1
      ST R13,SA+4 Keep savearea passed into us
      LR R13,R1 WA is new savearea
      USING WA,R13
      LA R7,GET 4
      LA R8,INPARMS+8 User parms start at 3rd
      STM R6,R8,PLIST handle, GET, Other parms
      LA R1,PLIST
      L R15,=V(EDCXSrvn) 5
      BALR R14,R15
      L R1,FEEDBK@ 6
      ST R15,0(R1)
      L R13,4(R13) 7
      L R14,12(R13)
      LM R0,R12,20(R13)
      BR R14
INPARMS DSECT
HANDLE@ DS F
FEEDBK@ DS F
LENGTH@ DS F
STRING@ DS F
WA DSECT
SA DS 18F
PLIST DS 4F
WALEN EQU *-WA
LIFO EQU 1
FIFO EQU 2
GET EQU 3
TERM EQU -1
YREGS
END

```

Figure 137. Example of server message stub-GET

1

Like the initialize stub, the QMGET message stub expects a standard save area pointed to by register 13. The parameters are passed with standard OS linkage (register 1 pointing to a list of addresses).

2

The *handle* contains the value that was placed there by the initialization stub at 8 Figure 134 on page 516. This is the address of the control block that is used to manage the interface between the user application and the server.

3

Recover the address of the stub work area for use as a Dynamic Storage Area (DSA). This value was saved here by the initialization stub at 7 Figure 134 on page 516. The save area back chain field is set according to usual conventions.

4

A parameter list consisting of the handle (as returned by EDCXSrvI at 5 Figure 134 on page 516. in the initialization stub), code for GET, and the address of the remaining parameters.

5

Call EDCXSrvN to *re-awaken* the server. This causes the server to resume control at 9 in Figure 133 on page 512 in the server. The server has control until it asks for the next *to-do*, in this example at 9 in Figure 133 on page 512, again.

6

The value passed to `__xsrv()` appears as the return code from EDCXSrvN. This value is passed back to the user application in the second parameter. *This is part of the API defined by this particular server, not something inherent in the user-server relationship.*

7

Control is returned to the user in the usual way.

The routine in Figure 138 on page 521 uses functions supplied in SCEESPC to load or locate the server code and initialize its environment.

```

* this is an example of a server message stub
QMGTERM  TITLE 'SERVER supplied stub for feeding strings TERM'
QMGTERM  CSECT
QMGTERM  AMODE ANY
QMGTERM  RMODE ANY
STM      R14,R12,12(R13)  1
LR       R3,R15
USING    QMGTERM,R3
LR       R5,R1
USING    INPARMS,R5
L        R6,HANDLE@
L        R6,0(R6)         Point to the handle  2
L        R1,4(R6)         Point to work area got by QMGINIT  3
USING    WA,R1
ST       R13,SA+4         Keep savearea passed into us
LR       R13,R1           WA is new savearea
USING    WA,R13
ST       R6,PLIST         Store handle as first parameter  4
MVC      PLIST+4,=A(TERM) Code for termination
LA       R1,PLIST
L        R15,=V(EDCXSRVN) 5
BALR     R14,R15
L        R13,4(R13)       6
L        R14,12(R13)
LM       R0,R12,20(R13)
BR       R14
INPARMS  DSECT
HANDLE@  DS    F
FEEDBK@  DS    F
LENGTH@  DS    F
STRING@  DS    F
WA        DSECT
SA        DS   18F
PLIST     DS   4F
WALEN     EQU  *-WA
LIFO      EQU   1
FIFO      EQU   2
GET       EQU   3
TERM      EQU  -1
YREGS
END

```

Figure 138. Example of server message stub-TERM

1

Like the initialize stub, the QMGTERM message stub expects a standard save area pointed to by register 13. The parameters are passed with standard OS linkage (register 1 pointing to a list of addresses).

2

The *handle* contains the value that was placed there by the initialization stub at **8** in Figure 134 on page 516. This is the address of the control block that is used to manage the interface between the user application and the server.

3

Recover the address of the stub work area for use as a Dynamic Storage Area (DSA). This value was saved here by the initialization stub at **7** in Figure 134 on page 516. The save area back chain field is set according to usual conventions.

4

A parameter list consisting of the handle (as returned by EDCXSRVI at **5** in Figure 134 on page 516), code for TERM, and the address of the remaining parameters.

5

Call EDCXSRVN to *re-awaken* the server. This causes the server to resume control at **9** in Figure 133 on page 512 in the server. The server has control until it asks for the next *to-do*, in this example at **9** in Figure 133 on page 512, again.

6

Control is returned to the user in the usual way.

The routines in the following section are used to create and use a persistent C environment for a server co-routine, written using z/OS XL C and EDCXSTRT, or EDCXSTRL and callable by a user application written in *any* language.

An initialization routine, EDCXSRVI, is called to start up a *server*. Control returns from the initialization call with the server code started and waiting for work.

As with the persistent C environment, the initialization call returns a *handle* that is used by EDCXSRVN for further communication with the created environment. EDCXSRVN suspends the execution of the calling routine and sends a message to the waiting server. When the server completes the function called for by the message its execution is suspended and the caller of EDCXSRVN resumes.

The server environment is terminated when a *Terminate* message is sent to the server.

Establishing a server environment

The EDCXSRVI routine creates a z/OS XL C environment for the server part of user-server application. It is intended that this routine be called by a stub routine supplied by the server and statically bound with the user application. The stub routine is responsible for loading the server application code. EDCXSRVI has the following parameters:

1. The address of the entry point of the server code. This must be the address of the EDCXSTRT or EDCXSTRL entry point.
2. The value to be in R1 when the server entry point is called. This can be used for communication between the initialization stub and the server mainline; its value can be retrieved in the server code. `__xregs(1)` will return a pointer to this list of parameters.
3. The address of a low-level get-storage routine (meeting the same interface as EDCXGET, but not necessarily EDCXGET).
4. The address of a low-level free-storage routine (meeting the same interface as EDCXFREE, but not necessarily EDCXFREE).

When this routine returns, the server environment is fully established and waiting for a message from the user. R15 points to a *handle* that is used in subsequent calls to EDCXSRVN to send messages to the server.

Initiating a server request

The EDCXSRVN routine is used by the stub routines that are linked with user application routines to send a message to an active server in a user-server application. EDCXSRVN has the following parameters:

1. The address of the handle returned by EDCXSRVI.
2. The function code for the function to be performed. The value -1 is used to indicate that the server should terminate. This value should not be used for any other purpose.
3. Other parameters, which are passed to the server code.

Upon return, R15 will contain the return code supplied by the server (as the parameter to EDCXSACC) for this service.

Accepting a request for service

The EDCXSACC routine operates in the server part of a user-server application. It is used to indicate acceptance or rejection of the last-requested service. EDCXSACC has the following parameter:

1. The return code of the last-requested service 0 indicating that the request was accepted and will be processed.

For more information on EDCXSACC, see “[__xsacc\(\) — Accept Request for Service](#)” on page 534.

Returning control from service

The EDCXSRVC routine operates in the server part of a user-server application. It is used to indicate completion of the last-requested service and to get information required for the next service to be performed. For more information on EDCXSRVC, see “[__xsrv\(\) — Return Control from Service](#)” on page 534. EDCXSRVC has the following parameter:

1. The return code for the last-requested service.

Constructing user-server stub routines

Part of building a server for use in a user-server environment is the construction of stub routines that load and initialize the server, pass messages to the server, and terminate the server. These stub routines are typically written in assembler language to allow them to be freely called from other environments without regard to the characteristics of the calling environment.

Building user-server environments

To build your server application, follow the rules for building a freestanding application as described in “[Building freestanding applications to run under z/OS](#)” on page 498.

There are no special considerations for building user applications. The automatic call facility will cause the correct routines from CEE.SCEESPC to be included.

Table 97. Parts used by or with application server routines				
Part Name	Function	Inclusion in Program		Location
		Notes		
EDCXSRVI	Used by a server-supplied stub routine to start up a server.	2	in the user module	Member of SCEESPC
EDCXSRVN	Used by a server-supplied stub routine to send a service-request message to a server.	2	in the user module	Member of SCEESPC
EDCXSRVC	Used by a server to wait for the next message to process.	2	in the user module	Member of SCEESPC
EDCXSAAC	Used by a server to accept the last message received.	2	in the user module	Member of SCEESPC
EDCXSPRT	System programming version of <code>sprintf()</code> .	3		Member of SCEESPC
EDCXEXIT	System programming version of <code>exit()</code> .	3		Member of SCEESPC
EDCXMEM	System programming version of <code>malloc()</code> , <code>calloc()</code> , <code>realloc()</code> , <code>free()</code> , <code>__4kmalc()</code> and <code>__24malc()</code> .	3		Member of SCEESPC
Notes:				
1. This module must be explicitly included in the program using the binder <code>INCLUDE</code> control statement.				
2. This module will normally be included by automatic call.				
3. This module must be explicitly included if you want to use the system programming version of the function.				

Tailoring the system programming C environment

Depending on the environment under which you want to run your z/OS XL C routines, you might want to replace some of the following routines for system-specific routines. To work correctly, your routines should match the interface as documented in this section. The routines as supplied by IBM with z/OS XL C meet the interface as documented.

Generating abends

The EDCXABND routine is called to generate an abend if there is an internal error during initialization or termination of a system programming C environment. This module must have the entry point name of @@XABND. It uses the following parameter:

R1

The address of the abend code and reason code

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4. [Figure 139 on page 524](#) shows an example.

```
* this is an example of a routine to generate an abend
@@XABND TITLE 'Generate an Abend'
EDCXABND CSECT
EDCXABND AMODE ANY
EDCXABND RMODE ANY
@@XABND DS 0H
ENTRY @@XABND
BALR R2,0
USING *,R2
SPACE 1

*
USING PARMs,R1
L R4,REAS_RC      get reason code
L R2,ERROR_RC     get error code
DROP R1,R2
ABEND ABEND (R2),REASON=(R4)
*
LTORG
EJECT
PARMS DSECT
ERROR_RC DS F
REAS_RC DS F
*
R1 EQU 1
R2 EQU 2
R3 EQU 3
R4 EQU 4
END
```

Figure 139. Example of routine to generate abend

Getting storage

The EDCXGET routine is called to get storage from the operating system. The entry point name for this routine must be @@XGET; see [Figure 140 on page 525](#) for an example. It uses the following parameter:

R0

The requested length, in bytes. If the high-order bit is zero or if the request was made in 24-bit addressing mode, the storage will be allocated below the 16M line. If the high-order bit is on and the request is made in 31-bit addressing mode, storage will be allocated anywhere with a preference for storage above the 16M line if available.

Upon return, the following values are set:

R0

The length of the storage block acquired, in bytes.

R1

The address of the acquired area or NULL.

R15

A system dependent return code, which must be zero on success and non-zero otherwise.

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

If you provide your own EDCXGET routine, it will be used when C library functions explicitly get storage. Whenever the library functions invoke operating system services, there may be implicit requests for storage that cannot be tailored.

```
* this is an example of a routine to get storage
@@XGET  TITLE  'Obtain memory as specified in R0'
EDCXGET  CSECT
EDCXGET  AMODE  ANY
EDCXGET  RMODE  ANY
@@XGET  DS      0H
        ENTRY  @@XGET
        SPACE  1
        BALR   R2,R0
        USING  *,R2
        LTR    R0,R0           Memory above or below?
        BNL    BELOW
        SLL    R0,1           Want memory anywhere
        SRL    R0,1
        LTR    R2,R2           are we running above the line?
        BNL    BELOW         no, so ignore above request
        GETMAIN RC,SP=0,LV=(R0),LOC=ANY
        LTR    R15,R15        Was it successful?
        BZR    R14            Yes...
        SR     R1,R1          No, indicate failure
        BR     R14
BELOW    DS      0H           Get memory below the line
        GETMAIN RC,SP=0,LV=(R0),LOC=BELOW
        LTR    R15,R15        Was it successful?
        BZR    R14            Yes...
        SR     R1,R1          no, indicate failure in R1
        BR     R14
*
R0       EQU    0
R1       EQU    1
R2       EQU    2
R4       EQU    4
R13      EQU    13
R14      EQU    14
R15      EQU    15
```

Figure 140. Example of routine to get storage

Getting page-aligned storage

The EDCX4KGT routine is called to get page-aligned storage from the operating system. Its entry point must be @@X4KGET. It has the following parameter:

R0

The requested length, in bytes. If the high-order bit of this register is zero or if the request was made in 24-bit addressing mode, the storage is allocated below the 16M line. If the high-order bit is on and the request is made in 31-bit addressing mode, storage is allocated above the 16M line. If this space is not available, storage is allocated elsewhere.

Upon return, the following values are set:

R0

The length of the storage block acquired, in bytes. This length may be greater than the size requested.

R1

The address of the acquired area or NULL.

R15

A system-dependent return code, which must be zero on success and nonzero otherwise.

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

Freeing storage

The EDCXFREE routine is called to return storage to the operating system. Its entry point must be @@XFREE. It uses the following parameters:

R0

The length of storage to be freed, in bytes

R1

The address of the area to be freed

Upon return, the following value is set.

R15

A system-dependent return code, which must be zero on success and nonzero otherwise

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

If you provide your own EDCXFREE routine, it will be used when C library functions explicitly free storage. Whenever the library functions invoke operating-system services, there may be implicit requests to free storage that cannot be tailored.

Figure 141 on page 526 shows an example of a routine that is used to free storage.

```
* this is an example of a routine to free storage
EDCXFREE CSECT
EDCXFREE AMODE ANY
EDCXFREE RMODE ANY
@@XFREE DS 0H
        ENTRY @@XFREE
        BALR R2,0
        USING *,R2

*
        FREEMAIN RC,SP=0,LV=(0),A=(1)
        BR      R14          return

*
R2      EQU     2
R14     EQU     14
END
```

Figure 141. Example of routine to free storage

Loading a module

The EDCXLOAD routine is called to load a named module into storage. Its entry point must be @@XLOAD. It has the following parameter:

R1

Points to the name of the routine to be loaded

On return, the following values are set.

R1

the address and amode of the routine or 0

R15

A system-dependent return code, which must be zero on success and nonzero otherwise

This routine *is* provided with a save area. Apart from the linkage registers, it must save and restore all registers used.

Deleting a module

The EDCXUNLD routine is called to delete a named module from storage. Its entry point must be @@XUNLD. It has the following parameter:

R1

Points to the name of the routine to be deleted

Upon return, the following values is set.

R15

A system-dependent return code, which must be zero on success and nonzero otherwise

This routine *is* provided with a save area. Apart from the linkage registers, it must save and restore all registers used.

Including a runtime message file

When you are running a freestanding environment and runtime messages are required, you must explicitly include a message file at link-edit time. One of the three following modules can be included to produce these messages:

EDCXLANE

Creates runtime error messages in uppercase and lowercase English

EDCXLANU

Creates runtime error messages in uppercase English

EDCXLANK

Creates runtime error messages in Kanji

If one of these message routines is not included and an exception occurs, the program could terminate without displaying a message. These error messages are directed to `stderr`. Refer to [z/OS Language Environment Debugging Guide](#) for more information.

Table 98 on page 527 contains the abend codes and reason codes specific to the system programming facilities.

Table 98. Abend and reason codes specific to system programming environments

Code Type	Code	Description
Abend	2100	No storage abend code
	2101	Error freeing storage
	2102	Error finding stack seg home
	2103	Error loading library
	2104	Error with heap allocation
	2105	Error with system level command
	2106	Error initializing statics
	2107	Error establishing error handler for EDCXSTRX
	2108	Error cleaning up heap for EDCXSTRX
Reason	4000	Error when handling abend
	7201	Error in initialization.
	7202	Error in termination.
	7203	Error when extending stack.
	7204	Error during longjmp/setjmp.
	7205	Can not locate static init. The routine EDCRCINT must be included in your module if you use the RENT compiler option.
	7206	Module EDCXABRT was not explicitly included at link edit time.
	7207	No initial heap allocation is specified and a heap is required.

Additional library routines

The following routines provide additional support that is unique to applications running in a system programming C environment. These routines are packaged as part of the link library. For more information on these routines refer to [Chapter 43, “Library functions for system programming C,”](#) on page 531.

--xregs()
Get registers on entry

--xusr()
Get address of User Word

--xusr2()
Get address of User Word

--4kmalc()
Allocate page-aligned storage

--24malc()
Allocate storage below 16mb line

Summary of application types

Table 99 on page 528 shows the summary of application types, how they are called, and the module entry points.

Table 99. Summary of types

Type of Application	How It Is Called	Module Entry Point	Data Sets Required at Execution Time	Runtime Options (1) and Other Considerations
A mainline function that requires no dynamic library facilities	From the command line, JCL, or an EXEC or CLIST.	EDCXSTRT, which must be explicitly included at bind time	None.	Runtime options are specified by #pragma runopts in compilation unit for the main() function. The heap and stack options are honored. The stack defaults to be above the line.
A mainline function that requires the z/OS XL C library functions	From the command line, JCL, or an EXEC or CLIST.	EDCXSTRL, which must be explicitly included at bind time	CEE.SCEERUN is required	Runtime options are specified by #pragma runopts in the compile unit for the entry point. The heap and stack options are honored, except that the stack will default to be above the line. The SPIE option is honored if a library is called for.
A C subroutine called from assembler language using a pre-established persistent environment	A <i>handle</i> , the address of the subroutine and a parameter list are passed to EDCXHOTU.		CEE.SCEERUN is optional, depending upon the way the <i>handle</i> was set up.	Runtime options are specified by #pragma runopts in any compile unit. The heap and stack options are honored, except that the stack will default to be above the line. The SPIE option is honored if a library is called for. The runopts in the first object module in the link edit that contains runopts will prevail, even if this compilation unit is part of the calling application. The environment is established by calling EDCXHOTC (or EDCXHOTL if library facilities are required). These functions return a value (the <i>handle</i>) which is used to call functions that use the environment.

Table 99. Summary of types (continued)

Type of Application	How It Is Called	Module Entry Point	Data Sets Required at Execution Time	Runtime Options (1) and Other Considerations
A Server	User code includes a stub routine that calls EDCXSRVI. This causes the server to be loaded and control to be passed to its entry point.	EDCXSTRT, or EDCXSTRL, depending upon whether the server needs the C runtime library or not	CEE.SCEERUN if required by the server code.	Runtime options are the same as for EDCXSTRL or EDCXSTRT. The author of the server must supply stub routines which call EDCXSRVI and EDCXSRVN to initialize and communicate with the server. These are bound with the user application.
A User of an Application Server			The server and CEE.SCEERUN if required by the server.	The author of the server must supply stub routines which call EDCXSRVI and EDCXSRVN to initialize and communicate with the server.

Chapter 43. Library functions for system programming C

This chapter describes the library functions specific to the System Programming C environment:

- `__xhotc()`
- `__xhotl()`
- `__xhott()`
- `__xhotu()`
- `__xregs()`
- `__xsacc()`
- `__xsrv()`
- `__xusr()`
- `__xusr2()`
- `__24malc()`
- `__4kmalc()`

`__xhotc()` — Set Up a Persistent C Environment (No Library)

Purpose

The function creates a persistent C environment that does not require the dynamic library facilities of z/OS Language Environment at run time.

For an extensive example of the use of `__xhotc()`, see [“Creating and using persistent C environments”](#) on page 505.

Format

```
#include <spc.h>

void *__xhotc(void *handle, int stack, int location);
```

The parameters are fullwords (four bytes).

handle

the field for the token (or handle) which is returned

stack

initial stack allocation required for the environment

location

location of the stack:

0

Below the line

1

Above the line

Returned value

`__xhotc()` returns a token (or handle) which is used in subsequent calls to `__xhotu()` and `__xhott()` to use or terminate a persistent C environment. This handle is found in both the first parameter passed and R15.

The RENT compiler option is not supported for routines called using this environment.

`__xhotl()` – Set Up a Persistent C Environment (With Library)

Purpose

The function creates a persistent C environment that will use the dynamic z/OS XL C/C++ library functions. All library facilities are available in this environment except:

- The RENT compiler option is not supported in the persistent environment described in this chapter.
- Exception handling is not supported in persistent C environments.

For an extensive example of the use of `__xhotl()`, see [“Creating and using persistent C environments” on page 505](#).

Format

```
#include <spc.h>

void *__xhotl(void *handle, int stack, int location);
```

The following parameters are fullwords (four bytes):

handle

the field for the token (or handle) which is returned

stack

the initial stack allocation required for the environment

location

location of the stack:

0

Below the line

1

Anywhere

Returned value

This routine returns a token (or handle) which is used in subsequent calls to `__xhotu()` and `__xhott()` to use or terminate a persistent C environment. This handle is found in both the first parameter passed and R15.

`__xhott()` – Terminate a Persistent C Environment

Purpose

This function terminates a persistent C environment created by `__xhotc()` or `__xhotl()`. `__xhott()` is specific to SP C. It is part of the group serving the persistent C environment. (The function is also available under the name EDCXHOTT.)

For an extensive example of the use of `__xhott()`, see [“Creating and using persistent C environments” on page 505](#).

Format

```
#include <spc.h>

void __xhott(void *handle);
```

The parameter of `__xhott()` is a handle returned by `__xhotc()` or `__xhotl()`.

__xhotu()

Purpose

This function is used to run a function in a persistent C environment. The function is also available under the name EDCXHOTU.

This routine, and the C function being called, must use OS linkage. As a result, you cannot make direct use of z/OS XL C/C++ Library functions with this function. C functions being invoked using `__xhotu()` must be compiled with `#pragma linkage(func_name, OS)`.

`__xhotu()` is specific to SP C. It is part of the group serving the persistent C environment.

For an extensive example of the use of `__xhotu()`, see [“Creating and using persistent C environments” on page 505](#).

Format

```
#include <spc.h>

void *__xhotu(void *handle, void *function, ...);
```

The parameters are fullwords (four bytes):

handle

a handle—returned by `__xhotc()` or `__xhotl()`

function

a function pointer, which points to the desired C function

- First parameter to pass to the function
- Second parameter to pass to the function

Returned value

The returned value from `__xhotu()` is the returned value from the function run in the persistent C environment.

__xregs() — Get Registers on Entry

Purpose

This routine finds the value a specified register had on entry to EDCXSTRT, EDCXSTRL, EDCXSTRX, or the *main* routine of an exit routine compiled with `#pragma environment(...)`.

`__xregs()` is available in these environments only. For more information about EDCXSTRT, EDCXSTRL, or EDCXSTRX, see [“Creating freestanding applications” on page 496](#).

`__xregs()` is specific to SP C. It is part of the client-server group of functions.

The function is also available under the name EDCXREGS.

Format

```
#include <spc.h>

int __xregs(int register);
```

Returned value

`__xregs()` returns the value found.

`__xsacc()` — Accept Request for Service

Purpose

This routine operates in the server part of a user-server application. It is used to indicate acceptance or rejection of the last-requested service. The function is also available under the name EDCXSACC.

Calls to `__xsacc` are optional but, if made, should be when the request is validated and all server references to user-owned storage are complete. `__xsacc` does not cause a return of control to the user; its sole purpose is to indicate that user-owned storage is no longer required by the application server.

In the case of a request that cannot be processed, possibly because the user's command is not recognized by the server or the parameter format is invalid, the call to `__xsacc` should be omitted.

`__xsacc()` is specific to SP C. It is part of the client-server group of functions.

Format

```
#include <spc.h>

void __xsacc( int message );
```

Returned value

The return code for the last-requested service, zero indicating that the request was accepted and will be processed.

`__xsrv()` — Return Control from Service

Purpose

This routine operates in the server part of a user-server application. It is used to indicate completion of the last-requested service and to get the information required for the next service to be performed. The function is also available under the name EDCXSRVC.

`__xsrv()` is specific to SP C. It is part of the client-server group of functions.

Format

```
#include <spc.h>

void *__xsrv(int message);
```

The *message* is the return code for the last-requested service.

__xusr() - __xusr2() — Get Address of User Word

Purpose

The `__xusr()` and `__xusr2()` functions are also available under the names `EDCXUSR` and `EDCXUSR2`, respectively. Two words in an internal control block are available for customer use. These words have an initial value of zero (that is, all bits are 0), but are otherwise ignored by compiled code, and by the z/OS XL C/C++-specific Library. The values in these words may be freely queried or set by application code using the pointers returned by these functions.

`__xusr()` and `__xusr2()` are specific to SP C.

Format

```
#include <spc.h>

void *__xusr(void);
void *__xusr2(void);
```

Returned value

`__xusr()` and `__xusr2()` return the addresses of these user words. The words, and `__xusr()` and `__xusr2()` themselves, are available in *any* environment, not only the system programming environments.

__24malc() — Allocate Storage below 16MB Line

Purpose

This function performs in the same manner as `malloc()` except that it allocates storage below the 16MB line in XA or ESA systems, even when the runtime option `HEAP (ANYWHERE)` is specified. Storage allocated by this function is not part of the heap, so you must free this storage explicitly using the `free()` function before this environment is terminated. Storage allocated using `__24malc()` is not automatically freed when the environment is terminated.

The function is available under the System Programming Environment.

Format

```
#include <spc.h>

void *__24malc(size_t size);
```

__4kmalc() — Allocate Page-Aligned Storage

Purpose

This function performs in the same manner as `malloc()`, except that it allocates page-aligned storage. Storage allocated by this function is not part of the heap, so you must free this storage explicitly using the `free()` function before this environment is terminated. Storage allocated using `__4kmalc()` is not automatically freed when the environment is terminated.

The function is available under the System Programming Environment.

Format

```
#include <spc.h>

void *_4kmalc(size_t size);
```

Chapter 44. Using runtime user exits

This chapter shows how to use runtime user exits with the z/OS Language Environment runtime library. This is general-use programming interface information and associated guidance information for using the library.

This section is provided here for your convenience. For further information on using runtime user exits in the z/OS Language Environment environment, refer to [z/OS Language Environment Programming Guide](#).

Note: Runtime user exits are not supported in AMODE 64 applications.

Using runtime user exits in z/OS Language Environment

z/OS Language Environment provides user exits that you can use for functions at your installation. You can use the assembler user exit (CEEBXITA) or the HLL user exit (CEEBINT). This section provides information about using these runtime user exits.

Note: You cannot code either the CEEBXITA user exit or the CEEBINT user exit as an XPLINK application.

Understanding the basics

User exits are invoked under z/OS Language Environment to perform enclave initialization functions and both normal and abnormal termination functions. User exits offer you a chance to perform certain functions at a point where you would not otherwise have a chance to do so. In an assembler initialization user exit, for example, you can specify a list of runtime options that establish characteristics of the environment. This is done before the actual execution of any of your application code. Another example is using an assembler termination user exit to request a dump after your application has terminated with an abend.

In most cases, you do not need to modify any user exit to run your application. Instead, you can accept the IBM-supplied default versions of the exits, or the defaults as defined by your installation. To do so, run your application normally and the default versions of the exits are invoked. You may also want to read the sections [“User exits supported under z/OS Language Environment”](#) on page 537 and [“Order of processing of user exits”](#) on page 538, which provide an overview of the user exits and describe when they are invoked.

If you plan to modify either of the user exits to perform some specific function, you must link the modified exit to your application before running, as described in [“Using installation-wide or application-specific user exits”](#) on page 539. In addition, the sections [“Using the Assembler user exit”](#) on page 539 and [“High level language user exit interface”](#) on page 550 describe the respective user exit interfaces to which you must adhere to change an assembler or HLL user exit.

PL/I and C/370 compatibility

For more information on compatibility support for the IBMBXITA and IBMFXITA assembler user exits, see [“PL/I and C/370 compatibility”](#) on page 550. Refer to *IBM C/370 Library Version 2 Release 2 Programming Guide* or to *PL/I for MVS & VM Compiler and Run-Time Migration Guide* for information about the IBMBINT HLL user exit. IBMBINT is not available under C++.

User exits supported under z/OS Language Environment

z/OS Language Environment provides two user exit routines, one written in assembler and the other in an HLL. You can find sample jobs containing these user exits in the SCEESAMP sample library. The user exits supported by z/OS Language Environment are shown in [Table 100 on page 538](#).

Table 100. User exits supported under z/OS Language Environment		
Name	Type of User Exit	When Invoked
CEEBXITA	Assembler user exit	Enclave initialization Enclave termination Process termination
CEEBINT	HLL user exit. CEEBINT can be written in z/OS XL C, PL/I, z/OS Language Environment-conforming assembler, or in C++ (see restrictions in “Order of processing of user exits” on page 538).	Enclave initialization

Order of processing of user exits

The location and order in which user exits are driven for your application are summarized in [Figure 142 on page 538](#).

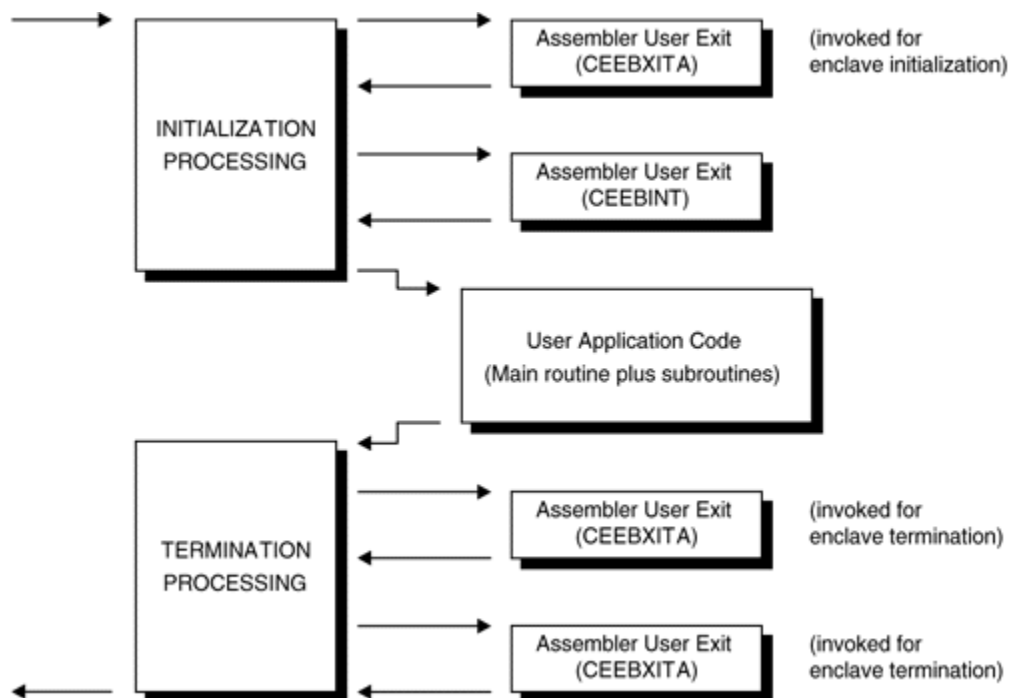


Figure 142. Location of user exits

In [Figure 142 on page 538](#), runtime user exits are invoked in the following sequence:

1. Assembler user exit is invoked for enclave initialization.

The assembler user exit (CEEBXITA) is invoked very early during the initialization process, before the enclave initialization is complete. Early invocation of the assembler exit allows the enclave initialization code to benefit from any changes that might be contained in the exit. If runtime options are provided in the assembler exit, the enclave initialization code is aware of the new options.

2. Environment is established.
3. HLL user exit is invoked.

The HLL initialization exit (CEEBINT) is invoked just before the invocation of the application code. In z/OS Language Environment, this exit can be written in z/OS XL C, PL/I, z/OS Language Environment-conforming assembler, or z/OS XL C++. However, you can only write CEEBINT in z/OS XL C++ if the following conditions are met:

- CEEBINT must be declared with C linkage. That is, it must be declared with `extern "C"`. If you are using C, you must compile your application code with the RENT compile-time option.
- You must bind your application code with the z/OS binder.
- CEEBINT must be used as an application-specific user exit, rather than as an installation-wide user exit (See [“Using installation-wide or application-specific user exits”](#) on page 539 for more information).
- The following information must be coded so that SMP/E can maintain the CSECT and properly link the intended user exit:

```
#pragma map(CEEEXIT, "CEEEXIT")
```

The HLL initialization exit *cannot* be written in COBOL, although COBOL applications can use this HLL user exit. At the time when CEEBINT is invoked, the runtime environment is fully operational and all z/OS Language Environment-conforming HLLs are supported.

4. Main routine is invoked.
5. Main routine returns control to caller.
6. Environment is terminated.
7. Assembler user exit is invoked for termination of the enclave.

CEEEXITA is invoked for enclave termination processing after all application code in the enclave has completed, but before any enclave termination activity.

8. Assembler user exit is invoked for termination of the process.

CEEEXITA is invoked again when the z/OS Language Environment process terminates.

Although both the assembler and HLL exits are invoked for initialization, they do not perform exactly the same functions. See [“CEEEXITA behavior during enclave initialization”](#) on page 540 and [“High level language user exit interface”](#) on page 550 for a detailed description of each exit.

z/OS Language Environment provides the CEEEXITA assembler user exit for termination but does not provide a corresponding HLL termination user exit.

Using installation-wide or application-specific user exits

IBM offers default versions of CEEEXITA and CEEBINT. You can use the IBM-supplied default version of either exit, or you can customize CEEEXITA or CEEBINT for use on an installation-wide basis. When CEEEXITA or CEEBINT is linked with the z/OS Language Environment initialization/termination library routines during installation, it functions as an installation-wide user exit.

Finally, you can customize CEEEXITA or CEEBINT yourself for use on your application. When CEEEXITA or CEEBINT is linked in your program, it functions as an application-specific user exit. The application-specific exit is used only when you run that application. The installation-wide assembler user exit is not executed.

To obtain an application-specific user exit, you must explicitly include it at bind time in the application using a binder INCLUDE control statement. Any time that the application-specific exit is modified, it must be relinked with the application.

The assembler user exit interface is described in [“Assembler user exit interface”](#) on page 541. The HLL user exit interface is described in [“High level language user exit interface”](#) on page 550.

Using the Assembler user exit

The assembler user exit CEEEXITA tailors the characteristics of the enclave before it is established. CEEEXITA must be written in assembler language because an HLL environment may not yet be established when the exit is invoked. CEEEXITA is driven for enclave initialization and enclave termination regardless of whether the enclave is the first enclave in the process or a nested enclave. CEEEXITA can differentiate easily between first and nested enclaves. For more information about nested enclaves, see [z/OS Language Environment Programming Guide](#).

CEEBXITA behaves differently depending on when it is invoked, as described in the following sections.

Using sample Assembler user exits

Sample assembler user exit programs are distributed with z/OS Language Environment. You can use them and modify the code for the requirements of your own application. Choose a sample program appropriate for your application. [Table 101 on page 540](#) lists the assembler exit user programs that are delivered with z/OS Language Environment.

Table 101. Sample Assembler user exits for z/OS Language Environment

Example User Exit	Operating System	Language (if Language Specific)
CEEBXITA	MVS (default)	
CEEBXITC	TSO	
CEECXITA	CICS (default)	
CEEBX05A	MVS	COBOL

Notes:

1. CEEBXITA and CEECXITA are the defaults on your system for MVS and CICS, if z/OS Language Environment is installed at your site without modification.
2. The source code for CEEBXITA, CEEBXITC, CEEDXITA, and CEEBX05A can be found on MVS in the sample library SCEESAMP.
3. CEEBX05A is an example user exit program for COBOL applications on z/OS.

CEEBXITA behavior during enclave initialization

The CEEBXITA assembler user exit is invoked before enclave initialization is performed. You can use it to help guide the establishment of the environment in which your application runs. For example, you can allocate data sets in the assembler user exit. The user exit can interrogate program parameters supplied in the JCL and change them if desired. In addition, you can specify runtime options in the user exit using the CEEAUE_OPTIONS field of the assembler interface (see [“Assembler user exit interface” on page 541](#) for information about how to do this).

CEEBXITA performs no special tasks other than to return control to z/OS Language Environment initialization.

CEEBXITA behavior during enclave termination

The CEEBXITA assembler exit is invoked after the user code for the enclave has completed, but before the occurrence of any enclave termination activity. For example, CEEBXITA is invoked before the storage report is produced (if one was requested), before data sets are closed, and before HLLs are invoked for enclave termination. In other words, the assembler user exit for termination is invoked when the environment is still active.

The assembler user exits allow you to request an abend. Under z/OS (as well as TSO and CICS), you can also request a dump to assist in problem diagnosis. Note that termination activities have not yet begun when the user exit is invoked. Thus, the majority of storage has not been modified when the dump is produced.

It is possible to request an abend and dump in the enclave termination user exit for all enclave-terminating events.

Example code that shows how to request an abend and dump when there is an unhandled condition of severity 2 or greater can be found in the member CEEBX05A in the sample library.

CEEBXITA behavior during process termination

The CEEBXITA assembler exit is invoked after:

- All enclaves have terminated.
- The enclave resources have been relinquished.
- Any z/OS Language Environment-managed files have been closed.
- Debug Tool has terminated.

This allows you to free files at this time, and it presents another opportunity to request an abend.

During termination, CEEBXITA can interrogate the z/OS Language Environment reason and return codes and, if necessary, request an abend with or without a dump. This can be done at either enclave or process termination.

The IBM-supplied CEEBXITA performs no special tasks other than to return control to z/OS Language Environment termination.

Specifying abend codes to be percolated by z/OS Language Environment

The assembler user exit, when invoked for initialization, can return a list of abend codes that are to be percolated by z/OS Language Environment. On non-CICS systems, this list is contained in the CEEAUE_A_AB_CODES field of the assembler user exit interface. (See “Assembler user exit interface” on page 541.) Both system abends and user abends can be specified in this list.

When TRAP(ON) is in effect, and the abend code is in the CEEAUE_A_AB_CODES list, z/OS Language Environment percolates the abend. Normal z/OS Language Environment condition handling is never invoked to handle these abends. This feature is useful when you do not want z/OS Language Environment condition handling to intervene for some abends, for example, when IMS issues abend code 777.

When TRAP(OFF) is specified, the condition handler is not invoked for any abends or program interrupts. The use of TRAP(OFF) is not recommended; refer to [z/OS Language Environment Programming Reference](#) for more information.

Actions taken for errors that occur within the Assembler user exit

If any errors occur during the enclave initialization user exit, the standard system action occurs because z/OS Language Environment condition handling has not yet been established.

Any errors occurring during the enclave termination user exit lead to abnormal termination (through an abend) of the z/OS Language Environment environment.

If a program check occurs during the enclave termination user exit and TRAP(ON) is in effect, the application ends abnormally with ABEND code 4044 and reason code 2. If a program check occurs during the enclave termination exit and "TRAP(OFF)" has been specified, the application ends abnormally without additional error checking support. z/OS Language Environment provides no condition handling; error handling is performed by the operating system. The use of TRAP(OFF) is not recommended; refer to [z/OS Language Environment Programming Guide](#) for more information.

z/OS Language Environment takes the same actions as described above for program checks during the process termination user exit.

Assembler user exit interface

You can modify CEEBXITA to perform any function desired, although the exit must have the following attributes after you modify it:

- The user-supplied exit must be named CEEBXITA.
- The exit must be reentrant.
- The exit must be capable of executing in AMODE(ANY) and RMODE(ANY).
- The exit must be relinked with the application after modification (if you want an application-specific user exit), or relinked with z/OS Language Environment initialization/termination routines after modification (if you want an installation-wide user exit).

If a user exit is modified, you are responsible for conforming to the interface shown in [Figure 143](#) on [page 542](#). This user exit must be written in assembler.

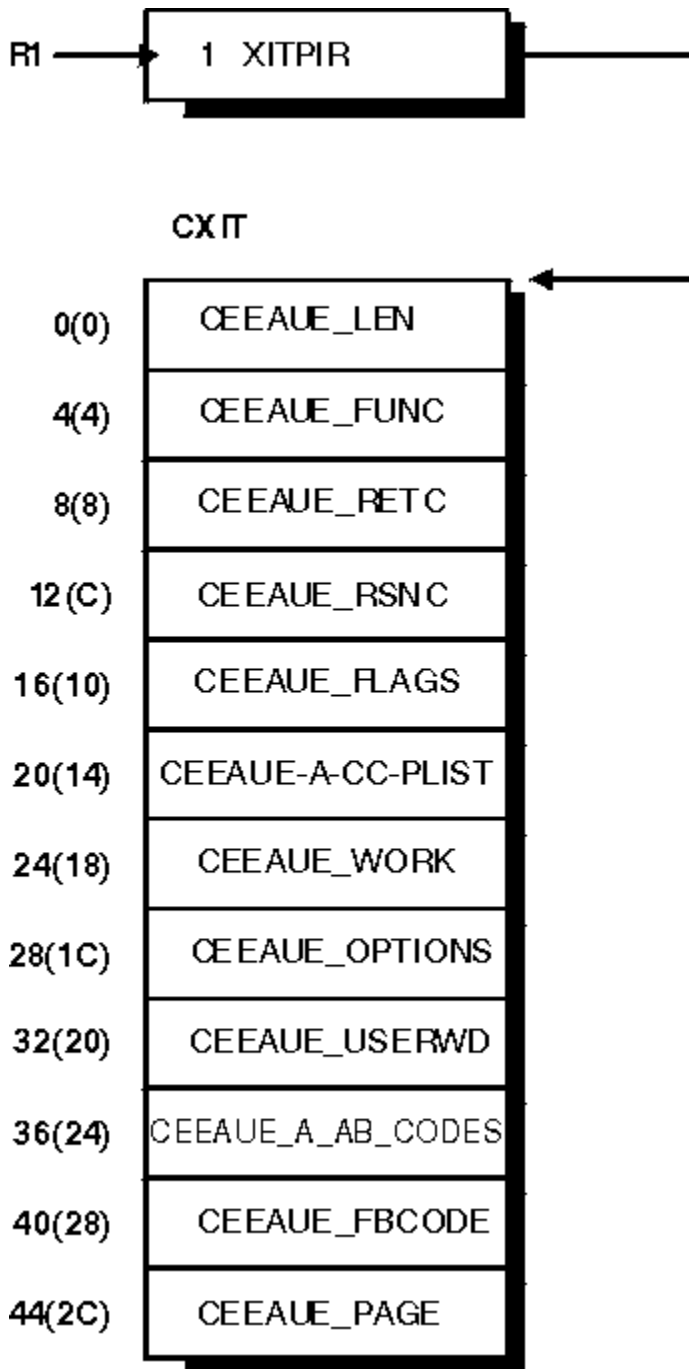


Figure 143. Interface for Assembler user exits

When the user exit is called, register 1 (R1) points to a word that contains the address of the CXIT control block. The high order bit is on. The CXIT control block contains the following fullwords:

CEEAUE_LEN (input parameter)

A fullword integer that specifies the total length of this control block. For z/OS Language Environment, the length is 48 bytes.

CEEAUE_FUNC (input parameter)

A fullword integer that specifies the function code. In z/OS Language Environment, the following function codes are supported:

- 1**
initialization of the first enclave within a process
- 2**
termination of the first enclave within a process
- 3**
nested enclave initialization
- 4**
nested enclave termination
- 5**
process termination

The user exit should ignore function codes other than those numbered from 1 through 5.

CEEAEU_RETC (input/output parameter)

A fullword integer that specifies the return or abend code. CEEAEU_RETC has different meanings depending on the flag CEEAEU_ABND:

- As an input parameter, this fullword is the enclave return code.
- As an output parameter, if the flag CEEAEU_ABND is on, this fullword is interpreted as an abend code that is used when an abend is issued. (This could be either an EXEC CICS ABEND or an SVC 13.)
- If the flag CEEAEU_ABND is off, this fullword is interpreted as the enclave return code that might have been modified by the exit.

See *z/OS Language Environment Programming Guide* for more information about how z/OS Language Environment computes return and reason codes.

CEEAEU_RSNC (input/output parameter)

A fullword integer that specifies the reason code for CEEAEU_RETC.

- As an input parameter, this fullword is the z/OS Language Environment return code modifier.
- As an output parameter, if the flag CEEAEU_ABND is on, CEEAEU_RETC is interpreted as an abend reason code that is used when an abend is issued. (This field is ignored when an EXEC CICS ABEND is issued.)
- If the flag CEEAEU_ABND is off, this fullword is the z/OS Language Environment return code modifier that might have been modified by the exit.

See *z/OS Language Environment Programming Guide* for more information about how z/OS Language Environment computes return and reason codes.

CEEAEU_FLAGS (input/output parameter)

Contains four flag bytes. CEEBXITA uses only the first byte but reserves the remaining bytes. All unspecified bits and bytes must be zero. The layout of these flags is shown in [Figure 144 on page 544](#).

Byte 0	x... .. - CEEAUE_ABTERM 0... .. - Normal termination 1... .. - Abnormal termination .x... .. - CEEAUE_ABND ..0... .. - Terminate with CEEAUE_RETC ..1... .. - Abend with CEEAUE_RETC and CEEAUE_RSNC given ..x... .. - CEEAUE_DUMP ..0... ..
Byte 1	00 - Reseved for future use
Byte 2	00 - Reseved for future use
Byte 3	00 - Reseved for future use

Figure 144. CEEAUE_FLAGS format

Byte 0 (CEEAE_FLAG1) has the following meaning:

CEEAE_ABTERM (input parameter)

When OFF, the enclave terminates normally (severity 0 or 1 condition).

When ON, the enclave terminates with the z/OS Language Environment return code modifier of 2 or greater. This could, for example, indicate that a condition of severity 2 or greater was raised that was unhandled.

CEEAE_ABND (output parameter)

When OFF, the enclave terminates without an abend. CEEAE_RETC and CEEAE_RSNC are placed in register 15 and register 0 and returned to the enclave creator.

When ON, the enclave terminates with an abend. Thus, CEEAE_RETC and CEEAE_RSNC are used by z/OS Language Environment in the invocation of the abend. While executing in CICS, an EXEC CICS ABEND command is issued.

CEEAE_RSNC is ignored under CICS. The TRAP option does not affect the setting of CEEAE_ABND.

CEEAE_DUMP (output parameter)

When OFF and you request an abend, an abend is issued without requesting a system dump.

When ON and you request an abend, an abend is issued requesting a system dump.

CEEAE_STEPS (output parameter)

When OFF and you request an abend, one is issued to abend the entire task. When ON and you request an abend, one is issued to abend the step.

Note: This fullword is ignored under CICS.

CEEAE-A-CC-PLIST (input/output parameter)

A fullword pointer to the parameter address list of the application program.

As an input parameter, this fullword contains the register 1 value passed to the main routine. The exit can modify this value, and the value is then passed to the main routine. If runtime options are present in the invocation command string, they are stripped off before the exit is called.

If the parameter inbound to the main routine is a character string, CEEAE-A-CC-PLIST contains the address of a fullword address that points to a halfword prefixed string. If this string is altered by the user exit, the string must not be extended in place.

CEEAE_WORK (input parameter)

Contains a fullword pointer to a 256-byte work area that the exit can use. On entry, it contains binary zeros and is doubleword-aligned. This area does not persist across exits.

CEEAE_OPTIONS (output parameter)

On return, this field contains a fullword pointer to the address of a halfword length prefixed character string that contains runtime options. These options are only processed for enclave initialization. When invoked for enclave termination, this field is ignored.

These runtime options override all other sources of runtime options except those that are specified as non-overrideable.

Under CICS, the STACK runtime option cannot be modified using the assembler user exit.

CEEAE_USERWD (input/output parameter)

Contains a fullword whose value is maintained without alteration and passed to every user exit. On entry to the enclave initialization user exit, it is zero. Thereafter, the value of the user word is not altered by z/OS Language Environment or any member libraries. The user exit can change the value of this field and z/OS Language Environment maintains this value. This allows a user exit to initialize the fullword and pass it to subsequent user exits.

CEEAE_A_AB_CODES (output parameter)

During the initialization exit, this field contains the fullword address of a table of abend codes that the z/OS Language Environment condition handler percolates while in the (E) STAE exit. Therefore, the application is not given the opportunity to field the abend. The table consists of:

- A fullword count of the number of abend codes that are to be percolated
- A fullword for each of the particular abend codes that are to be percolated

The abend codes can be user abend codes or system abend codes. User abend codes are specified by F'uuu'. For example, if you wanted user abend 777 to be percolated, an F'777' would be coded. System abend codes are specified by X'00sss000'. Avoid specifying the values 0C0 through 0CF as 'sss'. Language Environment ignores values between 0C0 and 0CF. No abend is percolated, and z/OS Language Environment condition handling semantics are in effect.

This function is not enabled under CICS.

CEEAE_FBCODE (input parameter)

Contains the fullword address of the condition token with which the enclave terminated. If the enclave terminates normally (that is, not because of a condition), the condition token is zero.

CEEAE_PAGE (input/output parameter)

Usage of this field is related to PL/I BASED variables that are allocated storage outside of AREAs. You can indicate whether storage should be allocated on a 4K-page boundary. You can specify the minimum number of bytes of storage that you want allocated. Your allocation request must be an exact multiple of 4K. The IBM-supplied default setting for CEEAE_PAGE is 32768 (32K).

If CEEAE_PAGE is set to zero, PL/I BASED variables can be placed on other than 4K-page boundaries.

CEEAE_PAGE is honored only during enclave initialization (that is, when CEEAE_FUNC is 1 or 3).

The offset of CEEAE_PAGE under z/OS Language Environment is different from the offset of IBMXITA under OS PL/I Version 2 Release 3.

Parameter values in the Assembler user exit

The parameters described in the following sections contain different values depending on how the user exit is used. Possible values are shown for the parameters based on how the assembler user exit is invoked.

First enclave within process initialization—entry

CEEAE_LEN

48

CEEAE_FUNC

1 (first enclave within process initialization function code).

CEEAE_RETC

0

CEEAE_RSNC

0

CEEAE_FLAGS

0

CEEAE-A-CC-PLIST

The register 1 value from the operating system.

CEEAE_WORK

Address of a 256-byte work area of binary zeros.

CEEAE_USERWD

0

CEEAE_FBCODE

0

CEEAE_PAGE

Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).

First enclave within process initialization—return

CEEAE_RETC

0, or if CEEAE_ABND = 1, the abend code.

CEEAE_RSNC

0, or if CEEAE_ABND = 1, the reason code for CEEAE_RETC.

CEEAE_FLAGS

CEEAE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

CEEAE_DUMP = 1 if the abend should request a dump.

CEEAE_STEPS = 1 if the abend should abend the step, or 0 if the abend should abend the task.

CEEAE-A-CC-PLIST

Register 1, used as the new parameter list.

CEEAE_OPTIONS

Pointer to the address of a halfword prefixed character string containing runtime options, or 0.

CEEAE_USERWD

Value of CEEAE_USERWD for all subsequent exits.

CEEAE_A_AB_CODES

Pointer to the abend code table, or 0.

CEEAE_PAGE

User-specified PAGE value. Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).

First enclave within process termination—entry

CEEAE_LEN

48

CEEAE_FUNC

2 (first enclave within process termination function code).

CEEAE_RETC

Return code issued by the application that is terminating.

CEEAE_RSNC

Reason code that accompanies CEEAE_RETC.

CEEAE_FLAGS

CEEAE_ABTERM = 1 if the application is terminating with the z/OS Language Environment return code modifier of 2 or greater, or 0 otherwise.

```
CEEAE_ABND = 0  
CEEAE_DUMP = 0  
CEEAE_STEPS = 0
```

CEEAE_WORK

Address of a 256-byte work area of binary zeros.

CEEAE_USERWD

Return value from the previous exit.

CEEAE_FBCODE

Feedback code causing termination.

First enclave within process termination—return

CEEAE_RETC

If CEEAE_ABND = 0, the return code placed in register 15 when the enclave terminates.

If CEEAE_ABND = 1, the abend code.

CEEAE_RSNC

If CEEAE_ABND = 0, the enclave reason code.

If CEEAE_ABND = 1, the abend reason code.

CEEAE_FLAGS

CEEAE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

CEEAE_DUMP = 1 if the abend should request a dump.

CEEAE_STEPS = 1 if the abend should abend the step, or 0 if the abend should abend the task.

CEEAE_USERWD

The value of CEEAE_USERWD for all subsequent exits.

Nested enclave initialization—entry

CEEAE_LEN

48

CEEAE_FUNC

3 (nested enclave initialization function).

CEEAE_RETC

0

CEEAE_RSNC

0

CEEAE_FLAGS

0

CEEAE-A-CC-PLIST

The register 1 value discovered in a nested enclave creation.

CEEAE_WORK

Address of a 256-byte work area of binary zeros.

CEEAE_USERWD

The return value from previous exit.

CEEAE_FBCODE

0

CEEAE_PAGE

Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).

Nested enclave initialization—return**CEEAE_RETC**

0, or if CEEAE_ABND = 1, the abend code.

CEEAE_RSNC

0, or if CEEAE_ABND = 1, the reason code for CEEAE_RETC.

CEEAE_FLAGS

CEEAE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

CEEAE_DUMP = 1 if the abend should request a dump.

CEEAE_STEPS = 1 if the abend should abend the step, or 0 if the abend should abend the task.

CEEAE-A-CC-PLIST

Register 1 used as the new parameter list.

CEEAE_OPTIONS

Pointer to a fullword address that points to a halfword prefixed string containing runtime options, or 0.

CEEAE_USERWD

The value of CEEAE_USERWD for all subsequent exits.

CEEAE_A_AB_CODES

Pointer to the abend code table, or 0.

CEEAE_PAGE

User-specified PAGE value. Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).

Nested enclave termination—entry**CEEAE_LEN**

48

CEEAE_FUNC

4 (termination function).

CEEAE_RETC

Return code issued by the enclave that is terminating.

CEEAE_RSNC

Reason code that accompanies CEEAE_RETC.

CEEAE_FLAGS

CEEAE_ABTERM = 1 if the application is terminating with the z/OS Language Environment return code modifier of 2 or greater, or 0 otherwise.

```
CEEAE_ABND   = 0
CEEAE_DUMP   = 0
CEEAE_STEPS   = 0
```

CEEAEU_WORK

Address of a 256-byte work area of binary zeros.

CEEAEU_USERWD

Return value from previous exit.

CEEAEU_FBCODE

Feedback code causing termination.

Nested enclave termination—return**CEEAEU_RETC**

If CEEAEU_ABND = 0, the return code from the enclave.

If CEEAEU_ABND = 1, the abend code.

CEEAEU_RSNC

If CEEAEU_ABND = 0, the enclave reason code.

If CEEAEU_ABND = 1, the enclave reason code.

CEEAEU_FLAGS

CEEAEU_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

CEEAEU_DUMP = 1 if the abend should request a dump.

CEEAEU_STEPS = 1 if the abend should abend the step, or 0 if the abend should abend the task.

CEEAEU_USERWD

Value of CEEAEU_USERWD for all subsequent exits.

Process termination—entry**CEEAEU_LEN**

48

CEEAEU_FUNC

5 (process termination function).

CEEAEU_RETC

Return code presented to the invoking system in register 15 that reflects the value returned from the first enclave within process termination.

CEEAEU_RSNC

Reason code accompanying CEEAEU_RETC that is presented to the invoking system in register 0 and reflects the value returned from the first enclave within process termination.

CEEAEU_FLAGS

CEEAEU_ABTERM = 1 if the last enclave is terminating abnormally (that is, the z/OS Language Environment return code modifier is 2 or greater). This reflects the value returned from the first enclave within process termination (function code 2).

CEEAEU_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing first enclave within process termination (function code 2).

CEEAEU_DUMP = 0

CEEAEU_STEPS = 0

CEEAEU_WORK

Address of a 256-byte work area of binary zeros.

CEEAEU_USERWD

The return value from previous exit.

CEEAEU_FBCODE

The feedback code causing termination.

Process termination—return

CEEAEU_RET

If CEEAEU_ABND = 0, the return code from the process.

If CEEAEU_ABND = 1, the abend code.

CEEAEU_RSNC

If CEEAEU_ABND = 0, the reason code for CEEAEU_RET from the process.

If CEEAEU_ABND = 1, reason code for the CEEAEU_RET abend reason code.

CEEAEU_FLAGS

CEEAEU_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

CEEAEU_DUMP = 1 if the abend should request a dump.

CEEAEU_STEPS = 1 if the abend should abend the step, or 0 if the abend should abend the task.

CEEAEU_USERWD

The value of CEEAEU_USERWD for all subsequent exits.

PL/I and C/370 compatibility

The following OS PL/I Version 2 Release 3 assembler user exits are supported for compatibility under z/OS Language Environment:

- IBMBXITA (MVS Batch version)
- IBMFXITA (CICS version)

For more information about IBMBXITA see *PL/I for MVS & VM Compiler and Run-Time Migration Guide*. These user exits are available only under C, not C++.

Default versions of the above exits are not supplied under z/OS Language Environment; instead, z/OS Language Environment supplies a default version of CEEBXITA. [Table 102 on page 550](#) describes the order of precedence if the IBMBXITA and IBMFXITA user exits are found in the same root program with CEEBXITA.

Table 102. Interaction of Assembler user exits

CEEBXITA Present	IBMBXITA Present under MVS Batch, IBMFXITA Present under CICS	Exit Driven
No	No	Default version of CEEBXITA
Yes	No	CEEBXITA
No	Yes	IBMBXITA under MVS Batch; IBMFXITA under CICS
Yes	Yes	CEEBXITA

CXIT_FUNC in IBMBXITA will map to CEEBXITA as follows:

- CXIT_FUNC = 1 when IBMBXITA is invoked for initial enclave initialization or nested enclave initialization
- CXIT_FUNC = 2 when IBMBXITA is invoked for initial enclave termination or nested enclave termination

CXIT_USERWD in IBMBXITA will persist across enclaves (for example, in system() calls).

High level language user exit interface

z/OS Language Environment provides CEEBINT, an HLL user exit, for enclave initialization. You can code CEEBINT in z/OS XL C, PL/I, or z/OS XL C++ (subject to the restrictions in [“Order of processing](#)

of user exits” on page 538), or z/OS Language Environment-conforming assembler. The HLL user exit cannot be written in COBOL. COBOL programmers can use an HLL exit written in z/OS XL C, PL/I, z/OS Language Environment-conforming assembler, z/OS XL C++ (again, subject to the restrictions in “Order of processing of user exits” on page 538), or default to the IBM-supplied default HLL user exit.

The HLL enclave initialization exit is invoked after the enclave has been established, after the Debug Tool initial command string has been processed, and prior to the invocation of compiled code. When invoked, it is passed a parameter list that conforms to the z/OS Language Environment definition. The parameters are all fullwords and are defined as follows:

Number of arguments in parameter list (input)

A fullword binary integer.

- On entry: Contains 7.
- On exit: Not applicable.

Return code (output)

A fullword binary integer.

- On entry: 0.
- On exit: Able to be set by the exit, but not interrogated by z/OS Language Environment.

Reason code (output)

A fullword binary integer.

- On entry: 0
- On exit: Able to be set by the exit, but not interrogated by z/OS Language Environment.

Function code (input)

A fullword binary integer.

- On entry: 1, indicating the exit is being driven for initialization.
- On exit: Not applicable.

Address of the main program entry point (input)

A fullword binary address.

- On entry: The address of the routine that gains control first.
- On exit: Not applicable.

User word (input/output)

A fullword binary integer.

- On entry: Value of the user word (CEEAE_USERWD) as set by the assembler user exit.
- On exit: The value set by the user exit, maintained by z/OS Language Environment and passed to subsequent user exits.

Exit List Address (output)

A fullword binary integer reserved for future use. This allows the establishment of one or more user exits when the enclave user exit sets this field to a list of user exits. Currently, only one user exit is supported in z/OS Language Environment.

A_Exits

The address of the exit list control block, `Exit_list`.

- On entry: 0.
- On exit: 0, unless you establish a hook exit, in which case you would set this pointer and fill in relevant control blocks. The control blocks for `Exit_list` and `Hook_exit` are shown in the following figure.

As supplied, CEEBINT has only one exit defined that you can establish: the hook exit described by the `Hook_exit` control block. This exit gains control when hooks generated by the PL/I compile-time TEST option are executed. You can establish this exit by setting appropriate pointers (`A_Exits` to `Exit_list` to `Hook_exit`). [Figure 145 on page 552](#) illustrates the `Exit_list` and `Hook_exit` control blocks.

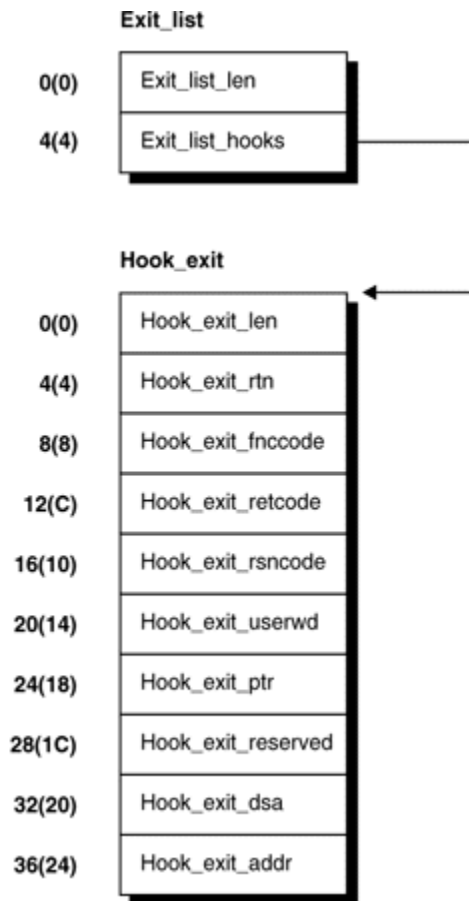


Figure 145. *Exit_list* and *hook_exit* control blocks

The control block *Exit_list* exit contains the following fields:

Exit_list_len

The length of the control block. It must be 1.

Exit_list_hooks

The address of the *Hook_exit* control block.

The control block for the hook exit must contain the following fields:

Hook_exit_len

The length of the control block.

Hook_exit_rtn

The address of a routine you want invoked for the exit. When the routine is invoked, it is passed the address of this control block. Because this routine is invoked only if the address you specify is nonzero, you can turn the exit on and off.

Hook_exit_fnccode

The function code with which the exit is invoked. This is always 1.

Hook_exit_retcode

The return code set by the exit. You must ensure it conforms to the following specifications:

- 0** Requests that Debug Tool be invoked next
- 4** Requests that the program resume immediately
- 16** Requests that the program be terminated

Hook_exit_rsncode

The reason code set by the exit. This is always zero.

Hook_exit_userwd

The user word passed to the user exits.

Hook_exit_ptr

An exit-specific user word.

Hook_exit_reserved

Reserved.

Hook_exit_dsa

The contents of register 13 when the hook was executed.

Hook_exit_addr

The address of the hook instruction executed.

Usage requirements

1. The user exit must not be a main-designated routine. For example, it cannot be a z/OS XL C or a z/OS XL C++ `main()` function.
2. The HLL exit routines must be linked with compiled code. If you do not provide an initialization user exit, an IBM-supplied default, which returns control to your application, is linked with the compiled code.
3. The exit cannot be written in COBOL/370.
4. The exit should be coded so that it returns for all unknown function codes.
5. z/OS XL C constructs such as the `exit()`, `abort()`, `raise(SIGTERM)`, and `raise(SIGABRT)` functions terminate the enclave.
6. A PL/I `EXIT` or `STOP` statement terminates the enclave.
7. Use the callable service `IBMHKS` to turn hooks on and off. For more information about `IBMHKS`, see *PL/I for MVS & VM Compiler and Run-Time Migration Guide*.
8. When `CEE Bint` is written in C/C++, the following information must be coded so that SMP/E can maintain the CSECT and properly link the intended user exit:

```
#pragma map(CEE Bint, "CEE Bint")
```

Chapter 45. Using the z/OS XL C MultiTasking Facility

This chapter describes how to use the MultiTasking Facility (MTF) with z/OS XL C. It explains how to organize, code, compile, link, and run a program using MTF. It also lists restrictions while using MTF.

MTF is a facility available under z/OS that can be used by application programs to improve turnaround time on multiprocessor and attached-processor configurations. When a program uses MTF on such a system, the elapsed time required to run the program can be reduced. You can run tasks, which can run independently of each other, simultaneously.

MTF is easy to use and requires very little knowledge of the multitasking capabilities upon which it depends. From the programmer's perspective, multitasking facilities are available through the library functions of z/OS XL C. Because of this simplicity, it is easy to introduce MTF to existing applications and code new MTF applications to gain the benefits of multitasking.

Notes:

1. Except for a few differences, the MTF support for z/OS XL C is the same as for the equivalent FORTRAN multitasking facilities. MTF is not supported under CICS, IMS, DB2, C++, or z/OS UNIX. In addition, IPA is not supported in an MTF environment.
2. XPLINK is not supported in an MTF environment.
3. AMODE 64 applications are not supported in an MTF environment.

Organizing a program with MTF

MTF takes advantage of the multitasking capabilities of the operating system to enable a single z/OS XL C application program to use more than one processor of a multiprocessing configuration simultaneously. The z/OS operating system organizes all work into units called *tasks*. These tasks are used by the operating system to assign work to the processors of the multiprocessor configuration.

MTF's facilities allow a single z/OS XL C application to be organized so it can be run in a *main task* and in one or more *subtasks*. As a result of this organization, the system can schedule these individual tasks to run simultaneously. This can significantly reduce the elapsed time needed to run the program.

When a program is organized in this manner, the main task runs the part of the program that controls the overall processing. This part is referred to as the *main task program* throughout this manual.

The subtasks run the portions of the program that can run independently of the main task program and of each other. These portions of the program are referred to as *parallel functions*. The library functions provided by MTF allow the main task program to schedule parallel functions and allow them to run independently. Parallel functions are queued for execution on the next available subtask. Scheduling a parallel function does not require that there be a free subtask at the time of the scheduling. MTF allows the main task program to schedule more parallel functions than there are actual MVS subtasks.

The parallel functions are coded the same way as normal C functions, with the exception of a few rules discussed in [“Designing and coding applications for MTF” on page 562](#). In particular, parallel functions cannot issue MTF calls.

MTF applications are link-edited as two separate load modules: a main task load module (containing the main task program) and a parallel load module (containing all parallel functions).

z/OS XL C provides the following MTF functions (for details, refer to [z/OS C/C++ Runtime Library Reference](#)):

- `tinit()` to initialize the MTF environment
- `tsched()` to schedule parallel functions to run
- `tsyncio()` to synchronize the completion of parallel functions
- `tterm()` to terminate all executing parallel functions.

z/OS XL C also provides the header file `mtf.h`, which must be included in your main task program if you are going to use the MTF facilities. The `mtf.h` header file contains the macros `MTF_ANY` and `MTF_ALL`, as well as the error-return codes and prototypes for library functions.

Ensuring computational independence

To use multitasking successfully, the parallel functions must have *computational independence*. This means that no data modified by either the main task program or a parallel function is examined or modified by a parallel function that might be running simultaneously.

Figure 146 on page 556 is a graphic example of hypothetical data in an array subscripted by I, J, and K. Each of the three divisions of the box represents a section of the array that can be operated on independently of the other sections. The same parallel function could be scheduled three times, with each instance of the function processing one of the three sections of the array.

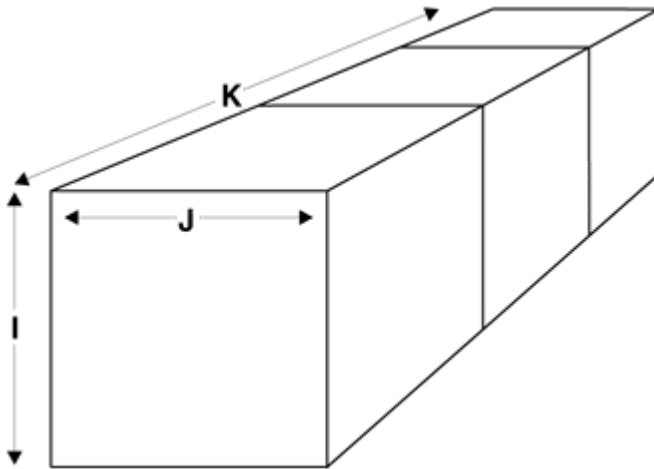


Figure 146. Computational independence

Your application may not have computational independence along the same subscript axis of K, as in this picture. The divisions might have been along one of the other subscript axes, I or J. Also, the computational independence in your application may not fall into neat, box-like divisions.

It is also possible to have computational independence that is not based on sections of the same array, but rather on separate arrays (perhaps with completely different types of data), the values of which do not depend on each other. In this case, separate parallel functions could be scheduled, with each function processing its own unique data.

Computational independence also applies to input/output files. One parallel function should not use a file while another is updating it. However, different functions can successfully read the same file. No single file pointer should be used concurrently by multiple parallel functions, because the behavior is undefined in such a case.

Running a C program without MTF

The following diagrams illustrate the way a z/OS XL C program runs without multitasking. The program and its functions must run in a strictly sequential manner, function following function, using one processor at a time. Consequently, your program takes more elapsed time to complete than it would if it could use several processors at the same time.

For example, as Figure 147 on page 557 shows, without multitasking, the z/OS XL C program and all its functions can only use one processor. While running, your program may be switched back and forth between the processors, but it can only run on one processor at a time.

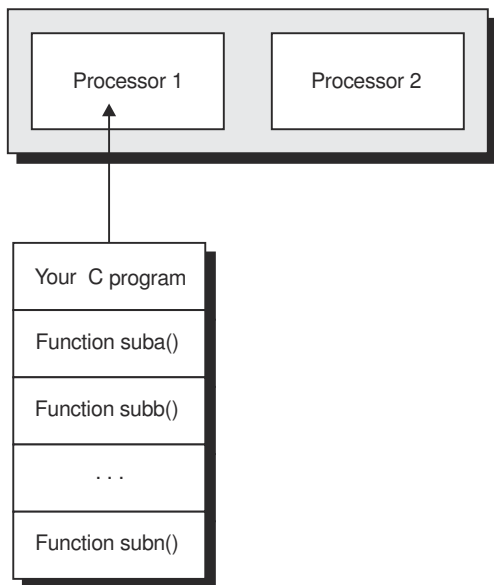


Figure 147. Example of a C program running without MTF

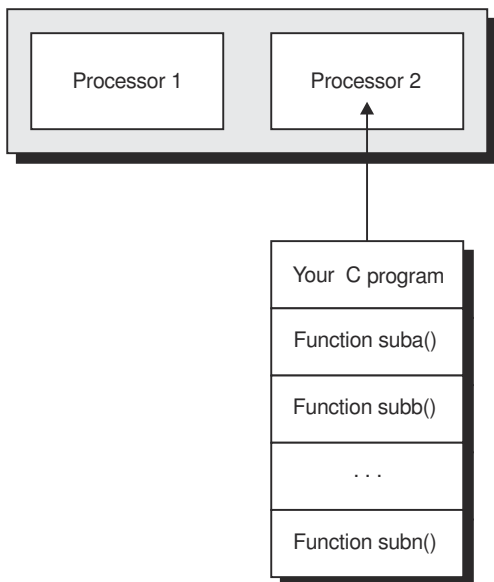


Figure 148. Example of a C program running without MTF (Part 2)

Running a C program with MTF

To illustrate the concept of multitasking, this section shows three examples of running a z/OS XL C program with MTF. These examples show programs using:

- One parallel function
- Two different functions
- Two or more instances of the same function

Each example provides an illustration of how the processors are used and how the program is organized to accomplish the particular use of the processors.

Running a C program with one parallel function

If your C program uses MTF, the main task program and a computationally-independent parallel function can run concurrently.

Processor use

In [Figure 149 on page 558](#), only the function `suba` has computations that can be done independently of the main task program, which includes the C main program plus its functions. Note that the arrows to Processor 1 and Processor 2 are for illustration only. The main task program could have run on Processor 2 and the parallel function, `suba`, on Processor 1; in fact, while they run, they may be switched between the processors.

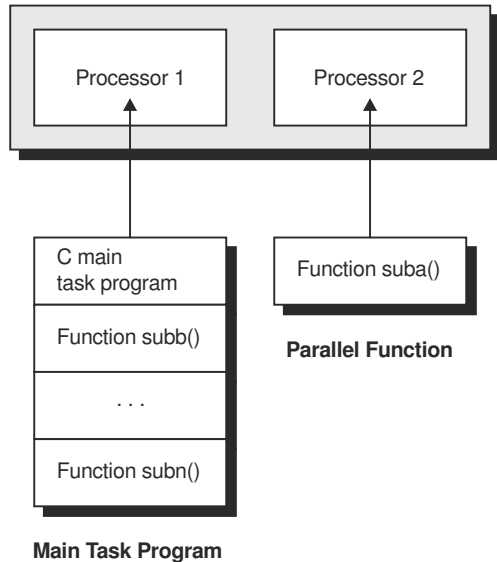


Figure 149. Processor usage with one parallel function

Sample program

With the appropriate MTF request, the parallel function, `suba`, is scheduled to run in a subtask. As [Figure 150 on page 559](#) shows, the MTF functions perform the following tasks:

- 1** `tinit()` names the parallel load module `plmod` and specifies one subtask.
- 2** `tsched()` schedules the parallel function `suba` to run. `suba` is computationally-independent of the main task.
- 3** At this point, `tsyncro()` makes the main task program wait until `suba` is finished before the main task program continues.

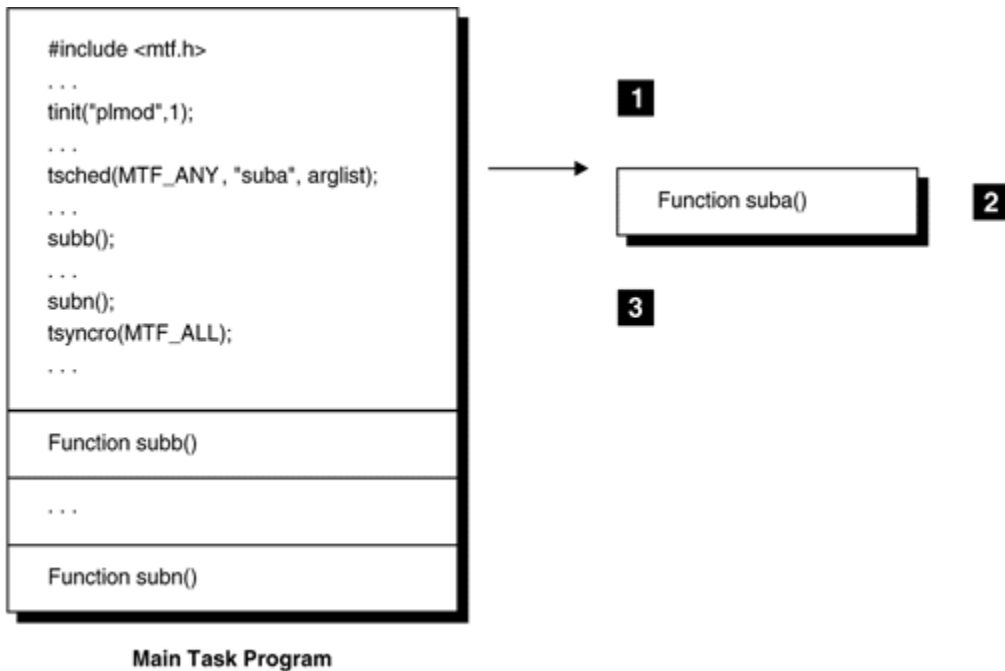


Figure 150. Sample program using one parallel function

Running a C program with two different parallel functions

If your C program uses MTF, the main task program and several different computationally-independent parallel functions can run concurrently.

Processor use

In Figure 151 on page 559, functions suba and subc are independent of each other and of the main task program. The arrows to Processors 1, 2, and 3 are for illustration only. The main task program and the parallel functions could run on any of the processors.

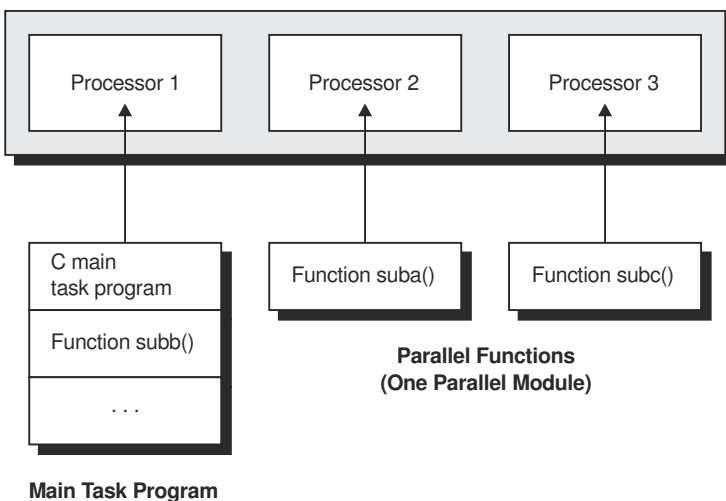


Figure 151. Processor usage with two parallel function

Sample program

Figure 151 on page 559 shows a sample program. The logic is similar to that for only one parallel function and can be extended to as many parallel functions as necessary to complete the logic of the program.

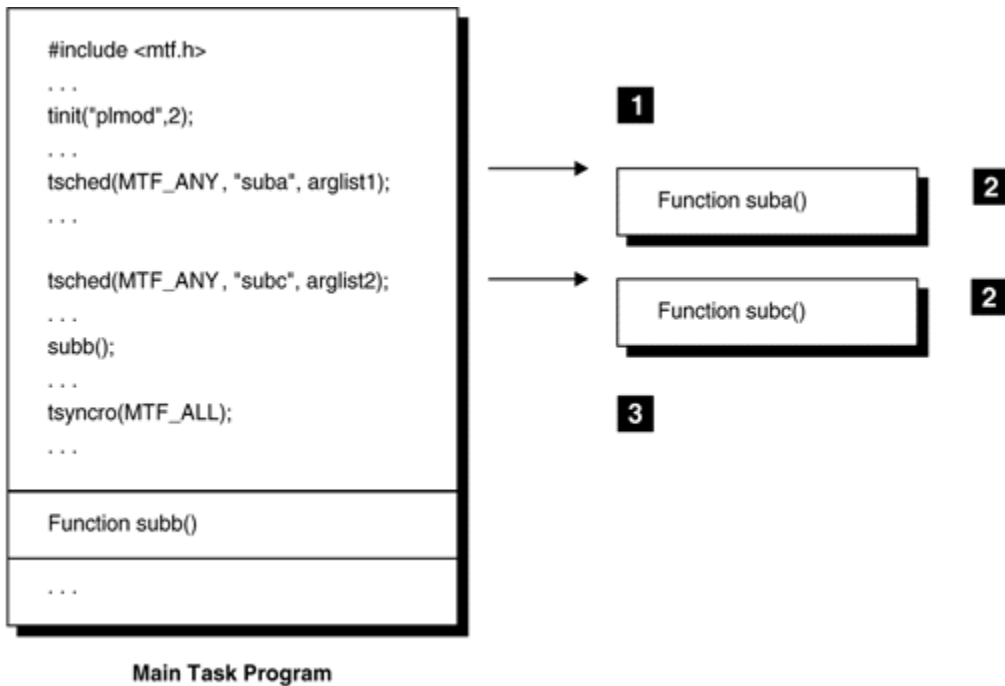


Figure 152. Sample program using two parallel functions

- 1 tinit() names the parallel load module plmod and specifies two subtasks.
- 2 Each call to tsched() schedules one of the parallel functions, passing different data to each for processing. suba and subc are computationally-independent parallel functions.
- 3 At this point, tsyncro() makes the main task program wait until both suba and subc are finished before the main task program continues its processing.

z/OS XL C with multiple instances of the same parallel function

If your C program uses MTF, the main task program and multiple instances of the same parallel function can run concurrently.

Processor use

In [Figure 153 on page 561](#), parallel function suba has data you can divide, so two instances of suba run independently of the main task program and of each other.

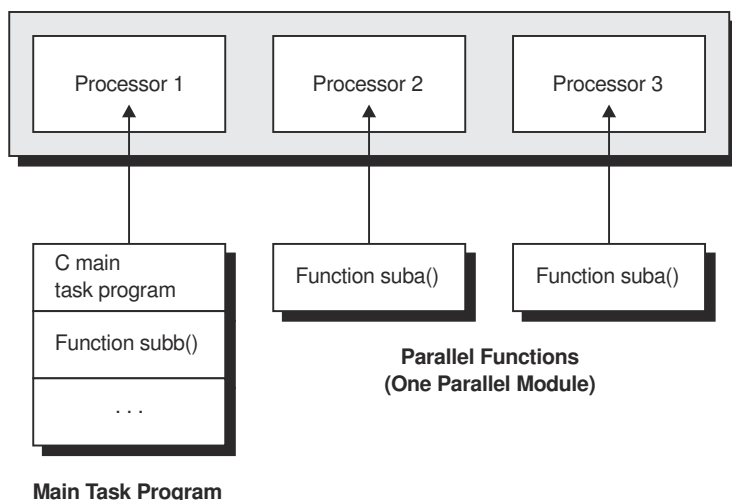


Figure 153. Processor use with multiple instances of the same parallel function

Sample program

As [Figure 154 on page 561](#) shows, the MTF functions perform the following tasks:

- 1** `tinit()` names the parallel load module `plmod` and specifies two subtasks.
- 2** Each call to `tsched()` schedules one instance of the parallel function to run and supplies separate data to be processed by that instance of `suba`. The data to be processed by each instance of the parallel function could be two different sections of the same array. Both instances of `suba` are computationally-independent of the main task program and each other, because each instance of `suba` processes different data.
- 3** At this point, `tsyncro()` makes the main task program wait until all instances of `suba` finish before the main task program continues.

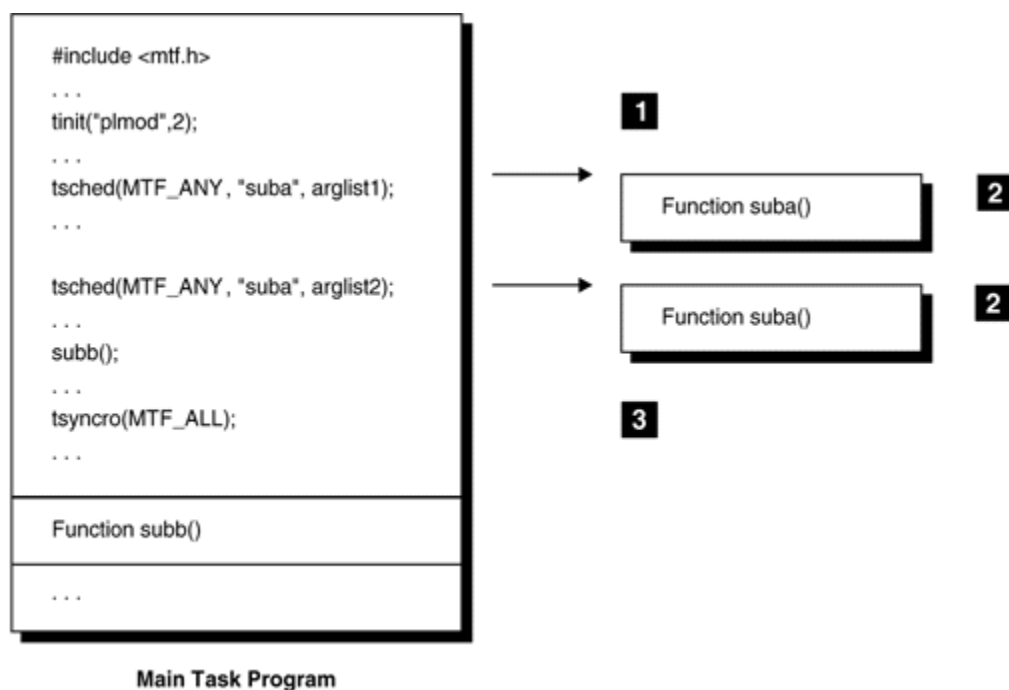


Figure 154. Sample program using multiple instances of the same parallel function

Designing and coding applications for MTF

You can use the following steps when preparing a z/OS XL C application to work with MTF:

1. Identify computationally-independent code
2. Create parallel functions
3. Insert calls to parallel functions in main task program

New programs can be designed to use MTF, and existing programs can be reconstructed.

Step 1: Identifying computationally-independent code

The first step in adapting an application program for MTF is to identify groups of computations that can be performed in parallel. To produce correct results, the computations that are done in parallel must be computationally-independent. This is further explained under [“Ensuring computational independence” on page 556](#).

Step 2: Creating parallel functions

After the segments of code that are computationally-independent are identified, they are separated from the main task program and placed in parallel functions. A parallel function is coded as a normal C function that follows several rules required for correct operation with MTF. Besides to data independence, there are rules for:

- Parallel functions
- Calling other functions
- Separate storage for separate modules
- Passing data
- Input and output
- Exception/signal handling
- Function termination

Parallel functions

- A parallel function must be written only in C.
- The return value of a parallel function must be void. If a parallel function attempts to return a value, the behavior will be undefined.
- External parallel function names must be 8 characters or shorter in length and will be uppercased.

Calling other functions

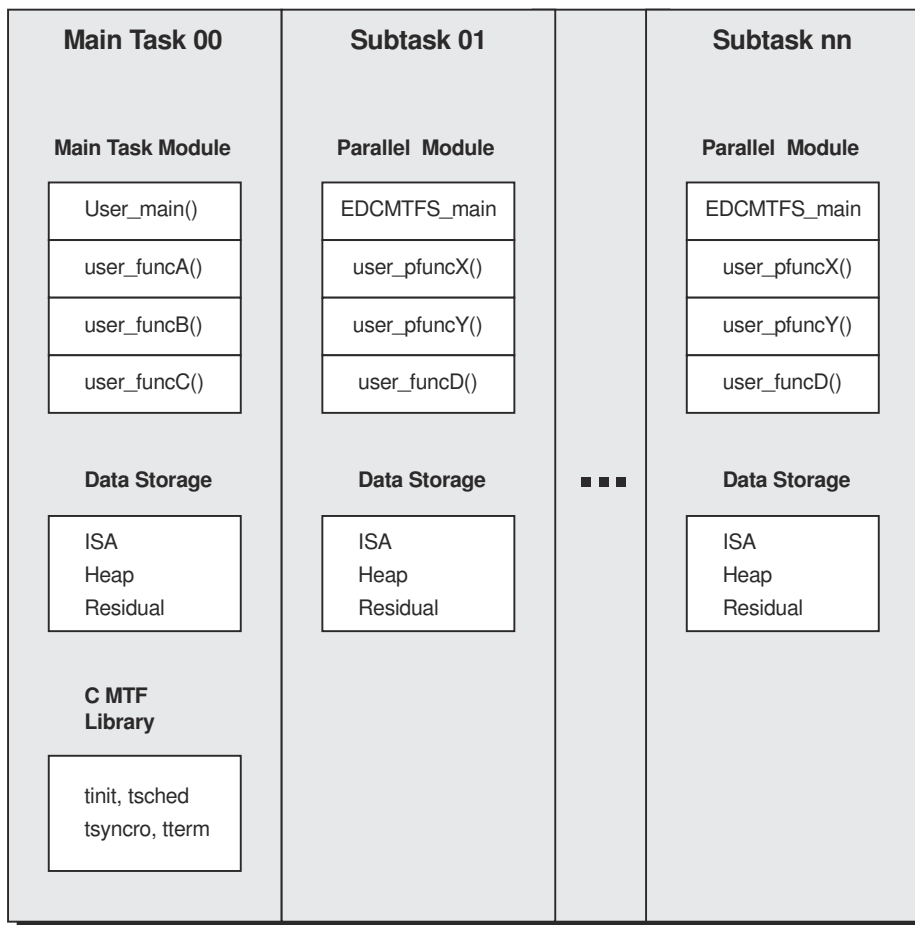
- A parallel function may actually be coded as a series of functions that call one another. All of these functions operate in the parallel function's subtask environment and must follow the rules of a parallel function except that they can be written in assembler as well as C, and they can have return values.
- A parallel function cannot call the MTF library functions `tinit()`, `tsched()`, `tsyncro()`, or `tterm()`. Such calls can only be used in the main task.

Separate storage for separate modules

- Every MTF application consists of two modules: the main task module which runs on the main task, and the parallel module that runs on the subtask(s). Each task (main or sub) has its own unique runtime storage structure consisting of ISA, heap, and residual storage. Each task has:
 - Separate writable static (whether reentrant or not)
 - Separate library-internal storage (for example, file and storage management control blocks)

- Separate exception and signal-handling environment (for example, `errno`, `__amrc`)
- Usually, functions must abide by the restrictions inherent in this arrangement. The remaining rules in this section mostly arise from this arrangement.

Single User Application/Single Address Space



Notes:

1. Each task has private and separate storage structure that leads to most of the parallel function idiosyncrasies:
 - All file operations from same task.
 - Storage must be `malloc()` or `free()`d from same task.
 - Independent signal handling environments.
2. MTF library functions are only operational in the main task.
3. `call/return` used for invocation within a task.
4. MTF only supports parallel load modules in a PDS. Parallel load modules in a PDSE are NOT supported.

Figure 155. Basic MTF layout

Passing data

- A parallel function is always invoked in its last-used state. If, for example, a parallel function has defined a static variable with an initializer, then the variable has that value the first time the parallel function executes on a given task. Should the value be modified, the modification is available the next time that parallel function is run only if the function is scheduled to the same task. If you don't schedule the parallel function to the same task, you cannot depend upon residual values from previous invocations of the function.

- Data can be passed between the main task program and parallel functions, and between parallel functions by passing a pointer to the storage area as a parameter. Care must be taken to ensure that the data remains valid and available until completion of the particular parallel function instance being scheduled.
- If heap storage is obtained on a given task, it must be freed on that task and no other. Other tasks may be given access to that storage by passing pointers but no other task can use that pointer to free the storage.

Input/Output

- File pointers must not be shared across subtasks. A given file pointer must only be used (for file access and closing) on the same task on that it was created {(using `fopen()`)}. File pointers must be utilized as a serial resource. z/OS XL C does not protect against misuse, and a program will have unpredictable behavior if this rule is not enforced.
- Each parallel function updates (writes or changes) a file as if it had complete control over the file; therefore, there should be *no* simultaneous read or update of a given file while any function on any task is updating that file (even if separate file pointers are used).
- Memory files cannot be shared across subtasks.

Exception/Signal handling

- The parallel functions on the subtasks run with TRAP(ON) runtime option, and each has a signal handling environment entirely independent from that of each other task. All signals are initialized to default handling on each task, and can be modified for a given task only through a signal statement from a parallel function on that task.
- All signal interrupts are eligible to be raised from the operating system or by the `raise()` function during execution of parallel functions. All signals, however, require special handling in the case of parallel functions because of the requirement that parallel functions always return normally. Signals must either be ignored or a handler must be established that does not terminate the program. If these signals are left to default handling or a handler is established that terminates the program, MTF will treat this as an abnormal termination of the parallel function.

Function termination

- Parallel functions run as called functions (from EDCMTFS, the z/OS XL C library supplied main function for parallel modules) and must terminate by simple return (to EDCMTFS). For more information on EDCMTFS, see [“Creating the parallel functions load module” on page 569](#).
- Termination with `exit()` and `abort()` calls is invalid because these functions interfere with EDCMTFS operation and they are treated by MTF as abnormal terminations.
- On the first valid call to MTF (`tsched()`, `tsyncro()`, `tterm()`) from the main task program after a parallel function has abnormally terminated (via `exit()`/`abort()` or otherwise) MTF will:
 - Abort all parallel functions scheduled or in progress
 - Remove the MTF environment
 - Return ETASKABND on that MTF call

A subsequent `tterm()` call is unnecessary and will simply return EINACTIVE. A `tinit()` can be reissued, but depending on the severity of the condition that caused the ETASKABND, the `tinit()` may or may not be successful.

Step 3: Inserting calls to parallel functions

In the main task, insert a call to `tinit()` to initialize the MTF environment before to any other MTF function call, or after `tterm()` is invoked. Replace each segment of code that was identified for parallel computation with a call to `tsched()` which schedules the corresponding parallel function. If more parallel function instances are scheduled than tasks are currently available, the additional instances are queued for subsequent execution in the order in which they were scheduled. They are queued for

any task or to a particular task according to the `task_id` parameter supplied on the `tsched()` call. If parallel operation is to be achieved by scheduling the same function multiple times with different data, the function call may be placed within a loop.

The arguments passed to the parallel function may be:

- A variable
- An array element
- An array name
- A constant
- A structure

The following items must not be used as the arguments supplied to the parallel function using `tsched()`:

- Function pointers
- A pointer to data or storage that will be modified or released before a `tsyncro()`.

After inserting calls to the parallel functions, insert a call to `tsyncro()` wherever the program requires that any subtask, one particular subtask, or all of the subtasks have finished executing the parallel functions previously scheduled to them. As the last MTF call, insert a call to `tterm()` before to exit/return from the main task program to remove the MTF environment.

To properly use MTF from the main task program it is necessary to include the `mtf.h` header file before to the first MTF call in your program. MTF calls themselves can be issued from non-main as well as main functions within the main task program, subject only to the restrictions already described above. MTF calls, however, can only be issued from C functions and not from functions written in any other language.

The next sections show examples of how to change existing C programs to use MTF following the steps just outlined.

Changing an application to use MTF

The following examples show how to change an application to use MTF by creating parallel functions and inserting calls to these functions.

Example 1

Figure 156 on page 565 shows a computation of the dot product on two long one-dimensional arrays of data. The processing within the loop structure may be separated so that the dot product is not a result of serial calculations but a result of parallel calculations. This is because the first part of the array is not dependent on the results computed in any other section of the array. Thus the calculations are therefore computationally independent of each other, and can be performed at the same time.

```
double dotprod(double *a, double *b, int len)
{
    int i;
    double res = 0;
    for (i=0; i < len; ++i)
        res += *a++ * *b++;
    return(res);
}
```

Figure 156. Identifying Computationally-Independent Code

Create parallel functions

The segments of the program that have been identified to run as parallel functions are then recoded as new z/OS XL C functions. In this case, there will be one parallel function, multiple instances of which will

be scheduled. The parallel function corresponding to the code in [Figure 156 on page 565](#) now looks like [Figure 157 on page 566](#).

```
void pdotprod(double *a, double *b, int len, int m, int n, double *pres)
{
    /* m = the section of the array */
    /* n = the number of subtasks. n must be a factor of len */

    int i, from, to;

    *pres = 0;

    /* Determine which section of the array to operate on */
    from = (m-1) * len / n;
    to   = (m * len) / n;

    /* Calculate the partial result on part of the array */
    for (a+= from, b+=from, i=from; i < to; ++i)
        *pres += *a++ * *b++;
}
```

Figure 157. Sample code as a parallel function

The variables `to` and `from` are used to determine on which part of the array the parallel function is to perform.

Insert calls to parallel functions

The segments of the program that have been removed to form parallel functions are replaced by calls to these new parallel functions. For the sample code in [Figure 156 on page 565](#), `sub:exph.` is scheduled for each subtask that will be used at run time. In order to do this, the computations controlled by the `k` index must be divided so that each instance of the function `sub` operates on a different part of the original range of the `k` variable. See [Figure 158 on page 566](#) for an example of how two instances of a parallel function can be scheduled.

```
#include <mtf.h>;

double dotprod(double *a, double *b, int len)
{
    :
    int i;
    double res = 0;
    double pres[MAXTASK];

    /* Schedule the parallel functions according to */
    /* how many subtasks exist */
    for (i=1; i < n; ++i)
        tsched(MTF_ANY, "pdotprod", a, b, len, i, n, &pres[i-1]);

    /* Perform the calculations on the last part of the array */
    pdotprod(a, b, len, n, n, &pres[n-1]);

    /* Wait until all of the partial results are determined */
    tsyncro(MTF_ALL);

    /* Add all the partial results to determine the final dot product */
    for (i=0; i < n; ++i)
        res += pres[i];

    return(res);
}
```

Figure 158. Scheduling instances of a parallel function

Also, within the main task program, the subtasks must be initialized and eventually terminated as shown in [Figure 159 on page 567](#).

```

#include <mtf.h>

int main(void)
{
    :
    /* other code */
    /* Attach and initialize a subtask */
    tinit(load_sub_name, n);
    :
    result = dotprod(vector1,vector2,len);
    :
    /* Terminate subtasks */
    tterm();
    /* more code */
}

```

Figure 159. Main task program to call dot product function

Example 2

Not all application programs contain parallelism within the iterations of a loop structure. The following example illustrates parallel computations that appear as different segments of code in the original program. Also illustrated is the use of pointer arguments for passing data, and I/O operations to files in parallel functions.

Figure 160 on page 567 (program CCNGMT1) shows two calls to the same function that performs the dot product on the values in two files of data. The values are read from each file and the function performs the dot product upon these values. The loop ends when the end of either file is reached. The two computations are independent of each other and thus can be performed simultaneously in two different parallel functions.

```

/* MTF example 2 */

#include <stdio.h>

void fdotprod(char *fn1, char *fn2)
{
    int i, res1;
    double result=0, val1, val2;
    FILE *file1, *file2;

    file1 = fopen(fn1, "r");
    file2 = fopen(fn2, "r");

    while (1)
    {
        res1 = fscanf(file1, "%lf", &val1);
        res1 += fscanf(file2, "%lf", &val2);
        if (res1 != 2)
            break;
        result += val1 * val2;
    }
    if (res1 == 1)
        printf("Error: Files of unequal length\n");
    else
        printf("Result: %lf\n", result);
}

int main(void)
{
    fdotprod("a.input", "b.input");
    fdotprod("c.input", "d.input");

    return(0);
}

```

Figure 160. Sample code to be changed to use MTF

Create parallel functions

The fdotprod routine is identified as a parallel function so it is recoded as a new C function in a separate file. Data is passed from the main function to the parallel functions by means of pointer arguments. The

main task program (CCNGMT2) is shown in [Figure 161 on page 568](#). The parallel functions are shown in [Figure 162 on page 568](#) (sample program CCNGMT3).

```
/* MTF example 2 */
/* part 2 of 2-other file is CCNGMT1 */

#include <stdio.h>
#include <mtf.h>

int main(void)
{
    tinit("plmod", 2);
    tsched(MTF_ANY, "fdotprod", "a.input", "b.input");
    tsched(MTF_ANY, "fdotprod", "c.input", "d.input");
    tsyncro(MTF_ALL);
    tterm();

    return(0);
}

void fdotprod(char *fn1, char *fn2)
{
    int i, res1;
    double result=0, val1, val2;
    FILE *file1, *file2;

    file1 = fopen(fn1, "r");
    file2 = fopen(fn2, "r");

    while(1)
    {
        res1 = fscanf(file1, "%lf", &val1);
        res1 += fscanf(file2, "%lf", &val2);
        if (res1 != 2)
            break;
        result += val1 * val2;
    }
    if (res1 == 1)
        printf("Error: Files of unequal length\n");
    else
        printf("Result: %lf\n", result);
}
```

Figure 161. Sample code (main routine)

```
/* MTF example 2 */
/* part 2 of 2-other file is CCNGMT2 */
#include <stdio.h>

void fdotprod(char *fn1, char *fn2)
{
    int i, res1;
    double result=0, val1, val2;
    FILE *file1, *file2;

    file1 = fopen(fn1, "r");
    file2 = fopen(fn2, "r");

    while(1)
    {
        res1 = fscanf(file1, "%lf", &val1);
        res1 += fscanf(file2, "%lf", &val2);
        if (res1 != 2)
            break;
        result += val1 * val2;
    }
    if (res1 == 1)
        printf("Error: Files of unequal length\n");
    else
        printf("Result: %lf-n", result);
}
```

Figure 162. Sample code (routine to create parallel functions)

Compiling and linking programs that use MTF

Programs that use MTF run using two MVS load modules: a load module that contains the main task program, and a load module that contains the parallel functions. You compile and link-edit the main task program in the same procedure as non-MTF C programs. The parallel function is compiled in the same procedure as non-MTF C programs and is linked with EDCMTFS.

Creating the main task program load module

The main task program load module is the load module that first receives control when MVS starts running your program. It is the load module named in the PGM keyword of the EXEC statement. This load module contains your application's C `main()` function plus all other functions that are to run as part of the main task. The MTF functions can be invoked from any of the C functions contained in the main task load module and do not necessarily have to be invoked from the C function called `main()`.

The procedures that you usually use to compile and link-edit a z/OS XL C program can be used to create the main task program load module. For example, the following JCL sequence (see [Figure 163 on page 569](#)) uses the standard z/OS XL C cataloged procedure EDCCL to compile and link-edit the C source for the main task program (stored in data set `USERPGM.C(MTASKPGM)`) and create a main task program load module named `MTASKPGM` in data set `USERPGM.LOAD`.

```
//MTASKPGM EXEC EDCCL ,  
//          INFILE='USERPGM.C(MTASKPGM)',  
//          OUTFILE='USERPGM.LOAD(MTASKPGM)',DISP=OLD'
```

Figure 163. Sample JCL to compile and link main task program

Creating the parallel functions load module

The parallel functions load module is the load module named in the call to the MTF library function `tinit()`. This single load module contains all of your main task program's parallel functions. It must not contain any user's C `main()` programs. z/OS XL C itself provides the EDCMTFS module to act as the C `main()` function in the parallel module. EDCMTFS controls processing of the parallel functions as they are scheduled (by way of `tsched()` calls) to the subtasks. The source code for the EDCMTFS module is included in [Figure 165 on page 570](#).

Note: The executable module for parallel function program must be a load module (in a PDS data set), created using the linkage editor (and prelinker if required due to the presence of C++ code or C code compiled with the RENT option). The MTF library functions used to access the parallel functions are not compatible with a program object executable module (in a PDSE data set).

The procedures that you usually use to compile and link-edit a z/OS XL C program must be modified such that the library module `CEESTART` will be the entry point of the parallel functions load module.

When you link-edit this load module, include the following linkage editor control statements:

```
INCLUDE SYSLIB(EDCMTFS)  
ENTRY CEESTART
```

For example, the JCL sequence in [Figure 164 on page 570](#) uses the standard z/OS XL C cataloged procedure EDCCL to compile and link-edit the C source for the parallel functions and create a parallel functions load module named `PLMOD` in data set `USERPGM.LOAD`. This load module contains the module EDCMTFS, and has EDCMTFS as the load module's entry point.

```
//MTASKPGM EXEC EDCCL,
//          INFILE='CBC.SCCNSAM(CCNGMT2)',
//          OUTFILE='USERPGM.LOAD(CCNGMT2),DISP=SHR'
//*
//PFUNC EXEC EDCCL,
//       INFILE='CBC.SCCNSAM(CCNGMT3)',
//       OUTFILE='USERPGM.LOAD(PLMOD),DISP=SHR'
//LKED.SYSLIN DD
//INCLUDE SYSLIB(EDCMTFS)
//ENTRY CEESTART
//*
```

Figure 164. Sample JCL to compile and link parallel functions

Note: First, we have a step that compiles and link-edits the main task program.

The addressing mode is subject to normal consideration as described in the [z/OS Language Environment Programming Guide](#).

Specifying the linkage-editor option

Do not specify the NE linkage-editor option when link-editing the parallel functions load module. MTF cannot schedule parallel functions that are contained in a load module link-edited with the NE option.

Modifying runtime options

You can alter the #pragma runopts options STACK and HEAP within the EDCMTFS module for each subtask, but you must recompile the module under the same name. The source code for EDCMTFS is shown in [Figure 165 on page 570](#).

```
/******
/* Modify the isa/isainc/heap subparameters in the following line */
/* as required to meet your needs. Ensure that your version (compiled*/
/* and linked) is then accessed in your link-edit of the parallel */
/* module in place of the prebuilt EDCMTFS found in SCEELKED. */
/******
/******
#pragma runopts(STACK(8K,4K,ANY,FREE),HEAP(4K,4K,ANY,FREE))
/******
/* The following lines must remain unmodified to ensure proper */
/* operation of MTF. */
/******
#pragma runopts(TRAP(ON),RPTSTG(OFF),\
               (STAE,SPIE,NOREPORT,NOTEST,\
                ARGPARSE,REDIR,NOEXECOPS)
int main(int argc, char **argv) { return tsetsbvt(argc,argv); }
```

Figure 165. Source code for EDCMTFS

You can also add a #pragma runopts statement with the LIBRARY and VERSION options to EDCMTFS, if required.

Running programs that use MTF

To run your program, use the usual MVS JCL for z/OS XL C programs, plus a few additional JCL statements that are required to run MTF.

STEPLIB DD statement

You must ensure that the library containing the load modules is specified on the STEPLIB DD statement in your JCL, as well as the other libraries usually specified, as follows:

```
//STEPLIB DD DSN=user.dsn,DISP=SHR
```

user.dsn

name of the load module library that contains the parallel functions load module. The parallel functions load module (*parallel_loadmod_name*), specified on the call to `tinit()`, must be in this data set. You must allocate the ddname EDCMTF to the *user.dsn* data set as well as adding *user.dsn* to the STEPLIB concatenation list.

DD statements for standard streams

For standard streams, MTF assigns a unique runtime output file to each parallel function. These output files contain diagnostic messages that the library can issue while the parallel functions are running. They also contain output directed to the standard streams (`stderr` and `stdout`) by parallel functions and input from the standard stream `stdin`.

Because these files are automatically allocated while the program is running, you need not supply DD statements for them unless you wish to override the default device type or other file characteristics. The default device type is a terminal in TSO or `SYSOUT=*` in batch.

If you do supply DD statements, use the following ddnames:

- `stdinstn` for files containing input for operations such as `getc()`
- `stderrstn` for files containing diagnostic messages
- `stdoutstn` for files containing output from operations such as `printf()`

Where *stn* is the 2-digit subtask number; that is, 01, 02, 03, and so on. Thus, for example, if you had four subtasks and the first two used `printf()` functions, you would use the ddnames `stdout01`, `stdout02`, `stderr01`, `stderr02`, `stderr03`, and `stderr04`.

Example of JCL

An example of the runtime JCL to run a program that uses MTF is shown in [Figure 166 on page 571](#). This figure shows the JCL that is unique to running MTF, as well as the other JCL the program would typically require. (Some programs might require additional DD statements.)

```
//GO      EXEC  PGM=MTASKPGM
//STEPLIB DD   DSN=USERPGM.LOAD,DISP=SHR
//STDIN01 DD   DSN=USERPGM.INPUT,DISP=SHR
//STDOUT02 DD  SYSOUT=S,DCB=(RECFM=F)
```

Figure 166. Example runtime JCL

MTASKPGM is the name of the main task program load module, and is the load module that gets control when MVS first starts running the program. In this example, this load module is contained in data set `USERPGM.LOAD`, which is referred to by the `STEPLIB DD` statement. `USERPGM.LOAD` also contains the parallel functions.

The `STDIN01 DD` statement specifies the data set that contains the program's input data for the first task. The `STDOUT02 DD` statement specifies that printed output aside from runtime error messages from the second subtask is to be written to `SYSOUT` class S and that the record format is to be fixed-length. These DD statements are necessary only if you do not want to accept the defaults.

Debugging programs that use MTF

Debug Tool can be used to interactively debug your main task program. It cannot, however, be used to debug your parallel functions.

Avoiding undesirable results when using MTF

To prevent undesirable results, be aware of the following concerns and restrictions:

- MTF only supports parallel load modules in a PDS. Parallel load modules in a PDSE are NOT supported.

- Do not update a file with one task if the other tasks read the same file. Files can be destroyed if this is attempted.
- The following products should not be used from the main task or any subtasks while MTF is active:
 - Information Management System (IMS)
 - The CICS command level interface
- The following products should not be used from subtasks while MTF is active but can be used from the main task:
 - Data Window Services (DWS)
 - Interactive System Productivity Facility (ISPF)
 - Graphical Data Display Manager (GDDM)
- All library functions can be issued from the main task program.
- The following library functions should not be issued from parallel functions (see [“Function termination” on page 564](#)):
 - `exit()`
 - `abort()`
 - `atexit()`
- The following library functions can be used with some restrictions from parallel functions:
 - `setjmp()/longjmp()` can be used from within any task/subtask but must not be used across tasks. That is, the stack environment saved via `setjmp()` on a given task may be restored by a `longjmp()` from that task but from no other task.
 - `setlocale()/localeconv()` are only effective within a task. Each task has its own distinct locale information. Thus `setlocale()/localeconv()` issued from one task have no effect on such functions issued from other tasks.
 - `tmpnam()` may produce identical file names across tasks and should be restricted to being invoked from a single task (subtask or main task).
 - `rand()/srand()` produce entirely independent series of pseudorandom integers on each task
 - All file manipulation functions (such as `fopen()/fread()/...`) - were identified earlier under the rules for parallel functions in [“Designing and coding applications for MTF” on page 562](#). These functions can only be used on the same task.

Note: When opening files under MTF, you incur additional overhead when `fopen()` and `freopen()` are called. This overhead would normally be performed at the first read or write to the stream and will not affect the performance of a program that does indeed perform at least one read or write to the stream.

 - `fetch()/release()` must only be issued from the same task.
 - `free()` must be issued on the same task as the `malloc()/calloc()/realloc()` functions were issued. Note also that a `realloc()` must be issued in the same task as the `malloc()`.
 - `signal()/raise()` also identified earlier under the rules for parallel functions in [“Designing and coding applications for MTF” on page 562](#). Basically, each task has its own distinct interrupt environment. Thus `signal()/raise()` issued from one task have no effect on the operation of any other task.
 - PL/I and COBOL interlanguage calls must not be made from parallel functions.
 - Busy waits (loops that iterate until a flag is changed by a cooperating task) violate the requirement for computational independence. In particular, they can result in deadlock because of the scheduling algorithm used by MVS. They must be avoided.

Part 7. Programming with Other Products

This part contains the following programming product information:

- [Chapter 46, “Using the CICS Transaction Server \(CICS TS\),” on page 575](#)
- [Chapter 47, “Using Cross System Product \(CSP\),” on page 597](#)
- [Chapter 48, “Using Data Window Services \(DWS\),” on page 609](#)
- [Chapter 49, “Using DB2 Universal Database,” on page 611](#)
- [Chapter 50, “Using Graphical Data Display Manager \(GDDM\),” on page 619](#)
- [Chapter 51, “Using the Information Management System \(IMS\),” on page 625](#)
- [Chapter 52, “Using the Query Management Facility \(QMF\),” on page 633](#)

Chapter 46. Using the CICS Transaction Server (CICS TS)

This chapter describes how to develop XL C/C++ programs for the CICS Transaction Server for z/OS (CICS TS). You can find more information about the general features of z/OS Language Environment and CICS in [z/OS Language Environment Programming Guide](#).

For information on using CSP/AD or CSP/AE under CICS, see [Chapter 47, “Using Cross System Product \(CSP\),” on page 597](#).

Notes:

1. AMODE 64 applications are not supported in a CICS TS environment.
2. As of this publication, the standalone CICS translator does not recognize the C compiler's support for alternative locales and coded character sets. Therefore, you should write all your CICS C code in coded character set IBM-1047 (APL 293).
3. XPLINK applications are not supported under CICS prior to CICS TS 3.1.
4. As of V1R2, a non-XPLINK Standard C++ Library DLL allows support for the Standard C++ Library in the CICS subsystem. For further information, see "Binding z/OS C/C++ Programs" in [z/OS XL C/C++ User's Guide](#).

Developing XL C/C++ programs for the CICS environment

When developing a program to run under CICS TS you must complete all of the following actions:

1. Prepare CICS for use with [z/OS Language Environment](#).
2. [Design and code the CICS program](#).
3. [Translate and compile the translated source for reentrancy](#).
4. [Prelink and link all object modules with the CICS stub](#).
5. [Define the program to CICS](#).

Preparing CICS for use with z/OS Language Environment

This section gives general instructions on enabling z/OS Language Environment to use a new CICS TS environment or to add z/OS Language Environment to an existing CICS TS environment. For more detailed information on CICS TS, refer to [CICS Transaction Server for z/OS \(www.ibm.com/docs/en/cics-ts\)](http://www.ibm.com/docs/en/cics-ts).

After CICS TS has been installed on your system, you must perform the following tasks:

- Create a CICS TS environment if one does not already exist. This involves creating a CICS System Definition (CSD), journals, and a Global Catalog Set (GCD).
- Copy CEECCICS from SCEERUN to an Authorized Program Facility (APF) data set. The data set should be concatenated in the STEPLIB when CICS is cold started.
- Create the CESO and CESE Transient Data Queues. Sample Destination Control Table (DCT) definitions are supplied in SCEESAMP (CEECDCT).
- Add required definitions to the CSD. Sample CSD definitions are provided in SCEESAMP (CEECCSD). These sample definitions create a group called CEE, which must be added to the installation LIST.
- Add SCEERUN and SCEECICS to the DFHRPL concatenation.

The C runtime event handler module CEEEV003 is required for CICS TS support (in addition to the z/OS Language Environment interface modules). CEEEV003 must be link-edited as AMODE=31, RMODE=ANY, and loaded above the 16M line.

If you will be using the I/O stream library, complex mathematics, collection, or Application Support Class DLLs provided with the z/OS XL C++ compiler, you must define these DLLs in the CSD, and the CBC . SCLBDLL library must be added to the DFHRPL concatenation. Sample CICS CSD definitions can be found in CBC . SCLBJCL (CLB3YCSD).

Designing and coding for CICS

This section describes what you must do differently when designing and coding a z/OS XL C/C++ program for CICS TS, such as using EXEC CICS commands in your code, using input and output, using z/OS XL C/C++ functions, managing storage, using interlanguage calls, and exception handling.

Using the CICS command-level interface

CICS TS provides a set of commands to access the CICS transaction server. The format of a CICS command is:

```
EXEC CICS function [option[(arg)]]...;
```

In the following CICS command, the function is SEND TEXT. This function has 4 options: FROM, LENGTH, RESP and RESP2. In this case, each of the options takes one argument.

```
EXEC CICS SEND TEXT FROM(mymsg)
                  LENGTH(mymsglen)
                  RESP(myresp)
                  RESP2(myresp2);
```

For further information on the EXEC CICS interface and a list of available CICS TS functions, refer to [CICS Transaction Server for z/OS \(www.ibm.com/docs/en/cics-ts\)](http://www.ibm.com/docs/en/cics-ts).

When you are designing and coding your CICS TS application, remember the following:

- The EXEC CICS command and options should be in uppercase. The arguments follow general C or C++ conventions.
- Before any EXEC CICS command is issued, the EXEC Interface Block (EIB) must be addressed by the EXEC CICS ADDRESS EIB command.
- z/OS XL C/C++ does not support the use of EXEC CICS commands in macros.

The example program in [Figure 167 on page 577 \(CCNGCI1\)](#) shows the use of several EXEC CICS commands to perform various tasks.

- 1** Initialize the CICS interface
- 2** Access the storage passed from the caller
- 3** Handle unexpected abends
- 4 and 7** I/O to RRDS files
- 5 and 6** Requesting and formatting time


```

/* program : GETSTAT          */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define FILE_LEN 40

void check_4_down_status( char *status_record ) ;
void sendmsg( char* status_record ) ;
void unexpected_prob( char* desc, int rc ) ;

struct com_struct {
    unsigned int quiet ;
} *commarea ;

DFHEIBLK *dfheiptr ;

main ()
{
    long int  vsamrrn;
    signed short int  vsamlen;
    unsigned char status_record[41];
    signed long int myresp;
    signed long int myresp2;

    /* get addressability to the EIB first */
    EXEC CICS ADDRESS EIB(dfheiptr); 1

    /* access common area sent from caller */
    EXEC CICS ADDRESS COMMAREA(commarea); 2

    /* call the CATCHIT prog. if it abends */
    EXEC CICS HANDLE ABEND PROGRAM("CATCHIT "); 3

    vsamrrn = 1;
    vsamlen = FILE_LEN;

    /* read the status record from the file*/
    EXEC CICS READ FILE("STATFILE") 4
        UPDATE
        INTO(status_record)
        RIDFLD(vsamrrn)
        RRN
        LENGTH(vsamlen)
        RESP(myresp)
        RESP2(myresp2);

    /* check cics response */
    /* -- non 0 implies a problem */
    if (myresp != DFHRESP(NORMAL))
        unexpected_prob("Unable to read from file",61);

    printf("The status_record from READ in GETSTAT = %s\n", status_record);

    if (memcmp(status_record,"DOWNTME ",8) == 0)
        check_4_down_status(status_record);

    if (commarea->quiet != 1)
        sendmsg(status_record);

    exit(11);
}

```

Example illustrating how to use EXEC CICS commands (Part 1 of 3)

Figure 167. Example illustrating how to use EXEC CICS commands

```

void check_4_down_status( char *status_record )
{
    unsigned char uptime[9];
    unsigned char update[9];
    char curabs[8];
    unsigned char curtime[9];
    unsigned char curdate[9];

    long int  vsmrrn;
    signed short int  vsmlen;
    signed long int  dnresp;
    signed long int  dnresp2;

    strncpy((status_record+8),update,8);
    strncpy((status_record+16),uptime,8);
    update[8] = '\0';
    uptime[8] = '\0';

    /* get the current time/date */
    EXEC CICS ASKTIME ABSTIME(curabs)          5
                   RESP(dnresp)
                   RESP2(dnresp2);

    if (dnresp != DFHRESP(NORMAL))
        unexpected_prob("Unexpected prob with ASKTIME",dnresp);

    /* format current date to YYMMDD */
    /* format current time to HHMMSS */
    EXEC CICS FORMATTIME ABSTIME(curabs)        6
                   YYMMDD(curdate)
                   TIME(curtime)
                   TIMESEP
                   DATESEP;

    if (dnresp != DFHRESP(NORMAL))
        unexpected_prob("Unexpected prob with FORMATTIME",dnresp);

    curdate[8] = '\0';
    curtime[8] = '\0';

    if ((atoi(curdate) > atoi(update)) ||
        (atoi(curdate) == atoi(update) && atoi(curtime) >= atoi(uptime)))
    {
        strcpy(status_record,"OK");

        vsmrrn = 1;
        vsmlen = FILE_LEN;

        /* update the first record to OK */

        EXEC CICS REWRITE FILE("STATFILE")      7
                   FROM(status_record)
                   LENGTH(vsmlen)
                   RESP(dnresp)
                   RESP2(dnresp2);

        if (dnresp != DFHRESP(NORMAL)) {
            printf("The dnresp from REWRITE = %d\n", dnresp) ;
            printf("The dnresp2 from REWRITE = %d\n", dnresp2) ;
            unexpected_prob("Unexpected prob with WRITE",dnresp);
        }
    }
}

```

Example illustrating how to use EXEC CICS commands (Part 2 of 3)

```

        printf("%s %s Changed status from DOWNTME to OK\n",curdate,
               curtime);
    }
}

void sendmsg( char* status_record )
{
    long int msgresp, msgresp2;
    char outmsgY80";
    int outlen;

    if (memcmp(status_record,"OK ",3)==0)
        strcpy(outmsg,"The system is available.");
    else if (memcmp(status_record,"DOWNTME ",8)==0)
        strcpy(outmsg,"The system is down for regular backups.");
    else
        strcpy(outmsg,"SYSTEM PROBLEM -- call help line for details.");

    printf("%s\n",outmsg);
    outlen=strlen(outmsg);

    EXEC CICS SEND TEXT FROM(outmsg)
                        LENGTH(outlen)
                        RESP(msgresp)
                        RESP2(msgresp2);

    if (msgresp != DFHRESP(NORMAL))
        unexpected_prob("Message output failed from sendmsg",71);
}

void unexpected_prob( char* desc, int rc)
{
    long int msgresp, msgresp2;
    int msglen;

    msglen = strlen(desc);

    EXEC CICS SEND TEXT FROM(desc)
                        LENGTH(msglen)
                        RESP(msgresp)
                        RESP2(msgresp2);

    fprintf(stderr,"%s\n",desc);

    if (msgresp != DFHRESP(NORMAL))
        exit(99);
    else
        exit(rc);}

```

Example illustrating how to use EXEC CICS commands (Part 3 of 3)

Using input and output

This section describes how to use z/OS XL C/C++ I/O with CICS TS. It describes the file and device support and the type of I/O used with CICS TS.

Note: You can set up a SIGIOERR handler to catch read or write system errors. See [Chapter 16](#), “Debugging I/O programs,” on page 161 for more information.

Standard stream support

Under CICS, if you are using the z/OS XL C++ standard streams, note the following:

- `cin` is not supported under CICS.
- `cout` maps to the Standard C I/O stream `stdout`.
- `cerr` and `clog` both map to the C standard stream `stderr`.

`stdout` and `stderr` are assigned to transient data destinations (queues). The type of queue, intrapartition or extrapartition, is determined during CICS initialization. Intrapartition queues are used for queueing messages and data within a CICS region. Extrapartition queues are used to send data outside the CICS region or to receive data from outside the CICS region.

The transient data queues associated with `stdout` and `stderr` are CES0 and CESE respectively. z/OS XL C/C++ supports VA and VBA queues with an `lrecl` of at least 137 bytes.

Records sent to the transient data queues associated with `stdout` and `stderr` take the form of a message. The entire message record can be preceded by an ASA Standard control character. [Figure 168](#) on [page 580](#) illustrates the recommended message format.

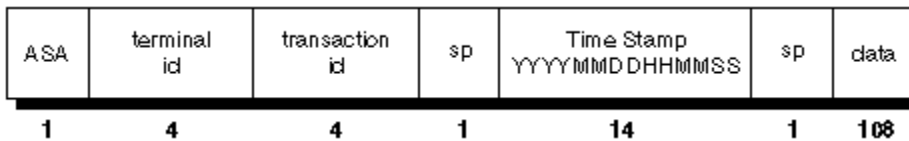


Figure 168. Format of data written to a CICS data queue

ASA

is the carriage-control character.

terminal id

is a 4-character terminal identifier.

transaction id

4-character transaction identifier

sp

a space

Time Stamp

date and time displayed in the format YYYYMMDDHHMMSS

data

data that is output to the standard streams `stdout` and `stderr`

The following are sample messages of data written to a CICS data queue:

```
SAMATST1 19940401080523 Hello World - from transaction TST1!
BOBATST3 19940401112348 Hello World - from transaction TST3!
TEDATST2 19940401112348 Hello World - from transaction TST2!
```

Standard streams can only be redirected to, or from, memory files. Because only one transient data queue can be associated with each of `stdout` and `stderr`, these queues can contain output written in chronological order from many C and C++ programs. This output must be sorted as necessary into the desired sequence.

Full memory file support

The full set of C I/O library functions is supported under CICS TS for memory files. Memory files are created with the parameter type set to memory on the `fopen()` call. If you are using C++, you can also use the I/O stream library to create and access memory files. Hiperspace memory files are not supported.

Support for disk files and other devices

There is no support by the C I/O library or the I/O stream library for using disk files and other devices with CICS TS. I/O to access methods supported by CICS TS must use the CICS TS Application Programming Interface.

Using z/OS XL C/C++ library support

This section discusses restrictions and support for the z/OS XL C/C++ library with CICS.

Arguments to C or main()

When a z/OS XL C/C++ program is running under CICS TS, you cannot pass command line arguments to it. The values for `argc` and `argv` have the following settings:

argc

1

argv[0]

4-character CICS transaction ID

Runtime options

Command line runtime options cannot be passed in CICS. To specify runtime options in XL C/C++, you must include the `#pragma runopts` directive in the code. [Figure 167 on page 577](#) shows how to do this. See *z/OS Language Environment Programming Guide* for information on other ways to supply runtime options when you are running under CICS TS.

Using packed decimal with CICS

The packed decimal data type is supported under CICS TS. However, the standalone CICS translator does not support packed decimal. CICS expects packed decimal streams to be passed to it as arrays of characters. If you want to manipulate these arrays as a packed decimal number, you should define the array of characters in union with the appropriate packed decimal definition. For information about how to define the data fields for the EXEC CICS commands you are using, refer to [CICS Transaction Server for z/OS \(www.ibm.com/docs/en/cics-ts\)](#).

Note: The z/OS XL C++ compiler does not support packed decimal data. Any program using the C or C++ character data type to handle packed decimal data must have its own functions for the manipulation of this data.

Locales

All locale functions are supported for locales that have been defined in the CSD. CSD definitions for the IBM-supplied locales are provided in `SCEESAMP(CEECCSD)`. `setlocale()` returns NULL if the locales are not defined.

Code set conversion tables

The code set conversion tables that are used by the `iconv()` functions must be defined in the CSD.

POSIX

There is no support for POSIX functions that are not already defined as part of ANSI/ISO. z/OS UNIX is not supported under CICS.

Multitasking facility

MTF functions are not supported under CICS TS.

System programming C facilities

There is no support for the System Programming C facilities (SP C) under CICS TS.

SVC99 and dynamic allocation functions

`svc99()` and the dynamic allocation functions `dynalloc()`, `dynfree()`, and `dyninit()` are not supported under CICS TS. The `svc99()` function returns 0 if the input is NULL, otherwise the return value is undefined.

IMS

There is no support for the `ctdli()` function under CICS TS. If you call `ctdli()` under CICS TS, the return value is -1. For information about the CICS TS method to access IMS, refer to [CICS Transaction Server for z/OS \(www.ibm.com/docs/en/cics-ts\)](http://www.ibm.com/docs/en/cics-ts).

Dump functions

The dump functions `csnap()`, `cdump()`, and `ctrace()` are supported under CICS TS. The output is sent to the CESE transient data queue. The dump can not be written if the queue does not have a sufficient LRECL. An LRECL of at least 161 is recommended.

Dynamic Linked Libraries (DLL)

All DLLs must be defined in the CSD.

fetch()

The `fetch()` function is supported under CICS TS. Modules to be fetched must be defined to the CSD and installed in the PPT.

release()

The `release()` function is supported under CICS TS.

system()

The `system()` function is not supported under CICS TS. However, there are two EXEC CICS commands that give you similar functionality:

EXEC CICS LINK

This command enables you to transfer control to another program and return to the calling program later. See [Figure 169 on page 585](#).

EXEC CICS XCTL

This command enables you to transfer control to another program. Control does not return to the caller after completion of the called program.

Time functions

All time functions are supported except the `clock()` function, which returns the value `(time_t)(-1)` if it is used under CICS TS.

iscics()

The `iscics()` function is an extension to the C library. It returns a non-zero value if your program is currently running under CICS. If your program is not running under CICS, `iscics()` returns the value 0. The following example shows how to use `iscics()` in your C or C++ program to specify non-CICS or CICS specific behavior.

```
if (iscics() == 0)
    <non-CICS behavior>
else
    <CICS-specific behavior>
```

Floating point arithmetic

The simulation of extended precision floating point is not supported in CICS TS.

Program termination

A C or C++ program running under CICS will terminate when:

- An `exit()` function call or a `return` statement is issued in the C or C++ program. The `atexit()` list of functions is run when the C or C++ program terminates.

Note: On return from a C or C++ application, the `return` statement or values passed by C or C++ through the `exit()` function are saved in the `EIBRESP2` field of the `EIB`.

- An abend occurs and is not handled.
- An `EXEC CICS RETURN` is issued in your C or C++ program. The `atexit()` list of functions runs after these calls.
- The `abort()` function is started.

Storage management

A z/OS XL C/C++ program can acquire storage from and release storage to CICS TS either implicitly or explicitly.

Storage is acquired and released *implicitly* by the runtime environment. This storage is used for automatic, external, and static variables. External variables are valid until program completion.

Storage is acquired and released *explicitly* by the user with the C library functions `malloc()`, `calloc()`, `realloc()`, `aligned_alloc()`, or `free()`, with z/OS Language Environment Callable Services (refer to [z/OS Language Environment Programming Guide](#)), with the C++ `new` and `delete` operators, or with the `EXEC CICS` commands `EXEC CICS GETMAIN`, or `EXEC CICS FREEMAIN`.

- If you request the storage by using the C functions `malloc()`, `realloc()`, `aligned_alloc()`, or `calloc()` you must deallocate it by using C functions as well.
- If you request the storage by using z/OS Language Environment Callable Services, you must deallocate it by using z/OS Language Environment Callable Services.
- If you request the storage by using `EXEC CICS GETMAIN`, you must deallocate it by using `EXEC CICS FREEMAIN`.
- If you request storage using the C++ `new` operator, you must deallocate it by using the C++ `delete` operator.

All other combinations of methods of requesting and deallocating storage are unsupported and lead to unpredictable behavior.

Partial deallocations are not supported. All storage allocated at a given time must be deallocated at the same time.

Under the z/OS Language Environment library, z/OS XL C/C++ uses the z/OS Language Environment Callable Services to allocate and free storage. Refer to [z/OS Language Environment Programming Guide](#) for specific information on memory and storage manipulation in CICS.

The z/OS XL C/C++ library functions acquire all storage from the Extended Dynamic Storage Area (EDSA) unless you specify otherwise using the `ANYHEAP`, `BELOWHEAP`, `HEAP`, `STACK`, or `LIBSTACK` runtime options.

Storage that is acquired with the `EXEC CICS GETMAIN` command exists for the duration of the CICS task.

If your application is multi-threaded or often uses `malloc()`, `realloc()`, `calloc()`, `aligned_alloc()`, and `free()`, you should consider using the `HEAPPOOLS` runtime option. Although storage requirements may increase, you can expect better performance.

Using ILC support

The z/OS Language Environment library supports a variety of different types of interlanguage calls (ILC) with CICS TS. For information on supported configurations, please refer to [z/OS Language Environment Writing Interlanguage Communication Applications](#).

Exception handling

You can use three different kinds of exception handlers when running C programs in a CICS TS environment: CICS exception handlers, z/OS Language Environment abend handlers, and C exception handlers. If you are using C++, you can use any of these three, or the C++ exception handling approach using `try`, `throw`, and `catch`. When a CICS condition is not handled under C++, the behavior of constructors and destructors for objects is undefined.

If the CICS command `EXEC CICS HANDLE ABEND PROGRAM(name)` was specified in the application, it will be called for any program exception that occurs (such as an operation exception or a protection exception) as well as for any `EXEC CICS ABEND ABCODE(. . .)` command that is run.

z/OS Language Environment provides facilities to set up a user handler. These facilities are discussed in detail in [z/OS Language Environment Programming Guide](#).

In CICS TS, the C error handling facilities have almost the same behavior as discussed in [Chapter 24, “Handling error conditions, exceptions, and signals,”](#) on page 273. A signal raised with the `raise()` function is handled by its corresponding signal handler or the default actions if no handler is installed. If a program exception such as a protection exception occurs, it is handled by the appropriate C handler if no CICS or z/OS Language Environment handler is present.

When a C or C++ application is invoked by an `EXEC CICS LINK PROGRAM(. . .)`, the invoked program inherits any handlers registered by `EXEC CICS HANDLE ABEND PROGRAM(. . .)` in the parent program. Any handlers registered in the child override the inherited handlers. C signal handlers are **not** inherited.

The following chart shows the process for handling abends in CICS TS.

Error handling in CICS

Procedure

1. Is this the result of a call to `raise()`?

Option	Description
Yes	See “2” on page 584
No	See “9” on page 585

2. Is `SIG_IGN` set for the signal?

Option	Description
Yes	See “3” on page 584
No	See “4” on page 584

3.
 - a. Resume at the next instruction.

4. Is z/OS Language Environment handler registered?

Option	Description
Yes	See “5” on page 584
No	See “6” on page 584

5.
 - a. Run z/OS Language Environment user handler. See [z/OS Language Environment Programming Guide](#) for more details.

6. Is a C or C++ handler established?

Option	Description
Yes	See “7” on page 585

Option	Description
No	See “8” on page 585

7. a. Run C or C++ handler.
8. a. Default handling the program check and percolate to next stack frame.
9. Has EXEC CICS HANDLE ABEND been issued?

Option	Description
Yes	See “10” on page 585
No	See “11” on page 585

10. a. Call z/OS XL C/C++-CICS interface for termination of program. CICS turns off signal and runs program in handler.
11. a. Continue at Step 005.

Example of error handling in CICS

The example program in [Figure 169 on page 585](#) (CCNGCI2) shows how to handle errors when using z/OS XL C/C++ with CICS.

```

/* program :   CHKSTAT                               */
/* transaction : called stand alone from transaction CHST */
/*           is also used by other transactions to determine */
/*           system status                                   */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>

#define FILE_LEN 40

void status_not_ok(int sig);
void unexpected_prob(char* desc, int rc);
volatile unsigned char status_record [41];

struct com_struct {
    int quiet;
} com_reg;

main (int argc, char *argv [ ])
{
    long int  vsamrrn;
    signed short int  vsamlen;

    signed long int myresp;
    signed long int myresp2;
    unsigned char status_downtme [41];

    if (strcmp(argv[0], "CHST") !=0) {
        printf("argv[0] = %s\n", argv[0]);
        com_reg.quiet = 1;
    }
    else
        com_reg.quiet = 0;

```

Example illustrating error handling under CICS (Part 1 of 3)

Figure 169. Example illustrating error handling under CICS

```

/* get addressability to the EIB first */
EXEC CICS ADDRESS EIB(dfheiptr);

EXEC CICS HANDLE ABEND PROGRAM("CATCHIT ");           1
signal(SIGUSR1,status_not_ok);                        2

EXEC CICS LINK PROGRAM("GETSTAT ")                    3
      RESP(myresp)
      RESP2(myresp2)
      COMMAREA(&com_reg)
      LENGTH(4);

/* check for failure in linked-to program */
if (myresp != DFHRESP(NORMAL)) {
    printf("The RESP of LINK = %d\n", myresp);
    printf("The RESP2 of LINK = %d\n", myresp2);
    unexpected_prob("CICS failure on EXEC CICS LINK\n",51);
}

if (myresp2 != 11)
    unexpected_prob("Unexpected rc from GETSTAT\n",myresp2);

vsamrrn = 1;
vsamlen = FILE_LEN;

/* following READ for UPDATE is for test purpose only. */
EXEC CICS READ FILE("STATFILE")
      UPDATE
      INTO(status_record)
      RIDFLD(vsamrrn)
      RRN
      LENGTH(vsamlen)
      RESP(myresp)
      RESP2(myresp2);

/* check for cics response - non-0 implies problem */
if (myresp != DFHRESP(NORMAL))
    unexpected_prob("Unable to read from file",52);

```

Example illustrating error handling under CICS (Part 2 of 3)

```

/* write DOWNTME back to file - for test purpose only */
strcpy(status_downtme,"DOWNTME");
EXEC CICS REWRITE FILE("STATFILE")
                FROM(status_downtme)
                LENGTH(vsamlen)
                RESP(myresp)
                RESP2(myresp2);

if (myresp != DFHRESP(NORMAL)) {
    printf("The dnresp from REWRITE = %d\n", myresp);
    printf("The dnresp2 from REWRITE = %d\n", myresp2);
    unexpected_prob("Unexpected prob with WRITE",myresp);
}

if (memcmp(status_record,"OK ",3) != 0)
    raise(SIGUSR1);

exit(11);
}

void unexpected_prob( char* desc, int rc)
{
    long int msgresp, msgresp2;
    int msglen;

    msglen = strlen(desc);

    EXEC CICS SEND TEXT FROM(desc)
                    LENGTH(msglen)
                    RESP(msgresp)
                    RESP2(msgresp2);

    fprintf(stderr,"%s\n",desc);

    if (msgresp != DFHRESP(NORMAL))
        exit(99);
    else
        exit(rc);
}

void status_not_ok( int sig )
{
    if (memcmp(status_record,"DOWNSTR ",8) != 0)
        exit(22);
    else
        exit(33);
}

```

Example illustrating error handling under CICS (Part 3 of 3)

1

The program CATCHIT has been installed as the CICS abend handler. Because this CICS abend handler is installed, C exception handlers will only catch signals raised with the `raise()` function.

2

Install a C signal handler to catch the user defined signal SIGUSR1. This handler will only be called if `raise(SIGUSR1)` is run.

3

This command causes the flow of control to shift to a child program called GETSTAT. GETSTAT will inherit CHKSTAT's CICS abend handler.

4

The C signal handler `status_not_OK` that will be invoked if this line is run. The `raise()` function will **not** trigger the CICS abend handler.

ABEND codes and error messages under z/OS XL C/C++

For information on ABEND Codes and error messages used by the z/OS Language Environment library, refer to [z/OS Language Environment Programming Guide](#) and [z/OS Language Environment Debugging Guide](#).

Coding hints and tips

- Do not use EXEC CICS commands in macros.
- Do not use EXEC CICS commands in header files. This makes the translation process much simpler.
- Do not set `atexit()` routines before an EXEC CICS XCTL. You will get unpredictable results.
- If you call `fclose()` or `freopen()` for a standard stream, you cannot redirect or reopen the link to the transient data queue. z/OS XL C/C++ does not provide a method of opening or reopening the transient data queues.
- The actual transient data queue is not closed when you call `fclose()` or `freopen()` for a standard stream; however, the transaction will lose access to the stream.
- You should not use the `stdin` stream unless you are redirecting it from a memory file.
- Closing the `cout`, `cerr`, or `clog` standard streams in a C++ application has the same effect as closing `stdout` or `stderr`.
- When CICS handlers (using EXEC CICS HANDLE ABEND PROG) are activated along with C or C++ signal handlers, the CICS handler is invoked when an abend occurs. The C or C++ signal handler that corresponds to that class of abends is ignored.

Note: The handler mentioned here is not a catch clause. It is a C signal handler exception registered by a C++ routine.

- If you do an EXEC CICS RETURN out of an `atexit()` routine, the resulting return code (RESP2) is undefined.

Translating and compiling for reentrancy

This section discusses translation of embedded CICS statements and provides examples. It also discusses reentrancy issues with respect to CICS.

Options for translating CICS statements

There are two options for translating CICS statements into C or C++ code: The [z/OS XL C/C++ integrated CICS translator](#) and the [standalone CICS translator](#), a CICS TS utility.

z/OS XL C/C++ integrated CICS translator

If you are using CICS Transaction Server 3.1 or later, you can compile XL C/C++ source code with embedded CICS commands and keywords without using the CICS TS language translation utility if you use the CICS compiler option. You can embed comments and macros within the embedded CICS commands.

When you use the z/OS XL C/C++ integrated CICS translator, you might experience the following benefits:

- More seamless operation of C/C++ applications that run in the CICS environment, especially under UNIX System Services
- Improved program readability
- Easier application maintenance
- Tighter coupling between the translation and compilation phases
- A more unified development approach across z/OS XL C, z/OS XL C++, COBOL, and PL/I

For more information, refer to the [CICS | NOCICS compiler option](#) in [z/OS XL C/C++ User's Guide](#).

In general, source code that can be processed successfully by the standalone CICS translator will be compatible with the integrated CICS translator.

Exception: The standalone CICS translator does not recognize C/C++ macros. For this reason, a CICS command that is processed with the integrated CICS translator can either fail to translate or change semantically if it (coincidentally) contains an identifier that is identical to a macro that is active within the scope of the CICS command.

Standalone CICS translator

The CICS TS utility called the CICS language translator is still supported. This program translates the EXEC CICS statements into C or C++ code. In this document, the CICS language translator is referred to as the *standalone CICS translator*.

Note: If you are using C++, you must use the CPP translator option to indicate to the compiler that you are using the C++ language, rather than the C language. The use of the CPP parameter specifies that the translator is to translate z/OS XL C++ programs. Code translated without the CPP option or with a translator released before version 4.1 of CICS is not supported by the z/OS XL C++ compiler and will not compile.

The standalone CICS translator supplies a control block (DFHEIBLK) for passing information between CICS TS and the application program. C or C++ function references for the EXEC CICS commands are generated. The translation step is not required if you do not use EXEC CICS statements.

The standalone CICS translator does not evaluate preprocessor directives such as **#include** or **#define**. You should ensure that all EXEC CICS statements are translated.

Translating example

The samples in this section are valid for both the integrated CICS translator and the standalone CICS translator. Figure 170 on page 589 (CCNGCI3) shows pieces of C and C++ code before they are translated with the standalone CICS translator. Figure 171 on page 590 shows the corresponding programs after translation.

```
/* program : CATCHIT */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct com_struct {
    unsigned int quiet ;
} *commarea ;

main () {

    signed long int myresp;
    signed long int myresp2;

    /* get addressability to the EIB first */
    EXEC CICS ADDRESS EIB(dfheiptr); 1

    /* access common area sent from caller */
    EXEC CICS ADDRESS COMMAREA(commarea); 2

    printf("The program is now inside CATCHIT.\n");

    /* statements required to handle the abend
    EXEC CICS .....
    EXEC CICS ..... */

    EXEC CICS RETURN;

}
```

Figure 170. Example illustrating how to use EXEC CICS commands

1 and 2

These programs each contain two EXEC CICS commands to be translated by the standalone CICS translator. A single instance of the EXEC CICS ADDRESS EIB command is required before any other call to the EXEC CICS interface. In this case, the main program (see [Figure 167 on page 577](#)) issues the ADDRESS EIB command. Since the two pieces of code make up one program there is no need to ADDRESS the EIB again.

[Figure 171 on page 590](#) shows how the programs appear after they are translated.

```
#ifndef __dfheita
#define __dfheita 1
    char *dfhlver = "LD TABLE DFHEITAB 320." ;
    unsigned short int dfheib0 = 0 ;
    char *dfheid0 = "\x00\x00\x00\x0c" ;
    char *dfheicb = " " ;
typedef struct {
    unsigned char eibtime [4] ;
    unsigned char eibdate [4] ;
    unsigned char eibtrnid [4] ;
    unsigned char eibtasn [4] ;
    unsigned char eibtrmid [4] ;
    signed short int eibfil01 ;
    signed short int eibcposn ;
    signed short int eibcalen ;
    unsigned char eibaid ;
    unsigned char eibfn [2] ;
    unsigned char eibrcoe [6] ;
    unsigned char eibds [8] ;
    unsigned char eibreqid [8] ;
    unsigned char eibrsrce [8] ;
    unsigned char eibsync ;
    unsigned char eibfree ;
    unsigned char eibrecv ;
    unsigned char eibfil02 ;
    unsigned char eibatt ;
    unsigned char eibeoc ;
    unsigned char eibfmh ;
    unsigned char eibcompl ;
    unsigned char eibsig ;
    unsigned char eibconf ;
    unsigned char eiberr ;
    unsigned char eiberrcd [4] ;
    unsigned char eibsynrb ;
    unsigned char eibnodat ;
    signed long int eibresp ;
    signed long int eibresp2 ;
    unsigned char eibrldbk ;
} DFHEIBLK;
DFHEIBLK *dfheiptr;
#endif

#ifndef __dfhtemps
#pragma linkage(dfhexec,OS) /* force OS linkage */
void dfhexec(); /* Function to call CICS */
#define __dfhtemps 1
    signed short int dfhb0020, *dfhbp020 = &dfhb0020 ;
    signed short int dfhb0021, *dfhbp021 = &dfhb0021 ;
    signed short int dfhb0022, *dfhbp022 = &dfhb0022 ;
    signed short int dfhb0023, *dfhbp023 = &dfhb0023 ;
    signed short int dfhb0024, *dfhbp024 = &dfhb0024 ;
    signed short int dfhb0025, *dfhbp025 = &dfhb0025 ;
    unsigned char dfhc0010, *dfhcp010 = &dfhc0010 ;
    unsigned char dfhc0011, *dfhcp011 = &dfhc0011 ;
    signed short int dfhdummy;
#endif
```

Child C program after translation (Part 1 of 3)

Figure 171. Child C program after translation

```

/* this is an example of a CICS program for C                                     */
/* program : GETSTAT ( part 2 - infrequent use routines )                         */
/*                                                                              */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void unexpected_prob( char* desc, int rc);

void sendmsg( char* status_record )
{
    long int msgresp, msgresp2;
    char outmsg[80];
    int outlen;

    if (memcmp(status_record,"OK ",3)==0)
        strcpy(outmsg,"The system is available.");
    else if (memcmp(status_record,"DOWNTIME ",8)==0)
        strcpy(outmsg,"The system is down for regular backups.");
    else
        strcpy(outmsg,"SYSTEM PROBLEM -- call help line for details.");

    outlen=strlen(outmsg);

    /* EXEC CICS SEND TEXT FROM(outmsg)                                     4
        LENGTH(outlen)
        RESP(msgresp)
        RESP2(msgresp2) */

    {
        dfhb0020 = outlen;
        dfhexec("\x18\x06\x60\x00\x2F\x00\x00\x00\x00\x00\x20\x04\x00\x00\x20\xF0\xF0\xF0\xF0\xF2\xF2\xF0\xF0",dfhdummy,outmsg,dfhbp020 ); 5
        msgresp = dfheiptr->eibresp;
        msgresp2 = dfheiptr->eibresp2;
    }

    if (msgresp != 0 )
        unexpected_prob("Message output failed from sendmsg",71);
}

void unexpected_prob( char* desc, int rc)
{
    long int msgresp, msgresp2;
    int msglen;

    msglen = strlen(desc);

    /* EXEC CICS SEND TEXT FROM(desc)
        LENGTH(msglen)
        RESP(msgresp)
        RESP2(msgresp2) */

```

Child C program after translation (Part 2 of 3)

```

{
    dfhb0020 = msglen;
    dfhexec("\x18\x06\x60\x00\x2F\x00\x00\x00\x00\x00\x20\x04\x00\x00\x20\xF0\xF0\xF0\xF0\xF4\xF1\xF0\xF0", dfhdummy, desc, dfhbp020 ); 6
    msgresp = dfheiptr->eibresp;
    msgresp2 = dfheiptr->eibresp2;
}

fprintf(stderr, "%s\n", desc);

if (msgresp != 0)
    exit(99);
else
    exit(rc);
}

```

Child C program after translation (Part 3 of 3)

3

This structure, DFHEIBLK, is used for passing information between CICS and the application program.

4

This is the CICS command that was interpreted by the translator. The translator comments out the EXEC CICS commands.

5

The translator inserts this call to the function dfhexec and comments out the EXEC CICS commands for further processing by the z/OS XL C/C++ compiler. The values msgresp and msgresp2 are set from the values in the DFHEIBLK structure.

6

This EXEC CICS command is similar in format to the one discussed in **4**. However, you should note that the generated call to dfhexec is different. For this reason it is important that EXEC CICS commands are not imbedded in macros.

Compiling XL C/C++ programs that were preprocessed by the standalone CICS translator

CICS requires that programs be reentrant at CICS entry points. If you are using C, this means:

- If your program is not naturally reentrant, you must compile with the RENT compiler option.
- If you are compiling code that was translated by the standalone CICS translator, you must compile with the RENT compiler option. The standalone CICS translator puts external writable static in the program.

For both C and C++, this means that if your program is naturally reentrant and has not been translated, you can compile and link it just as you would a non-CICS program.

Sample JCL to translate and compile

The sample JCL in [Figure 172 on page 593](#) shows to translate and compile C modules.


```

/*-----
/* Translate a C-CICS program
/*-----
/* Translate a C program for CICS
/*-----
//TRANSTEP EXEC PGM=DFHEDP1$,
//          REGION=2048K,
//          PARM='MAR(1,80,0),OM(1,80,0),NOS'
//STEPLIB  DD DSN=CICS.SDFHLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSPUNCH DD DSN=&&SYSCIN,DISP=(,PASS),UNIT=VIO,
//          DCB=BLKSIZE=400,SPACE=(400,(400,100))
//SYSIN    DD DSN=MYID.CHKSTAT.C,DISP=SHR
/*-----
/* Compile the translated C source.
/*-----
//C0010308 EXEC EDCC,
//          INFILE='MYID.CHKSTAT.C',
//          OUTFILE='MYID.OBJECT(CHKSTAT),DISP=SHR',
//          CPARM='OPT(0) NOSEQ NOMAR RENT ',
//          SYSOUT6='*'
//SYSIN    DD DSN=*.TRANSTEP.SYSPUNCH,DISP=(OLD,DELETE)
//USERLIB  DD DSN=MYID.MYHDR.FILES,DISP=SHR

```

Figure 172. JCL to translate and compile a C program

Figure 173 on page 593 shows an example of JCL to translate and compile a C++ program.

```

/*-----
/* Translate a C++-CICS program
/*-----
/* Translate C++ program for CICS
/*-----
//TRANSTEP EXEC PGM=DFHEDP1$,
//          REGION=2048K,
//          PARM='MAR(1,80,0),OM(1,80,0),NOS,CPP'
//STEPLIB  DD DSN=CICS.SDFHLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSPUNCH DD DSN=&&SYSCIN,DISP=(,PASS),UNIT=VIO,
//          DCB=BLKSIZE=400,SPACE=(400,(400,100))
//SYSIN    DD DSN=MYID.CHKSTAT.C,DISP=SHR
/*-----
/* Compile the translated C++ source.
/*-----
//C0010308 EXEC CBCC,
//          OUTFILE='MYID.OBJECT(CHKSTAT),DISP=SHR',
//          CPARM='NOSEQ NOMAR RENT ',
//          SYSOUT6='*'
//SYSIN    DD DSN=*.TRANSTEP.SYSPUNCH,DISP=(OLD,DELETE)

```

Figure 173. JCL to translate and compile a C++ program

Linking all object modules

If you are using C++, or if you have compiled your C source with the RENT compile-time option, you can use the binder to link all of the object modules together.

CICS provides a stub called DFHELII, which must be link-edited with the load module. For your convenience, the linkage editor commands required for CICS are provided with CICS in the DFHEILID member of the SDFHSAMP data set. The DFHEILID member must be reblocked before it is passed to the linkage editor. A name card should also be passed to the linkage editor. All applications **must** run AMODE=31. It is recommended that the object module is linked with AMODE(31) and RMODE(ANY). CICS does not require any other linkage editor options.

If you are using C, and your program will reside in one of the DFHRPL libraries, you do not need to link-edit the module with the RENT option. However, if the program is to be installed in one of the link pack areas, STEPLIBs, or data sets in the system link list, you should link-edit the module with the RENT option.

The following example shows how to link C and C++ modules.

```

/*
/* * Reblock CICS support link module
/* *
//COPYLINK EXEC PGM=IEBGENER
//SYSUT1 DD DSN=CICS.V5R3M0.SDFHSAMP(DFHEILID),DISP=SHR
//SYSUT2 DD DSN=&&COPYLINK,DISP=(,PASS),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200),UNIT=SYSDA,
// SPACE=(400,(20,20))
//SYSPRINT DD SYSOUT=*
//SYSIN DD DUMMY
/*
/* * Link MYMAIN with MYCICSTF and MYOTHSTF
/* *
//LKED EXEC PGM=IEWL,REGION=48M,
// PARM='LIST,MAP,LET,XREF'
//SYSLIB DD DSN=CICS.V5R3M0.SDFHLOAD,DISP=SHR
// DD DSN=CEE.SCEELKED,DISP=SHR
//OBJECT DD DSN=MYID.OBJECT,DISP=SHR
//SYSLIN DD DSN=*.COPYLINK.SYSUT2,DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSLMOD DD DSN=MYID.CICS.LOAD,DISP=SHR
//SYSUT1 DD DSN=&&SYSUT1L,DISP=(,PASS),
// DSN=&&SYSUT1L,DISP=(,PASS),UNIT=SYSDA,SPACE=(3200,(30,30)),
// DCB=(RECFM=FB,LRECL=80,
// BLKSIZE=3200)
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
INCLUDE OBJECT(MYMAIN)
INCLUDE OBJECT(MYCICSTF)
INCLUDE OBJECT(MYOTHSTF)
NAME MYMAIN(R)

```

Defining and running the CICS program

This section discusses the implications of program processing, link considerations for C programs, and CSD considerations. Sample JCL to install z/OS XL C/C++ application programs is provided.

Program processing

In a CICS environment, a single copy of a program is used by several transactions concurrently. One section of a program can process a transaction and then be suspended (usually as a result of an EXEC CICS command). Another transaction can then start or resume processing the same or any other section of the same application program. This behavior requires that the program be reentrant.

Link considerations for C programs

If your C program will reside in one of the DFHRPL libraries, following the translate, compile, and link steps detailed earlier in this chapter is sufficient; there is no requirement to link-edit the module with the RENT linkage editor option.

However, if the program is to be installed in one of the link pack areas, STEPLIBs, or data sets in the system link list, the module should be link-edited with the RENT option.

CSD considerations

Before you can run a program, you must define it in the CICS CSD. When defining a program to CICS, you should use LANGUAGE (LE). However, if the program is in C and does not use ILC support, you can use LANGUAGE (C).

If you use a copy of a reentrant C or C++ application program that was installed in the link pack area, you must specify USELPACOPY (YES) in the resource definition when you define the program in the CSD. You can use the CICS-supplied procedure DFYEITDL to translate, compile, prelink, and link-edit C or C++ programs. For C programs, you may have to change the compile step of this procedure. You will have to change the compile step to use it with the C++ compiler.

Sample JCL to install z/OS XL C/C++ application programs

Figure 174 on page 595 shows sample JCL to install a C or C++ application program. Your application is *anyname*; *x* can resolve to I or X.

```
//jobname      JOB   accounting info,name,MSGLEVEL=1
//             EXEC  PROC=DFHEXTEL
# //TRN.SYSIN   DD      *
#pragma XOPTS(Translator options . . .)
           :
           z/OS XL C/C++ source statements
           :
/*
//LKED.SYSIN   DD      *
           NAME      anyname(R)
/*
//
```

Figure 174. JCL to install z/OS XL C/C++ application programs

Chapter 47. Using Cross System Product (CSP)

This chapter briefly describes the interface between z/OS XL C and applications generated through the Cross System Product/Application Development (CSP/AD) and the Cross System Product/Application Execution (CSP/AE) Version 3 Release 2 Modification 2 or later. CSP refers to both CSP/AD and CSP/AE.

CSP/AD is an interactive application generator that provides methods for interactively defining, testing, and generating application programs. It can aid in improving productivity in application development.

CSP/AE takes the generated program and executes it in a production environment.

Notes:

- 1. XPLINK is not supported in a CSP environment.
- 2. AMODE 64 applications are not supported in a CSP environment.

Common data types

Table 103 on page 597 lists the data types common to both CSP and z/OS XL C. You must use the function `__csplist` to receive the parameter list from a CSP application. See [z/OS C/C++ Runtime Library Reference](#) for more information on this function.

Table 103. Common data types between z/OS XL C and CSP	
z/OS XL C	CSP
signed short	BIN - 2 bytes
signed int/long	BIN - 4 bytes
struct	RECORD
char array(size)	Characters

Passing control

You can pass control between CSP and z/OS XL C as follows:

CALL

Calls another application or subroutine to be run. When execution is completed, control is returned to the statement following the CALL statement in the original application.

XFER|DXFR

Transfers control and initiates execution of a CSP application or non-CSP program or transaction. The current application is terminated when the transfer statement is executed.

Under CICS, XFER is used to transfer control to another CICS transaction, while DXFR is used to transfer control to an application or program. If the target name is an application, control remains in CSP and the application is initiated immediately. If the target name is a program, CSP issues CICS XCTL to the program name.

Note: From a z/OS XL C program, you can pass control to a CSP application but you cannot pass control to another z/OS Language Environment-enabled language (COBOL, PL/I) from that CSP application. Only one z/OS Language Environment-enabled language can be in the chain of calls.

Running CSP under MVS

The following sections cover calling CSP applications from z/OS XL C and calling z/OS XL C from CSP.

Calling CSP applications from z/OS XL C

To call a CSP application from z/OS XL C, you must:

1. Define the CSP program to be called one of the following:
 - DCGCALL - calling under MVS/TSO
 - DCGXFER - transferring control under MVS/TSO with OS pragma linkage
2. Fetch the program dynamically.
3. Transfer control to the program. You must pass at least one parameter when calling CSP from z/OS XL C. This is the pointer to the ALF name and application name.

Example programs

The following example program (CCNGCP1 in [Figure 175 on page 598](#)) CALLs a CSP application in the z/OS environment. You must receive a structure.

Note: CSP cannot pass the DXFR statement to z/OS XL C under TSO.

```
/* this example shows how to CALL CSP from C under TSO */
/*          CALL          */
/* CCNGCP1 ==> R924A6 */
/* R924A6 is a CSP application */

#include <stdlib.h>
#include <math.h>

#pragma linkage(DCGCALL,OS)

void main(int argc , char * argv[])
{
    int ctr,base, power ;

    typedef void ASM_VOID();
    #pragma linkage (ASM_VOID,OS)
    ASM_VOID * fetch_ptr;

    int rc = 0;
    char module [ 8] = {"DCGCALL " } ;
    struct tag_a6progc {
        char alfx [ 8];
        char applx [ 8];
    } ;
```

C/370 CALLing CSP under TSO (Part 1 of 2)

Figure 175. C/370 CALLing CSP under TSO

```

struct tag_a6rec {
    char  a6ct   [ 4];
    char  a6lan  [ 4];
    char  fil1   [ 8];          /* packed fields for PLI */
    char  fil2   [ 8];          /* packed fields for PLI */
    char  fil3   [ 8];          /* packed fields for PLI */
    int   a6xbc;
    int   a6ybc;
    int   a6zbc;
};
struct {
    char  s_parm [ 240];
} s_parms = {"ALF=C   "};

struct tag_a6progc a6_progc = {"FZERSAM.", "R924A6  "};

_Packed struct tag_a6rec a6_rec = {"CALL" ,
                                     "C      ",
                                     "0000110C",
                                     "0000220C",
                                     "0000330C",
                                     12, 2, 0
                                     };

base = atoi(argv[1]) ;
power= atoi(argv[2]) ;

a6_rec.a6xbc = base;
a6_rec.a6ybc = power;
a6_rec.a6zbc = (int) pow((double) a6_rec.a6xbc,
                        (double) a6_rec.a6ybc);

if ((fetch_ptr = (ASM_VOID *) fetch(module)) == NULL ) {
    printf (" failed on fetch of CSP %s module \n", module);
}
else {
    fetch_ptr (&a6_progc, &a6_rec);
    rc = release((void (*)()) fetch_ptr) ;
    if ( rc != 0 ) {
        printf ("CCNGCP1: rc from release =%d\n", rc );
    }
}
}

```

C/370 CALLing CSP under TSO (Part 2 of 2)

Figure 176 on page 600 shows example program CCNGCP2, which uses an XFER command to transfer control to a CSP application. You must pass a structure.

```

/* this example shows how to transfer control to CSP from C under */
/* TSO, using XFER */
/*          XFER          */
/* CCNGCP2 ==> R924A5 */
/* R924A5 is a CSP application */

#include <stdlib.h>
#include <math.h>
#pragma linkage(DCGXFER,OS)

void main(int argc , char * argv[] )
{
    int ctr,base, power ;
    int   rc      = 0;
    char  module [ 8] = {"DCGXFER " } ;

    typedef void ASM_VOID();
    #pragma linkage (ASM_VOID,OS)
    ASM_VOID * fetch_ptr;

    struct tag_a5ws {
        short length ;
        char  filler [ 8];
        char  a5ct   [ 4];
        char  a5lan  [ 4];
        char  fil1   [ 8];          /* packed fields for PLI */
        char  fil2   [ 8];          /* packed fields for PLI */
        char  fil3   [ 8];          /* packed fields for PLI */
        int   a5xbc;
        int   a5ybc;
        int   a5zbc;
    };
    struct tag_a5progx {
        char  alfx   [ 8];
        char  applx  [ 8];
    };
    struct {
        char  s_parm [ 240];
    } s_parms = {"ALF=C "};
    struct tag_a5progx a5_progx = {"FZERSAM.", "R924A5 "};
    _Packed struct tag_a5ws a5_ws = { 54,
        "CCNGCP2",
        "XFER" ,
        "C " ,
        "0000110C",
        "0000220C",
        "0000330C",
        12, 2, 0
    };

    base = atoi(argv[1]) ;
    power= atoi(argv[2]) ;
    a5_ws.a5xbc = base;
    a5_ws.a5ybc = power;
    a5_ws.a5zbc = (int) pow((double) a5_ws.a5xbc,
                          (double) a5_ws.a5ybc);
    if ((fetch_ptr = (ASM_VOID *) fetch(module)) == NULL ) {
        printf (" failed on fetch of CSP %8s module \n", module);
    }
}

```

z/OS XL C transferring control to CSP under TSO using the XFER/DXFR statement (Part 1 of 2)

Figure 176. z/OS XL C transferring control to CSP under TSO using the XFER/DXFR statement


```

    else {
        fetch_ptr (&a5_ws , &a5_progx);
        rc = release((void (*) ())fetch_ptr) ;
        if ( rc != 0 ) {
            printf ("CCNGCP2: rc from release =%d\n", rc );
        }
    }
}
}

```

z/OS XL C transferring control to CSP under TSO using the XFER/DXFR statement (Part 2 of 2)

Calling z/OS XL C from CSP

To call a z/OS XL C program from CSP:

- PLIST(OS) must be specified in the z/OS XL C program so that input parameters will not be processed by the runtime environment.
- When CSP passes a parameter list to a z/OS XL C function, the list is in a different format from what z/OS XL C expects in a normal z/OS environment. To receive the parameters, use the macro `__csplist`, found in the `csp.h` header file and described in [z/OS C/C++ Runtime Library Reference](#).

Notes:

1. PLIST(OS) must be specified in the z/OS XL C program so that input parameters will not be processed by the runtime environment.
2. When CSP passes a parameter list to a z/OS XL C function, the list is in a different format from what z/OS XL C expects in a normal z/OS environment. To receive the parameters, use the macro `__csplist`, found in the `csp.h` header file and described in [z/OS C/C++ Runtime Library Reference](#).

Example programs

Figure 177 on page 601 shows example program CCNGCP3, which shows how parameters are received from a CSP application that uses a CALL statement to transfer control. You must pass three parameters:

- An int
- A string
- A struct

```

/* this example shows how to CALL C from CSP under TSO */

#pragma runopts (plist(os))
#include <csp.h>
#include <math.h>
#include <stdlib.h>

void main()
{
    struct date {
        char yy[2];
        char mm[2];
        char dd[2];
    };
    int *parm1_ptr ;
    char *parm2_ptr ;
    struct date * parm3_ptr ;
    parm1_ptr = (int *) __csplist[0];          /* get 1st  parm */
    parm2_ptr = (char *) __csplist[1];         /* get 2nd  parm */
    parm3_ptr = (struct date *) __csplist[2];  /* get 3rd  parm */
}

```

Figure 177. CSP CALLing z/OS XL C under TSO

Figure 178 on page 602 shows example program CCNGCP4, which shows how parameters are received from a CSP application that uses an XFER/DXFR statement to transfer control. You must pass a structure.

Notes:

1. Under TSO, CSP/AD cannot use the XFER statement to transfer control to z/OS XL C.
2. Under TSO, you cannot use the DXFR statement to transfer control to CSP.

```

/* this example shows how to transfer control from CSP to C          */
/*      This program will be called from CSP through                */
/*      "XFER" or DXFR call.                                         */
/*      Parameters are passed as a working storage record            */
/*      plus 10 bytes of filler information                           */
/*      2 bytes length                                                */
/*      8 bytes filler                                                */
/*      n bytes working storage record.                               */
/*                                                                    */
#pragma runopts (plist(os))
#include <stdlib.h>
#include <csp.h>
#include <math.h>
#include <string.h>

#pragma linkage(DCGXFER,OS)
#pragma linkage(DCGCALL,OS)

void xfer_rtn ();
void call_rtn ();

struct tag_a3ws {
    short length ;
    char  filler [ 8];
    char  a3ct   [ 4];
    char  a3lan  [ 4];
    char  fil1   [ 8];          /* packed fields for PLI */
    char  fil2   [ 8];          /* packed fields for PLI */
    char  fil3   [ 8];          /* packed fields for PLI */
    int   a3xbc;
    int   a3ybc;
    int   a3zbc;
};
struct tag_a3progx {
    char  alfx   [ 8];
    char  applx  [ 8];
};

void main()
{
    _Packed struct tag_a3ws *parm1 ;
    _Packed struct tag_a3ws a3_ws ;

    parm1 = (_Packed struct tag_a3ws *) __csplist[0];
    parm1->a3zbc = (int) pow((double) parm1->a3xbc,
                           (double) parm1->a3ybc);

    if (parm1->a3zbc > 255)
        xfer_rtn(parm1);      /* xfer to csp */
    else
        call_rtn(parm1);      /* call to csp */
}

```

CSP transferring control to z/OS XL C under TSO using the XFER statement (Part 1 of 3)

Figure 178. CSP transferring control to z/OS XL C under TSO using the XFER statement

```

/*****
*/
*/
/*****
void xfer_rtn(_Packed struct tag_a3ws * parm1 )
{
    #pragma linkage (ASM_VOID,OS)
    typedef void ASM_VOID();
    ASM_VOID * fetch_ptr;

    struct tag_a3progx a3_progx = {"FZERSAM.", "R924A5 " } ;
    int rc = 0;
    char pgm_xfer [ 8] = {"DCGXFER " } ;

    if ((fetch_ptr = (ASM_VOID *) fetch(pgm_xfer)) == NULL ) {
        printf (" failed on fetch of CSP %8s module \n", pgm_xfer);
    }
    else {
        fetch_ptr (parm1, &a3_progx);
        rc = release((void (*)()) fetch_ptr) ;
        if ( rc != 0 ) {
            printf ("xfer_rtn: rc from release =%d\n", rc );
        }
    }
}
/*****
*/
*/
/*****
void call_rtn(_Packed struct tag_a3ws * parm1 )
{
    typedef void ASM_VOID();
    ASM_VOID * fetch_ptr;
    char pgm_call [ 8] = {"DCGCALL " } ;
    int rc = 0;
    struct tag_a3progx a3_progx = {"FZERSAM.", "R924A6 " } ;
    struct tag_a6rec {
        char a6ct [ 4];
        char a6lan [ 4];
        char fil1 [ 8];          /* packed fields for PLI */
        char fil2 [ 8];          /* packed fields for PLI */
        char fil3 [ 8];          /* packed fields for PLI */
        int a6xbc;
        int a6ybc;
        int a6zbc;
    };
    struct tag_a6rec a6_rec ;
    memcpy(a6_rec.a6ct ,parm1->a3ct ,4);
    memcpy(a6_rec.a6lan,parm1->a3lan,4);
    memcpy(a6_rec.fil1 ,parm1->fil1 ,8);
    memcpy(a6_rec.fil2 ,parm1->fil2 ,8);
    memcpy(a6_rec.fil3 ,parm1->fil3 ,8);
    a6_rec.a6xbc = parm1->a3xbc;
    a6_rec.a6ybc = parm1->a3ybc;
    a6_rec.a6zbc = parm1->a3zbc;
}

```

CSP transferring control to z/OS XL C under TSO using the XFER statement (Part 2 of 3)

```

if ((fetch_ptr = (ASM_VOID *) fetch(pgm_call)) == NULL ) {
    printf (" failed on fetch of CSP %s module \n", pgm_call);
}
else {
    fetch_ptr (&a3_progx, &a6_rec);
    rc = release( (void (*)()) fetch_ptr) ;
    if ( rc != 0 ) {
        printf ("CCNGCP4: rc from release =%d\n", rc );
    }
}
}
}

```

CSP transferring control to z/OS XL C under TSO using the XFER statement (Part 3 of 3)

Running under CICS control

For CSP-CICS, note that because all z/OS XL C applications running under CICS must run with AMODE=31, when passing parameters to CSP, you must either

- Pass parameters below the line, or
- Relink the CSP load library with AMODE=31

Example programs

Figure 179 on page 604 shows example program CCNGCP5, which shows how parameters are received from a CSP application that uses a CALL statement to transfer control. The z/OS XL C program is expecting to receive an int as a parameter.

```

/* this example shows how to call C from CSP under CICS, and how */
/* parameters are passed */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

main()
{
    struct tag_commarea { /* commarea passed to z/OS C from R924A1 */
        int *ptr1 ;
        int *ptr2 ;
        int *ptr3 ;
    } * ca_ptr ;          /* commarea ptr */

    int *parm1_ptr ;
    int *parm2_ptr ;
    int *parm3_ptr ;

                                /* addressability to EIB control block */
                                /* and COMMUNICATION AREA */
    EXEC CICS ADDRESS EIB(dfheiptr) COMMAREA(ca_ptr) ;
    parm1_ptr = ca_ptr->ptr1 ;
    parm2_ptr = ca_ptr->ptr2 ;
    parm3_ptr = ca_ptr->ptr3 ;

    *parm3_ptr = (int) pow((double) *parm1_ptr,
                          (double) *parm2_ptr);

    EXEC CICS RETURN;
}

```

Figure 179. CSP CALLing z/OS XL C under CICS

Figure 180 on page 605 (example program CCNGCP6) shows how parameters are received from a CSP application that uses an XFER statement to transfer control.

```

/* this example shows how to XFER control to C from CSP under CICS */
/*      XFER      CALL      */
/* R924A3 ==> CCNGCP6 ==> R924A6 */
/* R924A3 and R924A6 are CSP applications */

#include <math.h>
#include <string.h>

/* structure passed to R924A6*/
void main()
{
    struct {
        char
        _Packed struct tag_a3rec    *appl_ptr;
        } parm_ptr ;
        /* Structure received R924A3*/
    struct tag_a3rec {
        char    a3ct    [ 4];
        char    a3lan    [ 4];
        char    fil1    [ 8];
        char    fil2    [ 8];
        char    fil3    [ 8];
        int     a3xbc;
        int     a3ybc;
        int     a3zbc;
    }
        /* packed fields for PLI */
        /* packed fields for PLI */
        /* packed fields for PLI */
        /* int field 1 for z/OS C */
        /* int field 2 for z/OS C */
        /* int field 3 for z/OS C */
}

```

CSP transferring control to z/OS XL C under CICS using the XFER statement (Part 1 of 2)

Figure 180. CSP transferring control to z/OS XL C under CICS using the XFER statement

```

_Packed struct tag_a3rec  a3rec ;
char  lk_appl[16] = "USR5ALF.R924A6  " ;

struct tag_a3progx {
    char  alfx  [ 8];
    char  applx  [ 8];
};
_Packed struct tag_a3progx a3progx = {"USR5ALF.", "R924A6  "} ;
short  length_a3rec = sizeof(a3rec) ;
char  * pa3rec ;
short  i ;

/*----- start of CSP XFER-ing to C under CICS -----*/

EXEC CICS ADDRESS EIB(dfheiptr);
/* retrieve data from CSP */
EXEC CICS RETRIEVE INTO(&a3rec) LENGTH(length_a3rec) ;

a3rec.a3zbc = (int) pow((double) a3rec.a3xbc,
                      (double) a3rec.a3ybc);

/*----- end of CSP XFER-ing to C under CICS -----*/

/* call CSP to display results*/
parm_ptr.appl_ptr = lk_appl ; /* alf.application */
parm_ptr.rec3_ptr = &a3rec ;

EXEC CICS LINK PROGRAM("DCBINIT ")
               COMMAREA(parm_ptr)
               LENGTH(8) ;

if (dfheiptr->eibresp2 != 0) {
    printf("CCNGCP6: EXEC CICS LINK  returned non zero \n");
    printf("          return code. eibresp2 =%d\n",
           dfheiptr->eibresp2);
}

/*----- end of C calling CSP under CICS -----*/
EXEC CICS RETURN ;
}

```

CSP transferring control to z/OS XL C under CICS using the XFER statement (Part 2 of 2)

Figure 181 on page 607 (example program CCNGCP7) The following example program shows how parameters are received from a CSP application that uses a DXFR statement to transfer control. You must receive a structure.

```

/* this example shows how to transfer control to C from CSP under    */
/* CICS, using the DXFR statement */

/*          DXFR          XCTL( equivalent to dxfr)                */
/* R924A3 ==> CCNGCP7 ==> DCBINIT   ( appl R924A5)                */
/* R924A3 is a CSP application */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

main ()
{
    struct tag_a3rec {
        char  a3ct   [ 4];
        char  a3lan  [ 4];
        char  fil1   [ 8];          /* packed fields for PLI */
        char  fil2   [ 8];          /* packed fields for PLI */
        char  fil3   [ 8];          /* packed fields for PLI */
        int   a3xbc;
        int   a3ybc;
        int   a3zbc;
    };

```

CSP Transferring Control to z/OS XL C under CICS Using the DXFR Statement (Part 1 of 2)

Figure 181. CSP Transferring Control to z/OS XL C under CICS Using the DXFR Statement

```

/* commarea passed to C/370 from R924A3 */
struct tag_commarea {
    char a3ct [ 4] ;
    char a3lan [ 4] ;
    char fil1 [ 8] ; /* packed fields for PLI */
    char fil2 [ 8] ; /* packed fields for PLI */
    char fil3 [ 8] ; /* packed fields for PLI */
    int a3xbc;
    int a3ybc;
    int a3zbc;
} * ca_ptr ; /* commarea ptr */

struct tag_a5progc {
    char alfc [ 8] ;
    char applc [ 8] ;
    struct tag_a3rec a3rec;
} a5progc = {"USR5ALF.", "R924A5 "};

short length_a3rec = sizeof(struct tag_a3rec) ;
short length_a5progc = sizeof(struct tag_a5progc) ;

/* addressability to EIB control block */
/* and COMMUNICATION AREA */

EXEC CICS ADDRESS EIB(dfheiptr) COMMAREA(ca_ptr) ;

if (dfheiptr->eibcalen == length_a3rec ) {
    memcpy(&a5progc.a3rec, ca_ptr , length_a3rec);

    /* calculate the pow(x,y) */
    a5progc.a3rec.a3zbc = (int) pow((double) a5progc.a3rec.a3xbc,
                                   (double) a5progc.a3rec.a3ybc);

    EXEC CICS XCTL
        PROGRAM("DCBINIT ")
        COMMAREA(a5progc)
        length(length_a5progc) ;

    if (dfheiptr->eibresp2 != DFHRESP(NORMAL)) {
        printf ("CCNGCP7: failed on xctl call to DCBINIT\n");
        printf (" \n");
    }
}
else {
    printf ("CCNGCP7:length of COMMAREA is different from expected\n");
    printf (" expected %d, actual %d\n",
            length_a3rec, dfheiptr->eibcalen);
    printf (" \n");
    EXEC CICS RETURN;
}

EXEC CICS RETURN;
}

```

CSP Transferring Control to z/OS XL C under CICS Using the DXFR Statement (Part 2 of 2)

Chapter 48. Using Data Window Services (DWS)

Data Window Services (DWS) is part of the CSL (Callable Services Library). DWS gives your C or C++ program the ability to manipulate data objects (temporary data objects known as TEMPSPACE, and VSAM linear data sets).

Notes:

1. XPLINK is not supported with DWS.
2. AMODE 64 applications are not supported with DWS.

To use DWS functions with C code, you do not have to specify a linkage pragma or add any specialized code. Code the DWS function call directly inside your z/OS XL C program just as you would a call to a C or C++ library function and then link-edit the DWS module containing the function you want (such as CSRIDAC, CSRVIEW, CSRSCOT, CSRSAVE or CSRREFR) with your C or C++ program.

To use DWS functions with C++ code, you must specify C linkage for any DWS function that you use. For example, if you wished to use CSRIDAC, you would use a code fragment as shown in [Figure 182 on page 609](#).

```
/* this example shows how DWS may be used with C++ */
#include <stdlib.h>

extern "C" {
    void csridac( char*, char*, char*, char*, char*,
                  char*, long int*, char*, long int*,
                  long int*, long int*);
}

int main(void)
{
    /* Set up the parameters that will be used by CSRIDAC. */

    char op_type[6]      = "BEGIN";
    char object_type[10] = "TEMPSPACE";
    char object_name[45] = "DWS.FILE ";
    char scroll_area[4]   = "YES";
    char object_state[4] = "NEW";
    char access_mode[7]  = "UPDATE";
    long int object_size = 8;
    char object_id[9];
    long int high_offset, return_code, reason_code;

    /* Access a DWS TEMPSPACE data object. */

    csridac(op_type, object_type, object_name, scroll_area, object_state,
            access_mode, OBJECT_size, object_id, &high_offset,
            &return_code, &reason_code);

    /* INSERT ADDITIONAL CODE HERE */
}
```

Figure 182. Example using DWS and C++

At link-edit time, you should link-edit the DWS module containing the function you want, just as you would for a C program.

In DWS, the data types of the parameters are specified differently from z/OS XL C/C++ data types. When invoking DWS functions from your C or C++ program, you must specify:

- A long int data type for DWS parameters of integer (I*4) type.
- Character strings (of the required length) for DWS parameters of character type. For example, if the DWS function requires a 9-character object name (in this example we will set the object name to TEMPSPACE) you can declare the parameter in your C or C++ function as follows:

```
char object_type[9] = "TEMPSPACE";
```

For more information on DWS, see *z/OS MVS Programming: Callable Services for High-Level Languages*.

As another example, Figure 183 on page 610 is an excerpt from a C program (CCNGDW1) that shows parameter declarations for the DWS CSRIDAC function and the function call.

```
/* this example shows how DWS may be used with C */
int main(void)
{
    /* Set up the parameters that will be used by CSRIDAC. */

    char op_type[5]      = "BEGIN";
    char object_type[9]  = "TEMPSPACE";
    char object_name[45] = "DWS.FILE ";
    char scroll_area[3]   = "YES";
    char object_state[3] = "NEW";
    char access_mode[6]  = "UPDATE";
    long int object_size = 8;
    char object_id[8];
    long int high_offset, return_code, reason_code;

    /* Access a DWS TEMPSPACE data object. */

    csridac(op_type, object_type, object_name, scroll_area, object_state,
            access_mode, OBJECT_size, OBJECT_id, &high_offset,
            &return_code, &reason_code);
    /* INSERT ADDITIONAL CODE HERE */

    return 0;
}
```

Figure 183. z/OS XL C/C++ Using Data Window Services

Chapter 49. Using DB2 Universal Database

Both z/OS Language Environment and z/OS XL C/C++ provide an interface to the IBM DB2 Universal Database Licensed Program. For a list of books describing DB2, refer to “DB2” on page 774.

An XL C/C++ program requests DB2 services by using SQL statements embedded in the program. This source code is translated into host language statements that perform assignments and call a database language interface module. After DB2 processes each request, it returns processing control to the XL C/C++ program.

Any errors that occur during database processing are handled by the database product. If a program is terminated during DB2 processing, DB2 takes appropriate action, depending on the nature of termination.

Preparing an XL C/C++ application to request DB2 services

Before a C/C++ program can request DB2 services, the code with embedded SQL statements must be converted into compilable code. There two ways to do this:

- Use the XL C/C++ DB2 coprocessor (provided by the z/OS XL C compiler).
- Use the DB2 C/C++ precompiler (provided by DB2).

To ensure that you are using compatible releases of z/OS XL C/C++ and DB2, see *z/OS Program Directory*.

Refer to the [SQL | NOSQL](#) compiler option in *z/OS XL C/C++ User's Guide* whenever your application performs the following operations:

- Declares global host variables.
- Declares host variables inside functions.
- Includes a header found in SYSLIB or in the LSEARCH path.
- Puts comments at the end of selected lines in the middle of a multiline SQL statement.
- Inserts, updates, or retrieves data using a host variable.
- Embeds SQL statements in template functions or classes.

Using the XL C/C++ DB2 coprocessor

If all the SQL statements are embedded in XL C programs, you can use the XL C DB2 coprocessor to prepare the program to request DB2 services. You can either run the program through the DB2 C/C++ precompiler before you compile, or you can specify the SQL compiler option when you compile the program. For detailed information about using the SQL option, refer to [SQL | NOSQL](#) in *z/OS XL C/C++ User's Guide*. If you are compiling code with SQL in effect, refer to *z/OS Program Directory* for a complete list of SQL suboptions.

When the XL C/C++ SQL(NOSTD) option is in effect, code should be written in codepage IBM-1047 (APL)293).

The following are advantages of using the XL C/C++ DB2 coprocessor instead of the DB2 C/C++ precompiler:

- Host variable names follow the same lexical scoping rules as C/C++ variables.
- Preprocessor directives (such as **#include** and **#define**) are supported.
- Variable-length source input is supported.

Notes:

1. Typically, NOSQL is the default compiler option. If your environment is customized to make SQL the default, be aware that the compiler will attempt to call the API that contains DBRMLIB DD even if the

source code does not contain SQL statements. When that happens, DB2 generates a message that you can ignore.

2. "SQLCODE", "SQLSTATE", and "sqlca" are not valid user host variable names, because they are reserved by the compiler and DB2 coprocessor as embedded SQL keywords. Previous to the compiler embedding the DB2 coprocessor, the DB2 precompiler did not recognize these three identifiers as keywords, but the new DB2 coprocessor correctly recognizes them and will not successfully compile any source file that attempts to use them as user host variable names. If you are already using any of these reserved keywords as table column names in your program (and referencing them through identically named user host variable names), and you are migrating from a compiler release that uses the DB2 precompiler to one that uses the DB2 coprocessor, you must restructure your queries in order to avoid any conflicts with these reserved keywords. For detailed information, see [Db2 for z/OS in IBM Documentation \(www.ibm.com/docs/en/db2-for-zos\)](http://www.ibm.com/docs/en/db2-for-zos).

Using the DB2 C/C++ precompiler

The DB2 C/C++ precompiler scans source code for potentially SQL-related keywords, such as the following:

- Host variables that can be used in SQL statements in the same source.
- SQL statements that start with the token pair EXEC SQL.

While the DB2 C/C++ precompiler can fully parse the SQL syntax, it has limited capacity for parsing compiler-language-related syntax. If you are compiling code with SQL in effect, all DB2 z/OS XL C/C++ code should be written in codepage IBM-1047 (APL293).

An advantage of using the DB2 C/C++ precompiler instead of the XL C/C++ DB2 coprocessor is that you can obtain a more useful message listing by preprocessing, precompiling, and then compiling source code with embedded SQL statements. Compiler diagnostics refer to line numbers of the translated output from the DB2 C/C++ precompiler, not to the line numbers of your source code. This means that you need both the DB2 C/C++ precompiler listing and the compiler listing to work through the compilation errors. Runtime troubleshooting tools also refer to coordinates of the DB2 C/C++ precompiler output.

Using DB2 services and stored procedures with XPLINK

XL C/C++ applications that are compiled with the XPLINK option can invoke DB2 services that are called through stubs defined as **#pragma linkage(..., OS)**.

When you embed DB2 stored procedures in a program that will be compiled with XPLINK, each CREATE PROCEDURE statement must include a RUN OPTIONS clause that specifies XPLINK(ON).

Examples of how to use XL C/C++ programs to request DB2 services

The examples in this section demonstrate how to code C and C++ programs with embedded SQL statements. You can use them with either the XL C/C++ DB2 coprocessor or the DB2 C/C++ precompiler.

Example CCNGDB4 demonstrates how to code a C program with embedded SQL statements. In [Figure 184 on page 613](#), a program CCNGDB4 creates a table called CTAB1, inserts literal values into the table, and drops the table. You can use this example either by compiling the program with the SQL option in effect or by running the program through the DB2 C/C++ precompiler, and then compiling the generated code with the NOSQL option in effect.

```

/* this example demonstrates how to use SQL with C */

#include <string.h>
#include <stdio.h>

EXEC SQL INCLUDE SQLCA;

int main(void)
{
    if (CreaTab() == -1)
    {
        printf("Test Failed in table-creation.\n");
        exit(-1);
    }

    if (DropTab() == -1)
    {
        printf("Test Failed in table-dropping.\n");
        exit(-1);
    }
    printf("Test Successful.\n");
    return(0);
}

/*
 * This routine creates the table CTAB1 and inserts some values
 * into it
 */

int CreaTab(void)
{
    EXEC SQL CREATE TABLE CTAB1
        ( EMPNO      CHAR(6) NOT NULL,
          FIRSTNME   VARCHAR(12) NOT NULL,
          LASTNME    VARCHAR(15) NOT NULL,
          WORKDEPT   CHAR(3) NOT NULL,
          PHONENO     CHAR(7),
          EDUCVLV    SMALLINT,
          SALARY      FLOAT(21) );

    if (sqlca.sqlcode != 0)
    {
        printf("ERROR - SQL code returned non-zero for "
            "creation of CTAB1, received %d\n",sqlca.sqlcode);
        return(-1);
    }
}

```

Using DB2 with C (Part 1 of 2)

Figure 184. Using DB2 with C

```

/* Now insert some values into the table */

EXEC SQL INSERT INTO CTAB1 VALUES
( '097892','John','Adams','003','8883945',3,29500.00 );
EXEC SQL INSERT INTO CTAB1 VALUES
( '000002','Joe','Smith','004','8883791',NULL,25500.00 );
EXEC SQL INSERT INTO CTAB1 VALUES
( '043929','Ralph','Holland','001','8888734',1,NULL);
EXEC SQL INSERT INTO CTAB1 VALUES
( '000010','Holly','Waters','001','8884590',3,29550.00 );

if (sqlca.sqlcode != 0)
{
    printf("ERROR - SQL code returned non-zero for "
           "insert into tables, received %d\n",sqlca.sqlcode);
    return(-1);
}
return(0);
}

/*
 * This routine will drop the table.
 */

int DropTab(void)
{
    EXEC SQL DROP TABLE CTAB1;
    if (sqlca.sqlcode != 0)
    {
        printf("ERROR - SQL code returned non-zero for "
               "drop of CTAB1 received %d??\n",sqlca.sqlcode);
        return(-1);
    }
    EXEC SQL COMMIT WORK;
    return(0);
}

```

Using DB2 with C (Part 2 of 2)

Figure 185 on page 615 is a C++ code example with embedded SQL statements. The sample code creates, populates, updates, and drops a table called CTAB1V. You can use this example either by compiling the program with the SQL option in effect or by running the program through the DB2 C/C++ precompiler, and then compiling the generated code with the NOSQL option in effect.

```

#include <iostream>
using namespace std;

// The test case information
typedef char TestType;

#define NUM_ROWS 3
#define IN_VALUE {'A', 'B', 'C'}
#define OUT_VALUE {'D', 'E', 'F'}

EXEC SQL INCLUDE SQLCA;

class SqlTestTable {
public:
    // The constructor and destructor create and drop the test table
    SqlTestTable() {
        EXEC SQL CREATE TABLE CTAB1V (
            ID          INTEGER      NOT NULL,
            TESTVAR     CHAR(1)      NOT NULL
        ) IN DATABASE DSNUCOMP;

        if (sqlca.sqlcode != 0) {
            std::cout << "ERROR - SQL code returned " << sqlca.sqlcode
                << " for creation of CTAB1V.\n";
        }
    }
    ~SqlTestTable() {
        EXEC SQL DROP TABLE CTAB1V;    // Clean up the database

        if (sqlca.sqlcode != 0) {
            std::cout << "ERROR - SQL code returned " << sqlca.sqlcode
                << " for drop of CTAB1V.\n";
        }
        EXEC SQL COMMIT WORK;
    }

    int insertRow(int idToAdd, TestType inputData) {
        int returnValue = 55;

        EXEC SQL BEGIN DECLARE SECTION;
        int idForRow = idToAdd;
        TestType inputValue = inputData;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL INSERT INTO CTAB1V
            VALUES ( :idForRow, :inputValue );

        if (sqlca.sqlcode != 0) {
            std::cout << "ERROR - SQL code returned " << sqlca.sqlcode
                << " for insert into tables.\n";
            returnValue = 66; // Not returned immediately in case cleanup is needed
        }

        return returnValue;
    }
}

```

Using DB2 with C/C++ (Part 1 of 3)

Figure 185. Using DB2 with C/C++

```

int updateTable(int idToChange, TestType inputData) {
    int returnValue = 55;

    EXEC SQL BEGIN DECLARE SECTION;
        int idForRow = idToChange;
        TestType inputValue = inputData;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL UPDATE CTAB1V
        SET TESTVAR = :inputValue
        WHERE ID = :idForRow;

    if (sqlca.sqlcode != 0) {
        std::cout << "ERROR - SQL code returned "
            << sqlca.sqlcode << " for update in tables.\n";
        returnValue = 66; // Not returned immediately in case cleanup is needed
    }

    return returnValue;
}

int checkTable(int idToCheck, TestType value) {
    int returnValue = 55;

    // Try other format variable names
    EXEC SQL BEGIN DECLARE SECTION;
        int idForRow = idToCheck;
        TestType check_var;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT TESTVAR INTO :check_var
        FROM CTAB1V
        WHERE ID = :idForRow;

    if (sqlca.sqlcode != 0) {
        std::cout << "ERROR - SQL code returned "
            << sqlca.sqlcode << " for SELECT of the data.\n";
        return 66; // Return immediately since no cleanup; nothing else to be done
    }

    if (check_var != value) {
        std::cout << "ERROR - Value in table
            " << check_var << " is not the expected value " << value << ".\n";
        returnValue = 66; // Not returned immediately in case cleanup is needed
    }

    return returnValue;
}
};

```

Using DB2 with C/C++ (Part 2 of 3)

```

int main(void) {
    SqlTestTable testTable;    // Creates the tables
    int i = 0;
    int returnValue = 55;

    TestType aLongVariableName[NUM_ROWS] = IN_VALUE;
    TestType expectedResults[NUM_ROWS] = OUT_VALUE;

    // SQL Declare's Not needed. Added to see what happens if not used as SQL vars.
    EXEC SQL BEGIN DECLARE SECTION;
        TestType inValue;
        TestType outValue;
    EXEC SQL END DECLARE SECTION;

    // Populate the table using non-host variables as function parameters
    for (i = 0; i < NUM_ROWS; i++) {
        returnValue = testTable.insertRow(i, aLongVariableName[i]);

        if (returnValue != 55) {
            return returnValue;
        }
    }

    // Check to see if the insert went OK using host variables as function parms
    for (i = 0; i < NUM_ROWS; i++) {
        inValue = aLongVariableName[i];
        returnValue = testTable.checkTable(i, inValue);

        if (returnValue != 55) {
            return returnValue;
        }
    }

    // Update the values using host variables as function parameters
    for (i = 0; i < NUM_ROWS; i++) {
        outValue = expectedResults[i];
        returnValue = testTable.updateTable(i, outValue);

        if (returnValue != 55) {
            return returnValue;
        }
    }

    // Check to see if the update went OK using non-host variables as function parms
    for (i = 0; i < NUM_ROWS; i++) {
        returnValue = testTable.checkTable(i, expectedResults[i]);

        if (returnValue != 55) {
            return returnValue;
        }
    }

    return returnValue;    // Deletes the table through the destructor
}

```

Using DB2 with C/C++ (Part 3 of 3)

Chapter 50. Using Graphical Data Display Manager (GDDM)

The Graphical Data Display Manager (GDDM) provides programmers with a comprehensive set of functions for displaying or printing information in the most effective manner. The major functions provided are:

- A windowing system that the user can tailor to display selected information
- Support for presentation and interaction through the keyboard
- Comprehensive graphics support
- Fonts, including support for double-byte character sets (DBCS)
- Business image support
- Saving and restoring graphics pictures
- Support for many types of display terminals, printers, and plotters.

Because GDDM uses OS-style linkage, calls from C to GDDM require the `#pragma linkage pragma`, as in the following example:

```
#pragma linkage(identifier, OS)
```

In C++ code, calls to and from GDDM require that any GDDM functions you use be prototyped as `extern "OS"`, as shown in the following example:

```
extern "OS" {  
    ASREAD( int *type, int *num, int *count );  
    CHAATT( int num, int *attrib );  
    CHHATT( int num, int *attrib );  
}
```

Because C++ does not support `#pragma linkage`, any existing C code that you are moving to C++ should use the `extern "OS"` specification instead.

When linking a GDDM application, you must add the GDDM library to your SYSLIB concatenation.

Notes:

1. XPLINK is not supported by GDDM.
2. AMODE 64 applications are not supported by GDDM.

Examples

The following examples demonstrate the interface between C and GDDM by drawing a polar chart to compare the characteristics of two cars.

[Figure 186 on page 620](#) shows a sample program (CCNGGD1) using GDDM and C.

```

/* this example demonstrates the use of C and GDDM */
#include <string.h>
#pragma linkage(asread,OS)
#pragma linkage(chaatt,OS)
#pragma linkage(chhatt,OS)
#pragma linkage(chhead,OS)
#pragma linkage(chkatt,OS)
#pragma linkage(chkey,OS)
#pragma linkage(chnatt,OS)
#pragma linkage(chnoff,OS)
#pragma linkage(chnote,OS)
#pragma linkage(chpolr,OS)
#pragma linkage(chset,OS)
#pragma linkage(chxlab,OS)
#pragma linkage(chxlat,OS)
#pragma linkage(chxtic,OS)
#pragma linkage(chyrng,OS)
#pragma linkage(chyset,OS)
#pragma linkage(fsinit,OS)
#pragma linkage(fsterm,OS)
/* Arrays are expected for int * and float * */
/* char * can be an array or a string */
extern int asread (int *type, int *num, int *count);
extern int chaatt (int num, int *attrib);
extern int chhatt (int num, int *attrib);
extern int chkatt (int num, int *attrib);
extern int chkey (int, int, char *);
extern int chnatt (int num, int *attrib);
extern int chnoff (double, double);
extern int chnote (char *string, int num, char *title);
extern int chpolr (int, int, float *xdata, float *ydata);
extern int chset (char *character);
extern int chxlab (int num, int, char *);
extern int chxlat (int num, int *attrib);
extern int chxtic (double x, double y);
extern int chyrng (double from, double to);
extern int chyset (char *character);
extern int fsinit (void);
extern int fsterm (void);
/*****
** Attribute arrays used for the chart. **
*****/
int i ;
int h_attrs[4] = { 3, 3, 0, 175 }; /* Head text attribute */
int n_attrs[4] = { 7, 3, 0, 200 }; /* Note text attribute */
int a_attrs[2] = { 7, 1 }; /* X-axis color and line */
int xl_attrs[1] = { 5 }; /* X-label color */
int k_attrs[1] = { 5 }; /* Key text color */
int type, num, count ;

```

Example using GDDM and C (Part 1 of 2)

Figure 186. Example using GDDM and C

```

float x_data[8] = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };
float y_data[16] = {
    14190.0, 260.0, 0.21, 0.066, 83.3, 6.0, 19.1, 14190.0,
    12986.0, 290.0, 0.23, 0.066, 95.6, 5.0, 16.2, 12986.0 };
float maxvals[16] = {
    15000.0, 300.0, 0.25, 0.070, 100.0, 6.0, 20.0, 15000.0,
    15000.0, 300.0, 0.25, 0.070, 100.0, 6.0, 20.0, 15000.0 };

int main(void)
{
    fsinit();
    chhatt( 4, h_attrs);
    chhead( 40, "TWO CARS COMPARED USING SEVEN PARAMETERS");
    chaatt( 2, a_attrs);
    chxtic( 1.0, 0.0);
    chxlat( 1, x1_attrs);
    chxlab( 7, 31,
        "PURCHASE PRICE ;      $15,000      INSURANCE      ;$300/YEAR      "
        "$0.25/MILE ;SERVICING      $0.070/MILE      ;FUEL      "
        "      100 BHP/TON; POWER/WT RATIO      6;      SEATS"
        "      BAGGAGE SPACE;      20 CU FT");
    chyrng ( 0.5,1.0);
    chyset( "NOAXIS");
    chyset( "NOLABEL");
    chyset( "PLAIN");
    chset( "KBOX");
    chkatt( 1, k_attrs);
    chkey( 2, 5, "CAR ACAR B");
    for(i=0; i<16; ++i)
        y_data[i] = y_data[i] / maxvals[i];
    chpolr(2, 8, x_data, y_data);
    chnatt( 4, n_attrs);
    chnoff( 0.0, 0.53);
    chnote( "Z2", 1, "+");
    chset("BNOTE");
    n_attrs[3] = 75;
    chnatt(4, n_attrs);
    chnoff(0.0, 0.60);
    chnote("Z2", 12, "CENTER VALUE");
    chnoff(0.0, 0.55);
    chnote("Z2", 23, "= 1/2 X PERIMETER VALUE");
    /*****
    ** Issue a screen read. When any interrupt is generated **
    ** by the terminal operator, the program terminates.      **
    *****/
    asread( &type, &num, &count);
    fsterm();
    exit(0);
}

```

Example using GDDM and C (Part 2 of 2)

Figure 187 on page 622 is a similar example program (CCNGGD2) in C++.

```

/* this example demonstrates the use of C++ and GDDM */
#include <stdlib.h>
#include <string.h>
/* Arrays are expected for int * and float *      */
/* char * can be an array or a string              */
extern "OS" {
    int asread (int *type, int *num, int *count);
    int chaatt (int num, int *attrib);
    int chhatt (int num, int *attrib);
    int chkatt (int num, int *attrib);
    int chkey  (int, int, char *);
    int chhead (int, char *);
    int chnatt (int num, int *attrib);
    int chnoff (double, double);
    int chnote (char *string, int num, char *title);
    int chpolr (int, int, float *xdata, float *ydata);
    int chset  (char *character);
    int chxlab (int num, int, char *);
    int chxlat (int num, int *attrib);
    int chxtic (double x, double y);
    int chying (double from, double to);
    int chyset (char *character);
    int fsinit (void);
    int fsterm (void);
}

```

Example using GDDM and C++ (Part 1 of 2)

Figure 187. Example using GDDM and C++

```

/*****
** Attribute arrays used for the chart. **
*****/
int i ;
int h_attrs[4] = { 3, 3, 0, 175 }; /* Head text attribute */
int n_attrs[4] = { 7, 3, 0, 200 }; /* Note text attribute */
int a_attrs[2] = { 7, 1 }; /* X-axis color and line */
int xl_attrs[1] = { 5 }; /* X-label color */
int k_attrs[1] = { 5 }; /* Key text color */
int type, num, count ;

float x_data[8] = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };
float y_data[16] = {
    14190.0, 260.0, 0.21, 0.066, 83.3, 6.0, 19.1, 14190.0,
    12986.0, 290.0, 0.23, 0.066, 95.6, 5.0, 16.2, 12986.0 };
float maxvals[16] = {
    15000.0, 300.0, 0.25, 0.070, 100.0, 6.0, 20.0, 15000.0,
    15000.0, 300.0, 0.25, 0.070, 100.0, 6.0, 20.0, 15000.0 };
int main(void)
{
    fsinit();
    chhatt( 4, h_attrs);
    chhead( 40, "TWO CARS COMPARED USING SEVEN PARAMETERS");
    chaatt( 2, a_attrs);
    chxtic( 1.0, 0.0);
    chxlat( 1, xl_attrs);
    chxlab( 7, 31,
        "PURCHASE PRICE ; $15,000 INSURANCE ;$300/YEAR "
        "$0.25/MILE ;SERVICING $0.070/MILE ;FUEL "
        " 100 BHP/TON; POWER/WT RATIO 6; SEATS"
        " BAGGAGE SPACE; 20 CU FT");
    chyrng ( 0.5,1.0);
    chyset( "NOAXIS");
    chyset( "NOLABEL");
    chyset( "PLAIN");
    chset( "KBOX");
    chkatt( 1, k_attrs);
    chkey( 2, 5, "CAR ACAR B");
    for(i=0; i<16; ++i)
        y_data[i] = y_data[i] / maxvals[i];
    chpolr(2, 8, x_data, y_data);
    chnatt( 4, n_attrs);
    chnoff( 0.0, 0.53);
    chnote( "Z2", 1, "+");
    chset("BNOTE");
    n_attrs[3] = 75;
    chnatt(4, n_attrs);
    chnoff(0.0, 0.60);
    chnote("Z2", 12, "CENTER VALUE");
    chnoff(0.0, 0.55);
    chnote("Z2", 23, "= 1/2 X PERIMETER VALUE");
/*****
** Issue a screen read. When any interrupt is generated **
** by the terminal operator, the program terminates. **
*****/
    asread( &type, &num, &count);
    fsterm();
    exit(0);
}

```

Example using GDDM and C++ (Part 2 of 2)

Chapter 51. Using the Information Management System (IMS)

This chapter explains how the Information Management System (IMS) and z/OS XL C/C++ coordinate error handling, and describes the limitations to using IMS with z/OS XL C/C++.

z/OS XL C/C++ provides the `ctdli()` C library function to invoke IMS facilities (see [z/OS C/C++ Runtime Library Reference](#) for more information).

You can also invoke IMS facilities with the callable service CEETDLI which is provided by the z/OS Language Environment. The CEETDLI interface performs essentially the same functions as `ctdli()`, but it offers some advantages, particularly if you plan to run an ILC application in IMS. If you use the CEETDLI interface instead of `ctdli()`, condition handling is improved because of the coordination between z/OS Language Environment and IMS condition handling facilities. For complete information on the CEETDLI interface, see [z/OS Language Environment Programming Guide](#).

For a description of writing IMS batch and online programs in C or C++, see the appropriate book listed in “IMS/ESA” on page 774.

To use IMS from z/OS XL C/C++, you must keep the following in mind:

- The file `<ims.h>` must be included in the program.
- `PLIST(OS)` and `TARGET(IMS)` must be used to compile IMS z/OS XL C/C++ application programs. `PLIST(OS)` establishes the correct parameter list format when invoked under IMS, and `TARGET(IMS)` establishes the correct operating environment. The descriptions that follow use the compile-time options. The compile-time option `PLIST(OS)` can alternatively be specified using `#pragma runopts(PLIST(OS))`.
- `TARGET(IMS)` is mandatory, as it establishes the correct operating environment. `PLIST(OS)` must also be used if the program is the initial `main()` program called under IMS. Programs in nested enclaves do not need to be compiled with `PLIST(OS)`.
- The argument count (`argc`) will be set to one (1), and the first element in the argument vector (`argv[0]`) will contain a NULL string when compiled with z/OS XL C compiler and the deprecated `#pragma runopts(PLIST(IMS))` is specified in the source. Otherwise, the argument vector will contain the name of the program.
- IMS provides a language interface module (DFSLI000) that gives a common interface to IMS and DL/I. This module must be link-edited with the application program.

The rest of this chapter is based on the assumption that you are using the `ctdli()` interface.

Notes:

1. AMODE 64 applications are not supported in an IMS environment.
2. As of V1R2, a non-XPLINK Standard C++ Library DLL allows support for the Standard C++ Library in the IMS subsystem. For further information, see [Binding z/OS XL C/C++ programs in z/OS XL C/C++ User's Guide](#).
3. XPLINK applications are supported under the IMS environment.

Handling errors

The IMS environments are sensitive to errors and error-handling issues. A failing IMS transaction or program can potentially corrupt an IMS database. IMS must know about the failure of a transaction or program that has been updating a database so that it can back out any updates made by that failing program.

z/OS XL C/C++ provides extensive error-handling facilities for the programmer, but special steps are required to coordinate IMS and C or C++ error handling so that IMS can do its database rollbacks when a program fails.

When you are using IMS from C or C++:

- Run your C or C++ program with the TRAP(ON) option, and use IMS interfaces by calling the `ctdli()` library function. If your application programs also use SQL facilities provided by DB2, you must modify the user exit CEEBXITA to add the user abend codes 777 and 778 to prevent the error handler from trapping these abends. This will allow deadlocks to be successfully resolved by IMS. See [z/OS Language Environment Programming Guide](#) for more information on CEEBXITA.
- The `ctdli()` library function will keep track of calls to and returns from IMS. If an abend or program check occurs and the C or C++ error handler gets control, it can determine if the problem arose on the IMS side of the interface or on the C or C++ side.
- If a program check or abend occurs in IMS, when the C or C++ exception handler gets control, it immediately issues an ABEND. The IMS Region Controller gets control next and ensures that the integrity of the database is preserved.
- If a program check occurs in the C or C++ program rather than in IMS, all the facilities of C or C++ error handling apply, provided that you meet certain conditions when you code your program. For any error condition that arises, you must do one of the following:
 1. Resolve the error completely so that the application can continue.
 2. Have IMS back out the program's updates by issuing a rollback call to IMS, and then terminate the program.
 3. Make sure that the program terminates abnormally and provide an installation-modified runtime user exit that turns all abnormal terminations into operating system ABENDs to effect IMS rollbacks. See [z/OS Language Environment Programming Guide](#) for more information.

The errors you most likely can fix in your program are arithmetic exception (SIGFPE) conditions. It is unlikely that you can resolve other types of program checks or system abends in your program.

Any program that invokes IMS by way of some other IMS interface should be executed with TRAP(OFF). You should be sure that the program contains code to issue a rollback call to IMS before terminating after an error. Refer to [z/OS Language Environment Programming Reference](#) for more information about the limitations of using TRAP(OFF).

Other considerations

A *program communication block* (PCB) is a control block used by IMS to describe results of a DL/I call (DB PCB) or the results of a message retrieval or insertion (I/O PCB) made by your program. A valid PCB is one that has been correctly initialized by IMS and passed to you through your C or C++ program. For details on PCBs, refer to “IMS/ESA” on page 774. See also the sample C-IMS and C++-IMS programs in [z/OS C/C++ Runtime Library Reference](#).

If you are running a C program under TSO or IMS, be aware of the effects of PLIST(OS), ENV(IMS), and their combinations when specified using the `#pragma runopts` preprocessor directive. [Table 104 on page 626](#) shows the combinations of PLIST(OS) and ENV(IMS) and the resulting PCB generated under each of the environments.

Table 104. PCB generated for C program under TSO and IMS

Combination	Running under TSO	Running under IMS
ENV(IMS) only	Invalid PCB	Valid PCB
PLIST(OS) only	Null PCB	Null PCB
ENV(IMS) and PLIST(OS)	Invalid PCB	Valid PCB

For more information on the runtime options ENV and PLIST, see [z/OS Language Environment Programming Reference](#).

If you are running a C or C++ program under TSO or IMS, be aware of the effects of specifying compiler options PLIST(OS), TARGET(IMS), and their combinations. Table 105 on page 627 shows the combinations of PLIST(OS) and TARGET(IMS) and the resulting PCB generated under each of the environments.

Table 105. PCB generated for C or C++ program under TSO and IMS

Combination	Running under TSO	Running under IMS
TARGET(IMS) only	Invalid PCB	Valid PCB
PLIST(OS) only	Null PCB	Null PCB
TARGET(IMS) and PLIST(OS)	Invalid PCB	Valid PCB

For both C and C++, specifying PLIST(OS) under either TSO or IMS results in an argc value of 1 (one), and argv[0] = NULL. For more information on the compiler options TARGET(IMS) and PLIST(OS), see [z/OS XL C/C++ User's Guide](#).

Examples

The sample C++ program CCNGIM1 (Figure 188 on page 627) makes an IMS call and checks the return code status of the call in IMS batch. Header file CCNGIM3 (Figure 190 on page 630) is included by this program.

```
/* this is an example of how to use IMS with C++ */

#pragma runopts(env(ims),plist(os))
#include <ims.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "ccngim3.h"

int main(void) {
/*****
/* Declare the database pointer control blocks for each database */
*****/
    PCB_STRUCT_8_TYPE *locdb_ptr,*orddb_ptr;
```

C++ Program using IMS (Part 1 of 2)

Figure 188. C++ Program using IMS

```

/*****
/*   IO areas used for DL/I calls                               */
*****/

    auto IOA2 aio_area, a2io_area;
    static IOA2 sio_area;
    IOA2 *io_area;
/*****
/*   SSAs for DL/I calls                                       */
*****/

    static char qual0[] =      "ORDER   (ORDKEY   =333333)";
    static char qual1[] =      "ORDITEM  ";
    static char qual2[] =      "DELIVERY ";
    static int six   = 6;
    static int four   = 4;
    static char gu[5]  = "GU  ";
    static char isrt[5] = "ISRT";

    int rc;
    int failed = 0;      /* Indicate if any part of test case failed.
*/

/*****
/*   Get the pointers to the databases from the parameter list */
*****/

    locdb_ptr = (__pcblist[1]);
    orddb_ptr = (__pcblist[2]);
/*****
/*   Make some calls to the database and change its contents   */
*****/

    printf("IMS Test starting\n");

    io_area = (IOA2 *)malloc(sizeof(IOA2));

/*****
/*   Issue a DL/I call with arguments below the line (using CTDLI) */
*****/

/*****
/* The first parameter for ctdli is an int specifying the number of */
/* arguments-this parameter was optional under C but is mandatory   */
/* under C++                                                         */
*****/
    rc = ctdli(six,gu,orddb_ptr,&aio_area,qual0,qual1,qual2);

    if ((orddb_ptr->stat_code[0] == ' ' && orddb_ptr->stat_code[1] == '
')
        && (rc == 0))
        printf("Call to CTDLI returned successfully\n");
    else
    {
        printf("Call to CTDLI returned status of %c%c.\n",
            orddb_ptr->stat_code[0],orddb_ptr->stat_code[1]);
        failed = 1;
    }
    if (failed == 0)
        printf("Test Successful\n");
    else printf("Test Failed");

    return(0);
}

```

C++ Program using IMS (Part 2 of 2)

Figure 189 on page 629 shows a sample C program (CCNGIM2) that makes an IMS call and checks the return code status of the call in IMS batch. Header file CCNGIM3 (Figure 190 on page 630) is included by this program.

```

/* This is an example of how to use IMS with C */

#pragma runopts(env(ims),plist(os))
#include <ims.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "ccngim3.h"

int main(void) {
/*****
/*  Declare the database pointer control blocks for each database */
*****/

    PCB_STRUCT_8_TYPE *locdb_ptr,*orddb_ptr;

/*****
/*  IO areas used for DL/I calls
*****/

    auto IOA2 aio_area, a2io_area;
    static IOA2 sio_area;
    IOA2 *io_area;

/*****
/*  SSAs for DL/I calls
*****/

    static char qual0[] =      "ORDER   (ORDKEY   =333333)";
    static char qual1[] =      "ORDITEM  ";
    static char qual2[] =      "DELIVERY ";
    static int six      = 6;
    static int four     = 4;
    static char gu[4]    = "GU  ";
    static char isrt[4]  = "ISRT";
    int rc;
    int failed = 0;    /* Indicate if any part of test case failed. */

/*****
/*  Get the pointers to the databases from the parameter list
*****/

    locdb_ptr = (__pcblist[1]);
    orddb_ptr = (__pcblist[2]);

/*****
/*  Make some calls to the database and change its contents
*****/

    printf("IMS Test starting\n");

    io_area = malloc(sizeof(IOA2));

```

C Program using IMS (Part 1 of 2)

Figure 189. C Program using IMS

```

/*****
/* Issue a DL/I call with arguments below the line (using CTDLI) */
*****/

rc = ctdli(six,gu,orddb_ptr,&aiio_area,qual0,qual1,qual2);

if ((orddb_ptr->stat_code[0] == ' ' && orddb_ptr->stat_code[1] == '
')
    && (rc == 0))
    printf("Call to CTDLI returned successfully\n");
else
{
    printf("Call to CTDLI returned status of %c%c.\n",
        orddb_ptr->stat_code[0],orddb_ptr->stat_code[1]);
    failed = 1;
}
if (failed == 0)
    printf("Test Successful\n");
else printf("Test Failed");

return(0);
}

```

C Program using IMS (Part 2 of 2)

Figure 190 on page 630 shows the header file (CCNGIM3) that is used by both the C and the C++ examples in [Figure 188 on page 627](#) and [Figure 189 on page 629](#), respectively.

```

/* this header file is used with the IMS example */

/*-----*/
/* DB PCB */
/*-----*/
typedef struct {
    char db_name[8];
    char seg_level[2];
    char stat_code[2];
    char proc_opt[4];
    int dli;
    char seg_name[8];
    int len_kfb;
    int no_senseg;
    char key_fb[2];
} DB_PCB;
/*-----*/
/* IO PCB */
/*-----*/
typedef struct {
    char term[8];
    char ims_res[2];
    char stat_code[2];
    char date[4];
    char time[4];
    int input_seq;
    char output_mess[8];
    char mod_nme[8];
    char user_id[8];
} IO_AREA;

```

Header file for IMS example (Part 1 of 2)

Figure 190. Header file for IMS example

```

/*-----*/
/*  SPA DATA      */
/*-----*/
typedef struct {
    short int uosplth;
    char uospres1[4];
    char uosptran[8];
    char uospuser;
    char fill[85];
} SPA_DATA;
/*-----*/
/*  INPUT MESSAGE  */
/*-----*/
typedef struct {
    short int ll;
    char zz[2];
    char fill[2];
    char numb[4];
    char nme[6];
} IN_MSG;
/*-----*/
/*  OUTPUT MESSAGE */
/*-----*/
typedef struct {
    short int ll;
    char z1;
    char z2;
    char fill[2];
    char sca[2];
} OUT_MSG;
/*-----*/
/*  IO AREA        */
/*-----*/
typedef struct {
    char key[20];
} IOA1;

typedef struct {
    char item[40];
} IOA2;

```

Header file for IMS example (Part 2 of 2)

Chapter 52. Using the Query Management Facility (QMF)

The z/OS XL C/C++ compiler's support of the Query Management Facility (QMF) interface, a query and report writing facility, enables you to write applications through the SAA callable interface. You can create applications to perform a variety of tasks such as data entry, query building, administration aids, and report analysis.

The z/OS XL C++ compiler itself does not support QMF. However, QMF can be accessed through C code that is statically or dynamically called from C++.

You must include the header file `DSQCOMM.C` (provided with the QMF application), which contains the function and structure definitions necessary to use the QMF interface.

For information on how to write your z/OS XL C/C++ applications with the QMF interface, see [product page for IBM® Query Management Facility \(www.ibm.com/products/db2-qmf\)](http://www.ibm.com/products/db2-qmf).

Notes:

1. AMODE 64 applications are not supported by QMF.
2. XPLINK is not supported by QMF.

Example programs

Figure 191 on page 633 is a sample program (CCNGQM1) that demonstrates the interface between the QMF facility and the z/OS XL C/C++ compiler.

```
/* this example shows how to use the interface between QMF and C */
#include <string.h>
#include <stdlib.h>
#include <DSQCOMM.C> /* QMF header file */

int main(void)
{
    struct dsqcomm communication_area; /* found in DSQCOMM */

    /******
    /* Query interface command length and commands */
    /******
    signed long command_length;
    static char start_query_interface [] = "START";
    static char set_global_variables [] = "SET GLOBAL";
    static char run_query [] = "RUN QUERY Q1";
    static char print_report [] = "PRINT REPORT (FORM=F1)";
    static char end_query_interface [] = "EXIT";

    /******
    /* Query command extension, number of parameters and lengths */
    /******
    signed long number_of_parameters;
    signed long keyword_lengths[10];
    signed long data_lengths[10];
```

QMF interface example (Part 1 of 3)

Figure 191. QMF interface example

```

/*****
/* Variable data type constants */
/*****
static char char_data_type[] = DSQ_VARIABLE_CHAR;
static char int_data_type[] = DSQ_VARIABLE_FINT;

/*****
/* Keyword parameter and value for START command */
/*****
static char start_keywords[] = "DSQSCMD";
static char start_keyword_values[] = "USERCMD1";

/*****
/* Keyword parameter and value for SET command */
/*****
#define SIZE_VAL 8
char set_keywords[3][SIZE_VAL];
signed long set_values[3];

/*****
/* Start a Query Interface Session */
/*****
number_of_parameters = 1;
command_length = sizeof(start_query_interface);
keyword_lengths[0] = sizeof (start_keywords);
data_lengths[0] = sizeof(start_keyword_values);
dsqcice(&communication_area,
        &command_length,
        START_query_interface[0],
        &number_of_parameters,
        &keyword_lengths[0],
        START_keywords[0],
        &data_lengths[0],
        START_keyword_values[0],
        char_data_type[0]);

/*****
/* Set numeric values into query using SET command */
/*****
number_of_parameters = 3;
command_length = sizeof(set_global_variables);
strcpy(set_keywords[0], "MYVAR01");
strcpy(set_keywords[1], "SHORT");
strcpy(set_keywords[2], "MYVAR03");
keyword_lengths[0] = SIZE_VAL;
keyword_lengths[1] = SIZE_VAL;
keyword_lengths[2] = SIZE_VAL;
data_lengths[0] = sizeof(long);
data_lengths[1] = sizeof(long);
data_lengths[2] = sizeof(long);
set_values[0] = 20;
set_values[1] = 40;
set_values[2] = 84;
dsqcice(&communication_area,
        &command_length,
        &set_global_variables[0],
        &number_of_parameters,
        &keyword_lengths[0],
        &set_keywords[0],
        &data_lengths[0],
        &set_values[0],
        &int_data_type[0]);

```

QMF interface example (Part 2 of 3)

```

/*****
/* Run a Query
*****/
command_length = sizeof(run_query);
dsqcic(&communication_area, &command_length,
      &run_query[0]);

/*****
/* Print the results of the query
*****/
command_length = sizeof(print_report);
dsqcic(&communication_area, &command_length,
      &print_report[0]);

/*****
/* End the query interface session
*****/
command_length = sizeof(end_query_interface);
dsqcic(&communication_area, &command_length,
      &end_query_interface[0]);

return 0;
}

```

QMF interface example (Part 3 of 3)

Figure 192 on page 635 is a sample program (CCNGQM2) that demonstrates how a C++ program may call a C program (see Figure 193 on page 636) that accesses QMF.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

extern "C" {
    int Gen_Report(void);
}
int main( int argc, char *argv[])
{
    int cmd;

    if (argc < 2 )
    {
        printf("ERROR - program takes at least one parm");
    }
    else
    {
        cmd=argv[1][0];
        cmd=toupper(cmd);
        switch (cmd)
        {
            case 'R':
            {
                Gen_Report();
                break;
            }
            default:
                printf("%d is an invalid option.\n");
        }
    }
}

```

Figure 192. C++ Calling a C program that accesses QMF

Sample program CCNGQM3 (Figure 193 on page 636) is called from the C program shown in Figure 192 on page 635.

```

/*
this example shows how C++ can access QMF by way of a C program */
/* part 2 of 2-this file is called from C */
/* other file is CCNGQM2 */

#include <string.h>
#include <stdlib.h>
#include <DSQCOMM.C> /* QMF header file */

int Gen_Report(void)
{
    struct dsqcomm communication_area; /* found in DSQCOMM */

    /******
    /* Query interface command length and commands */
    /******
    signed long command_length;
    static char start_query_interface [] = "START";
    static char set_global_variables [] = "SET GLOBAL";
    static char run_query [] = "RUN QUERY Q1";
    static char print_report [] = "PRINT REPORT (FORM=F1)";
    static char end_query_interface [] = "EXIT";

    /******
    /* Query command extension, number of parameters and lengths */
    /******
    signed long number_of_parameters;
    signed long keyword_lengths[10];
    signed long data_lengths[10];

    /******
    /* Variable data type constants */
    /******
    static char char_data_type[] = DSQ_VARIABLE_CHAR;
    static char int_data_type[] = DSQ_VARIABLE_FINT;

    /******
    /* Keyword parameter and value for START command */
    /******
    static char start_keywords[] = "DSQSCMD";
    static char start_keyword_values[] = "USERCMD1";

    /******
    /* Keyword parameter and value for SET command */
    /******
    #define SIZE_VAL 8
    char set_keywords[3][SIZE_VAL];
    signed long set_values[3];

```

C program that accesses QMF (Part 1 of 3)

Figure 193. C program that accesses QMF

```

/*****
/* Start a Query Interface Session
*/
/*****
    number_of_parameters = 1;
    command_length = sizeof(start_query_interface);
    keyword_lengths[0] = sizeof (start_keywords);
    data_lengths[0] = sizeof(start_keyword_values);
    dsqcice(&communication_area,
            &command_length,
            &start_query_interface[0],
            &number_of_parameters,
            &keyword_lengths[0],
            &start_keywords[0],
            &data_lengths[0],
            &start_keyword_values[0],
            &char_data_type[0]);

/*****
/* Set numeric values into query using SET command
*/
/*****
    number_of_parameters = 3;
    command_length = sizeof(set_global_variables);
    strcpy(set_keywords[0], "MYVAR01");
    strcpy(set_keywords[1], "SHORT");
    strcpy(set_keywords[2], "MYVAR03");
    keyword_lengths[0] = SIZE_VAL;
    keyword_lengths[1] = SIZE_VAL;
    keyword_lengths[2] = SIZE_VAL;
    data_lengths[0] = sizeof(long);
    data_lengths[1] = sizeof(long);
    data_lengths[2] = sizeof(long);
    set_values[0] = 20;
    set_values[1] = 40;
    set_values[2] = 84;
    dsqcice(&communication_area,
            &command_length,
            &set_global_variables[0],
            &number_of_parameters,
            &keyword_lengths[0],
            &set_keywords[0],
            &data_lengths[0],
            &set_values[0],
            &int_data_type[0]);

```

C program that accesses QMF (Part 2 of 3)

```

/*****
/* Run a Query
*/
/*****
    command_length = sizeof(run_query);
    dsqcic(&communication_area, &command_length,
           &run_query[0]);

/*****
/* Print the results of the query
*/
/*****
    command_length = sizeof(print_report);
    dsqcic(&communication_area, &command_length,
           &print_report[0]);

/*****
/* End the query interface session
*/
/*****
    command_length = sizeof(end_query_interface);
    dsqcic(&communication_area, &command_length,
           &end_query_interface[0]);

    exit(0);
}

```

C program that accesses QMF (Part 3 of 3)

Part 8. Internationalization: Locales and Character Sets

This part includes the following topics related to Locales and Character Sets:

- [Chapter 53, “Introduction to locale,” on page 641](#)
- [Chapter 54, “Building a locale,” on page 645](#)
- [Chapter 55, “Customizing a locale,” on page 691](#)
- [Chapter 56, “Customizing a time zone,” on page 697](#)
- [Chapter 57, “Definition of S370 C, SAA C, and POSIX C locales,” on page 699](#)
- [Chapter 58, “Code set conversion utilities,” on page 707](#)
- [Chapter 59, “Coded character set considerations with locale functions,” on page 731](#)
- [Chapter 60, “Bidirectional language support,” on page 747](#)

Chapter 53. Introduction to locale

This topic introduces locales and how they relate to the internationalization of programs.

Internationalization in programming languages

Internationalization in programming languages is a concept that comprises *externally stored cultural data*, a set of *programming tools* to create such cultural data, a set of *programming interfaces* to access this data, and a set of *programming methods* that enable you to use provided interfaces to write programs that do not make any assumptions about the cultural environments they run in. Such programs modify their behavior according to the user's cultural environment, specified during the program's execution.

Elements of internationalization

The typical elements of cultural environment are as follows:

Native language

The text that the executing program uses to communicate with a user or environment, that is, the natural language of the end user.

Character sets and coded character sets

Map an alphabet, the characters used in a particular language, onto the set of hexadecimal values (code points) that uniquely identify each character. This mapping creates the coded character set, which is uniquely identified by the character set it encodes, the set of code point values, and the mapping between these two.

For example IBM-273, also known as the German Code Page, and IBM-297, also known as the French Code Page, are two coded character sets which assign different EBCDIC encodings in the hexadecimal range 40 to FE to the same Latin Alphabet Number 1. IBM S/390 systems in Germany and France both use this Latin 1 alphabet, which is specified by International Standard ISO/IEC 8859-1. However, systems in Germany are configured for encodings of this alphabet given by IBM-273; whereas, systems in France are configured for encodings of this alphabet given by IBM-297.

IBM-1027, Japanese Latin Code Page, is another example of a coded character set. It assigns EBCDIC encodings in the hexadecimal range 40 to FE to characters specified by Japanese Industrial Standard JIS X 201-1978 plus encodings for a few more Latin characters selected by IBM. The resulting alphabet defined by IBM-1027 consists of some characters found in Latin Alphabet Number 1 and some Katakana characters. IBM S/390 systems in Japan are configured for encodings of this alphabet assigned by IBM-1027.

Collating and ordering

The relative ordering of characters used for sorting.

Character classification

Determines the type of character (alphabetic, numeric, and so forth) represented by a code point.

Character case conversion

Defines the mapping between uppercase and lowercase characters within a single character set.

Date and time format

Defines the way date and time data are formatted (names of weekdays and months; order of month, day, and year, and so forth).

Format of numeric and non-numeric numbers

Define the way numbers and monetary units are formatted with commas, decimal points, and so forth.

z/OS XL C/C++ Support for internationalization

The z/OS XL C/C++ compiler and library support of internationalization is based on the IEEE POSIX P1003.2 and X/Open Portability Guide standards for global locales and coded character set conversion. See Chapter 54, “Building a locale,” on page 645 for more information about locales.

Locales and localization

A *locale* is a collection of data that encodes information about the cultural environment. *Localization* is an action that establishes the cultural environment for an application by selecting the active locale. Only one locale can be active at one time, but a program can change the active locale at any time during its execution. The active locale affects the behavior of the locale-sensitive interfaces for the entire program. This is called the *global locale model*.

Locale-sensitive interfaces

The z/OS C/C++ runtime library provides many interfaces to manipulate and access locales. You can use these interfaces to write internationalized C programs. The following list summarizes all the z/OS XL C/C++ library functions which affect or are affected by the current locale.

Selecting locale

Changing the characteristics of the user's cultural environment by changing the current locale:

`setlocale()`

Querying locale

Retrieving the locale information that characterizes the user's cultural environment:

Monetary and numeric formatting conventions:

`localeconv()`

Date and time formatting conventions:

`localdtconv()`

User-specified information:

`nl_langinfo()`

Encoding of the variant part of the portable character set:

`getsyntax()`

Character set identifier:

`csid()`, `wcsid()`

Classification of characters:

Single-byte characters:

`isalnum()`, `isalpha()`, `isblank()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`

Wide characters:

`iswalnum()`, `iswalpha()`, `iswblank()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `wctype()`, `iswctype()`

Character case mapping:

Single-byte characters:

`tolower()`, `toupper()`

Wide characters:

`towlower()`, `towupper()`

Multibyte character and multibyte string conversion:

`mblen()`, `mbrlen()`, `mbtowc()`, `mbrtowc()`, `wctomb()`, `wcrtomb()`, `mbstowcs()`, `mbsrtowcs()`, `wcstombs()`, `wcsrtombs()`, `mbsinit()`, `wctob()`, `mbrtoc16()`, `mbrtoc32()`, `c16rtomb()`, `c32rtomb()`

String conversions to arithmetic:

strtod(), wcstod(), strtol(), wcstol(), strtoul(), wcstoul(), atof(), atoi(), atol()

String collating:

strcoll(), strxfrm(), wcscoll(), wcsxfrm()

Character display width:

wcswidth(), wcwidth()

Date, time, and monetary formatting:

strftime(), strptime(), wcsftime(), mktime(), ctime(), gmtime(), localtime(),
strfmon()

Formatted input/output:

printf() (and family of functions), scanf() (and family of functions), vswprintf(), swprintf(),
swscanf(), snprintf(), vsnprintf()

Processing regular expressions:

regcomp(), regexexec()

Wide character unformatted input/output:

fgetwc(), fgetws(), fputwc(), fputws(), getwc(), getwchar(), putwc(), putwchar(),
ungetwc()

Response matching:

rpmatch()

Collating elements:

ismccollet(), strtocoll(), colltostr(), collequiv(), collrange(), collorder(),
cclass(), maxcoll(), getmccoll(), getwmccoll()

Chapter 54. Building a locale

Cultural information is encoded in the locale source file using the locale definition language. One locale source file characterizes one cultural environment.

The locale source file is processed by the locale compilation tool, called the `localedef` tool.

To enhance portability of the locale source files, certain information related to the character sets can be encoded using the symbolic names of characters. The mapping between the symbolic names and the characters they represent and its associated hexadecimal value is defined in the *character set description file* or *chamap* file.

Figure 194 on page 645 shows the conceptual model of the locale build process.

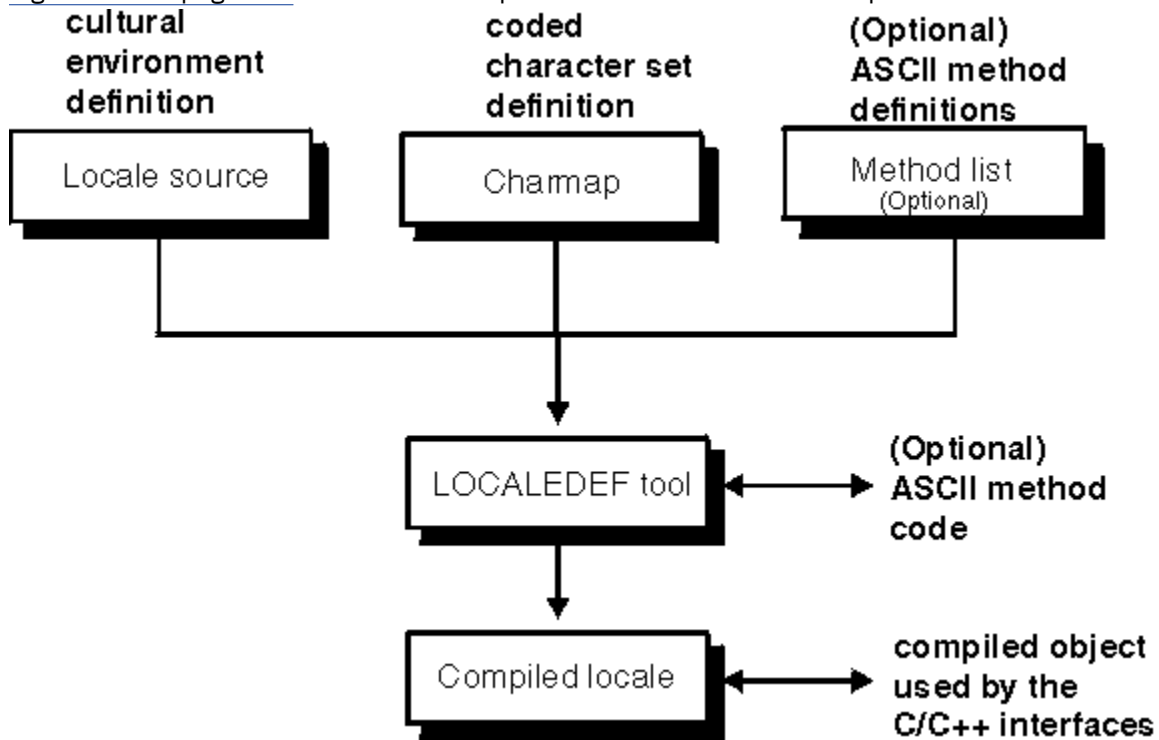


Figure 194. Conceptual model of the locale build process

Limitations of enhanced ASCII

This section explains under what conditions you can use Enhanced ASCII.

- A subset of C headers and functions is provided in ASCII. For more information, see [z/OS C/C++ Runtime Library Reference](#).
- The only way to get to the ASCII version of functions and the external variables **environ** and **tzname** is to use the appropriate IBM header files.
- ASCII applications may read, but not update, environment variables using the **environ** external variable. Updates to the environment variables using **environ** in an ASCII application causes unpredictable results and may result in an abend. Language Environment maintains two equivalent arrays of environment variables when running an ASCII application, one with EBCDIC encodings and the other with ASCII encodings. All ASCII compile units that use the **environ** external variable must include `<stdlib.h>` so that **environ** can be mapped to access the ASCII encoded environment strings. If `<stdlib.h>` is not included, **environ** will refer to the EBCDIC representation of the environment variable strings.

Enhanced ASCII provides limited conversion of ASCII to EBCDIC, and EBCDIC to ASCII. The character set or alphabet that is associated with any locale consists of the following:

- A common, XPG4-defined subset of characters such as POSIX portable characters
- A unique, locale-specific subset of characters such as NLS characters

The conversion only applies to the portable subset of characters that are associated with a locale. Only the EBCDIC IBM-1047 encoding of portable characters is supported.

You might encounter unexpected results in the following situations:

- If Enhanced ASCII applications run in locales that contain non-Latin Alphabet Number 1 NLS characters, C-RTL functions might copy some of the locale's non-Latin 1 NLS characters into buffers that the application is writing to stdout or another files in the z/OS UNIX file system. The non-Latin Alphabet Number 1 NLS characters would then cause problems during automatic conversion.
- Language Environment applications select non-English message files. If your NATLANG runtime option is not UEN or ENU, messages directed to the Language Environment message file are not converted to ASCII.

Using the charmap file

The charmap file defines a mapping between the symbolic names of characters and the hexadecimal values associated with the character in a given coded character set. Optionally, it can provide the alternate symbolic names for characters. Characters in the locale source file can be referred to by their symbolic names or alternate symbolic names, thereby allowing for writing generic locale source files independent of the encoding of the character set they represent.

Each charmap file must contain at least the definition of the portable character set and the character symbolic names associated with each character. The characters in the portable character set and the corresponding symbolic names, and optional alternate symbolic names, are defined in [Table 106 on page 646](#).

<i>Table 106. Characters in portable character set and corresponding symbolic names</i>				
Symbolic Name	Alternate Name	Character	Hex Value (EBCDIC)	Hex Value (ASCII)
<NUL>			00	00
<tab>	<SE10>		05	09
<vertical-tab>	<SE12>		0b	0b
<form-feed>	<SE13>		0c	0c
<carriage-return>	<SE14>		0d	0d
<newline>	<SE11>		15	0a
<backspace>	<SE09>		16	08
<alert>	<SE08>		2f	07
<space>	<SP01>		40	20
<period>	<SP11>	.	4b	2e
<less-than-sign>	<SA03>	<	4c	3c
<left-parenthesis>	<SP06>	(4d	28
<plus-sign>	<SA01>	+	4e	2b
<ampersand>	<SM03>	&	50	26
<right-parenthesis>	<SP07>)	5d	29
<semicolon>	<SP14>	;	5e	3b

Table 106. Characters in portable character set and corresponding symbolic names (continued)

Symbolic Name	Alternate Name	Character	Hex Value (EBCDIC)	Hex Value (ASCII)
<hyphen>	<SP10>	-	60	2d
<hyphen-minus>	<SP10>	-	60	2d
<slash>	<SP12>	/	61	2f
<solidus>	<SP12>	/	61	2f
<comma>	<SP08>	,	6b	2c
<percent-sign>	<SM02>	%	6c	25
<underscore>	<SP09>	_	6d	5f
<low-line>	<SP09>	_	6d	5f
<greater-than-sign>	<SA05>	>	6e	3e
<question-mark>	<SP15>	?	6f	3f
<colon>	<SP13>	:	7a	3a
<apostrophe>	<SP05>	'	7d	27
<equals-sign>	<SA04>	=	7e	3d
<quotation-mark>	<SP04>	"	7f	22
<a>	<LA01>	a	81	61
	<LB01>	b	82	62
<c>	<LC01>	c	83	63
<d>	<LD01>	d	84	64
<e>	<LE01>	e	85	65
<f>	<LF01>	f	86	66
<g>	<LG01>	g	87	67
<h>	<LH01>	h	88	68
<i>	<LI01>	i	89	69
<j>	<LJ01>	j	91	6a
<k>	<LK01>	k	92	6b
<l>	<LL01>	l	93	6c
<m>	<LM01>	m	94	6d
<n>	<LN01>	n	95	6e
<o>	<LO01>	o	96	6f
<p>	<LP01>	p	97	70
<q>	<LQ01>	q	98	71
<r>	<LR01>	r	99	72
<s>	<LS01>	s	a2	73
<t>	<LT01>	t	a3	74
<u>	<LU01>	u	a4	75
<v>	<LU01>	v	a5	76

Table 106. Characters in portable character set and corresponding symbolic names (continued)

Symbolic Name	Alternate Name	Character	Hex Value (EBCDIC)	Hex Value (ASCII)
<w>	<LW01>	w	a6	77
<x>	<LX01>	x	a7	78
<y>	<LY01>	y	a8	79
<z>	<LZ01>	z	a9	7a
<A>	<LA02>	A	c1	41
	<LB02>	B	c2	42
<C>	<LC02>	C	c3	43
<D>	<LD02>	D	c4	44
<E>	<LE02>	E	c5	45
<F>	<LF02>	F	c6	46
<G>	<LG02>	G	c7	47
<H>	<LH02>	H	c8	48
<I>	<LI02>	I	c9	49
<J>	<LJ02>	J	d1	4a
<K>	<LK02>	K	d2	4b
<L>	<LL02>	L	d3	4c
<M>	<SM02>	M	d4	4d
<N>	<LN02>	N	d5	4e
<O>	<LO02>	O	d6	4f
<P>	<LP02>	P	d7	50
<Q>	<LQ02>	Q	d8	51
<R>	<LR02>	R	d9	52
<S>	<LS02>	S	e2	53
<T>	<LT02>	T	e3	54
<U>	<LU02>	U	e4	55
<V>	<LV02>	V	e5	56
<W>	<LW02>	W	e6	57
<X>	<LX02>	X	e7	58
<Y>	<LY02>	Y	e8	59
<Z>	<LZ02>	Z	e9	5a
<zero>	<ND10>	0	f0	30
<one>	<ND01>	1	f1	31
<two>	<ND02>	2	f2	32
<three>	<ND03>	3	f3	33
<four>	<ND04>	4	f4	34
<five>	<ND05>	5	f5	35

Table 106. Characters in portable character set and corresponding symbolic names (continued)

Symbolic Name	Alternate Name	Character	Hex Value (EBCDIC)	Hex Value (ASCII)
<six>	<ND06>	6	f6	36
<seven>	<ND07>	7	f7	37
<eight>	<ND08>	8	f8	38
<nine>	<ND09>	9	f9	39
<vertical-line>	<SM13>		(4f)	7c
<exclamation-mark>	<SP02>	!	(5a)	21
<dollar-sign>	<SC03>	\$	(5b)	24
<circumflex>	<SD15>	^	(5f)	5e
<circumflex-accent>	<SD15>	^	(5f)	5e
<grave-accent>	<SD13>		(79)	60
<number-sign>	<SM01>	#	(7b)	23
<commercial-at>	<SM05>	@	(7c)	40
<tilde>	<SD19>		(a1)	7e
<left-square-bracket>	<SM06>	[(ad)	5b
<right-square-bracket>	<SM08>]	(bd)	5d
<left-brace>	<SM11>	{	(c0)	7b
<left-curly-bracket>	<SM11>	{	(c0)	7b
<right-brace>	<SM14>	}	(d0)	7d
<right-curly-bracket>	<SM14>	}	(d0)	7d
<backslash>	<SM07>	\	(e0)	5c
<reverse-solidus>	<SM07>	\	(e0)	5c

The portable character set is the basis for the syntactic and semantic processing of the localedef tool, and for most of the utilities and functions that access the locale object files. Therefore the portable character set must always be defined. It is conceptually divided into two parts:

Invariant

Characters for which encoding must be constant among all charmap files. The required encoded values are specified in [Table 106 on page 646](#). If any of these values change, the behavior of any utilities and functions on z/OS XL C/C++ is unpredictable. For example, if you are using charmaps such as Turkish IBM-1026 or Japanese IBM-290, where the characters encoded vary from the encoding in [Table 106 on page 646](#), you may get unpredictable results with the utilities and functions.

Variant

Characters for which encoding may vary from one EBCDIC charmap file to another. Only the following characters are allowed in this group:

```
<backslash>
<right-brace>
<left-brace>
<right-square-bracket>
<left-square-bracket>
<circumflex>
<tilde>
<exclamation-mark>
<number-sign>
<vertical-line>
<dollar-sign>
```

```
<commercial-at>  
<grave-accent>
```

The default EBCDIC encoding of each variant character is shown by a hexadecimal value in parentheses in [Table 106 on page 646](#). It is equivalent to the encoding in code page 1047.

The charmap file is divided into two main sections:

1. the charmap section, or CHARMAP
2. the character set identifier section, or CHARSETID

The following definitions can precede the two sections listed above. Each consists of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one or more <blank>s, followed by the value to be assigned to the symbol.

<code_set_name>

The string literal containing the name of the coded character set name (IBM-1047, IBM-273, etc.)

<mb_cur_max>

the maximum number of bytes in a multibyte character which can be set to a value between 1 and 4. EBCDIC locales have mb_cur_max settings of either 1 or 4. ASCII locales have mb_cur_max settings of 1, 2 or 3.

If it is 1, each character in the character set defined in this charmap is encoded by a one-byte value. If it is 4, each character in the character set defined in this charmap is encoded by a one-, two-, three-, or four-byte value. If it is not specified, the default value of 1 is assumed. If a value of other than 1 or 4 is specified for an EBCDIC locale, a warning message is issued and the default value of 1 is assumed.

For ASCII locales mb_cur_max is defined as 1, 2 or 3. The value 1 means the same as for EBCDIC locales, while the values 2 and 3 mean 2 and 3 bytes per character respectively.

<mb_cur_min>

The minimum number of bytes in a multibyte character. Can be set to 1 only. If a value of other than 1 is specified, a warning message is issued and the default value of 1 is assumed.

<escape_char>

Specifies the escape character that is used to specify hexadecimal or octal notation for numeric values. It defaults to the hexadecimal value 0xe0, which represents the \ character in the coded character set IBM-1047. For portability among the EBCDIC based systems, the escape character has been redefined to the / or <slash> character in all IBM-supplied charmap files, with the following statement:

```
<escape_char>  /
```

<comment_char>

Denotes the character chosen to indicate a comment within a charmap file. It defaults to the hexadecimal value 0x7b, which represents the # character in the coded character set IBM-1047. For portability among the EBCDIC based systems, the comment character has been redefined to the % or <percent-sign> character in all IBM-supplied charmap files, with the following statement:

```
<comment_char>  %
```

<shift_out>

Specifies the value of the shift-out control character that indicates the start of a string of double-byte characters. If specified, it must be the value of the EBCDIC shift-out (SO) character (hexadecimal value 0x0e). It is ignored if the <mb_cur_max> value is 1.

<shift_in>

Specifies the value of the shift-in control character that indicates the end of a string of double-byte characters. If specified, it must be the value of the EBCDIC shift-in (SI) character (hexadecimal value 0x0f). It is ignored if the <mb_cur_max> value is 1.

The CHARMAP section

The CHARMAP section defines the values for the symbolic names representing characters in the coded character set. Each charmap file must define at least the portable character set. The character symbolic names or alternate symbolic names (or both) must be used to define the portable character set. These are shown in [Table 106 on page 646](#).

Additional characters can be defined by the user with symbolic character names.

The CHARMAP section starts with the line containing the keyword CHARMAP, and ends with the line containing the keywords END CHARMAP. CHARMAP and END CHARMAP must both start in column one.

The character set mapping definitions are all the lines between the first and last lines of the CHARMAP section. The formats of the character set mappings for this section are as follows:

```
"%s %s %s\n", <symbolic-name>, <encoding>, <comments>
"%s...%s %s %s\n", <symbolic-name>, <symbolic-name>, <encoding>, <comments>
```

The first format defines a single symbolic name and a corresponding encoding. A symbolic name is one or more characters with visible glyphs, enclosed between angle brackets.

For reasons of portability, a symbolic name should include only the characters from the invariant part of the portable character set. If you use variant characters or decimal or hexadecimal notation in a symbolic name, the symbolic name will not be portable. A character following an escape character is interpreted as itself; for example, the sequence <\\> represents the symbolic name \> enclosed within angle brackets, where the backslash \ is the escape character. If / is the escape character, the sequence <///> represents the symbolic name />. In the supplied charmap files, the escape character has been redefined to the forward slash /.

The second format defines a group of symbolic names associated with a range of values. The two symbolic names are comprised of two parts, a prefix and suffix. The prefix consists of zero or more non-numeric invariant visible glyph characters and is the same for both symbolic names. The suffix consists of a positive decimal integer. The suffix of the first symbolic name must be less than or equal to the suffix of the second symbolic name. As an example, <j0101>...<j0104> is interpreted as the symbolic names <j0101>, <j0102>, <j0103>, <j0104>. The common prefix is 'j' and the suffixes are '0101' and '0104'.

The encoding part can be written in one of two forms:

```
<escape-char><number> (single byte value)
<escape-char><number><escape-char><number> (double byte value)
```

The number can be written using octal, decimal, or hexadecimal notation. Decimal numbers are written as a 'd' followed by 2 or 3 decimal digits. Hexadecimal numbers are written as an 'x' followed by 2 hexadecimal digits. An octal number is written with 2 or 3 octal digits. As an example, the single byte value x1F could be written as '\37', '\x1F', or '\d31'.

The double byte value of 0x1A1F could be written as '\32\37', '\x1A\x1F', or '\d26\d31'.

In lines defining ranges of symbolic names, the encoded value is the value for the first symbolic name in the range (the symbolic name preceding the ellipsis). Subsequent names defined by the range have encoding values in increasing order.

When constants are concatenated for multibyte character values, they must be of the same type, and are interpreted in byte order from first to last with the least significant byte of the multibyte character specified by the last constant. Each value is then prepended by the byte value of <shift_out> and appended with the byte value of <shift_in>. Such a string represents one EBCDIC multibyte character, as the following example shows:

```
<escape_char> /
<comment_char> %
<mb_cur_max> 4
<mb_cur_min> 1
<shift-out> /x0e
<shift-in> /x0f
CHARMAP
```

```
% many definition lines
<j0101>...<j0104>      /d129/d254
%many definition lines
END CHARMAP
```

is interpreted as:

```
<j0101>      /d129/d254
<j0102>      /d129/d255
<j0103>      /d130/d0
<j0104>      /d130/d1
```

It produces four 4-byte long multibyte EBCDIC characters:

```
<j0101>      x0Ex81xFEx0F
<j0102>      x0Ex81xFFx0F
<j0103>      x0Ex82x00x0F
<j0104>      x0Ex82x01x0F
```

The CHARSETID section

The character set identifier section of the charmap file maps the symbolic names defined in the CHARMAP section to a character set identifier.

Note: The two functions `csid()` and `wcsid()` query the locales and return the character set identifier for a given character. This information is not currently used by any other library function.

The CHARSETID section starts with a line containing the keyword `CHARSETID`, and ends with the line containing the keywords `END CHARSETID`. Both `CHARSETID` and `END CHARSETID` must begin in column 1. The lines between the first and last lines of the CHARSETID section define the character set identifier for the defined coded character set.

The character set identifier mappings are defined as follows:

```
"%s %c", <symbolic-name>, <value>
"%c %c", <value>, <value>
"%s...s %c", <symbolic-name>, <symbolic-name>, <value>
"%c...%c %c", <value>, <value>, <value>
"%s...%c %c", <symbolic-name>, <value>, <value>
"%c...s %c", <value>, <symbolic-name>, <value>
```

The individual characters are specified by the symbolic name or the value. The group of characters are specified by two symbolic names or by two numeric values (or combination) separated by an ellipsis (...). The interpretation of ranges of values is the same as specified in the CHARMAP section. The character set identifier is specified by a numeric value.

```
<comment_char>      %
<escape_char>       /
<code_set_name>      "IBM-930"
<mb_cur_max>         4
<mb_cur_min>         1
<shift_out>          /x0e
<shift_in>           /x0f

%
%      CHARMAP
%

CHARMAP
...
<j0110>              /x42/x5a
<j0111>...<j0112>    /x43/xbe
<judc2001>...<judc2094> /x72/x8d
...
END CHARMAP

%
%      CHARSETID
%

CHARSETID
...
```

```

<j0110>                                1
<j0111>...<j0112>                      1
<judc2001>...<judc2094>                3
...
END CHARSETID

```

Locale source files

Locales are defined through the specification of a locale definition file. The locale definition contains one or more distinct locale category source definitions and not more than one definition of any category. Each category controls specific aspects of the cultural environment. A category source definition is either the explicit definition of a category or the copy directive, which indicates that the category definition should be copied from another locale definition file.

ASCII locales must be specified using only the characters from the portable character set, and all character references must be symbolic names, not explicit code point values.

The definition file is composed of an optional definition section for the escape and comment characters to be used, followed by the category source definitions. Comment lines and blank lines can appear anywhere in the locale definition file. If the escape and comment characters are not defined, default code points are used (xE0 for the escape character and x7B for the comment character, respectively). The definition section consists of the following optional lines, where <character> in both cases is a single-byte character to be used:

```

escape_char    <character>
comment_char   <character>

```

For example, the following statement defines the escape character in this file to be ' / ' (the <slash> character).

```

escape_char    /

```

Locale definition files passed to the `localedef` utility are assumed to be in coded character set IBM-1047.

To ensure portability among EBCDIC systems, you should redefine these characters to characters from the invariant part of the portable character set. The suggested redefinition is:

```

escape_char    /
comment_char   %

```

This suggested redefinition is used in all locale definition files supplied by IBM. For reasons of portability, you should use the suggested redefinition in all your customized locale definition files. See [Chapter 55, “Customizing a locale,” on page 691](#) for information about customizing locales. These two redefinitions should be placed in the first lines of the locale definition source file, before any of the redefined characters are used.

Each category source definition consists of a category header, a category body, and a category trailer, in that order.

category header

consists of the keyword naming the category. Each category name starts with the characters `LC_`. The following category names are supported: `LC_CTYPE`, `LC_COLLATE`, `LC_NUMERIC`, `LC_MONETARY`, `LC_TIME`, `LC_MESSAGES`, `LC_TOD`, and `LC_SYNTAX`.

The `LC_TOD` and `LC_SYNTAX` categories, if present, must be the last two categories in the locale definition file.

category body

consists of one or more lines describing the components of the category. Each component line has the following format:

```

<identifier>  <operand1>
<identifier>  <operand1>;<operand2>;...;<operandN>

```

<identifier> is a keyword that identifies a locale element, or a symbolic name that identifies a collating element. <operand> is a character, collating element, or string literal. Escape sequences can be specified in a string literal using the <escape_character>. If multiple operands are specified, they must be separated by semicolons. White space can be before and after the semicolons.

category trailer

consists of the keyword END followed by one or more <blank>s and the category name of the corresponding category header.

Figure 195 on page 654 is an example of locale source containing the header, body, and trailer.

```
escape_char /
comment_char %
%
% Here is a simple locale definition file consisting of one
% category source definition, LC_CTYPE.
%
LC_CTYPE
upper <A>;...;<Z>
END LC_CTYPE
```

Figure 195. Example locale source containing header, body, and trailer

You do not have to define each category. Where category definitions are absent from the locale source, default definitions are used.

In each category, the keyword copy followed by a string specifies the name of an existing locale to be used as the source for the definition of this category.

If the locale is not found, an error is reported and no locale output is created.

For the batch (EDC(X) LDEF proc) and TSO (LOCALDEF) commands, the name must be the member name of a partitioned data set allocated to the EDCLOCL DD statement. For the UNIX System Services localedef command, the copy keyword specifies the path name of the source file.

You can continue a line in a locale definition file by placing an escape character as the last character on the line. This continuation character is discarded from the input. Even though there is no limitation on the length of each line, for portability reasons it is suggested that each line be no longer than 2048 characters (bytes). There is no limit on the accumulated length of a continued line. You cannot continue comment lines on a subsequent line by using an escaped <newline>.

Individual characters, characters in strings, and collating elements are represented using symbolic names, as defined below. Characters can also be represented as the characters themselves, or as octal, hexadecimal, or decimal constants. If you use non-symbolic notation, the resultant locale definition file may not be portable among systems and environments. The left angle bracket (<) is a reserved symbol, denoting the start of a symbolic name; if you use it to represent itself, you must precede it with the escape character.

The following rules apply to the character representation:

1. A character can be represented by a symbolic name, enclosed within angle brackets. The symbolic name, including the angle brackets, must exactly match a symbolic name defined in the charmap file. The symbolic name is replaced by the character value determined from the value associated with the symbolic name in the charmap file.

The use of a symbolic name not found in the charmap file constitutes an error, unless the name is in the category LC_CTYPE or LC_COLLATE, in which case it constitutes a warning. Use of the escape character or right angle bracket within a symbolic name is invalid unless the character is preceded by the escape character. For example:

<c>;<c-cedilla>

specifies two characters whose symbolic names are "c" and "c-cedilla"

"<M><a><y>"

specifies a 3-character string composed of letters represented by symbolic names "M", "a", and "y"

"<a><\>>"

specifies a 2-character string composed of letters represented by symbolic names "a" and ">" (assuming the escape character is \)

If the character represented by the symbolic name is a multibyte character defined by 2 byte values in the `charmap` file, and the shift-out and shift-in characters are defined, the value is enclosed within shift-out and shift-in characters before the `localedef` utility processes it any further.

2. A character can represent itself. Within a string, the double quotation mark, the escape character, and the left angle bracket must be escaped (preceded by the escape character) to be interpreted as the characters themselves. For example:

c

'c' character represented by itself

"may"

represents a 3-character string, each character within the string represented by itself

"%%>"

represents the three character long string ">", where the escape character is defined as %

3. A character can be represented as an octal constant. An octal constant is specified as the escape character followed by two or more octal digits. Each constant represents a byte value.

For example: `\131 "\212\129\168" \16\66\193\17`

4. A character can be represented as a hexadecimal constant. A hexadecimal constant is specified as the escape character, followed by an x, followed by two or more hexadecimal digits. Each constant represents a byte value.

For example: `\x83 "\xD4\x81\xA8"`

5. A character can be represented as a decimal constant. A decimal constant is specified as the escape character followed by a d followed by two or more decimal digits. Each constant represents a byte value.

For example: `\d131 "\d212\d129\d168" \d14\d66\d193\d15`

For multibyte characters, the entire encoding sequence, including the shift-out and shift-in characters, must be present. Otherwise, the sequence of bytes not enclosed between the shift-out and shift-in characters are interpreted as a sequence of single byte characters.

Multibyte characters can be represented by concatenating constants specified in byte order with the last constant specifying the least significant byte of the character. If the sequence of octal, hexadecimal, or decimal constants is to represent a multibyte character, it must be enclosed in shift-out and shift-in constants. For example: `\x0e\x42\xC1\x0f`

LC_CTYPE category

This category defines character classification, case conversion, and other character attributes. In this category, you can represent a series of characters by using three adjacent periods as an ellipsis symbol (...). An ellipsis is interpreted as including all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value following the ellipsis.

An ellipsis is valid within a single encoded character set. For example, `\x30;...;\x39;` includes in the character class all characters with encoded values from X'30' to X'39'.

The keywords recognized in the LC_CTYPE category are listed below. In the descriptions, the term "automatically included" means that it is not an error either to include or omit any of the referenced characters; they are assumed by default even if the entire keyword is missing and accepted if present. If a keyword is specified without any arguments, the default characters are assumed.

When a character is automatically included, it has an encoded value dependent on the charmap file in effect. If no charmap file is specified, the encoding of the encoded character set IBM-1047 is assumed.

copy

Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keywords are present in this category. If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

charclass

Defines one or more locale-specific character class names as strings separated by semicolons. Each named character class can then be defined subsequently in the LC_CTYPE definition. A character class name consists of at least one and at most {CHARCLASS_NAME_MAX} bytes of alphanumeric characters from the portable filename character set. The first character of a character class name cannot be a digit. The name cannot match any of the LC_CTYPE keywords defined in this document.

upper

Defines characters to be classified as uppercase letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. The uppercase letters A through Z are automatically included in this class. The `isupper()` and `iswupper()` functions test for any character and wide character, respectively, included in this class.

lower

Defines characters to be classified as lowercase letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. The lowercase letters a through z are automatically included in this class. The `islower()` and `iswlower()` functions test for any character and wide character, respectively, included in this class.

alpha

Defines characters to be classified as letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. Characters classified as either `upper` or `lower` are automatically included in this class. The `isalpha()` and `iswalph()` functions test for any character or wide character, respectively, included in this class.

digit

Defines characters to be classified as numeric digits. Only the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, can be specified. If they are, they must be in contiguous ascending sequence by numerical value. The digits 0 through 9 are automatically included in this class. The `isdigit()` and `iswdigit()` functions test for any character or wide character, respectively, included in this class.

space

Defines characters to be classified as whitespace characters. No character defined for the keywords `upper`, `lower`, `alpha`, `digit`, or `xdigit` can be specified for `space`. The characters `<space>`, `<form-feed>`, `<newline>`, `<carriage-return>`, `<horizontal-tab>`, and `<vertical-tab>`, and any characters defined in the class `blank` are automatically included in this class. The functions `isspace()` and `iswspace()` test for any character or wide character, respectively, included in this class.

cntrl

Defines characters to be classified as control characters. No character defined for the keywords `upper`, `lower`, `alpha`, `digit`, `punct`, `graph`, `print`, or `xdigit` can be specified for `cntrl`. The functions `iscntrl()` and `iswcntrl()` test for any character or wide character, respectively, included in this class.

punct

Defines characters to be classified as punctuation characters. No character defined for the keywords `upper`, `lower`, `alpha`, `digit`, `cntrl`, or `xdigit`, or as the `<space>` character, can be specified. The functions `ispunct()` and `iswpunct()` test for any character or wide character, respectively, included in this class.

graph

Defines characters to be classified as printing characters, not including the `<space>` character. Characters specified for the keywords `upper`, `lower`, `alpha`, `digit`, `xdigit`, and `punct` are automatically included. No character specified in the keyword `cntrl` can be specified for `graph`.

The functions `isgraph()` and `iswgraph()` test for any character or wide character, respectively, included in this class.

print

Defines characters to be classified as printing characters, including the `<space>` character. Characters specified for the keywords `upper`, `lower`, `alpha`, `digit`, `xdigit`, `punct`, and the `<space>` character are automatically included. No character specified in the keyword `cntrl` can be specified for `print`. The functions `isprint()` and `iswprint()` test for any character or wide character, respectively, included in this class.

xdigit

Defines characters to be classified as hexadecimal digits. Only the characters defined for the class `digit` can be specified, in contiguous ascending sequence by numerical value, followed by one or more sets of six characters representing the hexadecimal digits 10 through 15, with each set in ascending order (for example, `A, B, C, D, E, F, a, b, c, d, e, f`). The digits 0 through 9, the uppercase letters A through F, and the lowercase letters a through f are automatically included in this class. The functions `isxdigit()` and `iswxdigit()` test for any character or wide character, respectively, included in this class.

blank

Defines characters to be classified as blank characters. The characters `<space>` and `<tab>` are automatically included in this class. The functions `isblank()` and `iswblank()` test for any character or wide character, respectively, included in this class.

toupper

Defines the mapping of lowercase letters to uppercase letters. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma; the pair is enclosed in parentheses. The first character in each pair is the lowercase letter, and the second is the corresponding uppercase letter. Only characters specified for the keywords `lower` and `upper` can be specified for `toupper`. The lowercase letters a through z, their corresponding uppercase letters A through Z, are automatically in this mapping, but only when the `toupper` keyword is omitted from the locale definition. It affects the behavior of the `toupper()` and `towupper()` functions for mapping characters and wide characters, respectively.

tolower

Defines the mapping of uppercase letters to lowercase letters. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma; the pair is enclosed by parentheses. The first character in each pair is the uppercase letter, and the second is its corresponding lowercase letter. Only characters specified for the keywords `lower` and `upper` can be specified. If the `tolower` keyword is omitted from the locale definition, the mapping is the reverse mapping of the one specified for the `toupper`. The `tolower` keyword affects the behavior of the `tolower()` and `towlower()` functions for mapping characters and wide characters, respectively.

You may define additional character classes using your own keywords. A maximum of 31 classes are supported in total: the 12 standard classes, and up to 19 user-defined classes. The defined classes affect the behavior of `wctype()` and `iswctype()` functions.

[Figure 196 on page 658](#) is an example of the definition of the `LC_CTYPE` category.

Figure 196. Example LC CTYPE definition

A collation sequence definition defines the relative order between collating elements (characters and multicharacter collating elements) in the locale. This order is expressed in terms of collation values. It assigns each element one or more collation values (also known as collation weights). The collation sequence definition is used by regular expressions, pattern matching, and sorting and collating functions. The following capabilities are provided:

- Note:** This is an IBM extension; therefore, locales that use it may not be portable to localedef tools developed by other vendors.

Collating rules

Collation rules consist of an ordered list of collating order statements, ordered from lowest to highest. The <NULL> character is considered lower than any other character. The ellipsis symbol ("...") is a special collation order statement. It specifies that a sequence of characters collate according to their encoded character values. It causes all characters with values higher than the value of the <collating identifier> in the preceding line, and lower than the value for the <collating identifier> on the following line, to be placed in the character collation order between the previous and the following collation order statements in ascending order according to their encoded character values.

The use of the ellipsis symbol ties the definition to a specific coded character set and may preclude the definition from being portable among implementations.

The ellipsis symbol can precede or succeed the ellipsis symbol and may also have weights on the same line.

A collating order statement describes how a collating identifier is weighted.

Each <collating-identifier> consists of a character, <collating-element>, <collating-symbol>, or the special symbol UNDEFINED. The order in which collating elements are specified determines the character order sequence, such that each collating element is considered lower than the elements following it. The <NULL> character is considered lower than any other character. Weights are expressed as characters, <collating-symbol>s, <collating-element>s, or the special symbol IGNORE. A single character, a <collating-symbol>, or a <collating-element> represents the relative position in the character collating sequence of the character or symbol, rather than the character or characters themselves. Thus rather than assigning absolute values to weights, a particular weight is expressed using the relative "order value" assigned to a collating element based on its order in the character collation sequence.

A <collating-element> specifies multicharacter collating elements, and indicates that the character sequence specified by the <collating-element> is to be collated as a unit and in the relative order specified by its place.

A <collating-symbol> can define a position in the relative order for use in weights.

The <collating-symbol> UNDEFINED is interpreted as including all characters not specified explicitly. Such characters are inserted in the character collation order at the point indicated by the symbol, and in ascending order according to their encoded character values. If no UNDEFINED symbol is specified, and the current coded character set contains characters not specified in this clause, the localedef utility issues a warning and places such characters at the end of the character collation order.

The syntax for a collation order statement is:

```
<collating-identifier> <weight1>;<weight2>;...;<weightn>
```

Collation of two collating identifiers is done by comparing their relative primary weights. This process is repeated for successive weight levels until the two identifiers are different, or the weight levels are exhausted. The operands for each collating identifier define the primary, secondary, and subsequent relative weights for the collating identifier. Two or more collating elements can be assigned the same weight. If two collating identifiers have the same primary weight, they belong to the same *equivalence class*.

The special symbol IGNORE as a weight indicates that when strings are compared using the weights at the level where IGNORE is specified, the collating element should be ignored, as if the string did not contain the collating element. In regular expressions and pattern matching, all characters that are IGNOREd in their primary weight form an equivalence class.

All characters specified by an ellipsis are assigned unique weights, equal to the relative order of the characters. Characters specified by an explicit or implicit UNDEFINED special symbol are assigned the same primary weight (they belong to the same equivalence class).

One-to-many mapping is indicated by specifying two or more concatenated characters or symbolic names. For example, if the character "<ezset>" is given the string "<s><s>" as a weight, comparisons are performed as if all occurrences of the character <ezset> are replaced by <s><s> (assuming <s> has

the collating weight <s>). If it is desirable to define <ezet> and <s><s> as an equivalence class, then a collating element must be defined for the string "ss".

If no weight is specified, the collating identifier is interpreted as itself. For example, the order statement

```
<a>    <a>
```

is equivalent to

```
<a>
```

Collating keywords

The following keywords are recognized in a collation sequence definition.

copy

Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword shall be present in this category. If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

collating-element

Defines a collating-element symbol representing a multicharacter collating element. This keyword is optional. In addition to the collating elements in the character set, the `collating-element` keyword can be used to define multicharacter collating elements. The syntax is:

```
"collating-element %s from \"%s\"", <collating-element>, <string>
```

The <collating-element> should be a symbolic name enclosed between angle brackets (< and >), and should not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. The string operand is a string of two or more characters that collate as an entity. A <collating-element> defined with this keyword is only recognized within the LC_COLLATE category. For example:

```
collating-element <ch> from "<c><h>"
collating-element <e-acute> from "<acute><e>"
collating-element <ll> from "ll"
```

collating-symbol

Defines a collating symbol for use in collation order statements.

The `collating-symbol` keyword defines a symbolic name that can be associated with a relative position in the character order sequence. While such a symbolic name does not represent any collating element, it can be used as a weight. This keyword is optional.

This construct can define symbols for use in collation sequence statements, between the `order_start` and `order_end` keywords. The syntax is:

```
"collating-symbol \"%s\"", <collating-symbol>
```

The <collating-symbol> must be a symbolic name, enclosed between angle brackets (< and >), and should not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. A <collating-symbol> defined with this keyword is only recognized within the LC_COLLATE category.

For example:

```
collating-symbol <UPPER_CASE>
collating-symbol <HIGH>
```

substitute

The `substitute` keyword defines a substring substitution in a string to be collated. This keyword is optional. The following operands are supported with the `substitute` keyword:

```
"substitute %s with \"%s\", <regular-expr>, <replacement>
```

The first operand is treated as a basic regular expression. The replacement operand consists of zero or more characters and regular expression back-references (for example, `\1` through `\9`). The back-references consist of the backslash followed by a digit from 1 to 9. If the backslash is followed by two or three digits, it is interpreted as an octal constant.

When strings are collated according to a collation definition containing `substitute` statements, the collation behaves as if occurrences of substrings matching the basic regular expression are replaced by the replacement string, before the strings are compared based on the specified collation sequence. Ranges in the regular expression are interpreted according to the current character collation sequence and character classes according to the character classification specified by the `LC_CTYPE` environment variable at collation time. If more than one `substitute` statement is present in the collation definition, the collation process behaves as if the `substitute` statements are applied to the strings in the order they occur in the source definition. The substitution for the `substitute` statements are processed before any substitutions for one-to-many mappings. The support of the "substitute" keyword is an IBM z/OS XL C/C++ extension to the POSIX standard.

Note: This is an IBM extension; therefore, locales that use it may not be portable to `localedef` tools developed by other vendors.

order_start

Define collating rules. This statement is followed by one or more collation order statements, assigning character collation values and collation weights to collating elements.

The `order_start` keyword must precede collation order entries. It defines the number of weights for this collation sequence definition and other collation rules. The syntax of the `order_start` keyword is:

```
order_start <sort-rule1>;<sort-rule1>;...<sort-rulen>
```

The operands of the `order_start` keyword are optional. If present, the operands define rules to be applied when strings are compared. The number of operands define how many weights each element is assigned; if no operands are present, one forward operand is assumed. If any is present, the first operand defines rules to be applied when comparing strings using the first (primary) weight; the second when comparing strings using the second weight, and so on. Operands are separated by semicolons (;). Each operand consists of one or more collation directives separated by commas (,). If the number of operands exceeds the limit of 6, the `localedef` utility issues a warning message.

The following directives are supported:

forward

specifies that comparison operations for the weight level proceed from the start of the string towards its end.

backward

specifies that comparison operations for the weight level proceed from the end of the string toward its beginning.

no-substitute

no substitution is performed, such that the comparison is based on collation values for collating elements before any substitution operations are performed.

Notes:

1. This is an IBM extension; therefore, locales that use it may not be portable to `localedef` tools developed by other vendors.
2. When the `no-substitute` keyword is specified, one-to-many mappings are ignored.

position

specifies that comparison operations for the weight level must consider the relative position of non-IGNOREd elements in the strings. The string containing a non-IGNOREd element after the fewest IGNOREd collating elements from the start of the comparison collates first. If both strings contain a non-IGNOREd character in the same relative position, the collating values assigned to the elements determine the order. If the strings are equal, subsequent non-IGNOREd characters are considered in the same manner.

order_end

The collating order entries are terminated with an `order_end` keyword.

Figure 197 on page 662 is an example of an `LC_COLLATE` category.

```
LC_COLLATE
% ARTIFICIAL COLLATE CATEGORY

% collating elements
1  collating-element <ch> from "<c><h>"
   collating-element <Ch> from "<C><h>"
   collating-element <eszet> from "<s><z>"

%collating symbols for relative order definition

2  collating-symbol <LOW>
   collating-symbol <UPPER-CASE>
   collating-symbol <LOWER-CASE>
   collating-symbol <NONE>

3  order_start forward;backward;forward
4  <LOW>
   <UPPER-CASE>
   <LOWER-CASE>

5  UNDEFINED IGNORE;IGNORE;IGNORE

6  <space>
   ...
   <quotation-mark>
7  <a> <a>;<NONE>;<LOWER-CASE>
10 <a-acute> <a>;<a-acute>;<LOWER-CASE>
11 <a-grave> <a>;<a-grave>;<LOWER-CASE>
8  <A> <a>;<NONE>;<UPPER-CASE>
11 <A-acute> <a>;<a-acute>;<UPPER-CASE>
11 <A-grave> <a>;<a-grave>;<UPPER-CASE>
11 <ch> <ch>;<NONE>;<LOWER-CASE>
11 <Ch> <ch>;<NONE>;<UPPER-CASE>
9  <s> <s>;<s>;<LOWER-CASE>
12 <eszet> "<s><s>";"<eszet><s>";<LOWER-CASE>
9  <z> <z>;<NONE>;<LOWER-CASE>

order_end
```

Figure 197. Example `LC_COLLATE` definition

The example is interpreted as follows:

1. collating elements

- character `<c>` followed by `<h>` collate as one entity named `<ch>`
- character `<C>` followed by `<h>` collate as one entity named `<Ch>`
- character `<s>` followed by `<z>` collate as one entity named `<eszet>`

2. collating symbols `<LOW>`, `<UPPER-CASE>`, `<LOWER-CASE>` and `<NONE>` are defined to be used in relative order definition

3. up to 3 string comparisons are defined:

- first pass starts from the beginning of the strings
- second pass starts from the end of the strings, and
- third pass starts from the beginning of the strings

4. the collating weights are defined such that

- <LOW> collates before <UPPER-CASE>,
 - <UPPER-CASE> collates before <LOWER-CASE>,
 - <LOWER-CASE> collates before <NONE>;
5. all characters for which collation is not specified here are ordered after <NONE>, and before <space> in ascending order according to their encoded values
 6. all characters with an encoded value larger than the encoded value of <space> and lower than the encoded value of <quotation-mark> in the current encoded character set, collate in ascending order according to their values;
 7. <a> has a:
 - primary weight of <a>,
 - secondary weight <NONE>,
 - tertiary weight of <LOWER-CASE>,
 8. <A> has a:
 - primary weight of <a>,
 - secondary weight of <NONE>,
 - tertiary weight of <UPPER-CASE>,
 9. the weights of <s> and <z> are determined in a similar fashion to <a> and <A>.
 10. <a-acute> has a:
 - primary weight of <a>,
 - secondary weight of <a-acute> itself,
 - tertiary weight of <LOWER-CASE>,
 11. the weights of <a-grave>, <A-acute>, <A-grave>, <ch> and <Ch> are determined in a similar fashion to <a-acute>.
 12. <eszet> has a:
 - primary weight determined by replacing each occurrence of <eszet> with the sequence of two <s>'s and using the weight of <s>,
 - secondary weight determined by replacing each occurrence of <eszet> with the sequence of <eszet> and <s> and using their weights,
 - tertiary weight is the relative position of <LOWER-CASE>.

Comparison of strings

Compare the strings `s1="aAch"` and `s2="AaCh"` using the above `LC_COLLATE` definition:

1. `s1=> "aA<ch>"`, and `s2=> "Aa<Ch>"`
2. first pass:
 - a. substitute the elements of the strings with their primary weights: `s1=> "<a><a><ch>"`, `s2=> "<a><a><ch>"`
 - b. compare the two strings starting with the first element — they are equal.
3. second pass:
 - a. substitute the elements of the strings with their secondary weights: `s1=> "<NONE><NONE><NONE>"`, `s2=> "<NONE><NONE><NONE>"`
 - b. compare the two strings from the last element to the first — they are equal.
4. third pass:
 - a. substitute the elements of the strings with their third level weights:

```
s1=> "<LOWER-CASE><UPPER-CASE><LOWER-CASE>",  
s2=> "<UPPER-CASE><LOWER-CASE><UPPER-CASE>",
```

- b. compare the two strings starting from the beginning of the strings: s2 compares lower than s1, because <UPPER-CASE> is before <LOWER-CASE>.

Compare the strings s1="áß" and s2="àss":

1. s1=> "á<eszset>" and s2= "àss";
2. first pass:
 - a. substitute the elements of the strings with their primary weights: s1=> "<a><s><s>", s2=> "<a><s><s>"
 - b. compare the two strings starting with the first element — they are equal.
3. second pass:
 - a. substitute the elements of the strings with their secondary weights: s1=> "<a-acute><eszset><s>", s2=> "<a-grave><s><s>"
 - b. compare the two strings from the last element to the first — <s> is before <eszset>.

LC_MONETARY category

This category defines the rules and symbols used to format monetary quantities. The operands are strings or integers. The following keywords are supported:

copy

Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

int_curr_symbol

Specifies the international currency symbol. The operand is a four-character string, with the first three characters containing the alphabetic international currency symbol in accordance with those specified in ISO4217 *Codes for the Representation of Currency and Funds*. The fourth character is the character used to separate the international currency symbol from the monetary quantity.

The following value may also be specified, though it is not. If not defined, it defaults to the empty string ("").

currency_symbol

Specifies the string used as the local currency symbol. If not defined, it defaults to the empty string ("").

mon_decimal_point

The string used as a decimal delimiter to format monetary quantities. If not defined it defaults to the empty string ("").

mon_thousands_sep

Specifies the string used as a separator for groups of digits to the left of the decimal delimiter in formatted monetary quantities. If not defined, it defaults to the empty string ("").

mon_grouping

Defines the size of each group of digits in formatted monetary quantities. The operand is a sequence of integers separated by semicolons. Also, for compatibility, it may be a string of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not -1, then the size of the previous group (if any) is used repeatedly for the rest of the digits. If the last integer is -1, then no further grouping is performed. If not defined, mon_grouping defaults to -1 which indicates that no grouping. An empty string is interpreted as -1.

positive_sign

A string used to indicate a formatted monetary quantity with a non-negative value. If not defined, it defaults to the empty string ("").

negative_sign

Specifies a string used to indicate a formatted monetary quantity with a negative value. If not defined, it defaults to the empty string ("").

int_frac_digits

Specifies an integer representing the number of fractional digits (those to the right of the decimal delimiter) to be displayed in a formatted monetary quantity using `int_curr_symbol`. If not defined, it defaults to -1.

frac_digits

Specifies an integer representing the number of fractional digits (those to the right of the decimal delimiter) to be displayed in a formatted monetary quantity using `currency_symbol`. If not defined, it defaults to -1.

p_cs_precedes

Specifies an integer set to 1 if the `currency_symbol` or `int_curr_symbol` precedes the value for a non-negative formatted monetary quantity, and set to 0 if the symbol succeeds the value. If not defined, it defaults to -1.

p_sep_by_space

Specifies an integer set to 0 if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a non-negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the string sign, if adjacent. If not defined, it defaults to -1.

n_cs_precedes

An integer set to 1 if the `currency_symbol` or `int_curr_symbol` precedes the value for a negative formatted monetary quantity, and set to 0 if the symbol succeeds the value. If not defined, it defaults to -1.

n_sep_by_space

An integer set to 0 if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the string sign, if adjacent. If not defined, it defaults to -1.

p_sign_posn

An integer set to a value indicating the positioning of the `positive_sign` for a non-negative formatted monetary quantity. The following integer values are recognized; if not defined, it defaults to -1.

0

Parentheses surround the quantity and the `currency_symbol` or `int_curr_symbol`.

1

The sign string precedes the quantity and the `currency_symbol` or `int_curr_symbol`.

2

The sign string succeeds the quantity and the `currency_symbol` or `int_curr_symbol`.

3

The sign string immediately precedes the `currency_symbol` or `int_curr_symbol`.

4

The sign string immediately succeeds the `currency_symbol` or `int_curr_symbol`.

part of the POSIX standard.

5

Use `debit-sign` or `credit-sign` for `p_sign_posn` or `n_sign_posn`.

n_sign_posn

An integer set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity. The recognized values are the same as for `p_sign_posn`. If not defined, it defaults to -1.

left_parenthesis

The symbol of the locale's equivalent of (to form a negative-valued formatted monetary quantity together with right_parenthesis. If not defined, it defaults to the empty string ("").

Note: This is an IBM-specific extension.

right_parenthesis

The symbol of the locale's equivalent of) to form a negative-valued formatted monetary quantity together with left_parenthesis. If not defined, it defaults to the empty string ("");

Note: This is an IBM-specific extension.

debit_sign

The symbol of locale's equivalent of DB to indicate a non-negative-valued formatted monetary quantity. If not defined, it defaults to the empty string ("");

Note: This is an IBM-specific extension.

credit_sign

The symbol of locale's equivalent of CR to indicate a negative-valued formatted monetary quantity. If not defined, it defaults to the empty string ("");

Note: This is an IBM-specific extension.

int_p_cs_precedes

Specifies an integer set to 1 if the int_curr_symbol precedes the value for a non-negative formatted monetary quantity, and set to 0 if the symbol succeeds the value. If not defined, it defaults to -1.

int_n_cs_precedes

An integer set to 1 if the int_curr_symbol precedes the value for a negative formatted monetary quantity, and set to 0 if the symbol succeeds the value. If not defined, it defaults to -1.

int_p_sep_by_space

Specifies an integer set to 0 if no space separates the int_curr_symbol from the value for a non-negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the string sign, if adjacent. If not defined, it defaults to -1.

int_n_sep_by_space

An integer set to 0 if no space separates the int_curr_symbol from the value for a negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the string sign, if adjacent. If not defined, it defaults to -1.

int_p_sign_posn

For a non-negative monetary quantity, the following integer values are recognized:

- 0**
Parentheses surround the quantity and the int_curr_symbol.
- 1**
The sign string precedes the quantity and the int_curr_symbol.
- 2**
The sign string succeeds the quantity and int_curr_symbol.
- 3**
The sign string immediately precedes the int_curr_symbol.
- 4**
The sign string immediately succeeds the currency_symbol or int_curr_symbol.

int_n_sign_posn

For a negative monetary quantity, the following integer values are recognized:

- 0**
Parentheses surround the quantity and the int_curr_symbol.

- 1 The sign string precedes the quantity and the `int_curr_symbol`.
- 2 The sign string succeeds the quantity and `int_curr_symbol`.
- 3 The sign string immediately precedes the `int_curr_symbol`.
- 4 The sign string immediately succeeds the `currency_symbol` or `int_curr_symbol`.

Figure 198 on page 667 is an example of the definition of the LC_MONETARY category.

[illegible]

Figure 198. Example LC_MONETARY definition

LC_NUMERIC category

This category defines the rules and symbols used to format non-monetary numeric information. The operands are strings. The following keywords are recognized:

copy

Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

decimal_point

Specifies a string used as the decimal delimiter in numeric, non-monetary formatted quantities. This keyword cannot be omitted and cannot be set to the empty string.

thousands_sep

Specifies a string containing the symbol that is used as a separator for groups of digits to the left of the decimal delimiter in numeric, non-monetary, formatted quantities.

grouping

Defines the size of each group of digits in formatted non-monetary quantities. The operand is a sequence of integers separated by semicolons. Also, for compatibility, it may be a string of integers

d_fmt

Defines the appropriate date representation, corresponding to the %x field descriptor. The operand consists of a string, and may contain any combination of characters and field descriptors.

t_fmt

Defines the appropriate time representation, corresponding to the %X field descriptor. The operand consists of a string, which may contain any combination of characters and field descriptors.

am_pm

Defines the appropriate representation of the ante meridian and post meridian strings, corresponding to the %p field descriptor. The operand consists of two strings, separated by a semicolon. The first string represents the ante meridian designation, the last string the post meridian designation.

t_fmt_ampm

Defines the appropriate time representation in the 12-hour clock format with am_pm, corresponding to the %r field descriptor. The operand consists of a string and can contain any combination of characters and field descriptors.

era

Defines how the years are counted and displayed for each era (or emperor's reign) in a locale. No era is needed if the %E field descriptor modifier is not used for the locale. See the description of the `strftime()` function in [z/OS C/C++ Runtime Library Reference](#) for information about this field descriptor.

For each era, there must be one string in the following format:

```
direction:offset:start_date:end_date:name:format
```

direction

Either a + or - character. The + character indicates the time axis should be such that the years count in the positive direction when moving from the starting date towards the ending date. The - character indicates the time axis should be such that the years count in the negative direction when moving from the starting date towards the ending date.

offset

A number of the first year of the era.

start_date

A date in the form yyyy/mm/dd where yyyy, mm and dd are the year, month and day numbers, respectively, of the start of the era. Years prior to the year AD 0 are represented as negative numbers. For example, an era beginning March 5th in the year 100 BC would be represented as -100/3/5.

end_date

The ending date of the era in the same form as the start_date above or one of the two special values -* or +*. A value of -* indicates the ending date of the era extends to the beginning of time while +* indicates it extends to the end of time. The ending date may be either before or after the starting date of an era. For example, the strings for the Christian eras AD and BC would be:

```
+0:0000/01/01:+:AD:%EC %Ey
+:1:-0001/12/31:-*:BC:%EC %Ey
```

name

A string representing the name of the era which is substituted for the %EC field descriptor.

format

A string for formatting the %EY field descriptor. This string is usually a function of the %EC and %Ey field descriptors.

The operand consists of one string for each era. If there is more than one era, strings are separated by semicolons.

era_year

Defines the format of the year in alternate era format, corresponding to the %EY field descriptor.

era_d_fmt

Defines the format of the date in alternate era notation, corresponding to the %Ex field descriptor.

era_t_fmt

Defines the locale's appropriate alternative time format, corresponding to the %Ex field descriptor.

era_d_t_fmt

Defines the locale's appropriate alternative date and time format, corresponding to the %Ec field descriptor.

alt_digits

Defines alternate symbols for digits, corresponding to the %0 field descriptor modifier. The operand consists of semicolon-separated strings. The first string is the alternate symbol corresponding to zero, the second string the symbol corresponding to one, and so forth. A maximum of 100 alternate strings may be specified. The %0 modifier indicates that the string corresponding to the value specified by the field descriptor is used instead of the value.

For the definitions of the time formatting descriptors, see the description of the `strftime()` function in [z/OS C/C++ Runtime Library Reference](#).

LC_MESSAGES category

The LC_MESSAGES category defines the format and values for positive and negative responses. The following keywords are recognized:

copy

Specifies the name of an existing locale to be used as the source for the definition of this category. If you specify this keyword, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

yesexpr

The operand consists of an extended regular expression that describes the acceptable affirmative response to a question that expects an affirmative or negative response.

noexpr

The operand consists of an extended regular expression that describes the acceptable negative response to a question that expects an affirmative or negative response.

yesstr

The operand consists of a fixed string (not a regular expression) that can be used by an application for composition of a message that lists an acceptable affirmative response, such as in a prompt.

nostr

The operand consists of a fixed string that can be used by an application for composition of a message that lists an acceptable negative response.

[Figure 200 on page 671](#) shows an example of how to define the LC_MESSAGES category.

Figure 200. Example LC_MESSAGES definition

start_week

An integer specifying the week of the month when DST comes into effect. Acceptable values range from -4 to +4. A value of 4 means the fourth week of the month, while a value of -4 means fourth week of the month, counting from the end of the month. Sunday is considered to be the start of the week. If DST is not applicable to a locale, `start_week` is set to 0, which is also the default.

end_week

An integer specifying the week of the month when DST ceases to be in effect. The specifications are similar to those for `start_week`.

Note: The `start_week` and `end_week` need not be used. The `start_day` and `end_day` fields can specify either the day of the week or the day of the month. If day of month is specified, `start_week` and `end_week` become redundant.

start_day

An integer specifying the day of the week or the day of the month when DST comes into effect. The value depends on the value of `start_week`. If `start_week` is not equal to 0, this is the day of the week when DST comes into effect. It ranges from 0 through 6 inclusive, with 0 corresponding to Sunday and 6 corresponding to Saturday. If `start_week` equals 0, `start_day` is the day of the month (for the current year) when DST comes into effect. It ranges from 1 through to the last day of the month inclusive. The last day of the month is 31 for January, March, May, July, August, October, and December. It is 30 for April, June, September, and November. For February, it is 28 on non-leap years and 29 on leap years. If DST is not applicable to a locale, `start_day` is set to 0, which is also the default.

end_day

An integer specifying the day of the week or the day of the month when DST ceases to be in effect. The specifications are similar to those for `start_day`.

start_time

An integer specifying the number of seconds after 12:00 midnight, local standard time, when DST comes into effect. For example, if DST is to start at 2:00 am, `start_time` is assigned the value 7200; for 12:00 am (midnight), `start_time` is 0; for 1:00 am, it is 3600.

end_time

An integer specifying the number of seconds after 12 midnight, local standard time, when DST ceases to be in effect. The specifications are similar to those for `start_time`.

shift

An integer specifying the DST time shift, expressed in seconds. The default is 3600, for 1 hour.

uctname

A string specifying the name to be used for Coordinated Universal Time. If this keyword is not specified, the `uctname` will default to "UTC".

Figure 201 on page 673 is an example of how to define the `LC_TOD` category.


```

escape_char  /
comment-char %

%  

LC_TOD
%  

% the time zone difference is 8hrs; the name of the daylight saving  

% time is PDT, and it starts on the first Sunday of April at 2&00AM  

% and ends on the second Sunday of October at 2&00AM  

timezone_difference +480  

timezone_name      "<P><S><T>"  

daylight_name      "<P><D><T>"  

start_month        4  

end_month          10  

start_week         1  

end_week           2  

start_day          1  

end_day            30  

start_time         7200  

end_time           3600  

shift              3600  

END LC_TOD

```

Figure 201. Example LC_TOD definition

LC_SYNTAX category

The LC_SYNTAX category defines the variant characters from the portable character set. LC_SYNTAX is an IBM-specific extension. This category can be queried by the C library function `getsyntax()` to determine the encoding of a variant character if needed.



Attention: Customizing the LC_SYNTAX category is not recommended. You should use the LC_SYNTAX values obtained from the `charmap` file when you use the `localedef` utility.

The operands for the characters in the LC_SYNTAX category accept the single byte character specification in the form of a symbolic name, the character itself, or the decimal, octal, or hexadecimal constant. The characters must be specified in the LC_CTYPE category as a *punct* character. The values for the LC_SYNTAX characters must be unique. If symbolic names are used to define the encoding, only the symbolic names listed for each character should be used.

The code points for the LC_SYNTAX characters are set to the code points specified. Otherwise, they default to the code points for the respective characters from the `charmap` file, if the file is present, or to the code points of the respective characters in the IBM-1047 code page.

Note: LC_TOD and LC_SYNTAX are not supported for ASCII locales (a locale specification can not contain a definition for these categories). However, for consistency with EBCDIC locales, `localedef` generates default values for these categories in ASCII locale objects (the values generated for the C locale but with ASCII code points).

The following keywords are recognized:

copy

Specifies the name of an existing locale to be used as the source for the definition of this category. If you specify this keyword, no other keyword should be present. If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

backslash

Specifies a string that defines the value used to represent the backslash character. If this keyword is not specified, the value from the `charmap` file for the character `<backslash>`, `<reverse-solidus>`, or `<SM07>` is used, if it is present.

right_brace

Specifies a string that defines the value used to represent the right brace character. If this keyword is not specified, the value from the `charmap` file for the character `<right-brace>`, `<right-curly-bracket>`, or `<SM14>` is used, if it is present.

left_brace

Specifies a string that defines the value used to represent the left brace character. If this keyword is not specified, the value from the charmap file for the character <left-brace>, <left-curly-bracket>, or <SM11> is used, if it is present.

right_bracket

Specifies a string that defines the value used to represent the right bracket character. If this keyword is not specified, the value from the charmap file for the character <right-square-bracket>, or <SM08> is used, if it is present.

left_bracket

Specifies a string that defines the value used to represent the left bracket character. If this keyword is not specified, the value from the charmap file for the character <left-square-bracket>, or <SM06> is used, if it is present.

circumflex

Specifies a string that defines the value used to represent the circumflex character. If this keyword is not specified, the value from the charmap file for the character <circumflex>, <circumflex-accent>, or <SD15> is used, if it is present.

tilde

Specifies a string that defines the value used to represent the tilde character. If this keyword is not specified, the value from the charmap file for the character <tilde>, or <SD19> is used, if it is present.

exclamation_mark

Specifies a string that defines the value used to represent the exclamation mark character. If this keyword is not specified, the value from the charmap file for the character <exclamation-mark>, or <SP02> is used, if it is present.

number_sign

Specifies a string that defines the value used to represent the number sign character. If this keyword is not specified, the value from the charmap file for the character <number-sign>, or <SM01> is used, if it is present.

vertical_line

Specifies a string that defines the value used to represent the vertical line character. If this keyword is not specified, the value from the charmap file for the character <vertical-line>, or <SM13> is used, if it is present.

dollar_sign

Specifies a string that defines the value used to represent the dollar sign character. If this keyword is not specified, the value from the charmap file for the character <dollar-sign>, or <SC03> is used, if it is present.

commercial_at

Specifies a string that defines the value used to represent the commercial at character. If this keyword is not specified, the value from the charmap file for the character <commercial-at>, or <SM05> is used, if it is present.

grave_accent

Specifies a string that defines the value used to represent the grave accent character. If this keyword is not specified, the value from the charmap file for the character <grave-accent>, or <SD13> is used, if it is present.

[Figure 202 on page 675](#) is an example of how the LC_SYNTAX category is defined.

Figure 202. Example definition of LC_SYNTAX

Method files can be used when creating ASCII locales. They specify the method functions used by the C runtime's locale-sensitive interfaces when the ASCII locale is activated.

By replacing the CHARMAP related method functions in a method file, users can create a locale which supports a user-defined code page. For each replaced method, the method file supplies the user-written method function name, and optionally indicates where the method function code is to be found (.o file, archive library or DLL). The method source file maps method names to the National Language Support (NLS) subroutines that implement those methods. The method file also specifies the object libraries or DLL side decks where the implementing subroutines are stored. The methods correspond to those subroutines that require direct access to the data structures representing locale data.

Users can provide these CHARMAP methods via a DLL side deck, an archive library or an object file. The user-written method functions are used both by the locale-sensitive APIs they represent, and also by the `localedef` utility itself while generating the method-file based ASCII locale object. This second use by `localedef` itself causes a temporary DLL to be created while processing the CHARMAP file supplied on the `-f` parameter. The name of the file containing method objects or side deck information is passed by the `localedef` utility as a parameter on the `c89` command line, so the standard archive/object/side deck suffix naming conventions apply (for example, `.a`, `.o`, `.x`).

Chapter 54. Building a locale **675**

```

method_def :
    "METHODS"
    method_assign_list "END METHODS"
    ;
method_assign_list :
    method_assign_list method_assign
method_assign_list
method_assign
;

method_assign :
    "csid" meth_name meth_lib_path
    "fnmatch" meth_name meth_lib_path
    "is_wctype" meth_name meth_lib_path
    "mblen" meth_name meth_lib_path
    "mbstowcs" meth_name meth_lib_path
    "mbtowc" meth_name meth_lib_path
    "regcomp" meth_name meth_lib_path
    "regererror" meth_name meth_lib_path
    "regexexec" meth_name meth_lib_path
    "regfree" meth_name meth_lib_path
    "rpmatch" meth_name meth_lib_path
    "strcoll" meth_name meth_lib_path
    "strfmon" meth_name meth_lib_path
    "strftime" meth_name meth_lib_path
    "strptime" meth_name meth_lib_path
    "strxfrm" meth_name meth_lib_path
    "towlower" meth_name meth_lib_path
    "towupper" meth_name meth_lib_path
    "wcscoll" meth_name meth_lib_path
    "wcsftime" meth_name meth_lib_path
    "wcsid" meth_name meth_lib_path
    "wcstombs" meth_name meth_lib_path
    "wcswidth" meth_name meth_lib_path
    "wcsxfrm" meth_name meth_lib_path
    "wctomb" meth_name meth_lib_path
    "wcwidth" meth_name meth_lib_path
    ;

meth_name:
    global_name
    cfunc_name
    ;

```

Expected grammar for method files (Part 1 of 2)

Figure 203. Expected grammar for method files

```

global_name:
  CSID_STD
  FNMATCH_C
  FNMATCH_STD
  GET_WCTYPE_STD
  IS_WCTYPE_SB
  IS_WCTYPE_STD
  LOCALECONV_STD
  MBLEN_932
  MBLEN_EUCJP
  MBLEN_SB
  MBSTOWCS_932
  MBSTOWCS_EUCJP
  MBSTOWCS_SB
  MBTOWC_932
  MBTOWC_EUCJP
  MBTOWC_SB
  REGCOMP_STD
  REGERROR_STD
  REGEXEC_STD
  REGFREE_STD
  RPMATCH_C
  RPMATCH_STD
  STRCOLL_C
  STRCOLL_SB
  STRCOLL_STD
  STRFMON_STD
  STRFTIME_STD
  STRPTIME_STD
  STRXFRM_C
  STRXFRM_SB
  STRXFRM_STD
  TOWLOWER_STD
  TOWUPPER_STD
  WCSCOLL_C
  WCSCOLL_STD
  WCSFTIME_STD
  WCSID_STD
  WCSTOMBS_932
  WCSTOMBS_EUCJP
  WCSTOMBS_SB
  WCSWIDTH_932
  WCSWIDTH_EUCJP
  WCSWIDTH_LATIN
  WCSXFRM_C
  WCSXFRM_STD
  WCTOMB_932
  WCTOMB_EUCJP
  WCTOMB_SB
  WCWIDTH_932
  WCWIDTH_EUCJP
  WCWIDTH_LATIN
;

```

Expected grammar for method files (Part 2 of 2)

Where `cfunc_name` is the name of a user supplied subroutine, and `meth_lib_path` is an optional path name for the file containing the compiled subroutine or a side-deck for the DLL containing the subroutine.

The `localedef` command parses this information to determine the methods to be used for this locale. The following subroutines must be specified in the method file:

<code>mblen</code>	<code>mbstowcs</code>
<code>mbtowc</code>	<code>wcstombs</code>
<code>wcswidth</code>	<code>wctomb</code>
<code>wcwidth</code>	

The following additional subroutines are mandatory in AIX method files, but are not supported on z/OS and if specified are ignored:

```
mbtopc
mbstopcs
pctomb
pcstombs
```

Any other method not specified in the method file retains the default. Mixing of user-written method function names (represented as `cfunc_name` in the grammar) and IBM-provided method function names (represented by `global_name` in the grammar) is not allowed. A method file should not include both. If the `localedef` command encounters both `cfunc_name` values and `global_name` values in a method file, an error is generated and the locale is not created.

It is not mandatory that the `METHODS` section specify the `meth_lib_path` name for all methods. The following is an example of how to specify the `meth_lib_path` and what the `localedef` passes on the `c89` command invoking the binder when linking the method-based ASCII locale object:

```
METHODS
mblen "__mblen_myuni"
mbstowcs "__mbstowcs_myuni" "/u/my/libmyuni.a"
mbtowc "__mbtowc_myuni"
wcstombs "__wcstombs_myuni" "/u/gen/libgenuni.a"
wcswidth "__wcswidth_myuni"
wctomb "__wctomb_myuni"
wcwidth "__wcwidth_myuni" "./wcwidth.o"
```

In the example, `libmyuni.a` contains functions `__mbstowcs_myuni` and `__mbtowc_myuni`. Similarly, `libgenuni.a` contains functions `__wcstombs_myuni`, `__wcswidth_myuni` and `__wctomb_myuni`. The function `__wcwidth_myuni` is contained in the file `wcwidth.o`. If the function `__mblen_myuni` is not defined in either of the three files indicated, a locale object will not be created. For this example the `localedef` utility would invoke the binder using the following `c89` command line:

```
c89 -o myuni.locale -Wl,xplink ./localefBGgFfCgAo
./localeEgaBGaahA.o /u/my/libmyuni.a
/u/gen/libgenuni.a ./wcwidth.o
```

It is also possible to use the `-L` `localedef` option to specify the `c89` `-L` library flags and only reference the library names in the method file following the `liblibname.a` convention.

If an individual method does not specify a `meth_lib_path` name, the method inherits the most recently specified `meth_lib_path` name. If no `meth_lib_path` name is specified in the `METHODS` section, the default runtime library side-deck is assumed. The files indicated by `meth_lib_path` names of all methods in the method file are used when linking the locale object. A concatenated list of all `meth_lib_path` names is specified on the link step. If multiple object libraries or side decks are specified, the same routine should not be defined in more than one of them. Unexpected results may occur if the method functions appear in more than one file, particularly if the duplicate copies are not identical. The binder could resolve a method function from a file different from the one given in the method file itself.

The method for the `mbtowc` and `wcwidth` subroutines should avoid calling other methods where possible.

Using the `localedef` utility

The locale objects or locales are generated using the `localedef` utility. The `localedef` utility:

1. Reads the *locale definition file*
2. Resolves all the character symbolic names to the values of characters defined in the specified *character set definition file*, (`CHARMAP`)
3. Produces a z/OS XL C source file.

4. Compiles the source file using the z/OS XL C compiler and link-edits the produced text module to produce a locale object. `localedef` produces ASCII locale objects as XPLINK DLL's exclusively, while EBCDIC locales can be non-XPLINK objects or XPLINK DLL's.

Note: AMODE 64 locales are always XPLINK locales, while 31-bit locales may be XPLINK or non-XPLINK.

The locale DLL object can be loaded by the `setlocale()` function and then accessed by the z/OS XL C/C++ functions that are sensitive to the cultural information, or that can query the locales. For a list of all the library functions sensitive to locale, see [“Locale-sensitive interfaces”](#) on page 642. For detailed information on how to invoke `localedef`, see `localedef` utility in the *z/OS XL C/C++ User's Guide*.

The locale DLL object created by `localedef` must adhere to certain naming conventions so that the locale can be used by the system. These conventions are outlined in [“Locale naming conventions”](#) on page 679.

XPLINK applications require XPLINK locale objects, and non-XPLINK applications require non-XPLINK locale objects. Likewise, AMODE 64 applications require AMODE 64 locale objects. `localedef` creates non-XPLINK locales by default. The option `XPLINK` causes the TSO `localedef` command (`LOCALDEF`) to produce an XPLINK locale object. The batch `XPLINK localedef` command (`EDCXLDEF proc`) produces an XPLINK locale object (while the batch `localedef` command (`EDCLDEF`) produces a non-XPLINK locale object). The `-X` parameter causes the UNIX System Services `localedef` command to generate an XPLINK locale object.

The TSO `localedef` (`LOCALDEF`) command and the batch `XPLINK localedef` command (`EDCXLDEF proc`) cannot be used to generate ASCII locales or AMODE 64 locales. Only the UNIX System Services `localedef` command may be used. ASCII locales are generated by specifying the `-A` `localedef` option on the command line of the UNIX System Services `localedef` command. AMODE 64 locales are generated by specifying the `-6` option on the command line of the UNIX System Services `localedef` command. Specify both `-A` and `-6` to produce locale objects which are both ASCII and AMODE 64. AMODE 64 locales are always XPLINK locales. The `-X` option is implicitly specified whenever the `-6` option is specified. Users can supply functions for the methods referenced in the locale charmap category by indicating the `-m method_file` option on the command line.

The POSIX shell (`/bin/sh`) UNIX System Services shell, `/bin/sh`, is an example of a non-XPLINK application that uses locales. It needs non-XPLINK locales. If the shell invokes an XPLINK application that uses locales, the application will need an XPLINK version of the same locale. Usually, both XPLINK and non-XPLINK versions of a locale are needed whenever an XPLINK application is invoked from the shell, or when an XPLINK application invokes the shell or any other non-XPLINK application. Likewise, usually both AMODE 64 and non-XPLINK versions of a locale are needed whenever a AMODE 64 application is invoked from the shell, or when a AMODE 64 application invokes the shell or any other non-XPLINK application. The locale object naming conventions ensure that the runtime library loads the appropriate version of the locale.

Locale naming conventions

The `setlocale()` library function that selects the active locale maps the descriptive locale name into the name of the locale object before loading the locale and making it accessible.

In z/OS XL C/C++ programs, the locale modules are referred to by descriptive locale names. The locale names themselves are not case sensitive. They follow these conventions:

```
<Language>-<Territory>.<Codeset>
```

Language

is a two-letter uppercase abbreviation for the language name. The abbreviations come from the ISO 639 standard.

Territory

is a two-letter uppercase abbreviation for the territory name. The abbreviation comes from the ISO 3166 standard.

Codeset

is the name registered by the MIT X Consortium that identifies the registration authority that owns the specific encoding. A modifier may be added to the registered name but is not required. The modifier is of the form @codeset modifier and identifies the coded character set as defined by that registration authority.

The Codeset parts are optional. If they are not specified, Codeset defaults to IBM-nnn, where nnn is the default code page, which for EBCDIC locales is shown in [Table 108 on page 681](#) and for ASCII locales in [Table 109 on page 684](#). (The modifier portion defaults to nothing.)

For PDS resident locales, the mapping between the descriptive locale name and the eight-character name of the locale object is performed as follows:

1. The Language-Territory part is mapped into a two-letter LT code.
2. The Codeset part is mapped into a two-letter CC code.
3. The object name is built from a prefix, the two-letter LT code, and the two-letter CC code. The prefix is one of the following show in [Table 107 on page 680](#). (Note that the @-sign in the PDS and z/OS UNIX locale names always has Latin-1/Open Systems encoding; see IBM-1047 CHARMAP.)

Table 107. Locale object prefix			
Application	No modifier	@euro modifier	@preeuro modifier
non-XPLINK	EDC\$	EDC@	EDC3
XPLINK	CEH\$	CEH@	CEH3
XPLINK ASCII	CEJ\$	NA	NA
AMODE 64	CEQ\$	CEQ@	CEQ3
AMODE 64 ASCII	CEZ\$	NA	NA

Type	Mapping
Non-XPLINK	Fr_BE.IBM-1148 maps to EDC\$FBH0 Fr_BE.IBM-1148@euro maps to EDC@FBH0 Fr_BE.IBM-1148@preeuro maps to EDC3FBH0
XPLINK	Fr_BE.IBM-1148 maps to CEH\$FBH0 Fr_BE.IBM-1148@euro maps to CEH@FBH0 Fr_BE.IBM-1148@preeuro maps to CEH3FBH0
ASCII	Fr_BE.IS08859-1 maps to CEJ\$FBI1 Fr_BE.UTF-8 maps to CEJ\$FBU8
AMODE 64	Fr_BE.IBM-1148 maps to CEQ\$FBH0 Fr_BE.IBM-1148@euro maps to CEQ@FBH0 Fr_BE.IBM-1148@preeuro maps to CEQ3FBH0
AMODE 64 ASCII	Fr_BE.IS08859-1 maps to CEZ\$FBI1 Fr_BE.UTF-8 maps to CEZ\$FBU8

For resident locales in the z/OS UNIX file system, the mapping between the descriptive locale name and the z/OS UNIX file name is performed as follows:

1. The locale object file name starts out the same as the descriptive name.
2. If the locale object is XPLINK, add a suffix of ".xplink" to the end of the object file name.
3. If the locale object is AMODE 64, add a suffix of ".lp64" to the end of the object file name.

Type	Mapping
Non-XPLINK	Fr_BE.IBM-1148 maps to Fr_BE.IBM-1148 Fr_BE.IBM-1148@euro maps to Fr_BE.IBM-1148@euro Fr_BE.IBM-1148@preeuro maps to Fr_BE.IBM-1148@preeuro
XPLINK	Fr_BE.IBM-1148 maps to Fr_BE.IBM-1148.xplink Fr_BE.IBM-1148@euro maps to Fr_BE.IBM-1148@euro.xplink Fr_BE.IBM-1148@preeuro maps to Fr_BE.IBM-1148@preeuro.xplink
ASCII	Fr_BE.IS08859-1 maps to Fr_BE.IS08859-1.xplink Fr_BE.UTF-8 maps to Fr_BE.UTF-8.xplink
AMODE 64	Fr_BE.IBM-1148 maps to FR_BE.IBM-1148.lp64 Fr_BE.IBM-1148@euro maps to Fr_BE.IBM-1148@euro.lp64 Fr_BE.IBM-1148@preeuro maps to Fr_BE.IBM-1148@preeuro.lp64
AMODE 64 ASCII	Fr_BE.IS08859-1 maps to Fr_BE.IS08859-1.lp64 Fr_BE.UTF-8 maps to Fr_BE.UTF-8.lp64

The mapping between Language-Territory and the two-letter LT code is defined in the LT conversion table EDC\$LCNM, built with assembler macros as follows:

```
EDC$LCNM TITLE 'LOCALE NAME CONVERSION TABLE'
EDC$LCNM CSECT
    EDCLOCNM TYPE=ENTRY, LOCALE='DA_DK', CODESET='IBM-1047', CODE='DA'
    EDCLOCNM TYPE=ENTRY, LOCALE='DE_BE', CODESET='IBM-1047', CODE='DB'
    EDCLOCNM TYPE=ENTRY, LOCALE='DE_CH', CODESET='IBM-1047', CODE='DC'
    EDCLOCNM TYPE=ENTRY, LOCALE='DE_DE', CODESET='IBM-1047', CODE='DD'
    EDCLOCNM TYPE=ENTRY, LOCALE='JA_JP', CODESET='IBM-939', CODE='EJ'
:
    EDCLOCNM TYPE=END
END    EDC$LCNM
```

LOCALE specifies the name of Language-Territory, while CODE specifies the respective LT code.

You can customize this table by adding new LOCALE name mappings. z/OS XL C/C++ reserves alphabetic LT codes, but you can use codes containing numeric values for your own customized names.

The Language-Territory names and their mappings into LT codes that are provided are shown in Table 108 on page 681.

Table 108. Supported language-territory names and LT codes for EBCDIC locales				
Locale Name	Language	Country/Territory	EBCDIC Codeset	2-Byte LT Code
Ar_AA	Arabic	Algeria, Bahrain, Egypt, Iraq, Jordan, Kuwait, Lebanon, Libya, Morocco, Oman, Qatar, Saudi Arabia, Syria, Tunisia, U.A.E., Yemen	IBM-425	AR
Be_BY	Byelorussian	Belarus	IBM-1025	BB
Bg_BG	Bulgarian	Bulgaria	IBM-1025	BG
C			IBM-1047	CC
Ca_ES	Catalan	Spain	IBM-924	CS
Cs_CZ	Czech	Czech Republic	IBM-870	CZ
Da_DK	Danish	Denmark	IBM-1047	DA

Table 108. Supported language-territory names and LT codes for EBCDIC locales (continued)

Locale Name	Language	Country/Territory	EBCDIC Codeset	2-Byte LT Code
De_AT	German	Austria	IBM-924	DT
De_CH	German	Switzerland	IBM-1047	DC
De_DE	German	Germany	IBM-1047	DD
De_LU	German	Luxembourg	IBM-924	DL
El_GR	Greek	Greece	IBM-875	EL
En_AU	English	Australia	IBM-1047	NA
En_BE	English	Belgium	IBM-924	EB
En_CA	English	Canada	IBM-1047	EC
En_GB	English	United Kingdom	IBM-1047	EK
En_HK	English	China (Hong Kong S.A.R. of China)	IBM-1047	NH
En_IE	English	Ireland	IBM-924	EI
En_IN	English	India	IBM-1047	NI
En_JP	English	Japan	IBM-1027	EJ
En_NZ	English	New Zealand	IBM-1047	NZ
En_PH	English	Philippines	IBM-1047	NP
En_SG	English	Singapore	IBM-1047	NS
En_US	English	United States	IBM-1047	EU
En_ZA	English	South Africa	IBM-1047	EZ
Es_AR	Spanish	Argentina	IBM-1047	EA
Es_BO	Spanish	Bolivia	IBM-1047	EO
Es_CL	Spanish	Chile	IBM-1047	EH
Es_CO	Spanish	Colombia	IBM-1047	FG
Es_CR	Spanish	Costa Rica	IBM-1047	ER
Es_DO	Spanish	Dominican Republic	IBM-1047	ED
Es_EC	Spanish	Ecuador	IBM-1047	EQ
Es_ES	Spanish	Spain	IBM-1047	ES
Es_GT	Spanish	Guatemala	IBM-1047	EG
Es_HN	Spanish	Honduras	IBM-1047	FE
Es_MX	Spanish	Mexico	IBM-1047	EM
Es_NI	Spanish	Nicaragua	IBM-1047	FA
Es_PA	Spanish	Panama	IBM-1047	EP
Es_PE	Spanish	Peru	IBM-1047	EW
Es_PR	Spanish	Puerto Rico	IBM-1047	EX
Es_PY	Spanish	Paraguay	IBM-1047	EY
Es_SV	Spanish	El Salvador	IBM-1047	EV
Es_US	Spanish	United States	IBM-1047	ET

Table 108. Supported language-territory names and LT codes for EBCDIC locales (continued)

Locale Name	Language	Country/Territory	EBCDIC Codeset	2-Byte LT Code
Es_UY	Spanish	Uruguay	IBM-1047	FD
Es_VE	Spanish	Venezuela	IBM-1047	EF
Et_EE	Estonian	Estonia	IBM-1122	EE
Fi_FI	Finnish	Finland	IBM-1047	FI
Fr_BE	French	Belgium	IBM-1047	FB
Fr_CA	French	Canada	IBM-1047	FC
Fr_CH	French	Switzerland	IBM-1047	FS
Fr_FR	French	France	IBM-1047	FF
Fr_LU	French	Luxembourg	IBM-924	FL
He_IL	Hebrew	Israel	IBM-424	IL
Hr_HR	Croatian	Croatia	IBM-870	HR
Hu_HU	Hungarian	Hungary	IBM-870	HU
Id_ID	Indonesian	Indonesia	IBM-1047	II
It_CH	Italian	Switzerland	IBM-1047	IC
Is_IS	Icelandic	Iceland	IBM-871	IS
It_IT	Italian	Italy	IBM-1047	IT
Ja_JP	Japanese	Japan	IBM-939	JA
Ko_KR	Korean	Korea	IBM-933	KR
Iw_IL	Hebrew	Israel	IBM-424	IL
Lt-LT	Lithuanian	Lithuania	IBM-1112	LT
Lv_LV	Latvian	Latvia	IBM-1112	LL
Mk_MK	Macedonian	Macedonia	IBM-1025	MM
Ms_MY	Malay	Malaysia	IBM-1047	MY
NL_BE	Dutch	Belgium	IBM-1047	NB
NL_NL	Dutch	The Netherlands	IBM-1047	NN
No_NO	Norwegian	Norway	IBM-1047	NO
PL_PL	Polish	Poland	IBM-870	PL
Pt_BR	Portuguese	Brazil	IBM-1047	BR
Pt_PT	Portuguese	Portugal	IBM-1047	PT
Ro_RO	Romanian	Romania	IBM-870	RO
Ru_RU	Russian	Russia	IBM-1025	RU
Sh_SP	Serbian (Latin)	Serbia	IBM-870	SL
Sk_SK	Slovak	Slovakia	IBM-870	SK
Sl_SI	Slovene	Slovenia	IBM-870	SI
Sq_AL	Albanian	Albania	IBM-500	SA
Sr_SP	Serbian (Cyrillic)	Serbia	IBM-1025	SC

Table 108. Supported language-territory names and LT codes for EBCDIC locales (continued)

Locale Name	Language	Country/Territory	EBCDIC Codeset	2-Byte LT Code
Sv_SE	Swedish	Sweden	IBM-1047	SV
Th_TH	Thai	Thailand	IBM-838	TH
Tr_TR	Turkish	Turkey	IBM-1026	TR
UK_UA	Ukrainian	Ukraine	IBM-1125	UU
Zh_CN	Simplified Chinese	China (PRC)	IBM-935	ZC
Zh_TW	Traditional Chinese	Taiwan	IBM-937	ZT

Table 109 on page 684 shows the supported language-territory names and LT codes for ASCII locales. Note that ASCII locale names can also be coded <uppercase><lowercase>_<uppercase><uppercase>. For example, both en_US and En_US are valid ASCII locale names.

Table 109. Supported language-territory names and LT codes for ASCII locales

Locale Name	Language	Country/Territory	ASCII Codeset	2-Byte LT Code
be_BY	Byelorussian	Belarus	ISO8859-5	BB
bn_IN	Bengali	India	UTF-8	BN
en_CA	English	Canada	ISO8859-1	EC
cs_CZ	Czech	Czech Republic	ISO8859-2	CZ
en_ZA	English	South Africa	ISO8859-1	EZ
da_DK	Danish	Denmark	ISO8859-1	DA
de_CH	German	Switzerland	ISO8859-1	DC
de_DE	German	Germany	ISO8859-1	DD
el_GR	Greek	Greece	ISO8859-7	EL
en_AU	English	Australia	ISO8859-1	NA
en_GB	English	United Kingdom	ISO8859-1	EK
en_HK	English	China (Hong Kong S.A.R. of China)	ISO8859-1	NH
en_IN	English	India	ISO8859-1	NI
en_NZ	English	New Zealand	ISO8859-1	NZ
en_PH	English	Philippines	ISO8859-1	NP
en_SG	English	Singapore	ISO8859-1	NS
en_US	English	United States	ISO8859-1	EU
es_AR	Spanish	Argentina	ISO8859-1	EA
es_BO	Spanish	Bolivia	ISO8859-1	EO
es_CL	Spanish	Chile	ISO8859-1	EH
es_CO	Spanish	Colombia	ISO8859-1	FG
es_CR	Spanish	Costa Rica	ISO8859-1	ER
es_DO	Spanish	Dominican Republic	ISO8859-1	ED
es_EC	Spanish	Ecuador	ISO8859-1	EQ

Table 109. Supported language-territory names and LT codes for ASCII locales (continued)

Locale Name	Language	Country/Territory	ASCII Codeset	2-Byte LT Code
es_ES	Spanish	Spain	ISO8859-1	ES
es_GT	Spanish	Guatemala	ISO8859-1	EG
es_HN	Spanish	Honduras	ISO8859-1	FE
es_MX	Spanish	Mexico	ISO8859-1	EM
es_NI	Spanish	Nicaragua	ISO8859-1	FA
es_PA	Spanish	Panama	ISO8859-1	EP
es_PE	Spanish	Peru	ISO8859-1	EW
es_PR	Spanish	Puerto Rico	ISO8859-1	EX
es_PY	Spanish	Paraguay	ISO8859-1	EY
es_SV	Spanish	El Salvador	ISO8859-1	EV
es_US	Spanish	United States	ISO8859-1	ET
es_UY	Spanish	Uruguay	ISO8859-1	FD
es_VE	Spanish	Venezuela	ISO8859-1	EF
fi_FI	Finnish	Finland	ISO8859-1	FI
fr_BE	French	Belgium	ISO8859-1	FB
fr_CA	French	Canada	ISO8859-1	FC
fr_CH	French	Switzerland	ISO8859-1	FS
fr_FR	French	France	ISO8859-1	FF
gu_IN	Gujarati	India	UTF-8	GI
he_IL	Hebrew	Israel	ISO8859-8	IL
hi_IN	Hindi	India	UTF-8	IN
hr_HR	Croatian	Croatia	ISO8859-2	HR
hu_HU	Hungarian	Hungary	ISO8859-2	HU
id_ID	Indonesian	Indonesia	ISO8859-1	II
it_CH	Italian	Switzerland	ISO8859-1	IC
it_IT	Italian	Italy	ISO8859-1	IT
iw_IL	Hebrew	Israel	ISO8859-8	IL
ja_JP	Japanese	Japan	IBM-943	JA
kk_KZ	Kazakh	Kazakstan	UTF-8	KK
ko_KR	Korean	Korea	IBM-eucKR	KR
mr_IN	Marati	India	UTF-8	MI
ms_MY	Malay	Malaysia	ISO8859-1	MY
nl_NL	Dutch	Netherlands	ISO8859-1	NN
no_NO	Norwegian	Norway	ISO8859-1	NO
pa_IN	Punjabi	India	UTF-8	PI
pl_PL	Polish	Poland	ISO8859-2	PL

Table 109. Supported language-territory names and LT codes for ASCII locales (continued)

Locale Name	Language	Country/Territory	ASCII Codeset	2-Byte LT Code
pt_BR	Portuguese	Brazil	ISO8859-1	BR
pt_PT	Portuguese	Portugal	ISO8859-1	PT
ro_RO	Romanian	Romania	ISO8859-2	RO
ru_RU	Russian	Russia	ISO8859-5	RU
sk_SK	Slovak	Slovakia	ISO8859-2	SK
sl_SI	Slovene	Slovenia	ISO8859-2	SI
sv_SE	Swedish	Sweden	ISO8859-1	SV
ta_IN	Tamil	India	UTF-8	AN
te_IN	Telugu	India	UTF-8	EN
th_TH	Thai	Thailand	TIS-620	TH
tr_TR	Turkish	Turkey	ISO8859-9	TR
zh_CN	Simplified Chinese	China (PRC)	IBM-eucCN	ZC
zh_HKS	Simplified Chinese	China (Hong Kong S.A.R. of China)	UTF-8	ZG
zh_HKT	Traditional Chinese	China (Hong Kong S.A.R. of China)	UTF-8	ZU
zh_SGS	Simplified Chinese	Singapore	UTF-8	ZS
zh_TW	Simplified Chinese	Taiwan	BIG5	ZT

The mapping between Codeset and the two-letter CC code is defined in the CC conversion table EDCUCSNM. This table is built with assembler macros, as follows:

```
EDCUCSNM TITLE 'CODE SET NAME CONVERSION TABLE'
EDCUCSNM CSECT
EDCCSNAM TYPE=ENTRY, CODESET=' IBM-037 ', CODE=' EA '
EDCCSNAM TYPE=ENTRY, CODESET=' IBM-273 ', CODE=' EB '
EDCCSNAM TYPE=ENTRY, CODESET=' IBM-274 ', CODE=' EC '
EDCCSNAM TYPE=ENTRY, CODESET=' IBM-277 ', CODE=' ED '
EDCCSNAM TYPE=ENTRY, CODESET=' IBM-278 ', CODE=' EE '
:
EDCCSNAM TYPE=END
END EDCUCSNM
```

CODESET specifies the name Codeset; CODE specifies the respective CC code.

You can customize this table by adding new CODESET names. The alphabetic codes in the first byte of each CC name are reserved by IBM for future use, but you can use codes starting with numeric values for your own customized names.

Table 110 on page 686 lists the Codeset names and their mappings into CC codes that are provided.

Table 110. Supported codeset names and CC codes

Codesets	Primary Country or Territory	2-Byte CC code
Big5	Taiwan	BT
IBM-037	USA, Canada, Brazil	EA
IBM-273	Germany, Austria	EB
IBM-274	Belgium	EC
IBM-277	Denmark, Norway	EE

Table 110. Supported codeset names and CC codes (continued)

Codesets	Primary Country or Territory	2-Byte CC code
IBM-278	Finland, Sweden	EF
IBM-280	Italy	EG
IBM-282	Portugal	EI
IBM-284	Spain, Latin America	EJ
IBM-285	United Kingdom	EK
IBM-290	Japan (Katakana)	EL
IBM-297	France	EM
IBM-300	Japanese DBCS	EN
IBM-420	Algeria, Bahrain, Egypt, Iraq, Jordan, Kuwait, Lebanon, Libya, Morocco, Oman, Qatar, Saudi Arabia, Syria, Tunisia, U.A.E., Yemen	FF
IBM-424	Israel	FB
IBM-425	Algeria, Bahrain, Egypt, Iraq, Jordan, Kuwait, Lebanon, Libya, Morocco, Oman, Qatar, Saudi Arabia, Syria, Tunisia, U.A.E., Yemen	AR
IBM-500	International	EO
IBM-838	Thailand	EP
IBM-848	Ukraine with Euro (Cyrillic)	AS
IBM-870	Croatia, Czech Republic, Hungary, Poland, Romania, Serbia(Latin), Slovakia, Slovenia	EQ
IBM-871	Iceland	ER
IBM-875	Greece	ES
IBM-880	Cyrillic	ET
IBM-924	Latin 9/Open Systems	DL
IBM-930	Japan Katakana Extended (combined with DBCS)	EU
IBM-933	Korea	GZ
IBM-935	China(PRC)	GY
IBM-937	Taiwan	GW
IBM-943	Japan	JA
IBM-943	China (PRC)	No
IBM-943G	Japan	AN
IBM-1025	Bulgaria, Macedonia, Russia, Serbia (Cyrillic)	FE
IBM-1026	Turkey	EW
IBM-1027	Japan (Latin) Extended	EX
IBM-1047	Latin 1/Open Systems	EY
IBM-1112	Lithuania	GD
IBM-1122	Estonia	FD
IBM-1123	Ukraine (Cyrillic)	FH
IBM-1125	Ukraine (Cyrillic)	AT

Table 110. Supported codeset names and CC codes (continued)

Codesets	Primary Country or Territory	2-Byte CC code
IBM-1140	USA, Canada, Brazil	HA
IBM-1141	Austria, Germany	HB
IBM-1142	Denmark, Norway	HE
IBM-1143	Finland, Sweden	HF
IBM-1144	Italy	HG
IBM-1145	Spain, Latin America	HJ
IBM-1146	United Kingdom	HK
IBM-1147	France	HM
IBM-1148	International	HO
IBM-1149	Iceland	HR
IBM-1153	Czech Republic, Hungary, Poland, Slovakia, Slovenia	MB
IBM-1156	Latvia, Lithuania	HZ
IBM-1157	Estonia	HD
IBM-1158	Ukraine with Euro (Cyrillic)	FI
IBM-1165	Latin 2/Open Systems	FG
IBM-1364	Korea	KZ
IBM-1371	Taiwan	ZT
IBM-1388	China (PRC)	GV
IBM-1390	Japan	HU
IBM-1399	Japan	HV
IBM-4933	China (PRC)	FJ
IBM-4971	Greece	HS
IBM-13124	China (PRC)	FK
IBM-53668	Algeria, Bahrain, Egypt, Iraq, Jordan, Kuwait, Lebanon, Libya, Morocco, Oman, Qatar, Saudia Arabia, Syria, Tunisia, U.A.E., Yemen	FV
IBMEUCCN	China (PRC)	BY
IBMEUCKR	Korea	BZ
ISO8859-1	All Latin 1 Countries	I1
ISO8859-2	Croatia, Czech Republic, Hungary, Poland, Romania, Serbia (Latin), Slovakia, Slovenia	I2
ISO8859-5	Bulgaria, Macedonia, Russia, Serbia (Cyrillic)	I5
ISO8859-7	Greece	I7
ISO8859-8	Israel	I8
ISO8859-9	Turkey	I9
TIS-620	Thailand	BU
UTF-8	All Countries	F8

The exceptions to the rule above are the following special locale names, which are already recognized:

- C (EBCDIC and ASCII)
- POSIX (EBCDIC and ASCII)
- SAA (EBCDIC only)
- S370 (EBCDIC only)

The special names C, POSIX, SAA, and S370 always refer to the built-in locales, which cannot be modified. The S370 locale and the following names are for locales in an old format, created with the EDCLOC assembler macro, rather than with the localedef utility:

- GERM (EBCDIC only)
- FRAN (EBCDIC only)
- UK (EBCDIC only)
- ITAL (EBCDIC only)
- SPAI (EBCDIC only)
- USA (EBCDIC only)

The EDCLOC generated locales are not supported in AMODE 64 applications.

You can use the C macros in [Table 111](#) on page 689, which are defined in the `locale.h` header file, as synonyms for these special locale names. These macros can only be used for EBCDIC locales. The `<prefix>` in the Compiled locale column is EDC for non-XPLINK locales and CEH for XPLINK locales. The C macros for the locales which list a prefix in the Compiled locales column, are not defined for AMODE 64 compilations.

Table 111. C macros used as synonyms for special locale names

Macro	Locale	Compiled locale
LC_C	C	Not applicable
LC_POSIX	POSIX	Not applicable
LC_C_GERMANY	"GERM"	<prefix>\$GERM
LC_C_FRANCE	"FRAN"	<prefix>\$FRAN
LC_C_UK	"UK"	<prefix>\$UK
LC_C_ITALY	"ITAL"	<prefix>\$ITAL
LC_C_SPAIN	"SPAI"	<prefix>\$SPAI
LC_C_USA	"USA"	<prefix>\$USA

The predefined name for the built-in locale in the old format is S370.

The rest of the special names refer to the EBCDIC locale objects whose names are built by prepending the letters EDC\$ for non-XPLINK locales or CEH\$ for XPLINK locales to the special name, as for EDC\$FRAN.

Chapter 55. Customizing a locale

This chapter describes how you can create your own locales, based on the locale definition files supplied by IBM. The information in this chapter applies to the format of locales based on the `localedef` utility.

The following example assumes that the target of the generated locale will be a data set, but locales may also reside in a z/OS UNIX file system (see “[Locale naming conventions](#)” on page 679 for differences in object names). In this example you will build a locale named `TEXAN` using the `charmap` file representing the IBM-1047 encoded character set. The locale is derived from the locale representing the English language and the cultural conventions of the United States. We will assume that non-XPLINK, XPLINK, and AMODE 64 applications will use the `TEXAN` locale. All three versions of the `TEXAN` locale will be generated.

1. Determine the source of the locale you are going to use. In this case, it is the English language in the United States locale, the source for which is the member `EDC$EUEY` of the PDS `CEE.SCEELOCX`.
2. Copy the member `EDC$EUEY` from PDS `CEE.SCEELOCX` to the data set `hlq.LOCALE.SOURCE` which has been pre-allocated with the same attributes as `CEE.SCEELOCX`.
3. In your new file, change the locale variables to the desired values. For example, change

```
d_t_fmt "%a %b %e %H:%M:%S %Z %Y"
```

to

```
d_t_fmt "Howdy Pardner %a %b %e %H:%M:%S %Z %Y"
```

4. This locale's Language-Territory value is `TEXAN`. The Codeset value is `IBM-1047`. `TEXAN` is not a valid PDS resident locale name in the runtime library, because it does not appear in the runtime Locale Name Table. You must modify the table to include the `TEXAN` locale. Here are the steps to follow.
 - a. Copy the member `EDC$LCNM` from PDS `CEE.SCEESAMP` to the data set `hlq.LOCALE.TABLE` which has been pre-allocated with the same attributes as `CEE.SCEESAMP`. The z/OS XL C/C++ Library uses this table to map locale code registry prefixes into two-character codes.
 - b. For this example, insert a new line into the assembler table before the last `EDC$LCNM TYPE=END` entry:

```
EDC$LCNM TYPE=ENTRY, LOCALE='TEXAN', CODESET='IBM-1047', CODE='1T'
```

5. Now that your locale name table has been modified, you must make it available to the system. Assemble the `EDC$LCNM` member and link-edit it into the `hlq.LOCALE.LOADLIB` load library with the member name `EDC$LCNM`. For our example, this is done as follows:

```
//HLASM      EXEC PGM=ASMA90
//SYSPRINT   DD SYSOUT=*
//SYSLIB     DD DSN=SYS1.MACLIB,DISP=SHR
//           DD DSN=CEE.SCEEMAC,DISP=SHR
//           DD DSN=CEE.SCEELOCX,DISP=SHR
//SYSUT1     DD UNIT=VIO,DISP=(NEW,DELETE),SPACE=(32000,(30,30))
//SYSUT2     DD UNIT=VIO,DISP=(NEW,DELETE),SPACE=(32000,(30,30))
//SYSUT3     DD UNIT=VIO,DISP=(NEW,DELETE),SPACE=(32000,(30,30))
//SYSPUNCH   DD DUMMY
//SYSLIN     DD DSN=<hlq>.LOCALE.OBJECT(EDC$LCNM),DISP=SHR
//SYSIN      DD DSN=<hlq>.LOCALE.TABLE(EDC$LCNM),DISP=SHR
//*
//LKED       EXEC EDCL,
//           OUTFILE='<hlq>.LOCALE.LOADLIB(EDC$LCNM),DISP=SHR'
//LKED.SYSLIN DD DSN=<hlq>.LOCALE.OBJECT(EDC$LCNM),DISP=SHR
```

6. Generate the non-XPLINK, XPLINK and 64-bit locale objects into a load library. Note that both the XPLINK and 64-bit locale objects must be placed in a PDSE, while non-XPLINK locale objects may be in either a PDS or PDSE load library.

- a. Determine the correct locale object names, using the locale naming Conventions outlined in “[Locale naming conventions](#)” on page 679. PDS resident locale object names are of the form prefixLTCC .

For this non-XPLINK locale the prefix is EDC\$, the LT code for TEXAN is 1T and the CC code for IBM-1047 is EY. The non-XPLINK object name is therefore EDC\$1TEY.

For this XPLINK locale the prefix is CEH\$. The LT and CC codes remain the same. The XPLINK object name is therefore CEH\$1TEY.

For this 64-bit locale the prefix is CEQ\$. The LT and CC codes remain the same. The 64-bit locale object name is, therefore, CEQ\$1TEY.

- b. Use localedef to generate the locale objects.

- For non-XPLINK:

```
//GENLOCN EXEC PROC=EDCLDEF,  
// INFILE='hlq.LOCALE.SOURCE(TEXAN)',  
//  
OUTFILE='hlq.LOCALE.LODLIB(EDC$1TEY),DISP=SHR',  
// LOPT='CHARMAP(IBM-1047)'
```

- For XPLINK:

```
//GENLOCX EXEC PROC=EDCXLDEF,  
// INFILE='hlq.LOCALE.SOURCE(TEXAN)',  
//  
OUTFILE='hlq.LOCALE.PDSE.LODLIB(CEH$1TEY),DISP=SHR',  
// LOPT='CHARMAP(IBM-1047)'
```

- For 64-bit

The batch and TSO versions of the localedef utility cannot be used to generate 64-bit locales. The UNIX Systems Services utility must be used. To do this from TSO or batch the BPXBATCH utility can be used. See [z/OS UNIX System Services Command Reference](#) for more information about BPXBATCH. Here, we will assume we are in a UNIX System Services shell session:

```
cp "'hlq.LOCALE.SOURCE(TEXAN)'" texan.localedef  
localedef -6 -i texan.localedef -f /usr/lib/nls/charmap/IBM-1047  
TEXAN.IBM-1047.lp64  
cp TEXAN.IBM-1047.lp64 "'hlq.LOCALE.PDSE.LODLIB(CEQ$1TEY)'"
```

See [z/OS XL C/C++ User's Guide](#) for detailed information about the batch and TSO versions of localedef utility. The UNIX System Services version of the localedef utility is also described in [z/OS UNIX System Services Command Reference](#).

Note: The TEXAN locale uses one of the IBM supplied CHARMAPS. If you need to customize a CHARMAP, then you must define its two-letter CC code in the Codeset Name table EDCUCSNM. This is similar to defining the locale TEXAN in EDC\$LCNM. The two-letter CHARMAP codes beginning with a number are reserved for customer use. This is the same as the convention for customer-supplied Locale Name LT codes in the Locale Name table. The CC portion of your locale object names would then change to be the new CC value you added to the Codeset Name table.

Using the customized locale

Your locale objects must be made available to your program before they can be used. For PDS and PDSE resident locales, your load library must be included in your program search order. For resident locales in the z/OS UNIX file system, do one of the following:

- Copy your locales into the system default locale object directory /usr/lib/nls/locale.
- Update your LOCPATH environment variable to include the directory containing your locales.

For example, assume that the CCNGCL1 program has been compiled with LP64 into a UNIX file system executable called `getlocname`. Further assume that you have generated non-XPLINK, XPLINK and AMODE 64 UNIX file system resident versions of the TEXAN locale into your current directory. The following commands make TEXAN available to non-XPLINK, XPLINK and AMODE 64 applications:

```
$ ls
TEXAN.IBM-1047 TEXAN.IBM-1047.xplink TEXAN.IBM-1047.lp64 getlocname
$ export LOCPATH=$PWD
$ export LC_ALL=TEXAN.IBM-1047
$ getlocname
Default NULL locale = C
Default "" locale = /u/marcw/TEXAN.IBM-1047.lp64
$
```

If `getlocname` was compiled non-XPLINK then the output would look like the following:

```
$ getlocname
Default NULL locale = C
Default "" locale = /u/marcw/TEXAN.IBM-1047
$
```

If `getlocname` was compiled XPLINK then the output would look like the following:

```
$ getlocname
Default NULL locale = C
Default "" locale = /u/marcw/TEXAN.IBM-1047.xplink
$
```

The customized locale is now ready to be used in these ways:

- Explicitly referenced by name in z/OS XL C/C++ application code that uses `setlocale()` calls referring to the locale descriptive name (recommended) such as:

```
setlocale(LC_ALL, "TEXAN.IBM-1047");
```

or by a short internal name (not recommended) such as:

```
setlocale(LC_ALL, "1TEY");
```

- Explicitly referenced in the z/OS XL C/C++ initialization exit, using customized setup code in CEEBINT.
- Implicitly specified in each user environment with environment variables.

Note: You cannot customize the built-in locales, C, POSIX, SAA, or S370. The locale source files EDC\$POSX and EDC\$SAAC are provided for reference only.

Referring explicitly to a customized locale

Figure 204 on page 694 shows a non-XPLINK program (CCNGCL1) with an explicit reference to the TEXAN locale.

```

/* this example shows how to get the local time formatted by the */
/* current locale */

#include <stdio.h>
#include <time.h>
#include <locale.h>

int main(void){
    char dest[80];
    int ch;
    time_t temp;
    struct tm *timeptr;
    temp = time(NULL);
    timeptr = localtime(&temp);
    /* Fetch default locale name */
    printf("Default empty_str locale is %s\n",setlocale(LC_ALL,""));
    ch = strftime(dest,sizeof(dest)-1,
        "Local C datetime is %c", timeptr);
    printf("%s\n", dest);

    /* Set new Texan locale name */
    printf("New locale is %s\n", setlocale(LC_ALL,"Texan.IBM-1047"));
    ch = strftime(dest,sizeof(dest)-1,
        "Texan datetime is %c ", timeptr);
    printf("%s\n", dest);

    return(0);
}

```

Figure 204. Referring explicitly to a customized locale

Compile the program. Before you execute it, ensure the load library containing the non-XPLINK version of the TEXAN locale and updated table is available. If you compile your program XPLINK, ensure the load library containing the XPLINK version of the TEXAN locale and updated Locale Name table is available. If you compile your program LP64, ensure the load library containing the 64-bit version of the TEXAN locale and updated Locale Name table is available. The output should be similar to:

```

Default empty_str locale is S370
Local C datetime is Fri Aug 20 14:58:12 1993
New locale is TEXAN
Texan datetime is Howdy Pardner Fri Aug 20 14:58:12 1993

```

For programs which are run POSIX(OFF), and which are not 64-bit programs, if the second operand to `setlocale()` had been `NULL`, rather than `""`, the default locale name returned would have been `C`.

```

setlocale(LC_ALL,"") returns "S370"
setlocale(LC_ALL,NULL) returns "C"

```

Note: For `setlocale(LC_ALL, "")`, the result depends on the locale-related environment variables, the POSIX runtime option, and whether the program is AMODE 64 or not. See Chapter 57, “Definition of S370 C, SAA C, and POSIX C locales,” on page 699 for more information about the definition of the S370 locale.

Referring implicitly to a customized locale

An installation may require that a global mechanism should be used for all C programs. The exit CEEBINT may be used for this purpose. Users can insert a `setlocale()` call inside the routines referencing the locale required. [Figure 205 on page 695](#) shows an example program (CCNGCL2).

```

/* this example refers implicitly to a customized locale */

#ifdef __cplusplus
extern "C"{
#else
#pragma linkage(CEEBINT,OS)
#endif

void CEEBINT(int, int, int, int, void**, int, void**);
#pragma map(CEEBINT,"CEEBINT")

#ifdef __cplusplus
}
#endif

#include <locale.h>
#include <stdio.h>

int main(void){
    printf("Default NULL locale = %s\n", setlocale(LC_ALL,NULL));
    printf("Default \"\" locale = %s\n", setlocale(LC_ALL,""));
}

void CEEBINT(int number, int retcode, int rsncode, int fncode,
             void **a_main, int userwd, void **a_exits)
{
    /* user code goes here */
    printf("CEEBINT entry. number = %i\n", number);
    printf("Locale = %s\n", setlocale(LC_ALL,"Texan.IBM-1047"));
}

```

Figure 205. Referring implicitly to a customized locale

If the above example is compiled and executed with the TEXAN locale, the results are as follows:

```

CEEBINT entry. number = 7
Locale = TEXAN.IBM-1047
Default NULL locale = TEXAN.IBM-1047
Default "" locale = S370

```

The exit CEEBINT may provide a uniform way of restricting the use of customized locales across an installation. To do this, a system programmer can compile CEEBINT separately, and link it with the application program that will use it. The disadvantage to this approach is that CEEBINT must be link-edited into each user module explicitly. See [Chapter 44, “Using runtime user exits,” on page 537](#) for more information about user exits.

[Figure 206 on page 695](#) shows a sample program (CCNGCL3) that uses environment variables to select a locale. (For more information about setting environment variables, see [Chapter 28, “Using environment variables,” on page 327](#).)

```

/* this example can be used with setenv() to specify the name of a */
/* locale */

#include <locale.h>
#include <stdio.h>

int main(void){
    printf("Default NULL locale = %s\n", setlocale(LC_ALL,NULL));
    printf("Default \"\" locale = %s\n", setlocale(LC_ALL,""));

    return(0);
}

```

Figure 206. Using environment variables to select a locale

If you run this program in [Figure 206 on page 695](#) as is, without calling `setenv()`, you can expect the following result (for a 31-bit, POSIX(OFF), program):

```

Default NULL locale = C
Default "" locale = S370

```

On the other hand, if you issue the above `setenv()` call after `main()` but before the first `printf()` statement, the `LC_ALL` variable will be set to "TEXAN.IBM-1047" and you can expect this result instead:

```
Default NULL locale = C
Default "" locale = TEXAN.IBM-1047
```

In the example above, the default NULL locale returns C because the value of `LC_ALL` does not affect the current locale until the next `setlocale(LC_ALL, "")` is done. When this call is made, the `LC_ALL` environment variable will be used and the locale will be set to TEXAN.IBM-1047.

The names of the environment variables match the names of the locale categories:

- `LC_ALL`
- `LC_COLLATE`
- `LC_CTYPE`
- `LC_MONETARY`
- `LC_NUMERIC`
- `LC_TIME`
- `LC_TOD`
- `LC_SYNTAX`

See [z/OS C/C++ Runtime Library Reference](#) for information about `setlocale()`.

Customizing your installation

When z/OS XL C/C++ initializes its environment, it uses the C locale as its default locale. The only values that may be customized when z/OS Language Environment is installed are those defined in the `TZ` or `_TZ` environment variable, which can override `LC_TOD` category values in the default locale. Details on this customization are provided in [Chapter 56, "Customizing a time zone,"](#) on page 697.

Chapter 56. Customizing a time zone

For applications that work with local time, you can customize time zone information in the following two ways:

- TZ or _TZ environment variable

In a distributed environment, you may have users in several time zones. Use the TZ or _TZ environment variable to set each time zone. The user of your application can use the ENVAR runtime option with the TZ or _TZ environment variable to select the appropriate time zone.

For POSIX(ON) programs the TZ environment variable is used. For POSIX(OFF) programs the _TZ environment variable is used.

When neither TZ nor _TZ is defined, the current locale is interrogated for time zone information. If neither TZ nor _TZ is defined and LC_TOD time zone information is not present in the current locale, a default value is applied to local time. POSIX programs simply default to Coordinated Universal Time (UTC), while non-POSIX programs establish an offset from UTC based on the setting of the system clock.

The following functions are time zone sensitive: `ctime()`, `ctime64()`, `getdate()`, `localtime()`, `localtime64()`, `mktime()`, `mktime64()`, `strftime()`, and `tzset()`. The external variables `daylight`, `timezone`, and `tzname` and the thread specific functions `__dlight()` and `__tzone()` are also time zone sensitive.

LC_TOD category of a locale

You can customize the LC_TOD category in a locale to a particular time zone. The LC_TOD category binds each C/C++ locale to one time zone. For more information about customizing the LC_TOD category, see “LC_TOD category” on page 671 and Chapter 55, “Customizing a locale,” on page 691.

Using the TZ or _TZ environment variable to specify time zone

The C/C++ runtime library assumes times returned by the operating system are stored using Greenwich Mean Time (GMT) or Coordinated Universal Time (UTC). This time is referred to as the universal reference time. You can use the TZ or _TZ environment variable to specify information at run time. The C/C++ runtime library uses this information to map universal reference times to local times.

The TZ or _TZ environment variable has the following format.

```
TZ=standardHH[:MM[:SS]]
[daylightHH[:MM[:SS]]]
[,startdate[/starttime],enddate[/endtime]]
```

The value of the TZ or _TZ environment variable has the following five fields (two required and three optional):

standard

An alphabetic abbreviation for the local standard time zone (for example, GMT, EST, MESZ).

HH[:MM[:SS]]

The time offset westward from the universal reference time. A leading minus sign (-) means that the local time zone is east of the universal reference time. An offset of this form must follow *standard* and can also optionally follow *daylight*. An optional colon (:) separates hours from optional minutes and seconds.

If *daylight* is specified without a *daylight* offset, daylight savings time is assumed to be one hour ahead of the standard time.

[daylight]

The abbreviation for your local daylight savings time zone. If the first and third fields are identical, or if the third field is missing, daylight savings time conversion is disabled. The number of hours, minutes,

and seconds your local daylight savings time is offset from UTC when daylight savings time is in effect. If the daylight savings time abbreviation is specified and the offset omitted, the offset of one hour is assumed.

[,startdate[/starttime],enddate[/endtime]]

A rule that identifies the start and end of daylight savings time, specifying when daylight savings time should be in effect. Both the *startdate* and *enddate* must be present and must either take the form Jn, n, or Mm.n.d where:

- Jn is the Julian day n ($1 \leq n \leq 365$) and does not account for leap days.
- n is the zero-based Julian day ($0 \leq n \leq 365$). Leap days are counted; therefore, you can refer to February 29th.
- For Mm.n.d, the d'th day of week n of month m of the year. Day is ($0 \leq d \leq 6$), with day 0 = Sunday, day 1 = Monday, and so on. Week is ($1 \leq n \leq 5$) where week 1 is the first week in which day d occurs and week 5 is the last occurrence of day d in the month (may actually be in week 4 or 5). Month is ($1 \leq m \leq 12$).

Neither *starttime* nor *endtime* are required, and when omitted, their values default to 02:00:00. If this daylight savings time rule is omitted altogether, the values in the rule default to the standard American daylight savings time rules starting at 02:00:00 the second Sunday in March and ending at 02:00:00 the first Sunday in November.

Relationship between TZ or _TZ and LC_TOD

The C/C++ runtime library uses time zone information specified by the TZ or _TZ environment variable to convert universal reference times to local times. When neither the TZ nor _TZ variable is defined, the C/C++ runtime library uses time zone information specified by the LC_TOD category of the current locale to map universal reference times to local times. If LC_TOD in the current locale has not been customized, the C/C++ runtime library uses Coordinated Universal Time (UTC) for POSIX programs or, for non-POSIX programs, the time zone of the system on which C/C++ is installed.

Note: The time zone external variables, *tzname*, *timezone*, and *daylight*, declarations remain feature test protected in `time.h`. Definition of these external variables are only known to the C/C++ runtime library if the z/OS UNIX System Services C/C++ signature CSECT is link edited with your C/C++ application.

Chapter 57. Definition of S370 C, SAA C, and POSIX C locales

The POSIX, SAA, and S370 locales are pre-built locales used as defaults by the C runtime library. The POSIX locale complies with the standard UNIX definition and supports the z/OS UNIX environment. The SAA locale, which provides compatibility with previous releases of C/370, is consistent with the POSIX model, but varies slightly with respect to several values. The S370 locale, which is not supported for AMODE 64 applications, is compatible with an older format generated by the EDCLOC assembler macro rather than through the use of the `localedef` utility.

The POSIX definition of the C locale is described below, with the IBM extensions `LC_SYNTAX` and `LC_TOD` showing their default values.

The SAA and S370 definitions of the C locale are different from the POSIX definition; consistency with previous releases of z/OS XL C/C++ is provided for migration compatibility. The differences are described in [“Differences between SAA C and POSIX C locales” on page 705](#).

The relationship between the POSIX C and SAA C locales is as follows. If you are running with the runtime option `POSIX(OFF)`:

1. The SAA C locale definition is the default. "C", "SAA", and "S370" are treated as synonyms for the SAA C locale definition, which is prebuilt into the library.

The source file `EDC$SAAC LOCALE` is provided for reference, but cannot be used to alter the definition of this prebuilt locale.

2. Issuing `setlocale(category, "")` has the following effect:
 - First, locale-related environment variables are checked for the locale name to use in setting the *category* specified. Querying the locale with `setlocale(category, NULL)` returns the name of the locales specified by the appropriate environment variables.
 - If no non-null environment variable is present, then it is the equivalent of having issued `setlocale(category, "S370")`. That is, the locale chosen is the SAA C locale definition, and querying the locale with `setlocale(category, NULL)` returns "S370" as the locale name.
3. If no `setlocale()` function is issued, or `setlocale(LC_ALL, "C")`, then the locale chosen is the pre-built SAA C locale, and querying the locale with `setlocale(category, NULL)` returns "C" as the locale name.
4. For `setlocale(LC_ALL, "SAA")`, the locale chosen is the pre-built SAA C locale, and querying the locale with `setlocale(category, NULL)` returns "SAA" as the locale name.
5. For `setlocale(LC_ALL, "S370")`, the locale chosen is the pre-built SAA C locale, and querying the locale with `setlocale(category, NULL)` returns "S370" as the locale name. AMODE 64 applications do not support the "S370" locale, and `setlocale` will fail requests for that name.
6. For `setlocale(LC_ALL, "POSIX")`, the locale chosen is the pre-built POSIX C locale, and querying the locale with `setlocale(category, NULL)` returns "POSIX" as the locale name.

If you are running with the runtime option `POSIX(ON)`:

1. The POSIX C locale definition is the default. "C" and "POSIX" are synonyms for the POSIX C locale definition, which is pre-built into the library.

The source file `EDC$POSX LOCALE` is provided for reference, but cannot be used to alter the definition of this pre-built locale.

2. Issuing `setlocale(category, "")` has the following effect:
 - Locale-related environment variables are checked to find the name of locales that can set the *category* specified. Querying the locale with `setlocale(category, NULL)` returns the name of the locale specified by the appropriate environment variables.

Additional locale categories for POSIX C (Part 2 of 6)

```

<ENQ>
<ACK>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<S0>
<SI>
<DLE>
<DC1>
<DC2>
<DC3>
<DC4>
<NAK>
<SYN>
<ETB>
<CAN>
<EM>
<SUB>
<ESC>
<IS4>
<IS3>
<IS2>
<IS1>
<space>
<exclamation-mark>
<quotation-mark>
<number-sign>
<dollar-sign>
<percent-sign>
<ampersand>
<apostrophe>
<left-parenthesis>
<right-parenthesis>
<asterisk>
<plus-sign>
<comma>
<hyphen>
<period>
<slash>
<zero>
<one>
<two>
<three>
<four>
<five>
<six>
<seven>
<eight>
<nine>
<colon>
<semicolon>
<less-than-sign>
<equals-sign>
<greater-than-sign>
<question-mark>
<commercial-at>
<A>
<B>
<C>
<D>
<E>
<F>
<G>

```

Additional locale categories for POSIX C (Part 3 of 6)

END LC_COLLATE

```
int_curr_symbol      ""
currency_symbol      ""
mon_decimal_point    ""
mon_thousands_sep   ""
mon_grouping         ""
```



```

"<J><u><l><y>";/
"<A><u><g><u><s><t>";/
"<S><e><p><t><e><m><b><e><r>";/
"<O><c><t><o><b><e><r>";/
"<N><o><v><e><m><b><e><r>";/
"<D><e><c><e><m><b><e><r>"

% equivalent of AM/PM (%p)
am_pm      "<A><M>";"<P><M>"

% appropriate date and time representation (%c) "%a %b %e %H:%M:%S %Y"
d_t_fmt    "<percent-sign><a><space><percent-sign><b><space><percent-sign><e>/
<space><percent-sign><H><colon><percent-sign><M>/
<colon><percent-sign><S><space><percent-sign><Y>"

% appropriate date representation (%x) "%m/%d/%y"
d_fmt      "<percent-sign><m><slash><percent-sign><d><slash><percent-sign><y>"

% appropriate time representation (%X) "%H:%M:%S"
t_fmt      "<percent-sign><M><colon><percent-sign><M><colon><percent-sign><S>"

% appropriate 12-hour time representation (%r) "%I:%M:%S %p"
t_fmt_ampm "<percent-sign><I><colon><percent-sign><M><colon><percent-sign><S>/
<space><percent-sign><p>"

END LC_TIME

#####
LC_MESSAGES
#####

yesexpr "<circumflex><left-square-bracket><y><Y><right-square-bracket>"
noexpr  "<circumflex><left-square-bracket><n><N><right-square-bracket>"

END LC_MESSAGES

```

Additional locale categories for POSIX C (Part 6 of 6)

Differences between SAA C and POSIX C locales

In fact, there are three built-in locales, S370 C, SAA C, and POSIX C. The default locale at your site depends on the system that is running the application. Issuing `setlocale(LC_ALL, "")` sets the default, based on the current environment. Issuing `setlocale(LC_ALL, "SAA")` sets the SAA C locale, even when you are running with the POSIX(ON) runtime option. Likewise, `setlocale(LC_ALL, "POSIX")` sets the POSIX locale.

If you are running in a C locale, one way you can determine if the SAA C or the POSIX locale is in effect is to check if the cent sign (¢ at X'4A') is defined as a punctuation character. Under the default POSIX support, the cent sign is not part of the POSIX portable character set. [Figure 208 on page 705](#) (sample CCNGDL1) shows how to perform this test.

```

/* this example shows how to determine whether the SAA C or POSIX */
/* locale is in effect */

#include <stdio.h>
#include <ctype.h>
int main(void)
{
    if (ispunct(0x4A)) {
        printf(" cent sign is punct\n");
        printf(" current locale is SAA- or S370-like\n");
    }
    else {
        printf(" cent sign is not punct\n");
        printf(" default locale is POSIX-like\n");
    }
    return(0);
}

```

Figure 208. Determining which locale is in effect

Under the SAA or System/370 default locales, the lowercase letters collate before the uppercase letters; under the POSIX definition, the lowercase letters collate after the uppercase letters. The locale "" is the same locale as the one obtained from `setlocale(LC_ALL, "")`. For more detail on these special environment variables, see [Chapter 28, “Using environment variables,” on page 327](#). Other differences between the SAA C locale and the POSIX C locale are as follows:

<mb_cur_max>

The POSIX C locale is built using coded character set IBM-1047, with `<mb_cur_max>` as 1. The SAA C locale is built using coded character set IBM-1047, with `<mb_cur_max>` as 4.

The cent sign

In the default POSIX support, the cent sign (¢) is *not* part of the POSIX portable character set; in the SAA locale, it is defined as a punctuation character.

Collation weight by case

In the POSIX definition, the lowercase letters collate *after* the uppercase letters; in the SAA or System/370 default locales, the lowercase letters collate *before* the uppercase letters.

LC_CTYPE category

The SAA C locale has all the EBCDIC control characters defined in the 'cntrl' class. The POSIX C locale has only the ASCII control characters in the 'cntrl' class. The SAA C locale includes ¢ (the cent character) and ¦ (the broken vertical line) as 'punct' characters. The POSIX C locale does not group these characters as 'punct' characters.

LC_COLLATE category

The default collation for the SAA C locale is the EBCDIC sequence. The POSIX C locale uses the ASCII collation sequence; the first 128 ASCII characters are defined in the collation sequence, and the remaining EBCDIC characters are at the end of the collating sequence.

LC_TIME category

The SAA C locale uses the date and time format (`d_t_fmt`) as "%Y/%M/%D %X"; the POSIX C locale uses "%a %b %d %H/%M/%S %Y". The SAA C locale uses the strings "am" and "pm"; the POSIX C locale uses "AM" and "PM".

Chapter 58. Code set conversion utilities

This chapter describes the code set conversion utilities supported by the z/OS XL C/C++ compiler. These utilities are as follows:

genxlt utility

Generates a translation table for use by the iconv utility and `iconv()` functions.

iconv utility

Converts a file from one code set encoding to another.

iconv() functions

Perform code set translation. These functions are `iconv_open()`, `iconv()`, and `iconv_close()`. They are used by the iconv utility and may be called from any z/OS XL C/C++ program requiring code set translation.

uconvdef utility

Handles Universal-coded character sets. Creates binary conversion tables that define mapping between UCS-2 and multibyte code sets.

See [z/OS XL C/C++ User's Guide](#) for descriptions of the genxlt and iconv utilities, [z/OS C/C++ Runtime Library Reference](#) for descriptions of the `iconv()` functions, and [z/OS MVS Program Management: User's Guide and Reference](#) for descriptions of the uconvdef utility.

The genxlt utility

The genxlt utility reads a source translation file from `InputFile`, writes the compiled version to `OutputFile`, and then generates the translation load module. The source translation file provides the conversion specification from `fromCodeSet` to `toCodeSet`. The source translation file contains directives that are acted upon by the genxlt utility to produce the compiled version of the translation table.

The name of the conversion programs have the following naming conventions:

- The name starts with a four letter prefix. The prefix is EDCU for non-XPLINK converters, CEHU for XPLINK converters, and CEQU for AMODE 64 converters.
- The prefix is followed by the two-letter CC code that corresponds to the `CodesetRegistry.CodesetEncoding` name of the `fromCodeSet` defined in the [Table 110 on page 686](#).
- The first CC code is followed by the two-letter CC code than corresponds to the `CodesetRegistry.CodesetEncoding` name of the `toCodeSet` defined in the [Table 110 on page 686](#).

To generate your own conversions, you must modify the codeset name table EDCUCSNM with the macros described in “Locale naming conventions” on page 679. For descriptions of the genxlt and iconv utilities, refer to [z/OS XL C/C++ User's Guide](#). There is also a UNIX System Services iconv utility, which is described in [z/OS UNIX System Services Command Reference](#).

The iconv utility

The iconv utility reads characters from the input file, converts them from `fromCodeSet` encoding to `toCodeSet` encoding, and writes them to the output file.

The conversion is performed by the code conversion functions of the runtime library. They are described in “Code conversion functions” on page 708. The tables used are determined by the CC codes of the `fromCodeSet` and `toCodeSet` appended to the four-character prefix. The prefix is EDCU for non-XPLINK converters, CEHU for XPLINK converters, and CEQU for AMODE 64 converters. See [z/OS XL C/C++ User's Guide](#) for descriptions of the genxlt and iconv utilities. There is also a UNIX System Services iconv utility, which is described in [z/OS UNIX System Services Command Reference](#).

The `iconv` utility can also perform bidirectional layout transformation (such as shaping and reordering) while converting from `fromCodeSet` to `toCodeSet` according to the value of an environment variable called `_BIDIION`. The value of this variable is either set to `TRUE` to activate the BiDi layout transformation or `FALSE` to prevent the bidirectional layout transformation. If this variable is not defined in the environment it defaults to `FALSE`. The `_BIDIATTR` environment variable can be used to contain the bidirectional attributes (for information on bidirectional layout transformation see [Chapter 60, “Bidirectional language support,”](#) on page 747) which will determine the way the bidirectional transformation takes place. These two environment variables are described in [Chapter 28, “Using environment variables,”](#) on page 327.

Code conversion functions

The `iconv_open()`, `iconv()`, and `iconv_close()` library functions can be called from C or C++ source to initialize and perform the characters conversions from one character set encoding to another.

The `iconv()` family of functions has been modified to utilize character conversion services provided by Unicode Services. The `iconv_open()`, `iconv()`, and `iconv_close()` function interfaces remain unchanged except for the addition of the following:

- Four new errno values - `ECUNNOENV`, `ECUNNOCONV`, `ECUNNOTALIGNED`, and `ECUNERR`
- Two new environment variables - `_ICONV_MODE` and `_ICONV_TECHNIQUE`

For more information about these errno values and environment variables, see the `iconv_open()` function description in *z/OS C/C++ Runtime Library Reference*.

There are differences in externals between the `iconv()` family of functions and Unicode Services. However, the differences in externals are managed by the `iconv()` family of functions except where noted in the C/C++ Migration Guide for Application Developers. All conversions listed in [Table 113 on page 711](#) and [Table 114 on page 721](#) will continue to work as they do today. However, Unicode Services supports conversions between thousands of additional character sets not listed in [Table 113 on page 711](#) and [Table 114 on page 721](#). A complete list of conversions supported by Unicode Services can be found in the *z/OS Unicode Services User's Guide and Reference*. To set up a conversion using `iconv_open()` for any of the character sets listed in EBCDIC Conversion Table, ASCII Conversion Table, and Unicode Conversion Table, use a character string representing the CCSID's for `fromcode`/`toCode`. For example, to set up a conversion from CCSID 00256 to CCSID 00870 using conversion technique R, you need to set the `_ICONV_TECHNIQUE` environment variable to R and call `iconv_open()` as follows:

```
cd = iconv_open("00870", "00256");
```

and continue to use `iconv()` and `iconv_close()` as in previous releases.

`_ICONV_MODE` environmental variable

The `_ICONV_MODE` environmental variable selects the behavior mode for the `iconv_open()`, `iconv()`, and `iconv_close()` family of functions. The `_ICONV_MODE` value can be set to:

C

Users have created their own `iconv()` converter(s). Search for user created converters first. If the user-created converter is not located, try using Unicode Conversion Services to perform the conversion.

Note: `_ICONV_UCS2` and `_ICONV_PREFIX` environment variables will be honored while searching for user-created converters, but they will not be honored while using Unicode Conversion Services.

U

Use Unicode Conversion Services to perform all conversions not listed in the [Table 113 on page 711](#). This is the default value for `_ICONV_MODE`. The values of the `_ICONV_UCS2` and `_ICONV_PREFIX` will not be honored.

_ICONV_TECHNIQUE environmental variable

The `_ICONV_TECHNIQUE` environmental variable is the technique value used by Unicode Conversion Services. For more information regarding the Unicode conversion Services value, see [z/OS Unicode Services User's Guide and Reference](#). `_ICONV_TECHNIQUE` can be set to one of the following values:

R

Roundtrip conversion; roundtrip conversions between two CCSIDs assure that all characters making the roundtrip arrive as they were originally.

E

Enforced subset conversion; enforced subset conversions map only those characters from one CCSID to another that have a corresponding character in the second CCSID. All other characters are replaced by a substitution character.

C

Customized conversion; customized conversions use conversion tables that have been created to address some special requirements. Note that these customized conversion tables refer to Unicode Conversion Services customized tables, not user-supplied `iconv()` style conversion tables.

L

Language Environment-Behavior conversion; Language Environment-Behavior conversions use tables that map characters like the `iconv()` function of the Language Environment runtime Library does. These conversions differ from others primarily in their mapping of the EBCDIC newline (NL) character to ASCII and Unicode linefeed (LF).

M

Modified Language Environment-Behavior conversion; Modified Language Environment-Behavior conversions use tables that map characters like the `iconv()` function of the Language Environment runtime library does for converters ending with C; for example IBM-932C.

0-9

User-defined conversions. See Appendix C, "Defining CCSIDs and conversion tables" in the [z/OS Unicode Services User's Guide and Reference](#).

This refers to user defined tables for Unicode Conversion Services and should not be confused with user-defined converters for the Language Environment's `iconv()` function.

Notes:

1. You can specify more than one value for `_ICONV_TECHNIQUE`; the values will be tried in the order specified. For example, if a value of CE were specified as the `_ICONV_TECHNIQUE`, a converter using the Unicode Conversion Services technique of C will be searched for first. If it does not exist, a converter using the Unicode Conversion Services technique of E will be searched for.
2. If a value is not specified for the `_ICONV_TECHNIQUE` environment variable, a default value of LMREC will be used.

Code set converters supplied

A set of code set converters is provided in the National Language Resources component of z/OS Language Environment. Consult your system programmer to see if this component has been installed on your system. [Figure 209 on page 710](#) shows the converters.

Round Trip Conversions(RTC) or Customized
Round Trip Conversions(C-RTC), which means round trip with exceptions.

Conversions:
Latin-1 EBCDIC to/from Latin-1 EBCDIC: RTC
Non-Latin-1 EBCDIC to/from Latin-1 EBCDIC: RTC
Latin-1 ASCII to/from Latin-1 EBCDIC: C-RTC
Non_latin-1 ASCII to/from Latin-1 EBCDIC: C-RTC

Example of Customized Round Trip Conversions(C-RTC) is
IBM-850 to/from IBM-1047 conversion.

Customized Round Trip Conversion

IBM-850 Code Point		IBM-1047 Code Point
0A	<->	15
DA	->	3F
0A	<-	25

Figure 209. Supplied code set converters

The code set converters that are provided as programs are shown in Table 113 on page 711. The GENXLT source for the code set converters are shipped in the CEE.SCEEGXLT data set.

Notes:

1. Table 112 on page 710 shows the <prefix> values that appear in the Program Name column of Table 113 on page 711.

Table 112. Referencing data types

Converter	Prefix
31-bit	EDCU
31-bit XPLINK	CEHU
AMODE 64	CEQU

2. Specify IBM-932C or IBM-eucJC as the `iconv_open()` source or target code set name to set up for conversion of POSIX data encoded by IBM-932 or IBM-eucJP to or from a host code set encoding of the data such as IBM-930 or IBM-939.

Examples of POSIX data are C/C++ source and shell scripts. The data includes characters from the POSIX character set. The names IBM-932C and IBM-eucJC indicate that the <yen> and <overline> characters in POSIX data encoded by IBM-932 or IBM-eucJP map to the <backslash> and <tilde> characters, respectively, when the data is converted to or from host encodings.



CAUTION: The naming conventions provided for building genxlt conversion tables allow the `iconv` interfaces to recognize the converters. All genxlt conversion tables, whether customized or shipped in z/OS Language Environment, are intended for use with the C/C++ `iconv` interfaces or the `iconv` utility. Direct programming to these tables is not supported and will produce unpredictable results.

IBM makes no guarantee that converter binaries or source shipped with z/OS Language Environment will continue to be shipped in future releases.

UCS-2 converters, still shipped with z/OS Language Environment in z/OS V1R9, are removed in z/OS V1R11. See “Universal coded character set converters” on page 720 for more information.

Starting in z/OS V1R12, the C/C++ Runtime Library will not ship the genxlt source for character conversions performed by Unicode Services. However, if you require the affected genxlt converters and plan to build them by creating your own source or by migrating source from a prior release, you must continue to name them as shown in Table 113 on page 711. For more details, refer to “The genxlt utility” on page 707.

Table 113. Coded character set conversion tables

FromCode	ToCode	GENXLT source shipped	Program Name
IBM-858	IBM-1047	Yes	<prefix>AIEY
IBM-858	IBM-1140	Yes	<prefix>AIHA
IBM-858	IBM-1141	Yes	<prefix>AIHB
IBM-858	IBM-1142	Yes	<prefix>AIHE
IBM-858	IBM-1143	Yes	<prefix>AIHF
IBM-858	IBM-1144	Yes	<prefix>AIHG
IBM-858	IBM-1145	Yes	<prefix>AIHJ
IBM-858	IBM-1146	Yes	<prefix>AIHK
IBM-858	IBM-1147	Yes	<prefix>AIHM
IBM-858	IBM-1148	Yes	<prefix>AIHO
IBM-858	IBM-1149	Yes	<prefix>AIHR
IBM-037	IBM-924	Yes	<prefix>EAEZ
IBM-273	IBM-924	Yes	<prefix>EBEZ
IBM-278	IBM-924	Yes	<prefix>EFEZ
IBM-280	IBM-924	Yes	<prefix>EGEZ
IBM-284	IBM-924	Yes	<prefix>EJEZ
IBM-285	IBM-924	Yes	<prefix>EKEZ
IBM-297	IBM-924	Yes	<prefix>EMEZ
IBM-500	IBM-924	Yes	<prefix>EOEZ
IBM-871	IBM-924	Yes	<prefix>EREZ
IBM-1047	IBM-858	Yes	<prefix>EYAI
IBM-1047	IBM-923	Yes	<prefix>EYEZ
IBM-924	IBM-037	Yes	<prefix>EZEZ
IBM-924	IBM-273	Yes	<prefix>EZEB
IBM-924	IBM-278	Yes	<prefix>EZEF
IBM-924	IBM-280	Yes	<prefix>EZEG
IBM-924	IBM-284	Yes	<prefix>EZEJ
IBM-924	IBM-285	Yes	<prefix>EZ EK
IBM-924	IBM-297	Yes	<prefix>EZEM
IBM-924	IBM-500	Yes	<prefix>EZEZ
IBM-924	IBM-871	Yes	<prefix>EZ ER
IBM-923	IBM-1047	Yes	<prefix>EZEY
IBM-924	IBM-1140	Yes	<prefix>EZHA
IBM-924	IBM-1141	Yes	<prefix>EZHB
IBM-924	IBM-1142	Yes	<prefix>EZHE
IBM-924	IBM-1143	Yes	<prefix>EZHF
IBM-924	IBM-1144	Yes	<prefix>EZHG

Table 113. Coded character set conversion tables (continued)

FromCode	ToCode	GENXLT source shipped	Program Name
IBM-924	IBM-1145	Yes	<prefix>EZHJ
IBM-924	IBM-1146	Yes	<prefix>EZHK
IBM-924	IBM-1147	Yes	<prefix>EZHM
IBM-924	IBM-1148	Yes	<prefix>EZHO
IBM-924	IBM-1149	Yes	<prefix>EZHR
IBM-924	IBM-4971	Yes	<prefix>EZHS
IBM-924	IBM-923	Yes	<prefix>EZIF
IBM-1140	IBM-858	Yes	<prefix>HAAI
IBM-1140	IBM-924	Yes	<prefix>HAEZ
IBM-1141	IBM-858	Yes	<prefix>HBAI
IBM-1141	IBM-924	Yes	<prefix>HBEZ
IBM-1142	IBM-858	Yes	<prefix>HEAI
IBM-1142	IBM-924	Yes	<prefix>HEEZ
IBM-1143	IBM-858	Yes	<prefix>HFAI
IBM-1143	IBM-924	Yes	<prefix>HFEZ
IBM-1144	IBM-858	Yes	<prefix>HGAI
IBM-1144	IBM-924	Yes	<prefix>HGEZ
IBM-1145	IBM-858	Yes	<prefix>HJAI
IBM-1145	IBM-924	Yes	<prefix>HJEZ
IBM-1146	IBM-858	Yes	<prefix>HKAI
IBM-1146	IBM-924	Yes	<prefix>HKEZ
IBM-1147	IBM-858	Yes	<prefix>HMAI
IBM-1147	IBM-924	Yes	<prefix>HMEZ
IBM-1148	IBM-858	Yes	<prefix>HOAI
IBM-1148	IBM-924	Yes	<prefix>HOEZ
IBM-1149	IBM-858	Yes	<prefix>HRAI
IBM-1149	IBM-924	Yes	<prefix>HREZ
IBM-4971	IBM-924	Yes	<prefix>HSEZ
IBM-4971	IBM-4909	Yes	<prefix>HSIA
IBM-4909	IBM-4971	Yes	<prefix>IAHS
IBM-923	IBM-924	Yes	<prefix>IFEZ
IBM-850	IBM-037	No	<prefix>AAEA
IBM-850	IBM-273	No	<prefix>AAEB
IBM-850	IBM-277	No	<prefix>AAEE
IBM-850	IBM-278	No	<prefix>AAEF
IBM-850	IBM-280	No	<prefix>AAEG
IBM-850	IBM-284	No	<prefix>AAEJ

Table 113. Coded character set conversion tables (continued)

FromCode	ToCode	GENXLT source shipped	Program Name
IBM-850	IBM-285	No	<prefix>AAEK
IBM-850	IBM-297	No	<prefix>AAEM
IBM-850	IBM-500	No	<prefix>AAEO
IBM-850	IBM-871	No	<prefix>AAER
IBM-850	IBM-1047	No	<prefix>AAEY
IBM-850	IBM-1140	No	<prefix>AAHA
IBM-850	IBM-1141	No	<prefix>AAHB
IBM-850	IBM-1142	No	<prefix>AAHE
IBM-850	IBM-1143	No	<prefix>AAHF
IBM-850	IBM-1144	No	<prefix>AAHG
IBM-850	IBM-1145	No	<prefix>AAHJ
IBM-850	IBM-1146	No	<prefix>AAHK
IBM-850	IBM-1147	No	<prefix>AAHM
IBM-850	IBM-1148	No	<prefix>AAHO
IBM-850	IBM-1149	No	<prefix>AAHR
IBM-932	IBM-290	No	<prefix>ABEL
IBM-932	IBM-1027	No	<prefix>ABEX
IBM-932C	IBM-290	No	<prefix>AGEL
IBM-932C	IBM-1027	No	<prefix>AGEX
IBM-1254	IBM-1026	No	<prefix>DIEW
IBM-037	IBM-850	No	<prefix>EAAA
IBM-037	IBM-500	Yes	<prefix>EAEO
IBM-037	IBM-1047	Yes	<prefix>EA EY
IBM-037	ISO8859-1	No	<prefix>EAI1
IBM-273	IBM-850	No	<prefix>EBAA
IBM-273	IBM-500	Yes	<prefix>EBEO
IBM-273	IBM-1047	Yes	<prefix>EBEY
IBM-273	ISO8859-1	No	<prefix>EBI1
IBM-274	IBM-500	Yes	<prefix>ECEO
IBM-274	IBM-1047	Yes	<prefix>ECEY
IBM-274	IBM-1148	Yes	<prefix>ECHO
IBM-274	IBM-819	No	<prefix>ECI1
IBM-275	IBM-500	Yes	<prefix>EDEO
IBM-275	IBM-1047	Yes	<prefix>EDEY
IBM-275	IBM-1148	Yes	<prefix>EDHO
IBM-275	IBM-819	No	<prefix>EDI1
IBM-277	IBM-850	No	<prefix>EEAA

Table 113. Coded character set conversion tables (continued)

FromCode	ToCode	GENXLT source shipped	Program Name
IBM-277	IBM-500	Yes	<prefix>EEEE
IBM-277	IBM-1047	Yes	<prefix>EEFY
IBM-277	ISO8859-1	No	<prefix>EEI1
IBM-278	IBM-850	No	<prefix>EFAA
IBM-278	IBM-500	Yes	<prefix>EFE0
IBM-278	IBM-924	Yes	<prefix>EFEY
IBM-278	ISO8859-1	No	<prefix>EFI1
IBM-280	IBM-850	No	<prefix>EGAA
IBM-280	IBM-500	Yes	<prefix>EGE0
IBM-280	IBM-1047	Yes	<prefix>EGEY
IBM-280	ISO8859-1	No	<prefix>EGI1
IBM-281	IBM-500	Yes	<prefix>EHE0
IBM-281	IBM-1047	Yes	<prefix>EHEY
IBM-281	IBM-1148	Yes	<prefix>EHHO
IBM-281	ISO8859-1	No	<prefix>EHI1
IBM-282	IBM-500	Yes	<prefix>EIE0
IBM-282	IBM-1047	Yes	<prefix>EIEY
IBM-282	IBM-1148	Yes	<prefix>EIHO
IBM-282	ISO8859-1	Yes	<prefix>EII1
IBM-284	IBM-850	No	<prefix>EJAA
IBM-284	IBM-500	Yes	<prefix>EJE0
IBM-284	IBM-1047	Yes	<prefix>EJEY
IBM-284	ISO8859-1	No	<prefix>EJI1
IBM-285	IBM-850	No	<prefix>EKAA
IBM-285	IBM-500	Yes	<prefix>EKE0
IBM-285	IBM-1047	Yes	<prefix>EKEY
IBM-285	ISO8859-1	No	<prefix>EKI1
IBM-290	IBM-932	No	<prefix>ELAB
IBM-290	IBM-932C	No	<prefix>ELAG
IBM-290	IBM-500	Yes	<prefix>ELE0
IBM-290	IBM-1027	Yes	<prefix>ELEX
IBM-290	IBM-1047	Yes	<prefix>ELEY
IBM-290	IBM-1148	Yes	<prefix>ELHO
IBM-290	IBM-819	No	<prefix>ELI1
IBM-297	IBM-850	No	<prefix>EMAA
IBM-297	IBM-500	Yes	<prefix>EME0
IBM-297	IBM-1047	Yes	<prefix>EMEY

Table 113. Coded character set conversion tables (continued)

FromCode	ToCode	GENXLT source shipped	Program Name
IBM-297	ISO8859-1	No	<prefix>EMI1
IBM-500	IBM-850	No	<prefix>EOAA
IBM-500	IBM-037	Yes	<prefix>EOEA
IBM-500	IBM-273	Yes	<prefix>EOEB
IBM-500	IBM-274	Yes	<prefix>EOEC
IBM-500	IBM-275	Yes	<prefix>EOED
IBM-500	IBM-277	Yes	<prefix>EOEE
IBM-500	IBM-278	Yes	<prefix>EOEF
IBM-500	IBM-280	Yes	<prefix>EOEG
IBM-500	IBM-281	Yes	<prefix>EOEH
IBM-500	IBM-282	Yes	<prefix>EOEI
IBM-500	IBM-284	Yes	<prefix>EOEJ
IBM-500	IBM-285	Yes	<prefix>EOEK
IBM-500	IBM-290	Yes	<prefix>EOEL
IBM-500	IBM-297	Yes	<prefix>EOEM
IBM-500	IBM-871	Yes	<prefix>EOER
IBM-500	IBM-1027	Yes	<prefix>EOEX
IBM-500	IBM-1047	Yes	<prefix>EOEY
IBM-500	IBM-1140	Yes	<prefix>EOHA
IBM-500	IBM-1141	Yes	<prefix>EOHB
IBM-500	IBM-1142	Yes	<prefix>EOHE
IBM-500	IBM-1143	Yes	<prefix>EOHF
IBM-500	IBM-1144	Yes	<prefix>EOHG
IBM-500	IBM-1145	Yes	<prefix>EOHJ
IBM-500	IBM-1146	Yes	<prefix>EOHK
IBM-500	IBM-1147	Yes	<prefix>EOHM
IBM-500	IBM-1149	Yes	<prefix>EOHR
IBM-500	ISO8859-1	No	<prefix>EOI1
IBM-871	IBM-850	No	<prefix>ERAA
IBM-871	IBM-500	Yes	<prefix>EREO
IBM-871	IBM-1047	Yes	<prefix>EREY
IBM-871	IBM-924	No	<prefix>ERI1
IBM-871	ISO8859-1	Yes	<prefix>ESEY
IBM-875	ISO8859-7	No	<prefix>ESI7
IBM-930	IBM-1047	Yes	<prefix>EUEY
IBM-939	IBM-1047	Yes	<prefix>EVEY
IBM-1026	IBM-1254	No	<prefix>EWDI

Table 113. Coded character set conversion tables (continued)

FromCode	ToCode	GENXLT source shipped	Program Name
IBM-1026	IBM-1047	Yes	<prefix>EWEY
IBM-1026	ISO8859-9	No	<prefix>EWI9
IBM-1027	IBM-932	No	<prefix>EXAB
IBM-1027	IBM-932C	No	<prefix>EXAG
IBM-1027	IBM-290	Yes	<prefix>EXEL
IBM-1027	IBM-500	Yes	<prefix>EXEO
IBM-1027	IBM-1047	Yes	<prefix>EXEY
IBM-1027	IBM-1148	Yes	<prefix>EXHO
IBM-1027	ISO8859-1	No	<prefix>EXI1
IBM-1047	IBM-850	No	<prefix>EYAA
IBM-1047	IBM-037	Yes	<prefix>EYEA
IBM-1047	IBM-273	Yes	<prefix>EYEB
IBM-1047	IBM-274	Yes	<prefix>EYEC
IBM-1047	IBM-275	Yes	<prefix>EYED
IBM-1047	IBM-277	Yes	<prefix>EYEE
IBM-1047	IBM-278	Yes	<prefix>EYEF
IBM-1047	IBM-280	Yes	<prefix>EYEG
IBM-1047	IBM-281	Yes	<prefix>EYEH
IBM-1047	IBM-282	Yes	<prefix>EYEI
IBM-1047	IBM-284	Yes	<prefix>EYEJ
IBM-1047	IBM-285	Yes	<prefix>EYEK
IBM-1047	IBM-290	Yes	<prefix>EYEL
IBM-1047	IBM-297	Yes	<prefix>EYEM
IBM-1047	IBM-500	Yes	<prefix>EYEO
IBM-1047	IBM-871	Yes	<prefix>EYER
IBM-1047	IBM-875	Yes	<prefix>EYES
IBM-1047	IBM-930	Yes	<prefix>EYEU
IBM-1047	IBM-939	Yes	<prefix>EYEV
IBM-1047	IBM-1026	Yes	<prefix>EYEW
IBM-1047	IBM-1027	Yes	<prefix>EYEX
IBM-1047	IBM-836	Yes	<prefix>EYGL
IBM-1047	IBM-833	Yes	<prefix>EYGP
IBM-1047	IBM-937	Yes	<prefix>EYGW
IBM-1047	IBM-935	Yes	<prefix>EYGY
IBM-1047	IBM-933	Yes	<prefix>EYGZ
IBM-1047	IBM-1140	Yes	<prefix>EYHA
IBM-1047	IBM-1141	Yes	<prefix>EYHB

Table 113. Coded character set conversion tables (continued)

FromCode	ToCode	GENXLT source shipped	Program Name
IBM-1047	IBM-1142	Yes	<prefix>EYHE
IBM-1047	IBM-1143	Yes	<prefix>EYHF
IBM-1047	IBM-1144	Yes	<prefix>EYHG
IBM-1047	IBM-1145	Yes	<prefix>EYHJ
IBM-1047	IBM-1146	Yes	<prefix>EYHK
IBM-1047	IBM-1147	Yes	<prefix>EYHM
IBM-1047	IBM-1148	Yes	<prefix>EYHO
IBM-1047	IBM-1149	Yes	<prefix>EYHR
IBM-1047	ISO8859-1	No	<prefix>EYI1
IBM-836	IBM-1047	Yes	<prefix>GLEY
IBM-833	IBM-1047	Yes	<prefix>GPEY
IBM-937	IBM-1047	Yes	<prefix>GWEY
IBM-935	IBM-1047	Yes	<prefix>GYEY
IBM-933	IBM-1047	Yes	<prefix>GZEY
IBM-933 (IBM-833 SBCS subset only)	IBM-819	No	<prefix>GZI1
IBM-1140	IBM-850	No	<prefix>HAAA
IBM-1140	IBM-500	Yes	<prefix>HAEO
IBM-1140	IBM-1047	Yes	<prefix>HAEY
IBM-1140	IBM-1148	Yes	<prefix>HAHO
IBM-1140	ISO8859-1	No	<prefix>HAI1
IBM-1141	IBM-850	No	<prefix>HBAA
IBM-1141	IBM-500	Yes	<prefix>HBEO
IBM-1141	IBM-1047	Yes	<prefix>HBEY
IBM-1141	IBM-1148	Yes	<prefix>HBHO
IBM-1141	ISO8859-1	No	<prefix>HBI1
IBM-1142	IBM-850	No	<prefix>HEAA
IBM-1142	IBM-500	Yes	<prefix>HEEO
IBM-1142	IBM-1047	Yes	<prefix>HEEY
IBM-1142	IBM-1148	Yes	<prefix>HEHO
IBM-1142	ISO8859-1	No	<prefix>HEI1
IBM-1143	IBM-850	No	<prefix>HFAA
IBM-1143	IBM-500	Yes	<prefix>HFEO
IBM-1143	IBM-1047	Yes	<prefix>HFEY
IBM-1143	IBM-1148	Yes	<prefix>HFHO
IBM-1143	ISO8859-1	No	<prefix>HFI1
IBM-1144	IBM-850	No	<prefix>HGAA

Table 113. Coded character set conversion tables (continued)

FromCode	ToCode	GENXLT source shipped	Program Name
IBM-1144	IBM-500	Yes	<prefix>HGE0
IBM-1144	IBM-1047	Yes	<prefix>HGEY
IBM-1144	IBM-1148	Yes	<prefix>HGHO
IBM-1144	ISO8859-1	No	<prefix>HGI1
IBM-1145	IBM-850	No	<prefix>HJAA
IBM-1145	IBM-500	Yes	<prefix>HJEO
IBM-1145	IBM-1047	Yes	<prefix>HJEY
IBM-1145	IBM-1148	Yes	<prefix>HJHO
IBM-1145	ISO8859-1	No	<prefix>HJI1
IBM-1146	IBM-850	No	<prefix>HKAA
IBM-1146	IBM-500	Yes	<prefix>HKE0
IBM-1146	IBM-1047	Yes	<prefix>HKEY
IBM-1146	IBM-1148	Yes	<prefix>HKHO
IBM-1146	ISO8859-1	No	<prefix>HKI1
IBM-1147	IBM-850	No	<prefix>HMAA
IBM-1147	IBM-500	Yes	<prefix>HME0
IBM-1147	IBM-1047	Yes	<prefix>HMEY
IBM-1147	IBM-1148	Yes	<prefix>HMHO
IBM-1147	ISO8859-1	No	<prefix>HMI1
IBM-1148	IBM-850	No	<prefix>HOAA
IBM-1148	IBM-274	Yes	<prefix>HOEC
IBM-1148	IBM-275	Yes	<prefix>HOED
IBM-1148	IBM-281	Yes	<prefix>HOEH
IBM-1148	IBM-282	Yes	<prefix>HOEI
IBM-1148	IBM-290	Yes	<prefix>HOEL
IBM-1148	IBM-1027	Yes	<prefix>HOEX
IBM-1148	IBM-1047	Yes	<prefix>HOEY
IBM-1148	IBM-1140	Yes	<prefix>HOHA
IBM-1148	IBM-1141	Yes	<prefix>HOHB
IBM-1148	IBM-1142	Yes	<prefix>HOHE
IBM-1148	IBM-1143	Yes	<prefix>HOHF
IBM-1148	IBM-1144	Yes	<prefix>HOHG
IBM-1148	IBM-1145	Yes	<prefix>HOHJ
IBM-1148	IBM-1146	Yes	<prefix>HOHK
IBM-1148	IBM-1147	Yes	<prefix>HOHM
IBM-1148	IBM-1149	Yes	<prefix>HOHR
IBM-1148	ISO8859-1	No	<prefix>HOI1

Table 113. Coded character set conversion tables (continued)

FromCode	ToCode	GENXLT source shipped	Program Name
IBM-1149	IBM-850	No	<prefix>HRAA
IBM-1149	IBM-500	Yes	<prefix>HREO
IBM-1149	IBM-1047	Yes	<prefix>HREY
IBM-1149	IBM-1148	Yes	<prefix>HRHO
IBM-1149	ISO8859-1	No	<prefix>HRI1
ISO8859-1	IBM-037	No	<prefix>I1EA
ISO8859-1	IBM-273	No	<prefix>I1EB
ISO8859-1	IBM-274	No	<prefix>I1EC
ISO8859-1	IBM-275	No	<prefix>I1ED
ISO8859-1	IBM-277	No	<prefix>I1EE
ISO8859-1	IBM-278	No	<prefix>I1EF
ISO8859-1	IBM-280	No	<prefix>I1EG
ISO8859-1	IBM-281	No	<prefix>I1EH
ISO8859-1	IBM-282	No	<prefix>I1EI
ISO8859-1	IBM-284	No	<prefix>I1EJ
ISO8859-1	IBM-285	No	<prefix>I1EK
IBM-819	IBM-290	No	<prefix>I1EL
ISO8859-1	IBM-297	No	<prefix>I1EM
ISO8859-1	IBM-500	No	<prefix>I1EO
ISO8859-1	IBM-871	No	<prefix>I1ER
ISO8859-1	IBM-1027	No	<prefix>I1EX
ISO8859-1	IBM-1047	No	<prefix>I1EY
IBM-819	IBM-933	No	<prefix>I1GZ
ISO8859-1	IBM-1140	No	<prefix>I1HA
ISO8859-1	IBM-1140	No	<prefix>I1HB
ISO8859-1	IBM-1142	No	<prefix>I1HE
ISO8859-1	IBM-1143	No	<prefix>I1HF
ISO8859-1	IBM-1144	No	<prefix>I1HG
ISO8859-1	IBM-1145	No	<prefix>I1HJ
ISO8859-1	IBM-1146	No	<prefix>I1HK
ISO8859-1	IBM-1147	No	<prefix>I1HM
ISO8859-1	IBM-1148	No	<prefix>I1HO
ISO8859-1	IBM-1149	No	<prefix>I1HR
ISO8859-7	IBM-875	No	<prefix>I7ES
ISO8859-9	IBM-1026	No	<prefix>I9EW
IBM-943G	UCS-2	No	<prefix>ANU2
IBM-943G	UTF-8	No	<prefix>ANF8

Table 113. Coded character set conversion tables (continued)

FromCode	ToCode	GENXLT source shipped	Program Name
UCS-2	IBM-943G	No	<prefix>U2AN
UTF-8	IBM-943G	No	<prefix>F8AN

Universal coded character set converters

You can use the name UCS-2 to request setup for conversion to and from the Universal Two-Octet Coded Character Set, UCS-2, specified in ISO/IEC International Standard 10646-1. For example, `iconv_open("UCS-2", "IBM-1047")` requests setup for conversion from IBM-1047 character encoding to UCS-2 character encoding.

You can also use the name UTF-8 to request setup for conversion to and from Transform Format 8, UTF-8, specified in Unicode Standard, Version 2.1, Appendices A-7 and A-8. For example, `iconv_open("UTF-8", "IBM-1047")` requests setup for conversion from IBM-1047 character encoding to UTF-8 character encoding.

Before z/OS V1R12, source for UCS-2 converters was in a data set named `installation-prefix.SCEEUMAP`, where the installation prefix for z/OS XL C/C++ data sets defaults to CEE. UCS-2 source was also installed in the z/OS UNIX file system directory `/usr/lib/nls/locale/ucmap`. Starting in z/OS V1R12, IBM will no longer ship UCS-2 source with the C/C++ Runtime Library.

The `uconvdef` command, which is documented in *z/OS UNIX System Services Command Reference*, produces `uconvTable` binary files required by `iconv_open()` from UCS-2 source files.

The following notes apply only when you create your own converters:

- The `iconv()` family of functions uses Unicode Services to perform character conversion to or from UCS-2. Therefore, IBM no longer ships `uconvTable` binaries in either the `installation-prefix.SCEEUTBL` data set or the z/OS UNIX file system directory `/usr/lib/nls/locale/uconvTable`. Both the `installation-prefix.SCEEUTBL` data set and the `/usr/lib/nls/locale/uconvTable` directory have been removed. Users who create their own `uconvTable` converters need to create the `installation-prefix.SCEEUTBL` data set, the `/usr/lib/nls/locale/uconvTable` directory, or both to hold the converters. The `installation-prefix.SCEEUTBL` data set needs to be created with a fixed block record format and `lrecl` of 80.
- If your installation uses an installation-prefix different from CEE for z/OS XL C/C++ data sets, you must use the environment variable `_ICONV_UCS2_PREFIX` to specify the value of your installation-prefix before using `iconv_open()` to set up UCS-2 converters. Otherwise, `iconv_open()` cannot find your z/OS XL C/C++ `uconvTable` binary data set. One way to do this is to use the `ENVAR` runtime option when you start your application. For example, `ENVAR(..., _ICONV_UCS2_PREFIX=YOUR.PREFIX, ...)` has `iconv_open()` search for `uconvTable` binaries it requires in the data set `YOUR.PREFIX.SCEEUTBL`.
- If `uconvTable` binaries are installed in both the z/OS UNIX file system directory named `/usr/lib/nls/locale/uconvTable` and `installation-prefix.SCEEUTBL` data set, the `iconv_open()` function searches for `uconvTable` binaries in the z/OS UNIX file system before searching in the z/OS XL C/C++ UCS-2 data set.
- You can use the `LOCPATH` environment variable to give `iconv_open()` a colon-separated list of pathname prefixes to use instead of `/usr/lib/nls/locale/` to find `uconvTable` directories in your z/OS UNIX file system.
- If you have created your own conversion tables and want the `iconv()` family of functions to use them, you need to set the `_ICONV_MODE` environment variable to C.
- If you want to create customized conversion tables with any of the CCSIDs related to the conversion table source that is no longer shipped, create customized Unicode Services conversion tables instead of customized LE conversion tables. For information about how to generate and use customized Unicode Services conversion tables, see *z/OS Unicode Services User's Guide and Reference*.

You must set the `_ICONV_TECHNIQUE` environment variable to the same technique search order value used for the customized Unicode Services table in order for the `iconv()` family of functions to use

the customized Unicode Services table. For example, if you want the `iconv()` family of functions to use a user defined Unicode Services table with a technique search order of 2, you need to set the `_ICONV_TECHNIQUE` environment variable to 2LMREC.

- The `uconvdef` and `genxlt` utilities continue to be shipped and maintained in z/OS V1R12. If you do not want to use Unicode Services to create customized conversion tables, you can continue to use the `uconvdef` and `genxlt` utilities. However, you must obtain the `ucmap` or `genxlt` source in one of the following ways:
 - Migrate the `ucmap` or `genxlt` source forward from a previous z/OS release.
 - Create the `ucmap` or `genxlt` source on your own.
 - Get the `ucmap` or `genxlt` source from the Character Data Representation Architecture.

The UCS-2 and `genxlt` conversion table binaries produced by the `uconvdef` and `genxlt` utilities continue to be supported by the `iconv()` family of functions if you set the `_ICONV_MODE` environment variable to C. You must obtain the conversion table binaries in one of the following ways:

- Migrate the conversion table binaries from a previous z/OS release.
- Create the conversion table binaries by your own.
- Create the conversion table binaries by the `uconvdef` or `genxlt` utilities.

Note: IBM makes no guarantee that the `uconvdef` or `genxlt` utilities will continue to be supported in future releases.

Members in the UCS-2 source data sets have names of the form `EDCUUccU`, where `cc` is the CC-id associated with a particular coded character set name. Table 114 on page 721 shows the CC-id and member name associated with each coded character set name for which UCS-2 source is provided. The UCS-2 source is in a data set named `installation-prefix.SCEEUMAP`. The default value of the `installation-prefix` is CEE.



CAUTION: UCS-2 converter binaries added to the `SCEEUTBL` data set or the `uconvTable` directory must follow the `EDCUUccU` naming convention that allows `iconv` interfaces to recognize a UCS-2 converter.

All UCS-2 tables are intended for the use with the C/C++ `iconv` interfaces or the `iconv` utility. Direct programming to these tables is not supported and will produce unpredictable results.

Starting in z/OS V1R12, IBM will not ship conversion table source in either the `installation-prefix.SCEEUMAP` data set or the `/usr/lib/nls/locale/ucmap` directory. However, if you choose to migrate the conversion tables to your own `installation-prefix.SCEEUTBL` data set, you need to continue to name them as listed in Table 114 on page 721.

Table 114. UCS-2 converter names		
Codeset Name	CC-id	Table name in a user-created SCEEUTBL data set
IBM-850	AA	EDCUUAAU
IBM-4946	AA	EDCUUAAU
IBM-301	AB	EDCUUABU
IBM-932	AB	EDCUUABU
IBM-942	AB	EDCUUABU
IBM-943G	AN	EDCUUANU
EUCJP	AC	EDCUUACU
IBM-EUCJP	AC	EDCUUACU
IBM33722	AC	EDCUUACU
IBM-922	AD	EDCUUADU
IBM-1046	AF	EDCUUAFU

Table 114. UCS-2 converter names (continued)

Codeset Name	CC-id	Table name in a user-created SCEEUTBL data set
IBM-932C	AG	EDCUUAGU
IBM-EUCJC	AH	EDCUUAHU
IBM-858	AI	EDCUUAIU
IBM-943	AJ	EDCUUAJU
IBM-859	AK	EDCUUAKU
IBM-425	AR	EDCUUARU
IBM-848	AS	EDCUUASU
IBM-1125	AT	EDCUUATU
IBM-1124	AU	EDCUUAUU
IBM-437	AV	EDCUUAVU
IBM-921	BD	EDCUUBDU
IBM-866	BE	EDCUUBEU
IBM-862	BH	EDCUUBHU
GBK	BS	EDCUUBSU
IBM-874	BU	EDCUUBUU
TIS-620	BU	EDCUUBUU
EUCTW-1993	BW	EDCUUBWU
IBM-EUCTW	BW	EDCUUBWU
IBM-964	BW	EDCUUBWU
IBM-1383	BY	EDCUUBYU
EUCKR	BZ	EDCUUBZU
IBM-EUCKR	BZ	EDCUUBZU
IBM-970	BZ	EDCUUBZU
IBM-861	CA	EDCUUCAU
IBM-852	CB	EDCUUCBU
IBM-855	CE	EDCUUCEU
IBM-864	CF	EDCUUCFU
IBM-869	CG	EDCUUCGU
IBM-856	CH	EDCUUCHU
IBM-1115	CL	EDCUUCLU
IBM-1380	CM	EDCUUCMU
IBM-904	CN	EDCUUCNU
IBM-927	CO	EDCUUCOU
IBM-1088	CP	EDCUUCPU
IBM-951	CQ	EDCUUCQU
IBM-942	CR	EDCUUCRU
IBM-1386	CV	EDCUUCVU

Table 114. UCS-2 converter names (continued)

Codeset Name	CC-id	Table name in a user-created SCEEUTBL data set
IBM-938	CW	EDCUUCWU
IBM-948	CW	EDCUUCWU
IBM-1381	CY	EDCUUCYU
IBM-949	CZ	EDCUUCZU
IBM-1252	DA	EDCUUDAU
IBM-1250	DB	EDCUUDBU
IBM-1251	DE	EDCUUDEU
IBM-1256	DF	EDCUUDFU
IBM-1253	DG	EDCUUDGU
IBM-1255	DH	EDCUUDHU
IBM-1254	DI	EDCUUDIU
IBM-5348	DJ	EDCUUDJU
IBM-5349	DK	EDCUUDKU
BIG5	DW	EDCUUDWU
IBM-947	DW	EDCUUDWU
IBM-950	DW	EDCUUDWU
IBM-928	DY	EDCUUDYU
IBM-936	DY	EDCUUDYU
IBM-946	DY	EDCUUDYU
IBM-037	EA	EDCUUEAU
IBM-28709	EA	EDCUUEAU
IBM-273	EB	EDCUUEBU
IBM-274	EC	EDCUUECU
IBM-275	ED	EDCUUEDU
IBM-277	EE	EDCUUEEU
IBM-278	EF	EDCUUEFU
IBM-280	EG	EDCUUEGU
IBM-281	EH	EDCUUEHU
IBM-282	EI	EDCUUEIU
IBM-284	EJ	EDCUUEJU
IBM-285	EK	EDCUUEKU
IBM-290	EL	EDCUUELU
IBM-297	EM	EDCUUEMU
IBM-300	EN	EDCUUENU
IBM-4396	EN	EDCUUENU
IBM-500	EO	EDCUUEOU
IBM-838	EP	EDCUUEPU

Table 114. UCS-2 converter names (continued)

Codeset Name	CC-id	Table name in a user-created SCEEUTBL data set
IBM-870	EQ	EDCUUEQU
IBM-871	ER	EDCUUERU
IBM-875	ES	EDCUUESU
IBM-880	ET	EDCUUETU
IBM-930	EU	EDCUUEUU
IBM-5026	EU	EDCUUEUU
IBM-939	EV	EDCUUEVU
IBM-5035	EV	EDCUUEVU
IBM-1026	EW	EDCUUEWU
IBM-1027	EX	EDCUUEXU
IBM-1047	EY	EDCUUEYU
IBM-924	EZ	EDCUUEZU
UTF-8	F8	EDCUUF8U
IBM-424	FB	EDCUUFBU
IBM-1122	FD	EDCUUFDU
IBM-1025	FE	EDCUUFEU
IBM-420	FF	EDCUUFFU
IBM-1165	FG	EDCUUFGU
IBM-1123	FH	EDCUUFHU
IBM-1158	FI	EDCUUFIU
IBM-4933	FJ	EDCUUFJU
IBM13124	Fk	EDCUUFKU
IBM-53668	FV	EDCUUFVU
IBM-1112	GD	EDCUUGDU
IBM-836	GL	EDCUUGLU
IBM-837	GM	EDCUUGMU
IBM-835	GO	EDCUUGOU
IBM-833	GP	EDCUUGPU
IBM-834	GQ	EDCUUGQU
IBM-1388	GV	EDCUUGVU
IBM-937	GW	EDCUUGWU
IBM-935	GY	EDCUUGYU
IBM-5031	GY	EDCUUGYU
IBM-933	GZ	EDCUUGZU
IBM-1140	HA	EDCUUHAU
IBM-1141	HB	EDCUUHBU
IBM16804	HC	EDCUUHCU

Table 114. UCS-2 converter names (continued)		
Codeset Name	CC-id	Table name in a user-created SCEEUTBL data set
IBM-1157	HD	EDCUUHDU
IBM-1142	HE	EDCUUHEU
IBM-1143	HF	EDCUUHFU
IBM-1144	HG	EDCUUHGU
IBM12712	HH	EDCUUHHU
IBM-1145	HJ	EDCUUHJU
IBM-1146	HK	EDCUUHKU
IBM-1147	HM	EDCUUHMU
IBM-16684	HN	EDCUUHNU
IBM-1148	HO	EDCUUHOU
IBM-1160	HP	EDCUUHPU
IBM-1149	HR	EDCUUHRU
IBM-4971	HS	EDCUUHSU
IBM-1154	HT	EDCUUHTU
IBM-1390	HU	EDCUUHUU
IBM-1399	HV	EDCUUHVU
IBM-1155	HW	EDCUUHWU
IBM-5123	HX	EDCUUHXU
IBM-1156	HZ	EDCUUHZU
ISO8859-1	I1	EDCUUI1U
IBM-819	I1	EDCUUI1U
ISO8859-2	I2	EDCUUI2U
IBM-912	I2	EDCUUI2U
ISO8859-4	I4	EDCUUI4U
IBM-914	I4	EDCUUI4U
ISO8859-5	I5	EDCUUI5U
IBM-915	I5	EDCUUI5U
ISO8859-6	I6	EDCUUI6U
IBM-1089	I6	EDCUUI6U
ISO8859-7	I7	EDCUUI7U
IBM-813	I7	EDCUUI7U
ISO8859-8	I8	EDCUUI8U
IBM-916	I8	EDCUUI8U
ISO8859-9	I9	EDCUUI9U
IBM-920	I9	EDCUUI9U
IBM-4909	IA®	EDCUUIAU
IBM-923	IF	EDCUUIFU

Table 114. UCS-2 converter names (continued)

Codeset Name	CC-id	Table name in a user-created SCEEUTBL data set
ISO8859-15	IF	EDCUUIFU
ISO-2022-JP	JA	EDCUUJAU
IBM-956	JB	EDCUUJBU
IBM-957	JC	EDCUUJCU
IBM-956C	JD	EDCUUJDU
IBM-958	JD	EDCUUJDU
IBM-957C	JE	EDCUUJEU
IBM-959	JE	EDCUUJEU
IBM-5052	JF	EDCUUJFU
IBM-5053	JG	EDCUUJGU
IBM-5052C	JH	EDCUUJHU
IBM-5054	JH	EDCUUJHU
IBM-5053C	JI	EDCUUJIU
IBM-5055	JI	EDCUUJIU
IBM-1371	KA	EDCUUKAU
IBM-1364	KZ	EDCUUKZU
IBM-1370	LA	EDCUULAU
IBM-902	LD	EDCUULDU
IBM-872	LE	EDCUULEU
IBM-808	LF	EDCUULFU
IBM-9061	LG	EDCUULGU
IBM-901	LH	EDCUULHU
IBM-9238	LI	EDCUULIU
IBM-867	LJ	EDCUULJU
IBM-1161	LU	EDCUULUU
IBM-1363	LZ	EDCUULZU
IBM-1153	MB	EDCUUMBU
IBM-5346	NB	EDCUUNBU
IBM-5347	NE	EDCUUNEU
IBM-5352	NF	EDCUUNFU
IBM-9044	NG	EDCUUNGU
IBM-5351	NH	EDCUUNHU
IBM-5350	NI	EDCUUNIU
IBM17248	NJ	EDCUUNJU
UCS-2	U2	EDCUUU2U

Codeset conversion using UCS-2

z/OS XL C/C++ iconv supports use of UCS-2 as an intermediate code set for conversion of characters encoded in one code set to another. The `_ICONV_UCS2` environment variable instructs `iconv_open("Y", "X")` whether or not to set up indirect conversion from code set X to code set Y using UCS-2 as an intermediate code set. Values `iconv_open()` recognizes for `_ICONV_UCS2` are:

- 1**
Set up indirect conversion using UCS-2 first. The indirect conversions will use direct unicode converters if available, if not, `iconv_open()` will fopen/fread `uconvTable` binaries. If set up of indirect conversion fails, `iconv_open()` will try to set up direct conversion.
- 2**
Set up direct conversion first. If this fails, try to set up indirect conversion using UCS-2. The indirect conversions will use direct unicode converters if available, if not, `iconv_open()` will fopen/fread `uconvTable` binaries. This is the default.
- 3**
Set up direct conversion first. If this fails, try to set up indirect conversion using UCS-2. The indirect conversions will use direct unicode converters, if direct unicode converters are unavailable, the `iconv_open()` request fails.
- N**
Never set up indirect conversion using UCS-2. If a direct converter cannot be found, the `iconv_open()` request fails.
- D**
Never set up indirect conversion using UCS-2. If a direct converter cannot be found, the `iconv_open()` request fails.
- O**
Only set up indirect conversion using UCS-2. `iconv_open()` will fopen/fread `uconvTable` binaries. Direct unicode converters will **not** be used. If required `uconvTable` binaries cannot be found, the `iconv_open()` request fails..
- U**
Only set up indirect conversion using UCS-2. The indirect conversions will use direct unicode converters if available, if not, `iconv_open()` will fopen/fread `uconvTable` binaries.

Notes:

1. `_ICONV_UCS2` environment variable only has effect when `ICONV_MODE` is set to C.
2. If the value of the `_ICONV_UCS2` environment variable allows `iconv_open("Y", "X")` to use UCS-2 as an intermediate code set when it cannot find a direct converter from X to Y, `iconv_open()` will attempt to do so even if X and Y are not compatible code sets. That is, even if character sets encoded by X and Y are not the same, `iconv_open()` will set up conversion from X to UCS-2 to Y.
3. The application must specify compatible source and target code set names on various `iconv_open()` requests. For detailed information about code set characteristics, refer to the specific coded character set identifier in the CCSID repository.

UCMAP source format

A UCMAP source file defines UCS-2 (Unicode) conversion mappings for input to the `uconvdef` command. Conversion mapping values are defined using UCS-2 symbolic character names followed by character encoding (code point) values for the multibyte code set. For example:

<U0020>

`\x20` represents the mapping between the `<U0020>` UCS-2 symbolic character name for the space character and the `\x20` hexadecimal code point for the space character in ASCII.

In addition to the code set mappings, directives are interpreted by the `uconvdef` command to produce the compiled table. These directives must precede the code set mapping section. They consist of the following keywords surrounded by `<>` (angle brackets), starting in column 1, followed by white space and the value to be assigned to the symbol:

<comment_char>

Character used to denote start of escape sequence. Default escape character is <number_sign> (#). In ucmmap, source shipped by C/370 <percent_sign> (%) is specified for <comment_char>.

<escape_char>

Character used to denote start of escape sequence. Default escape character is <backslash> (\). In ucmmap source shipped by C/370 <slash> (/) is specified for <escape_char>.

<code_set_name>

The name of the coded character set, enclosed in quotation marks(""), for which the character set description file is defined.

<mb_cur_max>

The maximum number of bytes in a multibyte character. The default value is 1.

<mb_cur_min>

An unsigned positive integer value that defines the minimum number of bytes in a character for the encoded character set. The value is less than or equal to <mb_cur_max>. If not specified, the minimum number is equal to <mb_cur_max>.

<char_name_mask>

A quoted string consisting of format specifiers for the UCS-2 symbolic names. This must be a value of AXXXX, indicating an alphabetic character followed by 4 hexadecimal digits. Also, the alphabetic character must be a U, and the hexadecimal digits must represent the UCS-2 code point for the character. An example of a symbolic character name based on this mask is <U0020> Unicode space character.

<uconv_class>

Specifies the type of the code set. It must be one of the following:

SBCS

Single-byte encoding

DBCS

Stateless double-byte, single-byte, or mixed encodings

EBCDIC_STATEFUL

Stateful double-byte, single-byte, or mixed encodings

MBCS

Stateless multibyte encoding

This type is used to direct uconvdef on the type of table to build. It is also stored in the table to indicate the type of processing algorithm in the UCS conversion methods.

<locale>

Specifies the default locale name to be used if locale information is needed.

<subchar>

Specifies the encoding of the default substitute character in the multibyte code set.

The mapping definition section consists of a sequence of mapping definition lines preceded by a CHARMAP declaration and terminated by an END CHARMAP declaration. Empty lines and lines containing <comment_char> in the first column are ignored.

Symbolic character names in mapping lines must follow the pattern specified in the <char_name_mask>, except for the reserved symbolic name, <unassigned>, that indicates the associated code points are unassigned.

Each noncomment line of the character set mapping definition must be in one of the following formats:

1. This format defines a single symbolic character name and a corresponding encoding.

```
"%s%s%s/n", <symbolic_name>, <encoding>, <comments>
```

For example: <U3004> \x81\x57

The encoding part is expressed as one or more concatenated decimal, hexadecimal, or octal constants in the following formats:

- "%cd%d", <escape_char>, <decimal byte value>
- "%cx%x", <escape_char>, <hexadecimal byte value>
- "%co", <escape_char>, <octal byte value>

Decimal constants are represented by two or more decimal digits preceded by the escape character and the lowercase letter d, as in \d97 or \d143. Hexadecimal constants are represented by two or more hexadecimal digits preceded by an escape character and the lowercase letter x, as in \x61 or \x8f. Octal constants are represented by two or more octal digits preceded by an escape character.

Each constant represents a single—byte value. When constants are concatenated for multibyte character values, the last value specifies the least significant octet and preceding constants specify successively more significant octets.

2. This format defines a range of symbolic character names and corresponding encodings. The range is interpreted as a series of symbolic names formed from the alphabetic prefix and all the values in the range defined by the numeric suffixes.

```
"%s...%s %s %s/n", <symbolic-name>, <symbolic_name>, <encoding> <comments>
```

For example: <U3003><U3006> \x81\x56

The listed encoding value is assigned to the first symbolic name, and subsequent symbolic names in the range are assigned corresponding incremental values. For example, the line:

```
<U3003>...<U3006> \x81\x56
```

is interpreted as:

```
<U3003> \x81\x56
<U3004> \x81\x57
<U3005> \x81\x58
<U3006> \x81\x59
```

3. This format defines a range of one or more unassigned encodings.

```
"<unassigned>%s...%s %s/n", <encoding>, <comments>
```

For example, the line

```
<unassigned> \x9b...\x9c
```

is interpreted as:

```
<unassigned> \x9b <unassigned> \x9c
```


Chapter 59. Coded character set considerations with locale functions

Each EBCDIC *coded character set* consists of a mapping of all the available glyphs to their respective hex encodings and unique Graphic Character Global Identifiers (GCGIDs). GCGIDs are unique identifiers assigned to each character in the Unicode standard. A *glyph* is the printed appearance of a character. Each coded character set serves one linguistic environment.

There is wide variation among coded character sets; many glyphs do not appear in all coded character sets, and hexadecimal encodings for some glyphs differ from one coded character set to another. You may encounter problems when exporting a file from a system running in one coded character set, to a system running in another. For example, a left bracket ([) entered under the APL-293 or Open Systems IBM-1047 coded character set will appear as the capitalized Y-acute (Y). This occurs in such common coded character sets as International 500, France 297, Germany 273, and US or Canada 037.

z/OS XL C/C++ contains the following extensions to prevent such problems:

- The `#pragma filetag` directive allows you to specify the coded character set that was used when entering the source files. See “The pragma filetag directive” on page 737 for details on this pragma.
- The `LOCALE` compiler option enables you to tell the compiler what locale to use at compile time. See “Converting coded character sets at compile time” on page 739 for details on this compiler option.
- The `CONVLIT` compiler option enables you to change the assumed code page for string literals. See “CONVLIT compiler option” on page 739 for details on this compiler option.
- The `#pragma convert` directive allows you to change the assumed code page for string literals. It has the advantage of allowing more than one character encoding to be used for string literals in a single compilation unit. For more information, see `convert` in *z/OS XL C/C++ Language Reference*.

These facilities cause the compiler to respect your code page. Thus, you can enter source code with what appears to you to be the correct characters, and the compiler will recognize those characters.

The rest of this chapter discusses other ways to work efficiently in different locales.

Variant character detail

The POSIX Portable Character Set (PPCS) identifies the core set of 128 characters that are needed to write code and to run applications. Of these, 13 characters are variant among the EBCDIC coded character sets.

Table 115 on page 731 lists these 13 characters. It also displays their appearance when the Open Systems coded character set IBM-1047 hexadecimal values are entered on systems where different Country Extended Coded Character Sets are installed. These hex values are the ones expected by z/OS XL C/C++, and are consistent with the use of the APL-293 coded character set.

Table 115. Mappings of 13 PPCS variant characters

Character	Open Systems Hex Value (Default)	Open Systems IBM-1047 view	APL IBM-293 view	Inter-national IBM-500 view	France IBM-297 view	Germany IBM-273 view	US/Can IBM-037 view
left bracket	AD	[[Y	Y	Y	Y
right bracket	BD]]	ü	~	ü	..
left brace	C0	{	{	{	é	ä	{
right brace	D0	}	}	}	è	ü	}
backslash	E0	\	\	\	ç	Ö	\

Table 115. Mappings of 13 PPCS variant characters (continued)

Character	Open Systems Hex Value (Default)	Open Systems IBM-1047 view	APL IBM-293 view	Inter-national IBM-500 view	France IBM-297 view	Germany IBM-273 view	US/Can IBM-037 view
circumflex	5F	^	¬	^	^	^	¬
tilde	A1	~	~	~	ü .	ß	~
exclamation mark	5A	!	!]	\$	Ü	!
pound (number) sign	7B	#	#	#	£	#	#
vertical bar	4F			!	!	!	
accent grave	79	`	`	`	µ	`	`
dollar sign	5B	\$	\$	\$	\$	\$	\$
commercial "at"	7C	@	@	@	á	\$	@

Table 116 on page 732 lists the hexadecimal values assigned across some of the EBCDIC coded character sets for the 13 variant characters from the PPCS.

Table 116. Mappings of Hex encoding of 13 PPCS variant characters

Character Name	Glyph	GCGID	Open Systems IBM-1047 view	APL IBM-293 view	Inter-national 500 view	France 297 view	Germany 273 view	US/Can 037 view
left bracket	[SM060000	AD	AD	4A	90	63	BA
right bracket]	SM080000	BD	BD	5A	B5	FC	BB
left brace	{	SM110000	C0	C0	C0	51	43	C0
right brace	}	SM140000	D0	D0	D0	54	DC	D0
backslash	\	SM070000	E0	E0	E0	48	EC	E0
circumflex	^	SD150000	5F	5F	5F	5F	5F	B0
tilde	~	SD190000	A1	A1	A1	BD	59	A1
exclamation mark	!	SP020000	5A	5A	4F	4F	4F	5A
pound (number) sign	#	SM010000	7B	7B	7B	B1	7B	7B
vertical bar		SM130000	4F	4F	BB	BB	BB	4F
accent grave	`	SD130000	79	79	79	A0	79	79
dollar sign	\$	SC030000	5B	5B	5B	5B	5B	5B
commercial "at"	@	SM050000	7C	7C	7C	44	B5	7C

Two tables are available to show the full code—point mappings for Open Systems coded character set IBM-1047 and for the APL coded character set IBM-293. Upon examination of these coded character sets, you will notice that coded character set 1047 is a "Latinized" coded character set IBM-293. All the APL code points have been replaced by Latin 1 code points, allowing a one-to-one mapping among coded character set IBM-1047 and all other coded character sets in the Latin 1 group.

Although the official current coded character set for z/OS XL C/C++ is now coded character set IBM-1047 (Open Systems), the coded character set IBM-293 *syntax* points are still valid. Those points are the ones with syntactic relevance to the z/OS XL C/C++ compiler. Refer to [Table 115 on page 731](#) and [Table 116 on page 732](#) for more information.

Alternate code points

All syntactic code points that were supported in previous versions of z/OS XL C/C++ will continue to be supported *if* you are compiling with the NOLOCALE option.

To be compatible, the vertical bar character is represented by the following two encodings, provided you are not using the LOCALE compiler option or the NOLOCALE option:

- X'4F'
- X'6A'

If you do specify the LOCALE compiler option, each of these characters is represented by a unique value specified in the LC_SYNTAX category of the selected locale.

Coding without locale support by using a hybrid coded character set

If you want to avoid using the locale of the compiler, use a hybrid coded character set. A *hybrid* piece of code is in the local coded character set but the syntax is written *as if* it were in coded character set IBM-1047.

You can continue coding in the local coded character set, writing the syntax *as if* it were in coded character set IBM-1047. This solution uses the existing behavior of the compiler, but this method is not ideal for the following reasons; [Figure 210 on page 734](#) illustrates these difficulties.

- The code can be difficult to read and may not even look like C code anymore.
- There may be ambiguities in the code.
- Exporting code to another site can be difficult because the mapping between the hybrid characters used and the target coded character set may not be exact.

```

/* this has strings in codepage 273 with APL 293 syntax, and is a */
/* pre-locale source file for a user in Germany */
#define MAX_NAMES 20
#define MAX_NAME_LEN 80
#define STR(num) &hash273;num
#define SCAN_FORMAT(len) "%STR(len)"s %"STR(len)"s"

struct NameList &obrc273;
char first&obr273;MAX_NAME_LEN+1&cbrk273;;
char surname&obr273;MAX_NAME_LEN+1&cbrk273;;
&cbrk273;;
int compareNames(const void *elem1, const void *elem2) &obrc273;
struct NameList *name1 = (struct NameList *) elem1;
struct NameList *name2 = (struct NameList *) elem2;
int surnameComp = strcoll(name1&ptr273;surname,
                           name2&ptr273;surname);
int firstComp = strcoll(name1&ptr273;first,
                         name2&ptr273;first);

return(surnameComp ? surnameComp : firstComp);
&cbrk273;;

main() &obrc273;
int i, rc, numEntries;
struct NameList curName;
struct NameList nameList&obr273;MAX_NAMES&cbrk273;;
printf("Bitte geben Sie die Namen ein, "
       "im Format <Famlienname> <Voname> "
       "(Maximum %d Namen!)&esc273;",
       MAX_NAMES);
for (i=0; i<MAX_NAMES; &pp273;i &obrc273;
     printf("Name (oder EOF wenn fertig):&esc273;");
     rc = scanf(SCAN_FORMAT(MAX_NAME_LEN),
                 curName.surname, curName.first);
     if (rc &lnq273 2) &obrc273;
         break;
     &cbrk273;;
     nameList&obr273;i&cbrk273 = curName;
&cbrk273;;
numEntries = i+1;
qsort(nameList, numEntries, sizeof(struct NameList),
      compareNames);
for (i=0; i<numEntries; &pp273;i &obrc273;
     printf("Name %d:&lt; %s, s>&esc273;", i+1,
            nameList&obr273;i&cbrk273;.surname,
            nameList&obr273;i&cbrk273;.first);
     ü
     i != (MAX_NAMES << sizeof(int)/2);
     return(i);ü

```

Figure 210. Example of hybrid coded character set

The code points in Figure 210 on page 734, which have different glyphs in character code set IBM-273 and APL-293, are described below:

- 1** code point for the { character. In coded character set 273, this is the character ä.
- 2** code point for the [character. In coded character set 273, this is the character Ý.
- 3** code point for the] character. In coded character set 273, this is the character ¨.
- 4** code point for the } character. In coded character set 273, this is the character ü.
- 5** code point for the \ character. In coded character set 273, this is the character Ö.
- 6** code point for the ! character. In coded character set 273, this is the character Ü.

7

code point for the | character. In coded character set 273, this is the character !. This particular code point mapping is unfortunate because the | character and the ! character are both valid C syntax characters. Note that the ! character used in the printf() call at **8** will appear as ! on a terminal displaying in coded character set 273.

Writing code using a hybrid coded character set

Figure 210 on page 734 illustrates some of the problems with hybrid files. The following steps were done when writing this code:

1. Look up each variant character in coded character set IBM-1047 to find out what the compiler expects. For example, z/OS XL C/C++ expects the character [to have a byte value of X'AD'.
2. Determine which glyph is at X'AD' in the local coded character set, then use this in the code.
3. Always use the appropriate substitution. For example, to obtain a needed [in Germany, one would look up X'AD' in the German IBM-273 coded character set, and find the character Ÿ.

Converting hybrid code

Existing code that was written in a hybrid coded character set are supported.

Coded character set independence in developing applications

You can ensure that you are working effectively with the locale functionality if you use the appropriate functions, macros, and tools. Figure 211 on page 735 is a summary of the compile-edit work flow and shows which functions to use and where you can use them.

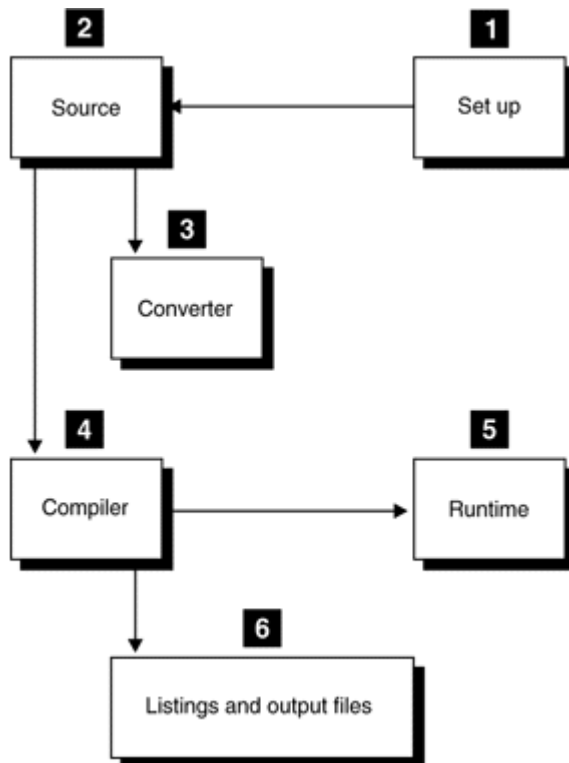


Figure 211. Compile-edit, related to locale function

1

Setup. The `localedef` information (see overview in [Chapter 55, “Customizing a locale,”](#) on page 691 and details in [“Locale source files”](#) on page 653).

2

Coded character set of source code, header files, and data. The compiler must support the coded character set used to create a source file so that it will recognize the variant C syntax characters correctly.

- The `#pragma filetag` directive identifies the coded character set of the source file as well as the library or user's `include` files (for an overview see [“The pragma filetag directive”](#) on page 737)
- Predefined macros `__LOCALE__`, `__FILETAG__`, and `__CODESET__` (for an overview see [“Using predefined macros”](#) on page 737)
- The function `setlocale()`
- The `pragma convlit` directive
- The `pragma convert` directive

3

Coded character set conversion utilities and functions. The coded character set of a file, or a stream of data, can be converted to another coded character set using the utilities `genxlt` and `iconv` (for an overview see Chapter 58, “Code set conversion utilities,” on page 707; for the details of the Coded character set and locale utilities, see *z/OS XL C/C++ User's Guide*), as well as the functions in the runtime library.

4

Coded character set conversion at compile time is determined by the compile-time locale and supported by the compiler options, `LOCALE` and `NOLOCALE` (for an overview, see [“Converting coded character sets at compile time”](#) on page 739; for details, see [LOCALE | NOLOCALE](#) in *z/OS XL C/C++ User's Guide*).

5

Runtime environment. During run time, the `setlocale()` function has an effect on runtime functions, such as `printf()`, `scanf()`, and `regcomp()`, which use variant characters.

6

Listings and output files. The coded character set used to create or to convert source files may affect listings, preprocessed source code, object modules, and SYSEVENT files (for an overview see [“Object modules and output listings”](#) on page 741). Your application can, however, include logic using the following to minimize the impact:

- `__LOCALE__`, `__FILETAG__`, and `__CODESET__` macros
- Locale functions such as `setlocale()`

Coded character set in source code and header files

There are five types of locale-related changes that you can make in your source code:

- You can tag your source code and other associated files with the `#pragma filetag` directive to specify the coded character set that was used while entering the file. You can then compile these to ensure that all variant characters in your files are correct.
- You can use the three macros: `__LOCALE__`, `__FILETAG__`, and `__CODESET__`. These *z/OS XL C/C++* macros expand to provide information about the `#pragma filetag` directive of the current source, and the locale and target coded character set used by the compiler at compile time. For more information, see [Compiler predefined macros](#) in *z/OS XL C/C++ Language Reference*.
- You can use the `setlocale()` function to set the runtime locale to be the same as the locale used to compile the application. This can be used when your application contains dependencies on the coded character set, as it would when comparing constants with external data. Using the macros forces the runtime locale to be the same as the one used to compile your code.
- You can use the `#pragma convlit` suspend and resume to exclude portions of your code from string literal conversion. See [CONVLIT | NOCONVLIT](#) in *z/OS XL C/C++ User's Guide* for more details on the compiler option and [#pragma convlit](#) in *z/OS XL C/C++ Language Reference* for more information on the pragma.

- You can use the `#pragma convert` directive to specify the coded character set to use for converting string literals. See [convert](#) in *z/OS XL C/C++ Language Reference* for more information on this pragma.

The pragma filetag directive

By using the `#pragma filetag` directive, you may write your programs in any convenient supported coded character set. The `#pragma filetag` directive instructs the z/OS XL C/C++ compiler how to "read" the source. Tagging the source files, the header files, and all data files (including messages) with the `#pragma filetag` directive enables you to keep the information about the coded character set used to create each source file, within the source file itself. This information can be helpful when moving source files to systems with different coded character sets. For more information, see [#pragma filetag](#) in *z/OS XL C/C++ Language Reference*.

The following example tag uses the German coded character set IBM-273:

```
??=pragma filetag("IBM-273")
```

Because the `#` character is variant in different coded character sets, you must use the trigraph `??=` for the `#pragma filetag` directive.

The `#pragma filetag` directive specifies the coded character set in which the source or data was entered. The coded character set specified in the `#pragma filetag` directive is in effect for the entire source file, but not for any other source file. This also applies to header files and data files.

The `#pragma filetag` directive can only appear once in each file, and it must appear before the first statement in a program. If encountered elsewhere, a warning appears and the directive does not change. If a comment contains variant characters and appears before the directive, the comment does not translate.



Attention: If you wish to use the `iconv` utility on a file that is tagged with the `??= #pragma filetag` directive, you must update the file manually to change the filetag to the correct converted coded character set. `iconv` does not update the pragma in source files.

Using predefined macros

There are three macros for z/OS XL C/C++ that relate to locale.

__LOCALE__

This macro expands to a string literal representing the locale of the `LOCALE` compiler option. This macro can be used to set the runtime locale to be the same as the compiled locale:

```
main() {
    setlocale(LC_ALL, __LOCALE__);
    :
}
```

The value of this macro is defined per compilation. If `NOLOCALE` compiler option is supplied, the macro is undefined.

__FILETAG__

This macro expands to a string literal representing the character coded character set of the `#pragma filetag` directive associated with the current file. For example, to convert to the coded character set specified by the `LOCALE` option from the coded character set specified by the `#pragma filetag` directive, you would use the `iconv_open()` function:

```
iconv_open(__FILETAG__, variable);
```

The value of this macro is defined per source file. If no `#pragma filetag` directive is present, the macro is undefined.

__CODESET__

This macro expands to a string literal representing the character coded character set of the `LOCALE` compiler option. The value of this macro is defined per compilation. If a value is not supplied, the macro is undefined.

Figure 212 on page 738 shows an example program (CCNGCC2) that uses the `__CODESET__` macro.

```
#include <iconv.h>
#include <string.h>
#include <stdio.h>

/* The following function could be in a header file */
#ifdef __CODESET__
    static int convstr(iconv_t convInfo, char *in, int inSize,
                      char *out, int outSize) {
        return(iconv(convInfo, in, inSize, out, outSize))
    }
#else
    static int convstr(iconv_t convInfo, char *in, int inSize,
                      char *out, int outSize) {
        memcpy(out, in, outSize > inSize ? inSize :
outSize);
        return(outSize > inSize ? -1 ::
0);
    }
#endif

iconv_t convInfo;

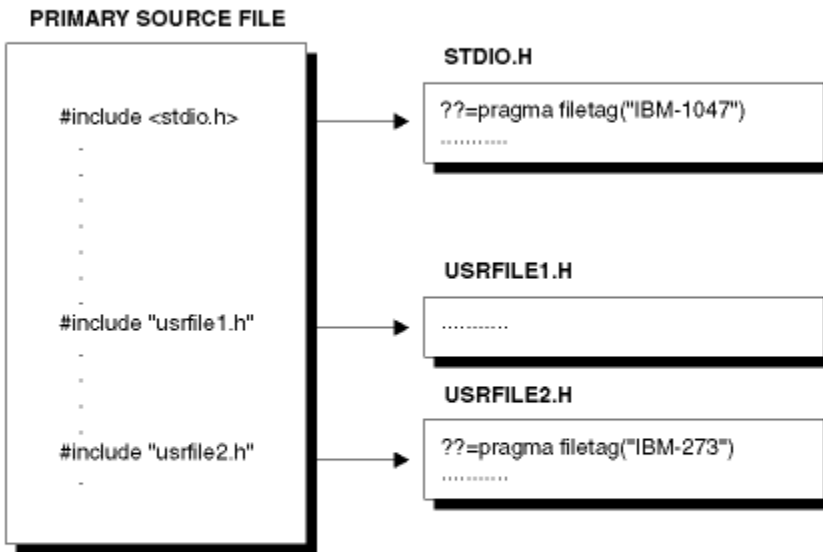
int main() {
#ifdef __CODESET__
    char *run-timeCodeSet;
    setlocale(LC_ALL, ""); /* set locale to default locale */
    run-timeCodeSet = nl_langinfo(CODESET);
    convInfo = iconv_open(run-timeCodeSet, __CODESET__);
#endif
    char intro[] = "Welcome to my variant world!\n";
    char nlIntro[sizeof(intro)];
    convstr(convInfo, intro, sizeof(intro),
nlIntro, sizeof(nlIntro));
    puts(nlIntro); /* string will print appropriately */
#ifdef __CODESET__
    iconv_close(convInfo);
#endif

    return(0);
}
```

Figure 212. Example of `__CODESET__` macro

Figure 213 on page 739 shows the values that these macros will take on, emphasizing that for `__FILETAG__`, a value is assigned for each source file, but for `__LOCALE__` and `__CODESET__`, a value is assigned for a compilation.

Assuming: Compiled source file with LOCALE("De_DE.IBM-273")



For the entire compilation: `__LOCALE__ = "De_DE.IBM273"`
`__CODESET__ = "IBM-273"`

In STDIO.H: `__FILETAG__ = "IBM-1047"`

In USRFILE1.H: `__FILETAG__` is undefined

In USRFILE2.H: `__FILETAG__ = "IBM-273"`

Figure 213. Values of macros `__FILETAG__`, `__LOCALE__`, and `__CODESET__`

Using `setlocale()`

You can change the runtime locale to any one of the other predefined locales. To use a defined locale, refer to it by its `setlocale()` parameter. To define a new locale, copy the source file provided, edit it, then assemble it (see [Chapter 55, "Customizing a locale,"](#) on page 691).

Converting coded character sets at compile time

The following section describe compiler options you can use to covert coded character sets.

CONVLIT compiler option

You can control the conversion of string literals in your code by using the CONVLIT compiler option. CONVLIT provides a means for changing the assumed code page for character string literals by supplying a codepage value. For more information, see [CONVLIT | NOCONVLIT in z/OS XL C/C++ User's Guide](#).

For example, if you used an ASCII client machine to write code that uses string literals, and then upload this to an EBCDIC server such as MVS, your string literals would be converted to EBCDIC. However, if you specified `"CONVLIT (IS08859-1)"` when you compiled your code, your string literals would have been converted to an ASCII code page.

For example, consider the program in [Figure 214 on page 740](#). When this program is compiled with the `CONVLIT (IS08859-1)` option, the string "Hi There!" will be converted to an ASCII string, but the string "Hello World" will not be converted.

```

/* header.h */
char *text="Hello World";

/* test.c */
#pragma convlit(suspend)
#pragma comment (user, "A user comment")

#include <stdio.h>
#include "header.h"
#pragma convlit(resume)

main () {
    char *text2 = "Hi There!";
}

```

Figure 214. Using the CONVLIT compiler option

LOCALE compiler option

The LOCALE compiler option enables you to instruct the compiler to use a specific locale at compile time, which then generates the output in the same coded character set.

The input files that are affected are:

- The primary source file
- Library header files
- User header files

The output files that are affected are:

- Object Modules
- Preprocessed source code
- Listings

To use the LOCALE option, you must supply a locale name value. The locale name is a string that represents the locale you want to compile source with; this will determine the characteristics of output, including the coded character set used for variant characters in the source. Usually, a locale name is of the format *territory name.coded character set*. For example, the German locale for coded character set 273 is De_DE . IBM-273. The *territory name* is De_DE and the *coded character set* is IBM-273. To determine the coded character set of the current locale, use the function `n1_langinfo(CODESET)`.

The special locale name "" gives you the default locale, which can be set using environment variables. The locale name "C" specifies the C default locale. Full details about the C locale are found in [Chapter 57, "Definition of S370 C, SAA C, and POSIX C locales,"](#) on page 699.

The default option setting is NOLOCALE. It instructs the compiler to do no conversion of text for input or for output.

You can create your own locales by using the `localedef` utility. See ["Locale source files"](#) on page 653 for details.

Examples

To compile a sample file, `userid.SORTNAME.C`, enter:

```
CC 'userid.SORTNAME.C' (LOCALE("De_DE.IBM-273"))
```

The compiler recognizes "De_DE . IBM-273" as a valid locale and automatically converts the source code to coded character set IBM-273, for its own use. The compiler would then generate listings in the German coded character set 273.

To generate a preprocessed file that can be sent to other sites, that use different coded character sets, enter:

```
CC 'userid.SORTNAME.C' (LOCALE("De_DE.IBM-273")) PPNLY
```

The compiler will insert the `#pragma filetag` directive at the start of the preprocessed file, using the coded character set specified in the `LOCALE` option. In this example, `??=pragma filetag("IBM-273")` is inserted.

Since the preprocessed file has been tagged, it can be compiled using the z/OS XL C/C++ compiler at any site, regardless of the locale used.

Summary of usage for LOCALE, NOLOCALE, and pragma filetag directive

The following list shows the results from different combinations of the `#pragma filetag` directive and the `LOCALE` compiler option.

Using LOCALE compiler option

In this case, the compiler does the following:

- Converts the source code from the coded character set specified with the `#pragma filetag` directive to the code set specified by the `LOCALE` compiler option.
- If no `#pragma filetag` directive is specified, the compiler assumes the source is in the same coded character set as specified by the locale, and does not perform any conversion.
- Converts compiler error messages from coded character set IBM-1047 to the coded character set specified in the `LOCALE` compiler option.
- Generates compiler output in the same coded character set as that of the locale specified in the `LOCALE` compiler option.
- If `PPONLY` was specified, the compiler inserts the `#pragma filetag` directive at the beginning of the preprocessor file, using the coded character set specified in the locale option.

Using NOLOCALE compiler option

In this case, the compiler does the following:

- Does not convert text in the input or output file, and uses the default coded character set IBM-1047 to interpret syntactic characters.
- If a `#pragma filetag` directive is specified, the compiler suppresses the `#pragma filetag` directive in the preprocessor file. The compiler issues warnings if the `#pragma filetag` directive specifies a coded character set other than IBM-1047, and uses IBM-1047 anyway.

Object modules and output listings

The compiler respects the locale specified by the `LOCALE` compiler option when it generates the listing.

If the `locale` option is specified, the object module is generated in the coded character set of your current locale. Otherwise, the object module is generated in the coded character set IBM-1047.

Code will run correctly if the runtime locale is the same as the locale of the object module.

If the object was generated with a different locale from the one you run under, you must ensure that your code can run under different locales. Refer to [Chapter 55, “Customizing a locale,” on page 691](#) for more information.

For information about exporting code to other sites, see [“Exporting source code to other sites” on page 744](#).

You can use the `LOCALE` compiler option to ensure that listings are sensitive to a specified locale.

[Figure 215 on page 742](#) shows the result from compiling source file `hello273.c` with:

```
xlc -F:c89 -o hello273 -qso -qlocale="De_DE.IBM-273" -qxplink -qgoff hello273.c
```

In [Figure 215 on page 742](#), notice the locale-specific information:

1

The date at the top right. The format of the date in the listing is that specified by the locale.

2

The name of the locale and the code set.

3

Code points for the }, /, and { characters.

```

15650Z0S V2.4 z/OS XL C                               ./hello273.c                12.06.19 10:40:10
Page      1

          * * * * * P R O L O G * * * * *
Compile Time Library . . . . . : 42040000
Command options:
  Program name. . . . . : ./hello273.c
  Compiler options. . . . . :
    *NOGONNUMBER *NOALIAS *RENT *TERMINAL *NOUPCONV *SOURCE *NOLIST
    *NOXREF *NOAGG *NOPPONLY *NOEXPMAC *NOSHOWINC *NOOFFSET *MEMORY
    *NOSSCOMM *NOSHOWMACROS *SKIPSRC(SHOW) *NOREPORT *NOMAKEDEP
    *PREFETCH *THREADED
    *LONGNAME *START *EXECOPS *ARGPARSE *NOEXPORTALL *NODLL(NOCALLBACKANY)
    *NOLIBANSI *NOWSIZEOF *REDIR *ANSIALIAS *DIGRAPH *NOROCONST *ROSTRING
    *TUNE(10) *ARCH(10) *SPILL(128) *MAXMEM(2097152) *NOCOMPACT
    *TARGET(LE,CURRENT) *FLAG(I) *NOTEST(SYM,BLOCK,LINE,PATH,HOOK) *NOOPTIMIZE
    *NOINLINE(AUTO,NOREPORT,100,1000) *NESTINC(255) *BITFIELD(UNSIGNED)
    *NOINFO
    *NODFP
    *NOVECTOR
    *FLOAT(HEX,FOLD,NOMAF,AFP(NOVOLATILE)) *ROUND(Z)
    *STRICT
    *NOSTACKPROTECT
    *NOCOMPRESS *NOSTRICT_INDUCTION *AGGRCOPY(NOOVERLAP) *CHARS(UNSIGNED)
    *NOIGNERRNO
    *NOINITAUTO
    *CSECT()
    *NOEVENTS
    *ASSERT(RESTRICT)
    *NORESTRICT
    *OBJECT(./hello273.o)
    *NOGENASM
    *NOOPTFILE
    *NOSERVICE
    *OE
    *NOIPA
    *SEARCH(/c390/archive/zosv2r4/D190529/util/usr/include)
    *NOLSEARCH
    *LOCALE *HALT(16) *PLIST(HOST)
    *NOCONVLIT
    *NOASCII

```

Figure 215. Example of output when locale option is used

```

: *GOFF *ILP32 *NOWARN64 *NOHGPR *NOHOT *NOMETAL *NOARMODE
: *XPLINK(NOBACKCHAIN,NOSTOREARGS,NOCALLBACK,GUARD,OSCALL(NOSTACK))
: *ENUMSIZE(SMALL)
: *NOHALTONMSG
: *NOSUPPRESS
: *NORTCHECK
: *NODEBUG
: *NOSQL
: *NOCICS
: *UNROLL(AUTO)
: *KEYWORD()
: *NOKEYWORD(asm,typeof)
: *NOSEVERITY
: *NODSAUSER
: *NOINCLUDE
: *NOSMP
: *SYSSTATE(NOASCENV,OSREL(NONE))
: *NOFUNCEVENT
: *NOASM

15650Z0S V2.4 z/OS XL C                               ./hello273.c                12.06.19 10:40:10
Page      2

          * * * * * P R O L O G * * * * *

: *NOASMLIB
: DEFINE(_OPEN_DEFAULT=1)
: DEFINE(errno=(*_errno()))
Version Macros. . . . . : __COMPILER_VER__=0x42040000 __LIBREL__=0x42040000 __TARGET_LIB__=0x42040000
Language level. . . . . : *ANSI:NOTEXTAFTERENDIF
Source margins. . . . . :
  Varying length. . . . . : 1 - 32760
  Fixed length. . . . . : 1 - 32760
Sequence columns. . . . . :
  Varying length. . . . . : none
  Fixed length. . . . . : 73 - 80
Locale Name . . . . . : De_DE.IBM-273
Code Set. . . . . : IBM-273

          * * * * * E N D   O F   P R O L O G * * * * *

```

Example of output when locale option is used (Part 1 of 2)

```

15650ZOS V2.4 z/OS XL C                      ./hello273.c                      12.06.19 10:40:10    Page    3

***** SOURCE *****

LINE STMT
INCNO
1      |?=?pragma filetag("IBM-273")
2      |#include <stdio.h>
3
4      |int main (void) a
5      |1 printf("Hello World|0n");
6
7      |2 return (0);
8      |ü
9
***** END OF SOURCE *****
15650ZOS V2.4 z/OS XL C                      ./hello273.c                      12.06.19 10:40:10
Page    4

***** INCLUDES *****

INCLUDE FILES --- FILE#   NAME
1      /c390/archive/zosv2r4/D190529/util/usr/include/stdio.h
2      /c390/archive/zosv2r4/D190529/util/usr/include/features.h
3      /c390/archive/zosv2r4/D190529/util/usr/include/sys/types.h
***** END OF INCLUDES *****
15650ZOS V2.4 z/OS XL C                      ./hello273.c                      12.06.19 10:40:10
Page    5

***** MESSAGE SUMMARY *****

Total      Informational(00)      Warning(10)      Error(30)      Severe Error(40)
0           0           0           0           0
***** END OF MESSAGE SUMMARY *****
***** END OF COMPILATION *****

```

Example of output when locale option is used (Part 2 of 2)

The pragma convert directive

You can control the conversion of string literals in your code by using the `#pragma convert` directive. It allows you to change the assumed code page for character string literals by supplying a codepage value. For more information, see [convert](#) in *z/OS XL C/C++ Language Reference*.

For example, if you use an ASCII client machine to write code with string literals and upload it to an EBCDIC server, then your string literals will be converted to EBCDIC. However, if you add the `pragma convert("IS08859-1")` directive to your source code, then your string literals will be converted to an ASCII code page.

For example, consider the program in Figure 216 on page 743. When this program is compiled, the string "Hello World" will be converted to an ASCII string, but the string "Hi There!" will not be converted.

```

/* header.h */
#pragma convert("IS08859-1")
char *text="Hello World";
#pragma convert(pop)

/* test.c */
#pragma comment (user, "A user comment")
#include "header.h"

main () {
    char *text2 ="Hi There!";
}

```

Figure 216. Using the pragma convert directive

Writing source code in coded character set IBM-1047

There are two reasons why you would want to write source in coded character set IBM-1047.

First, even though z/OS XL C/C++ provides support for multiple coded character sets, other tools may not do so. Tools such as CICS and DB2 may not support source code in any coded character set other than

the default coded character set, IBM-1047. If you are using these tools, and you write your code in a code page other than IBM-1047, you will need to use the z/OS XL C/C++ iconv utility to convert your code to coded character set IBM-1047 before you can use the tool.

Second, older versions of the C/370 product do not support source in coded character sets other than IBM-1047. This makes it difficult to share code with a site using an older compiler.

Exporting source code to other sites

This section deals with the *exporting* of code from one Latin-1 coded character set to another; that is, writing code that can be run in a locale that uses a different coded character set than the one used to write the source.

To export code, use the `iconv()` utility to convert each source file, header file, and data file to the target coded character set. You can then send all files to the target location for compilation.

Note: You must ensure that your code runs in the same locale that it was compiled under before running it with any other locales.

1. Use the `#pragma filetag` directive to tag each source file, header file, and data file.
2. Use message files for all external strings, such as prompts, help screens, and error messages. To write truly portable code, convert these strings to the runtime coded character set in your application code.
3. Use the `setlocale()` function so that the library functions are sensitive to the runtime coded character set.

Ensure that locale-sensitive information, such as decimal points, are displayed appropriately. Use either `nl_langinfo()` or `localeconv()` to obtain this information.

The `setlocale()` function does not change the CEE callable services under the z/OS Language Environment in such areas as date, time, currency, and time zones. Internationalization is specific to z/OS XL C/C++ applications. Also, the z/OS Language Environment CEE callable services do not change the z/OS XL C/C++ locales. For a list of these callable services, see the [z/OS Language Environment Programming Guide](#).

4. Compile with the locale specifying coded character set IBM-1047.

If you specify `locale("locale-name")`, your code will run correctly with libraries running in the same coded character set. However, if you compile with a different locale than you run under, you have to ensure that your code has no internal data, and also that all libraries you use are runtime locale sensitive.

For example, consider the following code fragment. If you compile with `locale("De_DE.IBM-273")`, the square brackets are converted to the hex values X'63' and X'FC'. If the default locale you then run under is not "De_DE.IBM-273", but instead "En_US.IBM-1047", and you have not used `setlocale()`, the square brackets will be interpreted as Ä and Ü, and the call to `scanf()` will not do what you intended.

```
int main() {
    setlocale(LC_ALL, "");
    :
    rc = scanf("%[1234567890abcdefABCDEF]", hexNum);
    :
}
```

If you only need to run your code locally or export it to a site that has your locale environment, you can solve this problem by using the following coding. This ensures that your code runs with the same locale it was compiled under. Library functions such as `printf()`, `scanf()`, `strfmon()`, and `regcomp()` are sensitive to the current coded character set. The `__LOCALE__` macro is described in [“Using predefined macros”](#) on page 737.


```
int main() {
    setlocale(LC_ALL, __LOCALE__);
    :
    rc = scanf("%[1234567890abcdefABCDEF]", hexNum);
    :
}
```

If you are generating code to export to a site that may not have your locale environment, you should write your code in IBM-1047.

Converting existing work

This section describes some conversion issues and presents some conversion scenarios. It is assumed that existing source code and libraries cannot be quickly converted from mixed coded character sets into a common coded character set; thus a staged approach is recommended.

- Code your new source in one coded character set, preferably IBM-1047. Tag all new source files to make them more portable by putting the `#pragma filetag` directive at the top of each one.
- If you need to interact with existing code, compile your new code using the locale in which the existing code was written.
- If you want to write code in a coded character set that does not have a one-to-one mapping to coded character set IBM-1047 (that is, a coded character set that is not Latin-1), create your own conversion table and compile it with the `genxlt` utility. Use your own conversion table with the `iconv` utility to convert your source code to coded character set IBM-1047.

Considerations with other products and tools

Note: Any software tool that scans source code or compiler listings is affected by the introduction of the locale functionality. Tools that read or generate source code now need to recognize the `#pragma filetag` directive. Tools that read listings need to recognize the coded character set in the title header.

Since the following tools scan source code, they may be affected:

- The Debug Tool does not support code written in any coded character set other than IBM-1047.
- Translators such as CICS and DB2 read source files and generate new source files. If they do not, then follow these steps:
 1. Convert the source file to coded character set IBM-1047 using the `iconv` utility.
 2. Remove the `#pragma filetag` directive from the source file, or change it to `??=pragma filetag("IBM-1047")`. Run the source that is in the IBM-1047 coded character set through the appropriate translator, if needed.

Chapter 60. Bidirectional language support

This chapter describes the characteristics of bidirectional languages, and provides an overview of the layout functions for bidirectional languages. For more information on the layout functions see [z/OS C/C++ Runtime Library Reference](#), and *X/Open Portable Layout Services: Context-dependent and Directional Text*.

Bidirectional languages

Bidirectional languages are languages such as Arabic and Hebrew, that are written and read mainly from right to left, but some portions of the text, such as numbers and embedded Latin languages (e.g. English) are written and read left to right. Additional characteristics of bidirectional languages include:

- visual order versus logical order
- symmetric swapping
- number formats
- cursive (shaping) versus non-cursive

In bidirectional text, it is important to note the difference between the logical order in which the text is processed or read, and the visual order in which the text is displayed. Bidirectional text is usually stored in logical order. For example, assume that the following text is Arabic, then the logical storage would contain:

```
maple street 25 entrance b
```

and the visual display would be (if read from right to left):

```
b ecnartne 25 teerts elpam
```

Some characters, such as the greater-than sign, have an implied directional meaning and have a complementary symmetric character with an opposite directional meaning (the less-than sign.) When used within a segment that is presented right-to-left but is inverted (left-to-right) when stored for processing, such a character might have to be replaced by its symmetric sibling to ensure that the correct meaning of the text is preserved. The replacement of such a character by its complement during the transformation of BiDi text is called "symmetrical swapping". Other graphic characters that need symmetrical swapping include the parentheses, square brackets, braces, and so on. Although symmetrical swapping is a characteristic of BiDi languages, it is not always mandatory for the software functions that transform different BiDi language text layouts. Sometimes this function is performed automatically by the workstation hardware or micro code.

Arabic numerals (Latin digits) are those numerals used with Latin text, while Hindi numerals are used within Arabic text, in some of the Arabian countries, like Egypt. However, the Implicit algorithm states the number storage should use Arabic numerals (Latin digit), and be displayed according to the user's settings.

Note that even though the text in the example is displayed right to left, the number "25" is still written left to right. That is because Arabic/Hebrew numbers are written and read left to right.

Arabic is a cursive language. Arabic characters are connected together, and each character has different shapes depending on its location within the word: initial, middle, final or isolated. Cursive languages are suited to handwriting rather than printing. Arabic is always cursive, whether in books, newspapers, signs or workstation displays. English can be handwritten in a cursive style, and it is often used that way in personal communications, but English is seldom published or displayed in a cursive style. Thus, English is not considered a cursive language.

To simplify processing, characters are usually stored in an unshaped form. (The unshaped form is also referred to as the abstract or basic form.) Shaping takes into account the character being shaped and the characters in its vicinity, and replaces the unshaped, abstract form with the proper shape. For example,

in Arabic, the unshaped character would be replaced with the initial, middle, final or isolated shaped character, depending on the context.

Note that Hebrew letters do not use shaping, and numbers used with Hebrew text are always displayed with the same digits as used for English.

Legacy operating systems like MVS used to store Arabic and Hebrew data in their visual format. Sometimes for specific needs, data might be stored in a specific shape, for example initial shape. Currently, most applications store text in its unshaped form in logical order. Reordering and shaping are done at display time. Storing text in its unshaped form in logical order makes it easier to process the data (sorting, comparison).

Overview of the layout functions

The layout functions are used to handle bidirectional languages correctly, to transform text from a format readable for the user to a format suitable for processing, and vice-versa. The layout functions include the following:

m_create_layout()

called at the beginning of the application to create the layout object that will be used by the other layout functions.

m_setvalues_layout()

sets the values that will be used inside the transform. `m_setvalues_layout()` must be called before calling `m_transform_layout` or `m_wtransform_layout`. This function is optional. Use this function if you need to change the values for the bidirectional attributes. You can eliminate it from the application, and use a modifier instead.

m_getvalues_layout()

queries the current layout values within a layout object.

m_transform_layout()

does the actual processing to convert the text format between different bidirectional layouts, according to the settings of the `LayoutObject`. Nothing will change if this function (or its wide character equivalent) is not called inside the application.

m_wtransform_layout()

works the same as `m_transform_layout()`, except that it handles Unicode wide characters (`wchar_t`).

m_destroy_layout()

called at the end of the application to destroy the layout object, and free up the allocated memory used by the layout object.

Those functions can be used to convert text from logical (implicit) unshaped forms to visual (display) shaped forms and vice versa. The layout functions also handle conversion of numerals.

Table 117 on page 749 lists supported layout attributes and their corresponding values. These are the attributes most commonly used to provide Bidi support. Each attribute has input and output values that can be specified for the layout transformation process. The default value for each attribute is indicated in the table.

A full list of attributes and values is available in *X/Open Portable Layout Services: Context-dependent and Directional Text*. Some attributes listed in Table 117 on page 749 are specific to the z/OS implementation and are noted with an asterisk (*) symbol.

You can set Bidi Layout Values in two ways:

- The `m_setvalues_layout()` function requires the Layout Attribute Names and Values to be specified by using keywords presented in columns 1 and 2.
- The `m_create_layout()` function allows a string to be passed to set the values. This string is preceded with the "@ls" characters and requires the names and values to be used as displayed in column 3. Multiple attribute and value pairs can be separated by commas in the following form:

```
@ls <attribute_name1>=<input1>:<output1>,  
<attribute_name2>=<input2>:<output2>...
```

Example:

```
@ls orientation=ltr:ltr,typeofText=implicit:visual,
numerals=nominal:national, swapping=no:yes,
bidiroundtrip=true
```

Table 117. Layout attribute and values				
Attribute Name	Attribute values	Modifier layout string names and values (@ls)		Description
Orientation	ORIENTATION_CONTEXTUAL ORIENTATION_LTR (Default) ORIENTATION_RTL	Name:	orientation=	The direction of the text.
		Values:	contextual ltr rtl	
Context	CONTEXT_LTR (Default) CONTEXT_RTL	Name:	context=	Contextual orientation when the orientation attribute is set to ORIENTATION_CONTEXTUAL
		Values:	ltr rtl	
TypeofText	TEXT_EXPLICIT TEXT_IMPLICIT (Input default) TEXT_VISUAL (Output default)	Name:	typeofText=	Type of the text.
		Values:	explicit implicit visual	
Swapping	SWAPPING_NO (Input default) SWAPPING_YES (Output default)	Name:	swapping=	Specifies if symmetric swapping is enabled.
		Values:	no yes	
Numerals	NUMERALS_CONTEXTUAL (Output default in Arabic locale) NUMERALS_NATIONAL NUMERALS_NOMINAL (Input default, Output default in Hebrew locale)	Name:	numerals=	How numerals are shaped. (Only valid for Arabic.)
		Values:	contextual national nominal	
TextShaping	TEXT_NOMINAL (Input default, Output default in Hebrew locale) TEXT_SHAPED (Output default in Arabic locale)	Name:	shaping=	Specifies if text is to be shaped. (Only valid for Arabic.)
		Values:	nominal shaped	
ShapeCharset	IBM-1046 IBM-1089 IBM-1256 IBM-420 IBM-424 IBM-425 IBM-53668 IBM-864 ISO8859-6 UCS-2 (Default)	Name:	shapcharset=	Code set of the output buffer to result from layout transformation.
		Values:	IBM-1046 IBM-1089 IBM-1256 IBM-420 IBM-424 IBM-425 IBM-53668 IBM-864 ISO8859-6 UCS-2 (Default)	
InputCharset*	IBM-1046 IBM-1089 IBM-1256 IBM-420 IBM-424 IBM-425 IBM-53668 IBM-864 ISO8859-6 UCS-2	N/A		Code set of the input buffer to use in layout transformation. The default value is the code set of the loaded locale.
BidiRoundTrip*	BIDIROUNDTRIP_OFF (Default) BIDIROUNDTRIP_ON	Name:	bidiroundtrip=	Specifies if the Bidi roundtrip algorithm is enabled. To take proper effect, BidiRoundTrip must be enabled in both legs of the layout transformations that complete the roundtrip.
		Values:	false true	

Table 117. Layout attribute and values (continued)

Attribute Name	Attribute values	Modifier layout string names and values (@ls)		Description
SeenTail*	SEEN_OFF (Default) SEEN_NEAR SEEN_ATBEGIN SEEN_ATEND	Name:	seentail=	Specifies the option for tail character after the Seen family character. See details about this attribute in Table 118 on page 750.
		Values:	off near atbegin atend	

Table 118. Description of the values of SeenTail attribute

SeenTail option	Transforming from implicit to visual	Transforming from visual to implicit
SEEN_OFF (Default) or off	<ul style="list-style-type: none"> If the output orientation is LTR, spaces are consumed from the end of the buffer and converted to Tail character. If the output orientation is RTL, spaces are consumed from the beginning of the buffer and converted to Tail character. 	<ul style="list-style-type: none"> If the output orientation is LTR, each Tail character is removed and a space is added to the end of the buffer. If the output orientation is RTL, each Tail character is removed and a space is added to the beginning of the buffer.
	Note: The option behavior is the same before the support for Sen tail is added for backward compatibility.	
SEEN_NEAR or near	Replace the space adjacent to the sen family character to Tail character.	Each Tail character is replaced by a space.
SEEN_ATBEGIN or atbegin	For each sen family character, add Tail character adjacent to the sen family character and check the orientation of the input and the output: <ul style="list-style-type: none"> If the orientation of the input and the output: is the same, remove a space from the start of the buffer. If the orientation of the input and the output are different, remove a space from the end of the buffer. 	Each Tail character is removed and a space is added at the beginning of the buffer.
SEEN_ATEND or atend	For each sen family character, add Tail character adjacent to the sen family character and: <ul style="list-style-type: none"> If the orientation of the input and the output are the same, remove a space from the end of the buffer. If the orientation of the input and the output are different, remove a space from the start of the buffer. 	Each Tail character is removed and a space is added at the end of the buffer.

m_create_layout()

This function creates a `LayoutObject` associated with the locale identified by `attrobj`. The `LayoutObject` is an opaque object containing all the data and methods necessary to perform the layout operations on context-dependent or directional characters of the locale identified by the `attrobj`.

The memory for the `LayoutObject` is allocated by `m_create_layout()`. The `LayoutObject` created has default layout values. (If the `modifier` argument is not `NULL`, the layout values specified by the `modifier` overwrite the default layout values associated with the locale).

```
#include <sys/layout.h>
LayoutObject m_create_layout(const AttrObject attrobj, const char* modifier);
```

attrobj argument

Is or may be an amalgam of many opaque objects. A locale object is just one example of the type of object that can be attached to an attribute object. The `attrobj` argument specifies a name that is usually associated with a locale category.

modifier argument

Can be used to announce a set of layout values when the `LayoutObject` is created.

m_setvalues_layout()

This function is used to change the layout values of a `LayoutObject`.

```
#include <sys/layout.h>
int m_setvalues_layout(LayoutObject layout_object, const LayoutValues values,
    int *index_returned);
```

layout_object argument

Specifies a `LayoutObject` returned by the `m_create_layout()` function.

values argument

Specifies the list of layout values that are to be changed. The values are written into the `LayoutObject` and may affect the behavior of subsequent layout functions.

m_getvalues_layout()

This function is used to query the current settings of the layout values within a `Layout Object`.

```
#include <sys/layout.h>
int m_getvalues_layout(const LayoutObject layout_object, LayoutValues values,
    int *index_returned);
```

layout_object argument

Specifies a `Layout Object` returned by the `m_create_layout()` function.

values argument

Specifies the list of layout values that are to be queried. Each value element of a `LayoutValueRec` must point to a location where the layout value is stored. That is, if the layout value is of type `T`, the argument must be of type `*T`. The values are queried from the `Layout Object` and represent its current setting. It is the user's responsibility to manage the memory allocation for the layout values queried. If the layout value name has `QueryValueSize` ORed to it, instead of the setting of the layout value, only its size is returned. This option can be used by the caller to determine the amount of memory needed to be allocated for the layout values queried.

m_transform_layout()

This function performs layout transformations (reordering and shaping), or it may provide additional information needed for layout transformation (such as the expected size of the transformed layout, the nesting level of different segments in the text and cross references between the locations of the corresponding elements before and after the layout transformation). Both the input text and output text are character strings. The `m_transform_layout()` function transforms the input text in `InpBuf` according to the current layout values in `layout_object`. Any layout value whose value type is `LayoutTextDescriptor` describes the attributes of the `InpBuf` and `OutBuf` arguments. If the attributes are the same for both `InpBuf` and `OutBuf`, a null transformation is performed with respect to that specific layout value. The `InpBuf` argument specifies the source text to be processed. The `InpSize` argument is the number of bytes within `InpBuf` to be processed by the transformation. Its value will not change after return from the transformation.

```
#include <sys/layout.h>
int m_transform_layout(LayoutObject layout_object,
                      const char *InpBuf,
                      const size_t InpSize,
                      void *OutBuf,
                      size_t *Outsize,
                      size_t *InpToOut,
                      size_t *OutToInp,
                      unsigned char *Property,
                      size_t *InpBufIndex);
```

LayoutObject argument

Specifies the Layout Object returned by m_create_layout().

InpBuf argument

Corresponds to the input string that the layout functions will process.

InpSize argument

Gives the input size of the input string specified by the InpBuf argument.

Note: If you need to pass -1 as a value for InpSize, you must cast it using (size_t)-1.

OutBuf argument

Any transformed data is stored here. This buffer will contain the data after converting it to the specified layout values and output code page.

Outsize argument

Gives the number of bytes in the Output Buffer.

InpToOut mapping argument

A cross-reference from each InpBuf code element to the transformed data. The cross-reference relates to the data in InpBuf starting with the first element that InpBufIndex points to (and not necessarily starting from the beginning of the InpBuf).

OutToInp mapping argument

A cross-reference to each InpBuf code element from the transformed data. The cross-reference relates to the data in InpBuf starting with the first element that InpBufIndex points to (and not necessarily starting from the beginning of the InpBuf).

Property argument

A weighted value that represents peculiar input string transformation properties with different connotations. If this argument is not a NULL pointer, it represents an array of values with the same number of elements as the source sub string text before the transformation. Each byte will contain relevant "property" information of the corresponding element in InpBuf starting from the element pointed by InpBufIndex.

InpBufIndex argument

InpBufIndex is an offset value to the location of the transformed text. When m_transform_layout() is called, InpBufIndex contains the offset to the element in InpBuf that will be transformed first. (Note that this is not necessarily the first element in InpBuf). At the return from the transformation, InpBufIndex contains the offset to the first element in the InpBuf that has not been transformed. If the entire sub string has been transformed successfully, InpBufIndex will be incremented by the amount defined by InpSize.

m_wtransform_layout()

The m_wtransform_layout is the same as m_transform_layout, except that it takes Unicode (wchar_t *) as an input buffer .

```
#include <sys/layout.h>
int m_wtransform_layout(LayoutObject layout_object,
                       const wchar_t *InpBuf,
                       const size_t InpSize, void *OutBuf,
                       size_t *Outsize,
                       size_t *InpToOut, size_t *OutToInp,
                       unsigned char *Property,
                       size_t *InpBufIndex );
```


m_destroy_layout()

This function destroys the layout object and frees up the allocated memory used by the layout object.

```
#include <sys/layout.h>
int m_destroy_layout(const LayoutObject layoutobject);
```

Using the layout functions

This section contains examples to illustrate how to call the BIDI layout engine. Note that to use the Bidi roundtrip algorithm, the option must be enabled in both legs of a layout transformation.

The example in [Figure 217 on page 753](#) sets the option using a layout string modifier.

```
Layout Transformation #1:
-----
"@ls orientation=ltr:ltr, typeoftext=visual:implicit,
bidiroundtrip=true"

Layout Transformation #2:
-----
"@ls orientation=ltr:ltr, typeoftext=implicit:visual,
bidiroundtrip=true"
```

Figure 217. Example of using a layout string modifier

[Figure 218 on page 753](#) demonstrates how to use the `m_setvalues_layout()` function to set the option.

```
LayoutValues layout = (LayoutValues)
calloc(2,sizeof(LayoutValueRec));
layout 0 .name = BidiRoundTrip;
layout 0 .value = (void *) BIDIROUNDTRIP_ON;
layout 1 .name = 0;
```

Figure 218. Example of using the `m_setvalues_layout()` function

To use the layout functions, perform the following steps.

1. Include the `sys/layout.h` header file to define the values and function prototypes.
-

```
#include <sys/layout.h>
```

2. Declare the program variables.
-

```
LayoutObject plh;
int error = 0, index;
size_t insize = 9, outsize;
LayoutValues layout;
LayoutTextDescriptor set_desc;

char *inbuffer;
char *outbuffer;
char *inShape;
char *outShape;
char *myModifier=
"@lstypeoftext=implicit:visual,shaping=nominal:shaped,orientation=ltr:rtl";
```

In the first line, declare a `LayoutObject` called "plh". This is the layout object that `m_create_layout()` creates later when invoked. `index` is the index of the returned error. `insize` is the size of the input buffer, and `outsize` is the size of the output buffer. The four integer variables in the second and third lines will be used later in the call of `m_setvalues_layout()` and `m_transform_layout()`. In the fourth line, declare a `LayoutValues` variable called "layout" and in the fifth line declare a `LayoutTextDescriptor` called "set_desc". These two variables are very important. They will be used with `m_setvalues_layout()` in the form of input/output pairs to specify new input and output

values for each one of the specified attributes. The next two lines add four strings (char *), that will be used as the input buffer, output buffer, input code page and, finally, the output code page. The last line adds a string that specifies the modifier to be used as specified earlier in the m_create_layout() function to create the layout object.

3. Allocate memory to the declared strings, layout values, layout text descriptor, and write the contents of the input buffer.

```
inbuffer=(char *)malloc(insize*sizeof(char));
outbuffer=(char *)malloc(outsize*sizeof(char));
layout  = (LayoutValues)malloc(6*sizeof(LayoutValueRec));
set_desc = (LayoutTextDescriptor)malloc(3*sizeof(LayoutTextDescriptorRec));
inShape = (char*) malloc(20 * sizeof(char));
outShape = (char*) malloc(20 * sizeof(char));
inbuffer[0] = 0xB0;
inbuffer[1] = 0xB1;
inbuffer[2] = 0xB2;
inbuffer[3] = 0xBF;
inbuffer[4] = 0x40;
inbuffer[5] = 0x9A;
inbuffer[6] = 0x75;
inbuffer[7] = 0x58;
inbuffer[8] = 0xDC;
```

The values of the input buffer are added one by one as an array of characters, but several alternatives could be used. For example, you can read the input buffer as a string from a file, or get it from another application.

4. Call the m_create_layout() function to create a layout object "plh".

```
plh = m_create_layout("Ar_AA",myModifier);
```

In the preceding example, the layout object "plh" is created with the locale Ar_AA with the modifier myModifier.

5. At this point of the program, there are two options: call m_setvalues_layout() or call the m_transform_layout() (or m_wtransform_layout()) directly.

Specify the input/output layout values. The first two lines below specify the two strings used as the input and output code pages. These two strings will be used by the other functions to specify the input code page for the input buffer and the output code page for the output buffer.

```
strcpy(outShape,"ibm-420");
strcpy(inShape,"ibm-425");

set_desc[0].inp = ORIENTATION_LTR;
set_desc[0].out = ORIENTATION_LTR;
set_desc[1].inp = TEXT_IMPLICIT;
set_desc[1].out = TEXT_VISUAL;
set_desc[2].inp = TEXT_NOMINAL;
set_desc[2].out = TEXT_SHAPED;
```

Add the input/output layout text descriptor pairs. These pairs are in the form of input descriptor and output descriptor; for example, the first statement specifies that the input orientation will be "orientation-left-to-right" and the second statement specifies that the output orientation will be also "orientation-left-to-right". All the above pairs follow the same rule to define the input/output pairs.

```

layout[0].name = ShapeCharset;
layout[0].value = (char *)outShape;

layout[1].name = InputCharset;
layout[1].value = (char *)inShape;

layout[2].name = Orientation;
layout[2].value = (LayoutTextDescriptor)&set_desc[0];

layout[3].name = TypeOfText;
layout[3].value = (LayoutTextDescriptor)&set_desc[1];

layout[4].name = TextShaping;
layout[4].value = (LayoutTextDescriptor)&set_desc[2];

layout[5].name = 0;

```

In the preceding lines, "set_desc" pairs create the new layout values attributes. Each one of these statements will be in the form of attribute_name/attribute_value pairs, for example in the fifth and sixth statements "Orientation" is the attribute name and set_desc[0] (as defined above) is the attribute value. The first two statements are used to declare the output code page and the following two lines are used to specify the input code page.

Call the `m_setvalues_layout()` function.

```

if((error =m_setvalues_layout(plh,layout,&index)))
printf("\n An error %d occurred in setting the value number %d\n",error,index);

```

Invoke `m_setvalues_layout()` using the layout object "plh", the layout values "layout" and an integer "index". If `m_setvalues_layout()` could not set any one of the layout values attributes, it will return -1 in the integer variable called "error", and also return the index of the layout value that caused the problem.

6. Call the `m_transform_layout()` function. The `m_transform_layout()` and `m_wtransform_layout()` functions are the same, except that `m_wtransform_layout()` is used for wide character (`wchar_t`). Both functions will do the actual reordering and shaping of the input buffer using the layout object (plh) created in step 4.
-

```

m_transform_layout(plh,inbuffer,insize,outbuffer,&outsize,NULL,
NULL,NULL,NULL);

```

plh

The Layout Object returned by `m_create_layout()`.

inbuffer

Corresponds to the input string to the function that the layout functions will process.

insize

Gives the input size of the input string specified by the Input Buffer argument.

outbuffer

Any transformed data is stored here. This buffer will contain the data after converting it to the specified output code page.

outsize

Gives the number of bytes in the Output Buffer.

The last four parameters are given here as NULL and they represent Input To Output Mapping, Output To Input Mapping, Property and Input Buffer Index as described above in the Overview of the Layout Functions. Each of these output arguments may be NULL to specify that no output is desired for the specific argument.

7. Call the `m_destroy_layout()` function. This function must be called at the end of the program to destroy the layout object or to free up the allocated memory used by the layout object.

```
m_destroy_layout(plh);
```

Finally, Figure 219 on page 756 is sample program (CCNGBID1) that shows how the bidirectional layout API are used.

```
*****
/* This is a simple program that explains how the layout API's are used */
/* This program will convert a simple implicit unshaped Arabic string */
/* to a visual shaped Arabic string . */
#include <sys/layout.h>
#include <stdio.h>

void main(int argc,char** argv)
{
    LayoutObject plh;
    int error = 0;
    int index;
    LayoutValues layout;
    LayoutTextDescriptor set_desc;
    size_t insize = 9,outsize = 9;

    char *inbuffer=NULL;
    char *outbuffer=NULL;
    char *inShape=NULL;
    char *outShape=NULL;
    char

    *myModifier="@lstypeoftext=implicit:visual,shaping=nominal:shaped,orientation=ltr:rtl";

    inbuffer =(char *)malloc((insize+1)*sizeof(char) ) ;
    outbuffer=(char *)malloc((outsize+1)*sizeof(char)) ;

    layout = (LayoutValues)malloc(6*sizeof(LayoutValueRec));
    set_desc = (LayoutTextDescriptor)malloc(3*sizeof(LayoutTextDescriptorRec));

    inShape = (char*) malloc(8 * sizeof(char));
    outShape = (char*) malloc(8 * sizeof(char));

    inbuffer[0] = 0xB0; /* These are the HEX code for Arabic characters in the IBM-425
codepage */
    inbuffer[1] = 0xB1;
    inbuffer[2] = 0xB2;
    inbuffer[3] = 0xBF;
    inbuffer[4] = 0x40;
    inbuffer[5] = 0x9A;
    inbuffer[6] = 0x75;
    inbuffer[7] = 0x58;
    inbuffer[8] = 0xDC;
```

Example of bidirectional layout API's (Part 1 of 2)

Figure 219. Example of bidirectional layout API's

```

plh = m_create_layout("Ar_AA",myModifier);

strcpy(outShape,"ibm-420");
strcpy(inShape,"ibm-425");

set_desc[0].inp = ORIENTATION_LTR;
set_desc[0].out = ORIENTATION_LTR;

set_desc[1].inp = TEXT_IMPLICIT;
set_desc[1].out = TEXT_VISUAL;

set_desc[2].inp = TEXT_NOMINAL;
set_desc[2].out = TEXT_SHAPED;

layout[0].name = ShapeCharset;
layout[0].value = (char *)outShape;

layout[1].name = InputCharset;
layout[1].value = (char *)inShape;

layout[2].name = Orientation;
layout[2].value = (LayoutTextDescriptor)&set_desc[0];

layout[3].name = TypeOfText;
layout[3].value = (LayoutTextDescriptor)&set_desc[1];

layout[4].name = TextShaping;
layout[4].value = (LayoutTextDescriptor)&set_desc[2];

layout[5].name = 0;

if( error=m_setvalues_layout(plh,layout,&index))
    printf("\n An error %d occurred in setting the value number %d\n",error,index);
m_transform_layout(plh,inbuffer,insize,outbuffer,&outsize,NULL,NULL,NULL,NULL);
m_destroy_layout(plh);

if(inbuffer)
    free(inbuffer);
if(outbuffer)
    free(outbuffer);
if(set_desc)
    free(set_desc);
if(layout)
    free(layout);
if(inShape)
    free(inShape);
if(outShape)
    free(outShape);
}

```

Example of bidirectional layout API's (Part 2 of 2)

Appendix A. Mapping variant characters for z/OS XL C/C++

If you are running a programmable workstation using host emulation software, you must:

1. Remap the hexadecimal values for the variant characters. Remapping will send the values that are correct for the installed z/OS XL C/C++ compiler to the host system. For instructions, see [“Specifying the appropriate code page for the compiler” on page 759](#).
2. Ensure that your current keyboard input generates the hexadecimal values that are expected by the z/OS XL C/C++ compiler and its library. For instructions, see [“Testing the display of variant characters” on page 759](#).
3. Enable your ISPF editor to support local use of variant characters. For instructions, see [“Inserting and viewing square brackets during an ISPF edit session” on page 762](#).

Note: See `LOCALE | NOLOCALE` and other appropriate sections in *z/OS XL C/C++ User's Guide* for information on the option and the list of IBM-supported locales available for use at compile time or run time. The default C locale is supported by code page IBM-1047. Consult your system programmer for the coded character set that your installation uses.

Specifying the appropriate code page for the compiler

To specify the code page based on the compiler version, use the `#pragma filetag` directive conditionally in the source and header file. The syntax is shown below, where *codepage* is the codepage in which the source code is written.

```
??=ifdef __COMPILER_VER__  
  ??=pragma filetag ("codepage")  
??=endif
```

Note: If you are running standard 3270 emulation in the U.S., your workstation software most likely uses code page 37. You can then use this alternative by specifying IBM-037 as *codepage*.

Testing the display of variant characters

If you use a programmable workstation or a 3270 terminal, you can follow [Figure 220 on page 760](#) to ensure that the host system uses the correct hexadecimal values to display the variant characters. See `iconv` in *z/OS XL C/C++ User's Guide* for more information on this utility. See [“Inserting and viewing square brackets during an ISPF edit session” on page 762](#) for instructions to insert square brackets into a C/C++ source file.

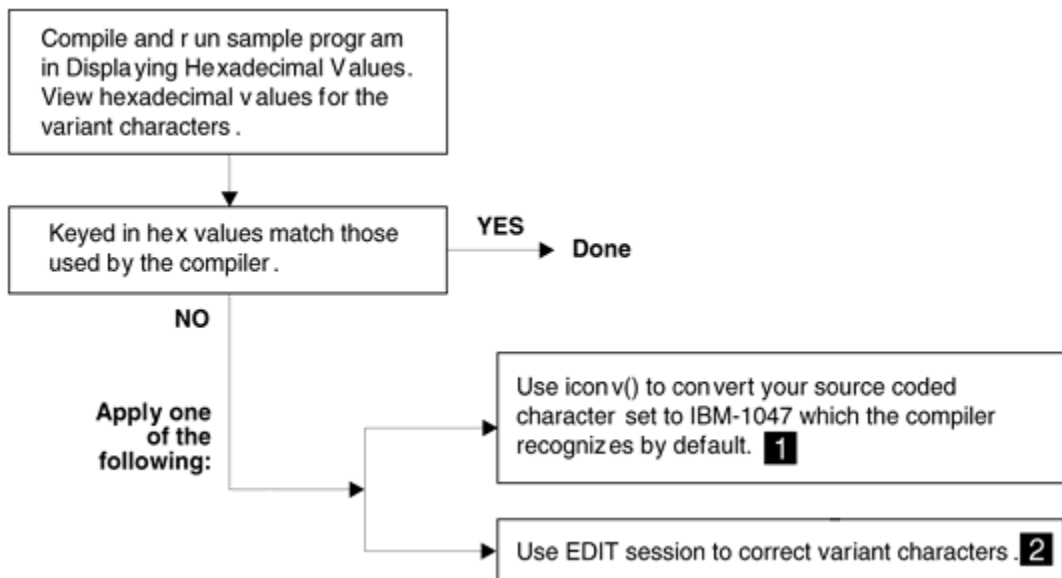


Figure 220. Variant characters

Displaying hexadecimal values

To ensure that your current keys generate the hexadecimal values that are expected by the z/OS XL C/C++ compiler and its library:

1. Create the input file MYFILE.DAT, typed in the following order on a single line, without spaces between them:
 - backslash \
 - right square bracket]
 - left square bracket [
 - right brace }
 - left brace {
 - circumflex ^
 - tilde ~
 - exclamation mark !
 - number sign #
 - vertical line |

Note: These ten variant characters are selected because they are syntactically important to the z/OS XL C/C++ compiler.
2. Run the program [Figure 221 on page 761](#) to display the following information:
 - Selected hexadecimal values for the variant characters that your current setup uses:
 - The values that the compiler and library expect for mapping the keyboard.
3. Perform the following steps until the hexadecimal values for the variant characters that your current setup matches the values that the compiler and library expect for mapping the keyboard.
 - a. Use the values that the compiler and library expect for mapping the keyboard to edit the input file MYFILE.DAT.
 - b. Run the program CCNGMV1 again.

Sample program

CCNGMV1 in [Figure 221 on page 761](#) performs the following actions:

- Reads the ten characters from MYFILE.DAT.
- Queries the current compile time locale for the character values that compiler would expect.
- Generates the codes as shown in the column EXPECTED BY COMPILER.

```

/* this example will display hexadecimal values for the variant */
/* characters */

#include <stdio.h>
#include <locale.h>
#include <variant.h>
#include <stdlib.h>

void read_user_data(char *, int);
void main() {
    char *user_char, *compiler_char;

    struct variant *compiler_var_char;
    int num_var_char, index;
    char *code_set;
    char *char_names[]={ "backslash",
                          "right bracket",
                          "left bracket",
                          "right brace",
                          "left brace",
                          "circumflex",
                          "tilde",
                          "exclamation mark",
                          "number sign",
                          "vertical line"};

    num_var_char=sizeof(char_names)/sizeof(char *);
    if ((user_char=(char*)calloc(num_var_char, 1)) == NULL)
    {
        printf("Error: Unable to allocate the storage\n");
        exit(99);
    }

    read_user_data(user_char, num_var_char);
    /* managed to read the users' characters from the file */

    code_set="default IBM-1047";
    compiler_char="\xe0\xbd\xad\xd0\xc0\x5f\xa1\x5a\x7b\x4f";
    /* standard compiler code page */

    printf("Compiler and library code page is : %s\n\n", code_set);
    printf("Variant character values:\n");
    printf(" %16s      expected by compiler    your current\n", "");
    for (index=0; index<num_var_char; index++)
        printf(" %16s :          %X              %X\n",
               char_names[index], compiler_char[index], user_char[index]);
    exit(0);
}

```

CCNGMV1: Displaying hexadecimal values (Part 1 of 2)

Figure 221. CCNGMV1: Displaying hexadecimal values

```

void read_user_data(char* char_array, int num_var_char)
{
    FILE *stream;
    int num;

    if (stream = fopen ("myfile.dat", "rb"))
        if (! (num = fread(char_array, 1, num_var_char, stream)))
        {
            printf("Error: Unable to read from the file\n");
            exit(99);
        }
        else { ; }
    else
    {
        printf("Error: Unable to open the file\n");
        exit(99);
    }
    fclose(stream);
    return;
}

```

CCNGMV1: Displaying hexadecimal values (Part 2 of 2)

Inserting and viewing square brackets during an ISPF edit session

When your workstation is sending correct hexadecimal values for the square brackets to the host system, you might find that they are still not correctly displayed during by ISPF. If you are using a programmable workstation or a 3270 terminal, you should include the sample ISPF macro CCNGMV2 in a regular CLIST library that is concatenated to the ddname SYSPROC. CCNGMV2 supports ISPF display of the "[" and "]" characters in text, trigraph, or hex form. You can toggle through the three settings.

Note: If you are using 3279-S3G-1 with ISPF, z/OS batch, or TSO, see [“Displaying square brackets when using ISPF on 3279 emulation” on page 763](#).

After you reset the environment by specifying the appropriate code page for the compiler, follow these steps:

1. Start ISPF and open a C or C++ source file with square brackets.
2. Run CCNGMV2 before editing to convert the compiler recognizable hexadecimal values of the square brackets to trigraphs.
3. Run CCNGMV2 again to convert the trigraphs to displayable characters.
4. Edit your C or C++ source code.
5. Run the CCNGMV2 macro again to convert the displayable characters back to the original hexadecimal values.
6. Save and file the C source file.

ISPF macro CCNGMV2

CCNGMV2 in [Figure 221 on page 761](#) is sample ISPF macro for displaying square brackets.

```

/* this ISPF macro can be used to display square brackets in different
/* formats

PROC 0
ISREDIT MACRO

SET RP = &STR()
/* Symbolic values for 6 C language symbols.
/* 1. left bracket, EBCDIC hex value
/* 2. right bracket, EBCDIC hex value
/* 3. left bracket, trigraph
/* 4. right bracket, trigraph
/* 5. left bracket, square
/* 6. right bracket, square
SET LBRACKET_HEX = X'AD'
SET RBRACKET_HEX = X'BD'
SET LBRACKET_TRI = &STR(??(
SET RBRACKET_TRI = &STR(??&RP)
SET LBRACKET_SQR = X'BA' /* LBRACKET_SQR = HEX BA */
SET RBRACKET_SQR = X'BB' /* RBRACKET_SQR = HEX BB */

ISREDIT FIND &LBRACKET_HEX ALL NX
ISREDIT (N1) = FIND_COUNTS
ISREDIT FIND &RBRACKET_HEX ALL NX
ISREDIT (N2) = FIND_COUNTS
IF (&N1 -=& &N2) THEN WRITE .....UNBALANCED HEX BRACKETS
IF (&N1 > 0) THEN DO
    ISREDIT CHANGE &LBRACKET_HEX &LBRACKET_TRI ALL NX
    ISREDIT CHANGE &RBRACKET_HEX &RBRACKET_TRI ALL NX
    EXIT
END

ISREDIT FIND &LBRACKET_TRI ALL NX
ISREDIT (N1) = FIND_COUNTS
ISREDIT FIND &RBRACKET_TRI ALL NX
ISREDIT (N2) = FIND_COUNTS
IF (&N1 -=& &N2) THEN WRITE .....UNBALANCED TRIGRAPH
IF (&N1 > 0) THEN DO
    ISREDIT CHANGE &LBRACKET_TRI &LBRACKET_SQR ALL NX
    ISREDIT CHANGE &RBRACKET_TRI &RBRACKET_SQR ALL NX
    EXIT
END

ISREDIT FIND &LBRACKET_SQR ALL NX
ISREDIT (N1) = FIND_COUNTS
ISREDIT FIND &RBRACKET_SQR ALL NX
ISREDIT (N2) = FIND_COUNTS
IF (&N1 -=& &N2) THEN WRITE .....UNBALANCED SQUARE BRACKETS
IF (&N1 > 0) THEN DO
    ISREDIT CHANGE &LBRACKET_SQR &LBRACKET_HEX ALL NX
    ISREDIT CHANGE &RBRACKET_SQR &RBRACKET_HEX ALL NX
    EXIT
END

```

Figure 222. Sample ISPF macro for displaying square brackets

Displaying square brackets when using ISPF on 3279 emulation

If you are using a 3279-S3G-1 with ISPF, z/OS batch, or TSO, you should have the APL keys on your keyboards.

1. Go to ISPF 0.1 and set the terminal type to 3278A.
2. Open the file that has the square brackets in the ISPF editor.
3. Whenever you want to enter square brackets:
 - a. Press ALT APLon.
 - b. Enter a bracket "[" or "]".
 - "[" is mapped to = X'AD'.
 - "]" is mapped to = X'BD'.
 - c. Press ALT APLoff.

Appendix B. Accessibility

Accessible publications for this product are offered through [IBM Documentation for z/OS \(www.ibm.com/docs/en/zos\)](http://www.ibm.com/docs/en/zos).

If you experience difficulty with the accessibility of any z/OS documentation see [How to Send Feedback to IBM](#) to leave documentation feedback.

Notices

This information was developed for products and services that are offered in the USA or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America*

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan*

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for IBM Documentation. Use of hyperlinks in other output formats of this information is at your own risk.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*IBM Corporation
Site Counsel
2455 South Road*

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 document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or

reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at ibm.com/privacy and IBM's Online Privacy Statement at ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMSdfp, JES2, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those

products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: [IBM Lifecycle Support for z/OS \(www.ibm.com/software/support/systemsz/lifecycle\)](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.SIBM CALL
- CEE.SIBM CAL2
- CEE.SIBMMATH
- CEE.SIBMTASK

Programming interface information

This publication documents *intended* Programming Interfaces that allow the customer to write z/OS XL C/C++ programs.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at [Copyright and Trademark information \(www.ibm.com/legal/copytrade.shtml\)](http://www.ibm.com/legal/copytrade.shtml).

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

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

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

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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

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

Standards

The following standards are supported in combination with the Language Environment element:

- The C language is consistent with *Programming languages - C (ISO/IEC 9899:1999)* and a subset of *Programming languages - C (ISO/IEC 9899:2011)*. For more information, see [International Organization for Standardization \(ISO\) \(www.iso.org\)](http://www.iso.org).
- The C++ language is consistent with *Programming languages - C++ (ISO/IEC 14882:1998)*, *Programming languages - C++ (ISO/IEC 14882:2003(E))*, and a subset of *Programming languages - C++ (ISO/IEC 14882:2011)*.

The following standards are supported in combination with the Language Environment and z/OS UNIX System Services elements:

- A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3)*. For more information, see [IEEE \(www.ieee.org\)](http://www.ieee.org).
- *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.
- *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, copyright 1985 by the Institute of Electrical and Electronic Engineers, Inc.
- *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*, copyright 1994 by The Open Group
- *X/Open CAE Specification, Networking Services, Issue 4*, copyright 1994 by The Open Group
- *X/Open Specification Programming Languages, Issue 3, Common Usage C*, copyright 1988, 1989, and 1992 by The Open Group
- United States Government's *Federal Information Processing Standard (FIPS) publication for the programming language C, FIPS-160*, issued by National Institute of Standards and Technology, 1991

Bibliography

This bibliography lists the publications for IBM products that are related to z/OS XL C/C++. It includes publications covering the application programming task. The bibliography is not a comprehensive list of the publications for these products, however, it should be adequate for most z/OS XL C/C++ users. Refer to [z/OS Information Roadmap](#) for a complete list of publications belonging to the z/OS product.

z/OS

- [z/OS Introduction and Release Guide](#)
- [z/OS Planning for Installation](#)
- [z/OS Release Upgrade Reference Summary](#)
- [z/OS Information Roadmap](#)
- [z/OS Licensed Program Specifications](#)
- [z/OS Upgrade Workflow](#)
- [z/OS Program Directory](#)

z/OS XL C/C++

- [z/OS XL C/C++ Programming Guide](#)
- [z/OS XL C/C++ User's Guide](#)
- [z/OS XL C/C++ Language Reference](#)
- [z/OS XL C/C++ Messages](#)
- [z/OS C/C++ Runtime Library Reference](#)
- [z/OS C Curses](#)
- [z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer](#)
- [Standard C++ Library Reference](#)

z/OS Metal C Runtime Library

- [z/OS Metal C Programming Guide and Reference](#)

z/OS Runtime Library Extensions

- [z/OS Common Debug Architecture User's Guide](#)
- [z/OS Common Debug Architecture Library Reference](#)
- [DWARF/ELF Extensions Library Reference](#)

Debug Tool

- Debug Tool documentation, which is available at [Debug Tool Utilities and Advanced Functions \(www.ibm.com/software/awdtools/debugtool\)](http://www.ibm.com/software/awdtools/debugtool).

z/OS Language Environment

- [z/OS Language Environment Concepts Guide](#)
- [z/OS Language Environment Customization](#)
- [z/OS Language Environment Debugging Guide](#)
- [z/OS Language Environment Programming Guide](#)

- [*z/OS Language Environment Programming Reference*](#)
- [*z/OS Language Environment Runtime Application Migration Guide*](#)
- [*z/OS Language Environment Writing Interlanguage Communication Applications*](#)
- [*z/OS Language Environment Runtime Messages*](#)

Assembler

Assembler documentation, which is available at [High Level Assembler and Toolkit Feature in IBM Documentation \(www.ibm.com/docs/en/hla-and-tf/1.6\)](http://www.ibm.com/docs/en/hla-and-tf/1.6).

COBOL

- COBOL documentation, which is available at the [Enterprise COBOL for z/OS documentation library \(www.ibm.com/support/docview.wss?uid=swg27036733\)](http://www.ibm.com/support/docview.wss?uid=swg27036733).

PL/I

- PL/I documentation, which is available at the [IBM Enterprise PL/I for z/OS library \(www.ibm.com/support/docview.wss?uid=swg27036735\)](http://www.ibm.com/support/docview.wss?uid=swg27036735).

VS FORTRAN

- VS FORTRAN documentation, which is available at the [VS FORTRAN Library \(www.ibm.com/software/awdtools/fortran/vsfortran/library.html\)](http://www.ibm.com/software/awdtools/fortran/vsfortran/library.html).

CICS Transaction Server for z/OS

- CICS Transaction Server for z/OS documentation, which is available at [CICS Transaction Server for z/OS \(www.ibm.com/docs/en/cics-ts\)](http://www.ibm.com/docs/en/cics-ts)

DB2

- DB2 for z/OS documentation, which is available at [Db2 for z/OS in IBM Documentation \(www.ibm.com/docs/en/db2-for-zos\)](http://www.ibm.com/docs/en/db2-for-zos).

IMS/ESA®

- IMS documentation, which is available at [IMS in IBM Documentation \(www.ibm.com/docs/en/ims\)](http://www.ibm.com/docs/en/ims).

MVS

- [*z/OS MVS Program Management: User's Guide and Reference*](#)
- [*z/OS MVS Program Management: Advanced Facilities*](#)

QMF

- QMF documentation, which is available at the [DB2 Query Management Facility Library \(www.ibm.com/support/docview.wss?uid=swg27021603\)](http://www.ibm.com/support/docview.wss?uid=swg27021603).

DFSMS

- [*z/OS DFSMS Introduction*](#)
- [*z/OS DFSMS Managing Catalogs*](#)
- [*z/OS DFSMS Using Data Sets*](#)
- [*z/OS DFSMS Macro Instructions for Data Sets*](#)

- *z/OS DFSMS Access Method Services Commands*

Index

Special Characters

square brackets ([and])
 displaying on workstation or 3270 [759](#)
 displaying square brackets [762](#)
 square brackets [762](#)
__abendcode macro, using for debugging [162](#)
__amrc structure
 debugging I/O programs [161](#)
 example [164](#)
 using with VSAM [98](#), [119](#)
__amrc2 structure
 usage [164](#)
__csplist() library function [597](#)
__last_op codes for __amrc [165](#)
__rsncode macro [162](#), [286](#)
__STDC_CONSTANT_MACROS feature test macro [416](#)
__STDC_FORMAT_MACROS feature test macro [416](#)
__STDC_LIMIT_MACROS feature test macro [416](#)
_24malc() library function [535](#)
_4kmalc() library function [535](#)
_EDC_ERRNO_DIAG environment variable [348](#)
_EDC_GLOBAL_STREAMS environment variable [350](#)
_EDC_IEEEV1_COMPATIBILITY_ENV environment variable [351](#)
_EDC_IO_ABEND environment variable [351](#)
_EDC_IO_TRACE environment variable [352](#)
_EDC_OPEN_CONCAT environment variable [354](#)
_EDC_POPEN environment variable [354](#)
_EDC_PTHREAD_YIELD environment variable [354](#)
_EDC_PTHREAD_YIELD_MAX environment variable [355](#)
_EDC_PUTENV_COPY environment variable [355](#)
_EDC_RRDS_HIDE_KEY environment variable [106](#)
_ICONV_UCS2 environment variable [727](#)
_ICONV_UCS2_PREFIX environment variable [720](#)
_ISOC99_SOURCE feature test macro [415](#)
_LP64 macro
 64-bit [245](#)
_TR1_C99 feature test macro [416](#)
_TZ environment variable [697](#)
_xhotc() library function [531](#)
_xhotl() library function [532](#)
_xhott() library function [532](#)
_xhotu() library function [533](#)
_xregs() library function [533](#)
_xsacc() library function [534](#)
_xsrvc() library function [534](#)
_xusr() library function [535](#)
_xusr2() library function [535](#)
! (exclamation mark) [759](#)
] (right square bracket) and [(left square bracket) [759](#)
{ (left brace) [759](#)
} (right brace) [759](#)
/* (EOF sequence for text terminal) [133](#)
// (double slash), part of MVS data set name [38](#), [101](#)
\a (alarm) [59](#)

\b (backspace) [59](#)
\f (form feed) [59](#)
\n (newline) [59](#)
\r (carriage return) [59](#)
\t (horizontal tab) [59](#)
\v (vertical tab) [59](#)
\x0E (DBCS shift out) [59](#)
\x0F (DBCS shift in) [59](#)
& (ampersand)
 using to specify temporary data set names [38](#)
(number sign) [759](#)
^ (caret) [759](#)
| (vertical bar) [759](#)
~ (tilde) [759](#)

Numerics

24malc() library function [535](#)
32-bit application
 recompiling as 64-bit [223](#)
4kmalc() library function [535](#)
64 bit offsets [90](#)
64-bit
 _LP64 macro [245](#)
 environment [221](#)
 migrating from 32-bit [226](#)
 pointers [236](#)
 printf [242](#)
 structure alignment [227](#)
64-bit virtual memory
 IPA(LINK) [227](#)

A

abend
 CICS and assembler user exit [544](#)
 codes
 CEEBOXITA, CEEAUE_RETC field [543](#)
 specifying those to be percolated [545](#)
 dumps, CEEAUE_DUMP [544](#)
 generating [524](#)
 percolating [541](#), [545](#)
 requesting dump [544](#)
 system [541](#), [545](#)
 TRAP runtime option [541](#)
 user [541](#), [545](#)
ABEND, compiler
 insufficient storage [227](#)
 MEMLIMIT system parameter and IMEMLIM variable [227](#)
acc parameter for fopen()
 memory file I/O [143](#)
 terminal I/O [132](#)
 VSAM data sets [103](#)
 z/OS OS I/O [51](#)
accept(), network example [300](#)
access control list (ACL) [94](#)

- access method selection [48](#)
- accessibility
 - contact IBM [765](#)
- accessing UNIX file system files
 - optimizing [448](#)
- ACL (access control list) [94](#)
- addressing
 - within AF_INET domain [298](#)
 - within AF_INET6 domain [298](#)
 - within AF_UNIX domain [299](#)
 - within sockets [297](#)
- addressing capabilities
 - ILP32 and LP64 [221](#)
- AF_INET domain
 - addressing [298](#)
 - defined [298](#)
- AF_INET6 domain
 - addressing [298](#)
- AF_UNIX domain
 - addressing [299](#)
 - defined [299](#)
- AGGRCOPY compiler option [465](#)
- alarm escape sequence \a [59](#)
- alignment
 - z/OS basic rule [228](#)
- alloca() library function [445](#)
- alternate code point support [733](#)
- AMODE processing option
 - for CEEBXITA user exit [541](#)
- AMODE/RMODE under CICS [575](#), [593](#)
- ANSI C++ 98 applications
 - C99 behavior [415](#)
- ANSIALIAS compiler option [465](#)
- application service routines [509](#)
- application, network [303](#)
- ARCH compiler option [454](#)
- argc under CICS [581](#)
- argv under CICS [581](#)
- arithmetic
 - constructions [435](#)
- ASA (American Standards Association)
 - control characters [25](#)
 - example [25](#)
 - overview [25](#)
- ASCII limitations [645](#)
- asis parameter, fopen()
 - memory file I/O [144](#)
 - terminal I/O [133](#)
 - VSAM data sets [104](#)
 - z/OS OS I/O [51](#)
- assembler
 - assembler user exit for termination of [543](#)
 - system programming alternative [493](#)
- ASSERT(RESTRICT) compiler option [465](#)
- assistive technologies [765](#)
- asynchronous I/O (z/OS) [53](#)
- atoi() library function [446](#)

B

- backspace escape sequence \b [59](#)
- BDAM data sets, restriction [37](#)

- BDW (block descriptor word) [49](#)
- bibliography [773](#)
- bidirectional languages [747](#)
- binary files
 - byte stream behavior [20](#)
 - fixed behavior [13](#)
 - undefined format behavior [19](#)
 - using fseek() and ftell(), OS I/O [66](#)
 - variable behavior [17](#)
- binary floating-point support
 - built-in instructions, hardware [406](#)
- binary I/O, description [8](#)
- bind(), network example [300](#)
- bit fields
 - referencing
 - and optimization [436](#)
- blksize parameter
 - memory file I/O [143](#)
 - terminal I/O [132](#)
 - VSAM data sets [103](#)
 - z/OS OS I/O [50](#)
- blocked
 - files, using fseek() and ftell() [67](#)
- blocked I/O, description [8](#)
- blocked records [12](#)
- BookManager documents [xlili](#)
- buffers
 - full buffering [23](#)
 - line buffering [23](#)
 - multiple [53](#)
 - no buffering
 - memory files [23](#)
 - UNIX file system files [23](#)
 - OS I/O [52](#)
 - terminal I/O [133](#)
 - terminology
 - buffering [23](#)
 - using [23](#)
- BUFNO subparameter, multiple buffering [53](#)
- built-in functions
 - __builtin_expect [444](#)
 - hardware [361](#)
- built-in instructions, hardware
 - binary floating-point instructions [406](#)
 - decimal floating-point [395](#)
 - decimal instructions [389](#)
 - floating-point instructions [394](#)
 - general instructions [361](#)
 - hexadecimal floating-point instructions [405](#)
 - Store Clock Fast [361](#)
 - transaction execution [409](#)
- byte order, network [297](#)
- bytesseek parameter in fopen()
 - effects on OS files [66](#)
 - memory file I/O [144](#)
 - terminal I/O [133](#)
 - VSAM data sets [104](#)
 - z/OS OS I/O [51](#)

C

- C locale

- C locale (*continued*)
 - comparing with POSIX and SAA locales [706](#)
 - defined [699](#)
- C or C++ interlanguage calls
 - with C++ [175](#)
 - with COBOL [175](#)
 - with FORTRAN [175](#)
 - with PL/I [175](#)
- C++
 - optimizing [427](#)
- c99 interfaces
 - exposure to XL C++ applications [415](#)
- C99 support
 - ANSI C and C++ 98 applications [415](#)
- CALL
 - command [597](#)
- calling
 - C from C++ [175](#)
 - C++ from C [175](#)
 - COBOL from C or C++ [175](#)
 - FORTRAN from C or C++ [175](#)
 - PL/I from C or C++ [175](#)
- card
 - punch output [48](#)
 - reader input [48](#)
- carriage return escape sequence \r [59](#)
- case sensitivity
 - linkage specification [178](#)
- catalogued procedure
 - changes for sockets [312](#), [313](#)
 - EDCC sample [313](#)
 - EDCCB sample [312](#)
 - link edit [312](#)
- catalogued procedures
 - IPA Link [227](#)
 - with IMEMLIM variable [227](#)
- catch [273](#)
- CCNGDB4
 - using DB2 with C [613](#)
- CCSID (coded character set id) [93](#)
- cds() library function [445](#)
- cdump() library function [582](#)
- CEEAE_ parameters [541](#)
- CEEBINT HLL user exit
 - customizing [539](#)
 - invoking [538](#)
 - using default version [539](#)
- CEEBXITA assembler user exit
 - abends [541](#)
 - customizing for your installation [539](#)
 - during enclave termination [540](#)
 - during process termination [540](#)
 - effects of runtime options [541](#)
 - error handling [541](#)
 - invoking [538](#), [539](#)
 - using default version [539](#)
- CEESTART
 - creating modules without [496](#)
 - using with MTF [569](#)
- cerr
 - predefined stream, usage [21](#)
- CESE, CICS data queue [153](#)
- CESO, CICS data queue [153](#)
- character shaping [747](#)

- character special files (z/OS UNIX file system)
 - creating [76](#)
 - I/O rules [87](#)
 - using [75](#)
- charmap section [651](#)
- CHARSETID section [652](#)
- CHECKNEW compiler option [466](#)
- CICS (Customer Information Control System)
 - AMODE/RMODE considerations [575](#), [593](#)
 - arguments to C or C++ main() [581](#)
 - C program reentrancy [592](#)
 - cdump() library function [582](#)
 - CESE data queue [153](#)
 - CESO data queue [153](#)
 - clock() library function [582](#)
 - compile [588](#)
 - compiling XL C/C++ programs after preprocessing [592](#)
 - Cross System Product (CSP) [597](#)
 - CSD considerations [594](#)
 - csnap() library function [582](#)
 - ctdli() library function [582](#)
 - ctrace() library function [582](#)
 - define and run the program [594](#)
 - designing and coding a program [576](#)
 - developing and XL C/C++ program [575](#)
 - DLL [582](#)
 - dump functions [582](#)
 - dynamic allocation [581](#)
 - EXEC CICS LINK [582](#)
 - EXEC CICS statements
 - example [589](#)
 - EXEC CICS XCTL [582](#)
 - fetch() library function [582](#)
 - floating point arithmetic [582](#)
 - input and output [153](#)
 - interlanguage support [583](#)
 - iscics() library function [582](#)
 - JCL to translate and compile [592](#)
 - link considerations [594](#)
 - Linking [593](#)
 - linking for reentrancy [593](#)
 - locale support [581](#), [745](#)
 - memory file support [580](#)
 - migrating to a newer translation option [588](#)
 - MTF support [581](#)
 - overview [575](#)
 - packed decimal support [581](#)
 - POSIX support [581](#)
 - preparing for use with z/OS Language Environment [575](#)
 - program processing [594](#)
 - program termination [582](#)
 - reentrancy [594](#)
 - release() library function [582](#)
 - requirements [575](#)
 - runtime options [581](#)
 - SP C support [581](#)
 - standalone CICS translator [589](#)
 - standard stream support [579](#)
 - storage management [583](#)
 - svc99() library function [581](#)
 - system() library function [582](#)
 - translating example [589](#)

- CICS (Customer Information Control System) (*continued*)
 - translating options [588](#)
 - using with IMS [582](#)
 - z/OS XL C/C++ integrated CICS translator [588](#)
 - z/OS XL C/C++ library support [580](#)
- cin
 - predefined stream, usage [21](#)
- CINET [308](#)
- class libraries
 - optimizing [427](#)
- clearenv() library function [334](#)
- clearing memory [445](#)
- client perspective [301](#)
- client/server
 - allocation with socket() [299](#)
 - conversation [299](#)
 - exchanging data [299](#)
 - server perspective [299](#)
- clock() library function [582](#)
- clog
 - predefined stream, usage [21](#)
- closing
 - memory files [150](#)
 - OS I/O files [68](#)
 - terminal files [138](#)
 - the CELQPIPI MSGRTN file [157](#)
 - VSAM data sets [118](#)
- clrmemf() library function
 - memory I/O files [150](#)
- COBOL
 - assembler user exit [540](#)
 - using linkage specifications [175](#)
- code
 - independence [562](#)
 - motion [452](#)
- coded character set
 - CICS support [575](#)
 - considerations with locale [731](#)
 - conversion during compile [740](#)
 - conversion utilities [707](#)
 - converters supplied [709](#)
 - IBM-1047
 - converting code to [735](#)
 - IBM-1047 vs. IBM-293 [732](#)
 - independence [735](#)
 - related to compile-edit cycle [735](#)
- coded character set id [93](#)
- command
 - syntax diagrams xxxv
- common expression elimination [451](#)
- Common INET [308](#)
- Common Programming Interface (CPI) [633](#)
- communication, network basics [293](#)
- communications, interprocess
 - asynchronous signal delivery [281](#)
 - z/OS TCP/IP considerations [310](#)
- COMPACT compiler option [465](#)
- compile-edit cycle related to coded character set [735](#)
- compiler diagnostics
 - ensuring code portability [224](#)
- compiler options
 - LOCALE compiler option [759](#)
- compiling
 - for a locale [740](#)

- compiling (*continued*)
 - include files [311](#)
 - linking [309](#)
 - procedures [309](#)
 - sockets programs [309](#)
 - under batch
 - for Berkeley Sockets [312](#)
 - for X/Open Sockets [313](#)
 - with X Windows [313](#)
 - using c89
 - for Berkeley Sockets [313](#)
 - with X Windows [313](#)
- COMPRESS compiler option [465](#)
- computational independence [556](#), [562](#)
- concatenation
 - compatibility rules [44](#)
 - in-stream data sets [45](#)
 - sequential and partitioned [43](#)
- condition variable [248](#)
- configuration file access, z/OS TCP/IP [311](#)
- constant
 - propagation [452](#)
- constructed reentrancy
 - and XPLINK [262](#)
- contact
 - z/OS [765](#)
- control characters
 - ASA text files [25](#)
 - OS I/O text files [59](#)
 - terminal I/O files [136](#)
 - z/OS XL C/C++ recognized by text files [7](#)
- conversation [299](#)
- conversions
 - 32-bit to 64-bit [237](#)
 - code set [707](#)
 - hybrid code to IBM-1047 [735](#)
 - integers [237](#)
 - pointers [237](#)
- convert pragma [743](#)
- converters, locale code set [709](#)
- cout
 - predefined stream, usage [21](#)
- CPI (Common Programming Interface) [633](#)
- creat() library function [75](#)
- cs() library function [445](#)
- CSECT (control section)
 - CEESTART [496](#)
- csid() library function [642](#)
- csnap() library function [582](#)
- CSP (Cross System Product) [597](#)
- CSP/AD (Cross System Product/Application Development) [597](#)
- CSP/AE (Cross System Product/Application Execution) [597](#)
- csplist library function
 - passing parameters [597](#)
- ctdli() library function [582](#)
- ctrace() library function [582](#)
- cursive languages [747](#)
- CVFT compiler option [465](#)
- CXIT control block [541](#)

D

- DASD (Direct-Access Storage Device)
 - input and output [37](#)
 - multivolume data sets, input and output [47](#)
 - sequential and partitioned concatenation [43](#)
 - striped data sets, input and output [47](#)
- data alignment
 - 64-bit [239](#)
- data independence [556](#), [562](#)
- data models
 - ILP32 and LP64 [221](#)
- data sets
 - in-stream [45](#)
 - large format sequential [47](#)
 - multivolume [47](#)
 - name
 - opening a memory file [142](#)
 - opening an MVS data set [100](#)
 - opening z/OS OS files [37](#)
 - sequential vs. partitioned concatenation [43](#)
 - striped [47](#)
 - temporary [38](#)
- data structures
 - rule of alignment [228](#)
- data type sizes
 - ILP32 and LP64 [221](#)
- data types
 - referencing bit fields
 - and optimization [436](#)
- datagram
 - definition [294](#)
 - sockets [296](#)
- DB2
 - codepage [611](#)
 - DB2 C/C++ precompiler [612](#)
 - host variables
 - example [614](#)
 - invoking DB2 services with z/OS XL C/C++ [612](#)
 - locale support [745](#)
 - preprocessor directives [611](#)
 - stored procedures and XPLINK [612](#)
 - variable-length source input [611](#)
 - when NOSQL is the default compiler option [611](#)
 - with z/OS XL C/C++
 - examples [612](#)
 - XL C/C++ DB2 coprocessor [611](#)
- DBCS (Double-Byte Character Support)
 - input and output functions [29](#)
 - reading [30](#)
 - shift in character [59](#)
 - shift out character [59](#)
 - writing [31](#)
- DCB (Data Control Block)
 - OS I/O [53](#)
 - parameter on a DD statement [40](#)
 - parameters, optimizing code [447](#)
- ddname
 - opening a z/OS UNIX System Services file system I/O file under MVS [78](#)
 - opening an OS I/O file under z/OS [39](#)
- dead code elimination [452](#)
- dead store elimination [452](#)
- Debug Tool
 - (continued)*
 - CEEBINT and [551](#)
 - debugging I/O programs [161](#)
 - decimal floating-point support
 - built-in instructions, hardware
 - biased exponent definitions [404](#)
 - FPC register-rounding macros [403](#)
 - macros [403](#)
 - Test Data Class masks [404](#)
 - Test Data Group masks [405](#)
 - declarations
 - and optimization
 - referencing bit fields [436](#)
 - extern, using for linkage to other languages [175](#)
 - default
 - C locales for POSIX, SAA, and S370 [699](#)
 - DCB attributes for SYSOUT data set [46](#)
 - locales [699](#), [706](#)
 - definition side-deck [192](#), [193](#)
 - delete
 - named module from storage [526](#)
 - optimizing [427](#)
 - pipes with z/OS UNIX System Services file system [85](#)
 - VSAM records [108](#)
 - z/OS UNIX System Services file system files [83](#)
 - delimiter in JCL statements [45](#)
 - delivery, signals
 - asynchronous [281](#)
 - ISO C rules [279](#)
 - POSIX rules [279](#)
 - differences among C, POSIX, and SAA locales [706](#)
 - direct processing [106](#)
 - directories (z/OS UNIX file system)
 - creating [76](#)
 - deleting [83](#)
 - using [75](#)
 - disabled signals [282](#)
 - disjoint pragma [438](#)
 - DISP=MOD specification, DD statement
 - OS I/O, fopen() modes [39](#)
 - DL/I (Data Language I) [625](#)
 - DLL code [199](#)
 - DLLs (Dynamic Link Libraries)
 - applications [182](#)
 - binding a DLL [192](#)
 - binding a DLL application [193](#)
 - C example [206](#), [212](#)
 - C++ example [210](#)
 - calling explicitly [183](#)
 - calling implicitly [183](#)
 - CICS [582](#)
 - compatibility with non-DLL [202](#)
 - Complex
 - assigning pointers [203](#)
 - compatibility issues [202](#)
 - creating [199](#)
 - guidelines [201](#)
 - modifying source [200](#)
 - creating
 - C [190](#)
 - description [190](#)
 - export pragma [191](#)

- DLLs (Dynamic Link Libraries) (*continued*)
 - creating (*continued*)
 - exporting functions [191](#)
 - guidelines [201](#)
 - entry point [196](#)
 - example [194](#)
 - freeing [190](#)
 - load-on-call [183](#)
 - loading [188](#)
 - managing the use of [188](#)
 - performance [197](#)
 - restrictions [196](#)
 - sharing among application executable files [189](#)
 - using [193](#)
- domain
 - AF_INET [298](#)
 - AF_UNIX [299](#)
- DSQCOMM.C header file [633](#)
- DUMMY data set output [48](#)
- dumps
 - requesting in the CEEBXITA assembler user exit [540](#), [544](#)
- duplicate alternate index keys
 - retrieval sequence [106](#)
 - under VSAM [103](#)
- DWS (Data Window Services) [609](#)
- DXFR, transfer control [597](#)
- dynamic memory [481](#)

E

- EDCCB
 - cataloged procedure [312](#), [313](#)
 - changes for sockets [312](#), [313](#)
 - sample [312](#), [313](#)
- EDCDPLNK macro [264](#)
- EDCDXD macro [264](#)
- EDCLA macro [264](#)
- EDCRCINT routine [500](#)
- EDCX4KGT routine [525](#)
- EDCXABND routine [524](#)
- EDCXABRT module
 - using during link edit [498](#)
- EDCXABRT routine [500](#), [505](#)
- EDCXENV module [505](#)
- EDCXENVL module [505](#)
- EDCXEXIT module
 - exit(), system programming version [505](#), [509](#), [523](#)
 - freestanding applications [500](#)
- EDCXFREE routine [526](#)
- EDCXGET routine [524](#)
- EDCXHOTC library function [531](#)
- EDCXHOTC routine [509](#)
- EDCXHOTL library function [532](#)
- EDCXHOTL routine [509](#)
- EDCXHOTT library function [532](#)
- EDCXHOTT routine [509](#)
- EDCXHOTU library function [533](#)
- EDCXHOTU routine [509](#)
- EDCXISA module
 - entry point [498](#)
 - in freestanding applications [500](#)
- EDCXLANE module [527](#)
- EDCXLANU module [527](#)

- EDCXLOAD routine [526](#)
- EDCXMEM module
 - freestanding applications [500](#)
 - persistent environment [509](#)
 - system programming memory management [505](#), [523](#)
- EDCXREGS library function [533](#)
- EDCXSAAC library function [534](#)
- EDCXSAAC routine
 - accepting a request for service [522](#)
- EDCXSPRT module
 - in freestanding applications [500](#)
 - sprintf(), system programming version [509](#)
 - sprintf(), system programming version of [505](#)
 - System programming version of sprintf() [523](#)
- EDCXSRC routine
 - xsrc library function [534](#)
- EDCXSRVC routine [523](#)
- EDCXSRVN routine
 - initiating a server request [522](#)
- EDCXSTRL module
 - in freestanding applications [500](#)
 - usage [497](#)
- EDCXSTRT module
 - in freestanding applications [500](#)
 - usage [497](#)
- EDCXSTRX module
 - in freestanding applications [500](#)
 - usage [497](#)
- EDCXUNLD routine [526](#)
- EDCXUSR library function [535](#)
- EDCXUSR2 library function [535](#)
- ELPA (Extended Link Pack Area) [262](#)
- empty records
 - _EDC_ZERO_RECLEN [18](#), [359](#)
- enabled signals [282](#)
- enclave
 - terminating with CEEAUE_ABND [544](#)
- encoded offset [66](#)
- ENGLISH runtime messages [527](#)
- Enhanced ASCII
 - limitations of [645](#)
- environment
 - 64-bit [221](#)
- environment variables
 - _BPXK_AUTOCVTS [327](#)
 - _BPXK_CCIDS [328](#)
 - _BPXK_PCCSID [329](#)
 - _BPXK_SIGDANGER [329](#)
 - _CEE_CONDWAIT_PAUSE [336](#)
 - _CEE_DLLLOAD_XPCOMPAT [337](#)
 - _CEE_DMPTARG [337](#)
 - _CEE_ENVFILE [338](#)
 - _CEE_ENVFILE_S [339](#)
 - _CEE_HEAP_MANAGER [340](#)
 - _CEE_RUNOPTS [342](#)
 - _EDC_ADD_ERRNO2 [343](#)
 - _EDC_ANSI_OPEN_DEFAULT [54](#), [344](#)
 - _EDC_AUTO_MAP64 [344](#)
 - _EDC_AUTOCVT_BINARY [344](#)
 - _EDC_BYTE_SEEK [51](#), [66](#), [345](#)
 - _EDC_C99_NAN [346](#)
 - _EDC_CLEAR_SCREEN [136](#), [345](#)
 - _EDC_COMPAT [345](#)

environment variables (*continued*)

- [_EDC_CONTEXT_GUARD 346](#)
- [_EDC_DLL_DIAG 347](#)
- [_EDC_EOVERFLOW 348](#)
- [_EDC_ERRNO_DIAG 348](#)
- [_EDC_GLOBAL_STREAMS 350](#)
- [_EDC_IO_ABEND 351](#)
- [_EDC_IO_TRACE 352](#)
- [_EDC_OPEN_CONCAT 354](#)
- [_EDC_POPEN 354](#)
- [_EDC_PTHREAD_YIELD 354](#)
- [_EDC_PTHREAD_YIELD_MAX 355](#)
- [_EDC_PUTENV_COPY 355](#)
- [_EDC_RRDS_HIDE_KEY 356](#)
- [_EDC_SIG_DFLT 356](#)
- [_EDC_STOR_INCREMENT 356](#)
- [_EDC_STOR_INCREMENT_B 357](#)
- [_EDC_STOR_INITIAL 357](#)
- [_EDC_STOR_INITIAL_B 357](#)
- [_EDC_STRPTM_STD 358](#)
- [_EDC_SUSV3 358](#)
- [_EDC_ZERO_RECLEN 359](#)
- [_ICONV_UCS2 727](#)
- [_ICONV_UCS2_PREFIX 720](#)
- [BIDIATTR 327](#)
- [BIDION 327](#)
- [locale 332](#)
- [naming conventions 335](#)
- [using 334](#)

EOF (end of file)

- [resetting terminal I/O 133](#)

ERRCOUNT runtime option [279](#)

errors, debugging [286](#)

ESCON channels, striped data sets [47](#)

ESDS (Entry-Sequenced Data Set)

- [alternate index keys 98](#)
- [use of 95](#)

established signals [282](#)

examples

- [ccngas1 25](#)
- [ccngbid1 756](#)
- [ccngcc2 738](#)
- [ccngch1 275](#)
- [ccngch2 276](#)
- [ccngci1 577](#)
- [ccngci3 589](#)
- [ccngcl1 693](#)
- [ccngcl2 695](#)
- [ccngcl3 695](#)
- [ccngcp1 598](#)
- [ccngcp2 600](#)
- [ccngcp3 601](#)
- [ccngcp4 602](#)
- [ccngcp5 604](#)
- [ccngcp6 605](#)
- [ccngcp7 607](#)
- [CCNGDB4 613](#)
- [ccngdi1 164](#)
- [ccngdi2 169](#)
- [ccngdl1 705](#)
- [ccngdw1 610](#)
- [ccngdw2 609](#)

examples (*continued*)

- [ccngec1 291](#)
- [ccngev1 359](#)
- [ccnggd1 619](#)
- [ccnggd2 622](#)
- [ccnghf1 84](#)
- [ccnghf2 86](#)
- [ccnghf3 88](#)
- [ccnghf4 90](#)
- [ccngim1 627](#)
- [ccngim2 629](#)
- [ccngim3 630](#)
- [ccngmf1 145](#)
- [ccngmf2 146](#)
- [ccngmf3 152](#)
- [ccngmf4 152](#)
- [ccngmt1 567](#)
- [ccngmt2 568](#)
- [ccngmt3 568](#)
- [ccngmv1 760](#)
- [ccngmv2 762](#)
- [ccngop1 455](#)
- [ccngop3 435](#)
- [ccngos1 41](#)
- [ccngos2 41](#)
- [ccngos3 62](#)
- [ccngos4 70](#)
- [ccngqm1 633](#)
- [ccngqm2 635](#)
- [ccngqm3 636](#)
- [ccngre1 263](#)
- [ccngre2 264](#)
- [ccngre3 265](#)
- [ccngre4 266](#)
- [ccngsp1 499](#)
- [ccngsp2 499](#)
- [ccngsp3 503](#)
- [ccngsp4 506](#)
- [ccngsp5 507](#)
- [ccngsp6 510](#)
- [ccngsp7 512](#)
- [ccngsp8 516](#)
- [ccngsp9 517](#)
- [ccngspa 524](#)
- [ccngspb 525](#)
- [ccngspc 526](#)
- [ccngspd 518](#)
- [ccngspe 520](#)
- [ccngspf 521](#)
- [ccngth1 253](#)
- [ccngvs1 98](#)
- [ccngvs2 119](#)
- [ccngvs3 125](#)
- [ccngvs4 128](#)
- [ccngci2 585](#)
- [machine-readable xliv](#)
- [naming of xliv](#)
- [softcopy xliv](#)
- [using DB2 with C 613](#)

exception handling

- [C exceptions under C++ 273](#)
- [C-IMS 625](#)
- [C++-IMS 625](#)
- [CEEEXITA assembler user exit 541](#)

- exception handling (*continued*)
 - description [273](#)
 - hardware exceptions under C++ [274](#)
 - optimizing [427](#)
- EXEC CICS commands
 - FREEMAIN [583](#)
 - GETMAIN [583](#)
 - how to use [576](#)
 - LINK [582](#)
 - RETURN [583](#)
 - WRITEQ TD [166](#)
 - XCTL [582](#)
- exec family of functions
 - data definition considerations [78](#)
 - described [448](#)
- execution_frequency pragma [438](#)
- EXH compiler option [465](#)
- export pragma [438](#)
- EXPORTALL compiler option [465](#)
- exporting functions [182](#)
- exporting source to other sites [744](#)
- expressions, optimizing
 - recommendations [434](#)
- extern declaration
 - using for linkage to other languages [175](#)
- external
 - static [262](#)
 - variables [432](#), [436](#)

F

- F-format records [12](#)
- families
 - address [297](#)
 - socket [296](#)
- fclose() library function
 - _EDC_COMPAT environment variable [346](#)
- fcntl() library function [75](#)
- fdelrec() library function
 - using to delete records [100](#), [108](#)
- feature model [413](#)
- fetch() library function
 - and writable statics [259](#)
 - calling other z/OS XL C/C++ modules in C [437](#)
 - system programming C environment [495](#)
 - under CICS [582](#)
- fflush() library function
 - _EDC_COMPAT environment variable [346](#)
 - optimizing code [448](#)
- fgetpos() library function
 - _EDC_COMPAT environment variable [346](#)
 - optimizing code [448](#)
- fgets() library function
 - optimizing code [447](#)
- fgetwc_unlocked() library function [30](#)
- fgetwc() library function [30](#)
- fgetws_unlocked() library function [30](#)
- fgetws() library function [30](#)
- FIFO
 - mkfifo() [83](#), [84](#)
 - special files
 - creating [76](#)
 - using [75](#), [84](#)

- File I/O trace
 - Locating the file I/O trace [171](#)
 - Sample file I/O trace [170](#)
- files
 - conversion [93](#)
 - large support [90](#)
 - memory
 - closing [150](#)
 - extending [149](#)
 - flushing [149](#)
 - opening [141](#)
 - positioning [149](#)
 - reading [147](#)
 - repositioning [149](#)
 - writing [148](#)
 - named pipe [84](#)
 - origin of OS attributes [53](#)
 - OS
 - flushing [62](#)
 - opening [37](#)
 - reading from [55](#)
 - removing [70](#)
 - renaming [70](#)
 - repositioning [64](#), [68](#)
 - writing to [57](#)
 - tagging [93](#)
 - VSAM
 - closing [118](#)
 - deleting a record [108](#)
 - flushing [110](#)
 - locating a record [108](#)
 - reading a record [105](#)
 - repositioning [108](#)
 - updating a record [107](#)
 - writing a record [106](#)
 - z/OS, opening [37](#)
- filetag pragma [737](#)
- fixed-format records
 - overview [12](#)
 - standard format [12](#)
- fldata() library function
 - memory file I/O [150](#)
 - OS I/O files [71](#)
 - terminal I/O [138](#)
 - z/OS UNIX System Services file system I/O [92](#)
- FLOAT compiler option [465](#)
- floating-point support
 - binary [406](#)
 - built-in instructions [394](#)
 - decimal [395](#)
 - hexadecimal [405](#)
- flocate() library function
 - VSAM data sets [98](#), [108](#)
- flushing
 - binary streams, wide character I/O [34](#)
 - buffers for terminal files [137](#)
 - memory files [149](#)
 - OS I/O files [62](#)
 - terminal files [137](#)
 - text streams, wide character I/O [33](#)
 - the CELQPIPI MSGRTN file [157](#)
 - VSAM data sets [110](#), [116](#)

- flushing (*continued*)
 - z/OS Language Environment message file [156](#)
 - z/OS UNIX System Services file system records [82](#)
- fopen() library function
 - list of parameters, for
 - memory file I/O [143](#)
 - terminal I/O [131](#)
 - VSAM I/O [102](#)
 - z/OS OS I/O [49](#)
 - z/OS UNIX System Services file system I/O [79](#)
 - under MTF [572](#)
 - z/OS UNIX System Services file system files [75](#)
- for statement [435](#)
- fork() library function
 - data definition considerations [78](#)
 - using with memory files [448](#)
- form feed escape sequence [\f 59](#)
- Format-D files restriction [11](#)
- fputc() library function
 - optimizing code [447](#)
- fputs() library function
 - optimizing code [447](#)
- fputc_unlocked() library function [31](#)
- fputwc() library function [31](#)
- fputws_unlocked() library function [31](#)
- fputws() library function [31](#)
- fread() library function
 - optimizing code [447](#), [448](#)
- FREE=CLOSE parameter, DD statement [39](#)
- freestanding applications
 - EDCXISA [498](#)
 - EDCXSTRL [497](#)
 - EDCXSTRT [497](#)
 - EDCXSTRX [497](#)
- freopen() library function
 - noseek parameter
 - in-stream data sets [45](#)
 - under MTF [572](#)
 - VSAM data sets [100](#)
 - z/OS UNIX System Services file system files [75](#)
- fseek() library function
 - _EDC_COMPAT environment variable [346](#)
 - optimizing code [447](#)
- fsetpos() library function
 - optimizing code [448](#)
- fstream class [22](#)
- ftell() library function
 - _EDC_COMPAT environment variable [346](#)
- full buffering [23](#)
- function prototypes
 - default linkage [178](#)
- functions
 - arguments [433](#)
 - descriptors [181](#)
 - exported [182](#)
 - imported [182](#)
- fupdate() library function [100](#), [107](#)
- fwprintf_unlocked() library function [31](#)
- fwprintf() library function [31](#)
- fwrite() library function
 - optimizing code [447](#), [448](#)

- fwscanf_unlocked() library function [30](#)
- fwscanf() library function [30](#)

G

- GDDM (Graphical Data Display Manager)
 - interface [619](#)
 - with z/OS XL C/C++ [619](#)
- GDG (Generation Data Group)
 - C example [41](#)
 - C++ example [41](#)
 - input and output [40](#)
- genxlt utility [707](#)
- getenv() library function [334](#)
- getsyntax() library function [642](#)
- getwc_unlocked() library function [30](#)
- getwc() library function [30](#)
- getwchar_unlocked() library function [30](#)
- getwchar() library function [30](#)
- global assembler user exit [539](#)
- global variables [432](#)
- graph coloring register allocation [452](#)
- graphics support [619](#)

H

- hard-coding [733](#)
- hardware built-in functions [361](#)
- hardware model [413](#)
- hardware signals [282](#)
- HEAP runtime option
 - system programming C environment [495](#)
- hexadecimal floating-point support
 - built-in instructions, hardware [405](#)
- hexadecimal values
 - keyboard, mapping variant characters [759](#)
- HGPR compiler option [466](#)
- high-level
 - qualifier
 - defaults [39](#), [142](#)
 - running without RACF [39](#), [142](#)
 - setting the user prefix under TSO [39](#), [142](#)
- hiperspace memory files
 - input and output [141](#)
 - specifying buffer size, setvbuf() [141](#)
- horizontal tab escape sequence [\t 59](#)
- HOT compiler option [458](#)
- hybrid coded character set, using [733](#)

I

- I/O
 - binary stream [8](#)
 - blocked [8](#)
 - card input and output [48](#)
 - CICS [153](#), [579](#)
 - debugging [161](#)
 - DUMMY data set output [48](#)
 - errors [161](#)
 - hiperspace memory files [141](#)
 - in-stream data sets [45](#)
 - low-level z/OS UNIX System Services [88](#)
 - memory file [141](#)

I/O (*continued*)

- multivolume data sets [47](#)
- object-oriented [21](#)
- optical reader input [48](#)
- OS [37](#)
- pipe [83](#)
- printer output [48](#)
- record
 - introduction [8](#)
 - model [11](#)
 - rules, z/OS UNIX System Services file system [79](#)
- striped data sets [47](#)
- sysout data set [46](#)
- tapes [46](#)
- terminal [131](#)
- text stream [7](#)
- wide characters [29](#)
- z/OS Language Environment message file [155](#)
- z/OS UNIX System Services file system
 - functions [88](#)
 - using with I/O [75](#)
- i/o stream libraries
 - optimizing [449](#)
- I/O Streams File I/O [21](#)
- IBM-1047 (APL 293), CICS [575](#)
- IBM-1047 coded character set
 - converting code to [735](#)
- iconv utility
 - converting code sets [707](#)
 - preparing source code for exporting [744](#)
- iconv() library function [708](#)
- IEBGENER utility (TSO)
 - tape files [46](#)
- IEEE Binary Floating-Point [267](#)
- IEEE Decimal Floating-Point [267](#)
- IEFUSI exit routine
 - MEMLIMITvalue [227](#)
- if statement [436](#)
- ifstream class [22](#)
- IGNERRNO compiler option [466](#)
- ILP32
 - and LP64 [221](#)
- ILP32 to LP64 migrations
 - alignment differences [228](#)
 - alignment issues [227](#)
 - assignment issues [232](#)
 - availability of suboptions [227](#)
 - conditional compiler directives [245](#)
 - conversions between int and pointer [237](#)
 - converters [246](#)
 - customized locales [246](#)
 - debugging [245](#)
 - ensuring portability [224](#)
 - explicit types [243](#)
 - function prototypes [245](#)
 - header files [236, 243](#)
 - localedef utility [246](#)
 - locales [246](#)
 - LONG_MAX [242](#)
 - padding [243](#)
 - pointer cast conversions [238](#)
 - pointer declarations [236](#)

ILP32 to LP64 migrations (*continued*)

- portability issues [226, 232](#)
- portable coding [243](#)
- post-migration activities [224](#)
- pre-migration activities [223](#)
- precision [238](#)
- SAA [246](#)
- shared structures [239, 243](#)
- suffixes [243](#)
- type definitions [243](#)
- unsuffixed numbers [241](#)
- IMEMLIM variable
 - to override the MEMLIMIT default [227](#)
- IMS (Information Management System)
 - default high-level qualifier [39, 142](#)
 - error handling [625](#)
 - opening files [39, 142](#)
 - other considerations [626](#)
 - using with CICS [582](#)
 - with z/OS XL C/C++ [625](#)
- in-stream data sets
 - delimiter for data [45](#)
 - input [45](#)
 - noseek parameter [45](#)
- include files
 - with z/OS UNIX System Services sockets [311](#)
- INCLUDE statement, MVS [539](#)
- INFO compiler option
 - ensuring portability to LP64 [225](#)
- initialization
 - nested enclave
 - CEEEXITA's function code for [543](#)
 - using CEEEXITA [539](#)
- inlining
 - optimization [455](#)
 - suggestions [455](#)
 - under IPA [457](#)
- installation-wide assembler user exit [539](#)
- instruction scheduling [452](#)
- integer constants
 - 64-bit [238](#)
- interface
 - CICS [575](#)
 - DB2 [611](#)
 - DWS [609](#)
 - GDDM [619](#)
 - IMS [625](#)
 - locale-sensitive [642](#)
- interlanguage calls
 - linkage specification [178](#)
 - using linkage specifications [175](#)
- international enabling
 - for programming languages [641](#)
 - z/OS XL C/C++ support for [642](#)
- internationalization
 - display of hexadecimal values [759](#)
- Internet address [297](#)
- internetworking concepts [293](#)
- interprocess communication
 - asynchronous signal delivery [281](#)
 - z/OS TCP/IP considerations [310](#)
- INTRDR, using to create job stream within a program [46](#)
- ios class [21](#)

- iostream
 - optimizing [427](#)
- iostream header file [21](#)
- iostream.h header file [21](#)
- IPA
 - flow of processing
 - IPA [461](#)
 - IPA Compile step [461](#)
 - IPA Link step [463](#)
 - non-IPA [461](#)
- IPA(LINK) compiler option
 - as of z/OS V1R8 XL C/C++ [227](#)
- isalnum() macro [445](#)
- isalpha() macro [445](#)
- ISAM data sets, restriction [37](#)
- ISASIZE runtime option
 - system programming C environment [495](#)
- iscics() library function [582](#)
- isctrl() macro [445](#)
- isdigit() macro [445](#)
- ISearchByClassExample [744](#)
- isgraph() macro [445](#)
- islower() macro [445](#)
- isolated_call [438](#)
- isprint() macro [445](#)
- ispunct() macro [445](#)
- isspace() macro [445](#)
- isupper() macro [445](#)
- isxdigit() macro [445](#)

J

- JCL procedures
 - 64-bit virtual memory [227](#)
 - setting MEMLIMIT value [227](#)

K

- KANJI runtime messages [527](#)
- keyboard
 - navigation [765](#)
 - PF keys [765](#)
 - shortcut keys [765](#)
- keyboard, mapping variant characters [759](#)
- KSDS (Key-Sequenced Data Set)
 - alternate index, under VSAM [98](#)
 - description [95](#)

L

- LANGLVL compiler option [466](#)
- large format sequential data sets [47](#)
- LC_ALL locale variable [653](#)
- LC_COLLATE locale variable [653](#)
- LC_CTYPE locale variable [653](#)
- LC_MONETARY locale variable [653](#)
- LC_NUMERIC locale variable [653](#)
- LC_SYNTAX locale variable [673](#)
- LC_TIME locale variable [653](#)
- LC_TOD locale category [697](#)
- LC_TOD locale variable [653](#)
- leaves pragma [438](#)

- LIBANSI compiler option [466](#)
- library extensions [437](#)
- library lookasides
 - optimizing [483](#)
- line buffering [23](#)
- linear data sets [95](#)
- link edit [312](#)
- link files (z/OS UNIX file system), creating [75](#)
- link pack areas
 - optimizing [483](#)
- link() library function [76](#)
- linkage editor, CICS [575](#)
- linking
 - case sensitivity [178](#)
 - function prototypes [178](#)
 - kinds of linkage [176](#)
 - sockets programs [309](#)
 - syntax [175](#)
- listen(), network example [300](#)
- listings, locale sensitive [741](#)
- loading
 - named module into storage [526](#)
 - VSAM data sets [106](#)
- local
 - constant propagation [451](#)
 - expression elimination [451](#)
 - variables [432](#)
- localdtconv() library function [642](#)
- locale
 - ASCII method files [675](#)
 - C [699](#)
 - categories
 - LC_ALL [653](#)
 - LC_COLLATE locale variable [653](#)
 - LC_MONETARY locale variable [653](#)
 - LC_NUMERIC locale variable [653](#)
 - LC_SYNTAX locale variable [673](#)
 - LC_TIME locale variable [653](#)
 - LC_TOD locale variable [653](#)
 - LC_TYPE locale variable [653](#)
 - CICS support [575](#)
 - compiler option examples [740](#)
 - converting existing work [745](#)
 - customizing [691](#)
 - environment variables [332](#)
 - generating an object module [741](#)
 - hybrid coded character set, using [733](#)
 - library functions
 - localdtconv() [642](#)
 - localeconv() [642](#)
 - setlocale() [642](#)
 - LOCALE compiler option [740](#)
 - localeconv() library function [653](#)
 - macros [737](#)
 - overview of z/OS XL C/C++ support [642](#)
 - predefined [739](#)
 - source-code functions summary [736](#)
 - summary of support in compiler [741](#)
 - tests for SAA or POSIX [706](#)
 - TZ or _TZ environment variable [697](#)
 - using with CICS [581](#)
- LOCALE compiler option
 - display of hexadecimal values [759](#)

- localeconv() library function [642](#)
- localedef utility [246](#)
- locales
 - display of hexadecimal values [759](#)
 - under ILP32 and LP64 [246](#)
- locales, customized [246](#)
- loop optimization [469](#)
- loop statements, optimizing [435](#)
- low-level z/OS UNIX System Services I/O [88](#)
- LP64
 - and ILP32 [221](#)
- LP64 environment
 - advantages and disadvantages [222](#)
 - application performance and program size [222](#)
 - migrating applications to [223](#)
 - pointer assignment [237](#)
 - restrictions [223](#)
- LP64 strategy [223](#)
- LPA (Link Pack Area) [262](#)
- LRECL (logical record length) parameter
 - fopen() library function
 - memory file I/O [143](#)
 - terminal I/O [132](#)
 - VSAM data sets [103](#)
 - z/OS OS I/O [50](#)
 - lrecl=X, OS I/O [50](#)

M

- m_create_layout() library function [750](#)
- m_destroy_layout() library function [753](#)
- m_getvalues_layout() library function [751](#)
- m_setvalues_layout() library function [751](#)
- m_transform_layout() library function [751](#)
- m_wtransform_layout() library function [752](#)
- machine print-control codes [12](#)
- macros
 - use with locale [737](#)
- main task for MTF [555](#)
- mainframe
 - education xliv
- malloc() library function
 - system programming C environment [505](#), [509](#), [523](#)
- MB_CUR_MAX, effect on DBCS [29](#)
- member, PDS and PDSE [42](#)
- memcmp library function [445](#), [446](#)
- MEMLIMIT default value
 - 64-bit virtual memory [227](#)
 - overriding [227](#)
 - setting [227](#)
- memory
 - optimizing [481](#)
- memory files
 - automatic name generation [144](#)
 - closing [150](#)
 - example [152](#)
 - example program [151](#)
 - extending [149](#)
 - flushing [149](#)
 - in hiperspace [141](#)
 - input and output [141](#)
 - opening [141](#)
 - optimizing [449](#)

- memory files (*continued*)
 - positioning within [149](#)
 - reading from [147](#)
 - repositioning within [149](#)
 - return values for fldata() [150](#)
 - simulated partitioned data sets
 - description [145](#)
 - example [145](#), [146](#)
 - specifying asterisk as file name [144](#)
 - support under CICS [580](#)
 - text mode treated as binary [145](#)
 - ungetc() considerations [149](#)
 - using to optimize code [449](#)
 - writing to [148](#)
- memset library function [445](#)
- method files [675](#)
- migrating applications
 - from ILP32 to LP64 [223](#)
- migration issues, ILP32-to-LP64 [226](#)
- mkdir() library function [76](#)
- mkfifo() library function
 - with z/OS UNIX System Services file system files [76](#), [83](#), [84](#)
- mknod() library function [76](#), [84](#)
- MSGCLASS, matching for SYSOUT data sets [46](#)
- MSGFILE (z/OS Language Environment)
 - closing [156](#), [157](#)
 - flushing buffers [156](#), [157](#)
 - opening files [155](#), [157](#)
 - output [155](#)
 - reading from [155](#), [157](#)
 - repositioning within [156](#), [157](#)
 - writing to [155](#), [157](#)
- MTF (multitasking facility)
 - coding for [562](#)
 - compiling [569](#)
 - concepts illustrated [557](#)
 - DD statements [571](#)
 - designing for [562](#)
 - dynamic commons [567](#)
 - EDCMTFS [569](#)
 - examples [565](#)
 - independence requirement [562](#)
 - introduction to [555](#)
 - Job Control Language (JCL) [569](#), [571](#)
 - link-editing considerations [570](#)
 - linking [569](#)
 - load modules [569](#)
 - modifying runtime options [570](#)
 - multithreading [259](#)
 - passing data [563](#)
 - restrictions [571](#)
 - rules [562](#)
 - running under [570](#)
 - tasks [555](#)
 - with z/OS XL C/C++ [491](#)
- multibyte characters
 - effect of MB_CUR_MAX [29](#)
 - reading [30](#)
 - writing [31](#)
- multiple buffering [53](#)
- multiple threads [247](#)
- multithreading [469](#)

multivolume data sets, opening [47](#)

mutex [248](#)

MVS (Multiple Virtual System)

alternative initialization routine [496](#)

building freestanding applications [498](#)

Data Window Services (DWS) [609](#)

file names [37](#)

file names for memory files [142](#)

reentrant modules [499](#)

MVS data sets

optimizing [447](#)

N

named pipes

example [86](#)

using [84](#)

naming environment variables [335](#)

natural reentrancy

and XPLINK [262](#)

navigation

keyboard [765](#)

NCP subparameter

multiple buffering [53](#)

network byte order [297](#)

network communication basics [293](#)

network, application example [303](#)

new

optimizing [427](#)

newline escape sequence `\n` [59](#)

nl_langinfo() library function [642](#)

non-DASD devices, I/O [48](#)

Non-VSAM keyed data sets, restriction [37](#)

nonoverrideable runtime options in the user exit [545](#)

NOSEEK parameter

in-stream data sets [45](#)

memory file I/O [144](#)

sequential concatenations [45](#)

terminal I/O [133](#)

VSAM data sets [104](#)

z/OS OS I/O [51](#)

O

object-oriented model for I/O [21](#)

OBJECTMODEL compiler option [466](#)

ofstream class [22](#)

Open Socket [293](#)

open() library function

for low-level z/OS UNIX System Services files [76](#)

with pipes [84](#)

z/OS UNIX System Services file system files [75](#)

opening

memory I/O files [141](#)

multibyte character files [30](#)

terminal files [131](#)

the CELQPIPI MSGRTN file [157](#)

VSAM data sets [100](#)

z/OS Language Environment message files [155](#)

z/OS UNIX System Services file system files [77](#)

OpenMP directives [469](#)

optica/reader input [48](#)

optimization

accessing UNIX file system files [448](#)

additional compiler options [465](#)

ANSI aliasing [429](#)

application performance [485](#)

arithmetic constructions [435](#)

built-in functions

examples [445](#)

C++ [427](#)

code motion [452](#)

common expression elimination [451](#)

compilation time [485](#)

constant propagation [452](#)

control constructs [435](#)

conversions [434](#)

dead code elimination [452](#)

dead store elimination [452](#)

declarations [436](#)

dynamic memory [481](#)

expressions [434](#)

fixed standard format records [12](#)

function arguments [433](#)

general notes [427](#)

graph coloring register allocation [452](#)

i/o stream libraries [449](#)

inlining [455](#)

inlining under IPA [457](#)

instruction scheduling [452](#)

levels [453](#), [458](#)

library extensions [437](#)

library lookasides [483](#)

link pack areas [483](#)

loop statements [435](#)

loops [469](#)

memory [481](#)

memory files [449](#)

MVS data sets [447](#)

noseek parameter for OS I/O [52](#)

OPTIMIZE [451](#)

pointers [433](#)

programming recommendations [12](#)

progression [453](#)

referencing bit fields [436](#)

storage [481](#)

straightening [451](#)

strength reduction [452](#)

value numbering [451](#)

variables [432](#)

virtual lookasides [483](#)

XPLINK [457](#)

OPTIMIZE

optimizing [451](#)

optimized code

troubleshooting with dbx [489](#)

option_override [439](#)

order, network byte [297](#)

OS I/O

acc= parameter [51](#)

asis parameter [51](#)

asynchronous reads [52](#), [53](#)

asynchronous writes [52](#), [53](#)

buffering [52](#)

bytewise parameter [51](#)

- OS I/O (*continued*)
 - closing files [68](#)
 - fgetpos() and ftell() values [65](#)
 - flushing records
 - description [62](#)
 - example [62](#)
 - I/O stream library [37](#)
 - in-stream data sets [45](#)
 - lrecl=X [50](#)
 - multivolume data sets [47](#)
 - opening files [37](#)
 - overview [37](#)
 - password= parameter [51](#)
 - PDS and PDSE considerations
 - BLKSIZE values [50](#)
 - LRECL values [50](#)
 - overview [42](#)
 - RECFM values [50](#)
 - reading from files [55](#)
 - repositioning within files [64](#)
 - space= parameter [50](#)
 - striped data sets [47](#)
 - tapes [46](#)
 - type= parameter [51](#)
 - ungetc() considerations [63](#), [65](#)
 - writing to files [57](#)
- OS linkage [175](#)
- os parameter, fopen()
 - memory file I/O [144](#)
 - terminal I/O [133](#)
 - VSAM I/O [104](#)
 - z/OS OS I/O [52](#)
- overlapped I/O [53](#)
- overrideable runtime options in the user exit [545](#)

P

- packed decimal
 - using with CICS [581](#)
- parallel functions [555](#)
- parallelization
 - OpenMP directives [469](#)
 - shared and private variables [470](#)
 - shared memory parallelism (SMP) [442](#)
- partitioned concatenation
 - compatibility rules [44](#)
 - data sets [43](#)
- passing parameters
 - CSP [597](#)
- password= parameter
 - memory file I/O [143](#)
 - VSAM data sets [103](#)
 - z/OS OS I/O [51](#)
- PATH, under VSAM [98](#)
- pathname, under POSIX.1 [77](#)
- PDF documents [xlili](#)
- PDS (partitioned data set)
 - input and output [42](#)
 - memory files simulation
 - description [145](#)
 - example [145](#), [146](#)
 - opening [50](#)

- PDS (partitioned data set) (*continued*)
 - OS I/O, restriction on opening [43](#)
- PDSE (partitioned data set extended)
 - input and output [42](#)
 - opening [50](#)
 - OS I/O, restriction on opening [43](#)
- performance
 - impact from BYTESEEK mode for OS files [66](#)
 - improvements by using fixed standard format records [12](#)
 - memory files [141](#)
 - noseek parameter for OS I/O [52](#)
 - opening memory files [144](#)
 - specifying FBS format [50](#)
- persistent C environments [505](#)
- pipe() library function [76](#)
- pipes
 - creating [76](#)
 - I/O [83](#)
 - named [84](#)
 - unnamed
 - description [76](#)
 - example [84](#)
- PL/I
 - using linkage specifications [175](#)
- PLIST
 - system programming environment [495](#)
- plotters, Graphical Data Display Manager (GDDM) [619](#)
- pointer assignments
 - under LP64 [237](#)
- pointers
 - 64-bit [236](#)
 - assigning in DLLs [203](#)
 - optimization [433](#)
- portability
 - between ILP32 and LP64 [243](#)
 - from ILP32 to LP64 [225](#)
 - ILP32-to-LP64 issues [226](#)
 - INFO [225](#)
 - long and int [232](#)
 - VM/CMS and z/OS filenames [39](#)
 - WARN64 [225](#)
- portable character set [731](#)
- ports
 - description [297](#)
 - locating [303](#)
- positioning
 - memory files [149](#)
 - OS I/O files [64](#)
 - terminal files [138](#)
 - the CELQPIPI MSGRTN file [157](#)
 - z/OS Language Environment message file [156](#)
 - z/OS UNIX System Services file system files [82](#)
- POSIX
 - locale, defined [699](#)
 - POSIX C locale and SAA C locale differences [705](#), [706](#)
- pragma
 - omp [469](#)
- pragmas
 - convert [743](#)
 - disjoint [438](#)

- pragmas (*continued*)
 - environment [501](#), [504](#)
 - execution_frequency [438](#)
 - export [438](#)
 - filetag
 - ??=pragma filetag directive [737](#)
 - inline [438](#)
 - isolated_call [438](#)
 - leaves [438](#)
 - linkage [498](#)
 - noinline [439](#)
 - option_override [439](#)
 - reachable [439](#)
 - runopts
 - description [527](#), [528](#)
 - heap [570](#)
 - plist [495](#)
 - stack [570](#)
 - strings [439](#)
 - unroll [439](#)
 - variable
 - NORENT [261](#)
 - RENT [261](#)
 - with XPLINK and SQL [612](#)
- predefined locale [739](#)
- PREFETCH compiler option [467](#)
- presentation interface [619](#)
- printer output
 - Graphical Data Display Manager (GDDM) [619](#)
- printf
 - 64-bit [242](#)
- problem determination
 - location of compile-time options information [421](#)
 - run time [421](#)
 - stepping through optimized code using dbx [489](#)
- protocols, transport [294](#)
- putc() library function
 - optimizing code [447](#)
- putwc_unlocked() library function [31](#)
- putwc() library function [31](#)
- putwchar_unlocked() library function [31](#)
- putwchar() library function [31](#)

Q

- QMF (Query Management Facility)
 - with has SAA callable interface [633](#)

R

- RACF (Resource Access Control Facility)
 - no hyphens in names for [38](#)
 - qualifier required in data set name [39](#)
- raise() library function
 - error handling [278](#)
- RBA (Random Byte Address)
 - in VSAM [98](#)
- RDW (record descriptor word) [49](#)
- reachable pragma [439](#)
- read-write lock [248](#)
- read() library function
 - with pipes [84](#)

- read() library function (*continued*)
 - z/OS UNIX System Services file system files [81](#)
- reading
 - from memory files [147](#)
 - from OS I/O files [55](#)
 - from terminal files [133](#)
 - from the CELQPIPI MSGRTN file [157](#)
 - from the z/OS Language Environment message file [155](#)
 - from VSAM data sets [105](#)
 - from z/OS UNIX System Services file system files [80](#)
 - multibyte characters [30](#)
 - using recfm=U [49](#)
- realloc() library function
 - system programming C environment [509](#), [523](#)
- reason codes
 - in user exits [543](#)
- RECFM (record format)
 - F (fixed-format) [12](#)
 - memory file I/O [143](#)
 - overview [11](#)
 - recfm=* extension [50](#)
 - recfm=A extension [50](#)
 - S (fixed standard) [12](#)
 - S (variable spanned) [16](#)
 - terminal I/O [132](#)
 - U (undefined format)
 - overview [18](#)
 - reading OS files [49](#)
 - V (variable format)
 - overview [15](#)
 - VSAM data sets [103](#)
 - z/OS OS I/O [49](#)
- record
 - empty
 - _EDC_ZERO_RECLLEN [18](#), [359](#)
 - files, using fseek() and ftell() [67](#)
 - fixed standard format [12](#)
 - I/O
 - byte stream behavior [20](#)
 - fixed-format behavior [15](#)
 - introduction [8](#)
 - restriction [30](#)
 - undefined-format behavior [19](#)
 - variable-format behavior [18](#)
 - spanned [16](#)
 - undefined-length [18](#)
 - variable-length [15](#)
 - z/OS UNIX System Services file system I/O rules [79](#)
 - zero-byte
 - _EDC_ZERO_RECLLEN [18](#), [359](#)
- redirection
 - standard streams in a system programming C environment [495](#)
- reentrancy
 - and XPLINK [262](#)
 - constructed [262](#)
 - in z/OS XL C/C++ [261](#)
 - limitations [262](#)
 - modified CEEBXITA must be reentrant [541](#)
 - natural [262](#)
 - with respect to CICS [588](#)

- register
 - allocation [452](#)
 - variables [433](#)
- regular z/OS UNIX System Services file system files [75](#)
- relative byte offset [66](#)
- remove() library function
 - memory I/O files [150](#)
 - OS I/O files [70](#)
- rename() library function
 - OS I/O files [70](#)
- RENT compiler option [261](#)
- repositioning
 - binary streams, wide character I/O [35](#)
 - memory files [149](#)
 - OS I/O files [64](#)
 - terminal files [138](#)
 - text streams, wide character I/O [34](#)
 - the CELQPIPI MSGRTN file [157](#)
 - VSAM records [108](#)
 - z/OS Language Environment message file [156](#)
 - z/OS UNIX System Services file system files [82](#)
- RESTRICT compiler option [467](#)
- restrictions, compiler [309](#)
- return
 - codes
 - __amrc structure [119](#)
 - CEEAEUE_RET field of CEEBXITA and [543](#)
 - in user exits [543](#)
 - value under CICS [583](#)
- RMODE processing option
 - for CEEBXITA user exit [541](#)
- ROCONST compiler option
 - controlling external static [262](#)
- ROSTRING compiler option
 - controlling writable strings [263](#)
- RPC (Remote Procedure Call) [310](#)
- RRDS (Relative Record Data Set)
 - choosing whether key and data are contiguous [105](#)
 - choosing whether key is returned with data on read [106](#)
 - key structure [105](#)
 - related environment variable [356](#)
 - use of [95](#)
- RRN (Relative Record Number)
 - under VSAM [99](#)
- RTTI
 - optimizing [427](#)
- RTTI compiler option [467](#)
- runtime
 - messages
 - EDCXLANE [527](#)
 - EDCXLANK [527](#)
 - UENGLISH [527](#)
 - options
 - in the user exit [540](#), [545](#)
 - TRAP [541](#), [544](#)
 - user exits [537](#)
- runtime environment
 - changing the code page [759](#)
- Runtime type identification
 - optimizing [427](#)

S

- S370 locale [699](#)
- SAA (Systems Application Architecture)
 - applications using QMF callable interface [633](#)
 - differences between C and POSIX locales [705](#), [706](#)
 - locale [699](#)
- saved options information [421](#)
- saved options information layout [421](#)
- screen layouts [619](#)
- SEEK_CUR macro
 - effects of ungetc() [65](#)
 - effects of ungetwc() [35](#)
- seeking
 - OS I/O files [64](#)
 - terminal files [138](#)
 - the CELQPIPI MSGRTN file [157](#)
 - within memory files [149](#)
 - within z/OS UNIX System Services file system files [82](#)
 - z/OS Language Environment message file [156](#)
- select(), network example [300](#)
- sequential
 - concatenation
 - compatibility rules [44](#)
 - data sets [43](#)
 - noseek parameter [45](#)
 - processing [105](#), [106](#)
- sequential data sets, large format [47](#)
- server
 - allocation with socket() [299](#)
 - locating the port [303](#)
 - perspective [299](#)
- service routines [509](#)
- session
 - typical TCP socket [301](#)
 - typical UDP socket [302](#)
- setenv() library function
 - setting environment variables [334](#)
- setlocale() library function
 - description [642](#)
 - not thread-safe [259](#)
- setvbuf() library function
 - hiperspace memory files [23](#), [141](#)
 - specifying size of buffer for hiperspace [141](#)
 - usage [448](#)
 - zFS file systems [448](#)
- severity of a condition
 - CEEBXITA assembler user exit and [544](#)
- shaping characters [747](#)
- shared programs [261](#)
- shareoptions specification, VSAM
 - deleting records [107](#)
 - opening a data set [101](#)
- shift-in character (DBCS) [59](#)
- shift-out character (DBCS) [59](#)
- shortcut keys [765](#)
- SIGABND signal [283](#)
- SIGABRT signal [283](#)
- SIGDANGER signal [283](#)
- SIGDUMP signal [283](#)
- SIGFPE signal
 - error condition [283](#)
- SIGILL signal [283](#)

- SIGINT signal [283](#)
- SIGIOERR signal [168](#), [283](#)
- signal
 - actions, defaults [286](#)
 - delivery
 - asynchronous [281](#)
 - ISO C rules [279](#)
 - POSIX rules [279](#)
 - handling
 - default [286](#)
 - disabled [282](#)
 - enabled [282](#)
 - established [282](#)
 - hardware [282](#)
 - raise [278](#)
 - software [283](#)
 - with signal() and raise() [278](#)
 - with z/OS Language Environment [278](#)
- SIGSEGV signal [283](#)
- SIGTERM signal [283](#)
- SIGUSR1 signal [283](#)
- SIGUSR2 signal [283](#)
- socket
 - address [297](#)
 - address families [297](#)
 - addressing within [297](#)
 - AF_INET domain [298](#)
 - AF_UNIX domain [299](#)
 - client perspective [301](#)
 - compiling [309](#)
 - data sets [311](#)
 - datagram [296](#)
 - defined [293](#), [294](#)
 - domains [297](#)
 - families [296](#)
 - include files [311](#)
 - Internet [293](#)
 - linking [309](#)
 - local [293](#)
 - types
 - datagram [296](#)
 - guidelines for using [296](#)
 - stream [296](#)
 - typical TCP session [301](#)
 - typical UDP session [302](#)
 - using over TCP/IP [293](#)
 - z/OS TCP/IP [310](#)
 - z/OS UNIX System Services specific [296](#)
- software signals [283](#)
- space= parameter
 - memory file I/O [143](#)
 - terminal I/O [132](#)
 - VSAM data sets [103](#)
 - z/OS OS I/O [50](#)
- spanned records [16](#)
- SPIII compiler option [467](#)
- spool data sets [346](#)
- sprintf() library function
 - in freestanding routines [499](#)
 - system programming C environment [505](#), [509](#), [523](#)
- sscanf() library function
 - (continued)*
 - character to integer conversions [446](#)
 - stand-alone modules [496](#)
 - standalone CICS translator [592](#)
 - standard
 - records [12](#)
 - stream
 - cerr [21](#)
 - cin [21](#)
 - clog [21](#)
 - cout [21](#)
 - global behavior [350](#)
 - restrictions in threaded applications [257](#)
 - support under CICS [579](#)
- static variables [432](#)
- STEPLIB DD statement [570](#)
- stepping through optimized code using dbx [489](#)
- storage
 - allocating with the system programming C environment [494](#)
 - freeing with EDCXFREE [526](#)
 - getting with EDCXGET [524](#)
 - optimizing [481](#)
 - page-aligned, getting with EDCX4KGT [525](#)
 - under CICS [583](#)
- Store Clock Fast
 - built-in instruction [361](#)
- straightening [451](#)
- strcat() library function [446](#)
- stream sockets [296](#)
- streambuf class [21](#)
- streams, orientation of [29](#)
- strength reduction [452](#)
- STRICT compiler option [467](#)
- STRICT_INDUCION compiler option [467](#)
- strings
 - comparisons [445](#), [446](#)
 - pragma [439](#)
 - processing [446](#)
- striped data sets [47](#)
- strlen library function [445](#)
- structure alignment
 - 64-bit [227](#)
- structure comparison [445](#)
- structures
 - ILP32 to LP64 alignment problems [239](#)
 - rule of alignment [228](#)
- stub routines
 - in a user-server environment [523](#)
- summary of changes
 - z/OS XL C/C++ Programming Guide [xlvi](#)
- svc99() library function [581](#)
- symbolic link (z/OS UNIX System Services file system) files, creating [75](#)
- syntax diagrams
 - how to read [xxxv](#)
- SYSOUT data set
 - DCB attributes, defaults [46](#)
 - output [46](#)
- system
 - exit routines [501](#)
 - functions
 - built-in [494](#)

- system (*continued*)
 - functions (*continued*)
 - memory management [494](#)
 - programming facilities
 - additional library routines [528](#)
 - building persistent C environments [505](#), [506](#)
 - building system exit routines [502](#)
 - building user-server environments [523](#)
 - freestanding applications [496](#)
 - runtime messages [527](#)
 - tailoring the environment [524](#)
 - with z/OS XL C++ [491](#)
- system() library function
 - CICS [582](#)
 - library extension [438](#)
 - programming C environment [495](#)

T

- tab, horizontal [59](#)
- tab, vertical [59](#)
- tapes
 - input and output [46](#)
 - multivolume data sets [47](#)
- tasks
 - setting up a stopping point for dbx in optimized code
 - steps for [489](#)
 - using an implicitly loaded DLL in your simple DLL application>
 - steps for [193](#)
 - using dbx to step through optimized code
 - steps for [489](#)
- tasks, using MTF [555](#)
- TCP socket session [301](#)
- TCP/IP [310](#)
- templates
 - TEMPINC
 - examples of source files [320](#), [321](#)
 - JCL to compile examples [322](#)
 - regenerating the template instantiation file [323](#)
 - TEMPLATEREGISTRY
 - changing and recompiling parts of program [323](#)
 - instantiation with template registry [323](#)
 - using TEMPINC or NOTEMPINC
 - multipurpose header file [319](#)
- temporary data sets (MVS)
 - using & names [38](#)
- temporary files [141](#)
- terminals
 - closing [138](#)
 - flushing [137](#)
 - Graphical Data Display Manager (GDDM) [619](#)
 - I/O
 - overview [131](#)
 - reading from files [133](#)
 - writing to files [135](#)
 - opening I/O files [131](#)
 - positioning within [138](#)
 - responses to fldata() [138](#)
- termination
 - enclave
 - as indicated in CEEAUE_ABND field of CEEAUE_FLAGS [544](#)

- termination (*continued*)
 - enclave (*continued*)
 - as indicated in CEEAUE_ABTERM field of CEEAUE_FLAGS [544](#)
 - CEEBXITA's behavior during [540](#)
 - CEEBXITA's function codes for [543](#)
 - process [540](#), [543](#)
- text files
 - ASA RECFM fixed-format behavior [14](#)
 - ASA RECFM undefined-format behavior [19](#)
 - ASA RECFM variable-format behavior [18](#)
 - non-ASA RECFM fixed-format behavior [13](#)
 - non-ASA RECFM undefined-format behavior [19](#)
 - non-ASA RECFM variable-format behavior [17](#)
 - RECFM byte stream behavior [20](#)
 - using fseek() and ftell() [67](#)
- text I/O [7](#)
- THREADED compiler option [467](#)
- threads
 - cancel [254](#)
 - cleanup [256](#)
 - condition variable [248](#)
 - create [247](#)
 - functions [247](#)
 - low-level z/OS UNIX System Services I/O [88](#)
 - management [247](#)
 - mutex [248](#)
 - read-write lock [248](#)
 - recoverable resources [260](#)
 - signals [253](#)
 - thread-specific data [252](#)
 - using in z/OS UNIX System Services applications [247](#)
 - using with MVS files [257](#)
- throw [273](#)
- time zone
 - customizing [697](#)
 - specifying [653](#)
- tolower() macro [445](#)
- toupper() macro [445](#)
- traceback [274](#)
- transaction execution
 - built-in instructions, hardware [409](#)
- translation tables [707](#)
- transport protocols [294](#)
- TRAP runtime option
 - CEEBXITA assembler user exit and [541](#)
 - how CEEAUE_ABND is affected by [544](#)
 - IMS considerations [626](#)
- troubleshooting
 - stepping through optimized code using dbx [489](#)
- try [273](#)
- TSO (Time Sharing Option)
 - default high-level qualifier [39](#), [142](#)
 - opening files [39](#), [142](#)
 - setting the user prefix [39](#), [142](#)
- TUNE compiler option [454](#)
- type= parameter
 - memory file I/O [144](#)
 - terminal I/O [132](#)
 - VSAM data sets [103](#)
 - z/OS OS I/O [51](#)

types, sockets [296](#)
TZ environment variable [697](#)
tzset() library function
 not thread-safe [259](#)

U

UDP socket session [302](#)
ulimit command
 MEMLIMIT system parameter [227](#)
unbuffered I/O
 setvbuf() function [53](#)
undefined format records [18](#)
ungetc() library function
 _EDC_COMPAT environment variable [346](#)
 memory file I/O, effect on fflush() [149](#)
 OS I/O, effect on fflush() [63](#)
 OS I/O, effect on fgetpos() and ftell() [65](#)
 SEEK_CUR [65](#)
ungetwc() library function
 effect on fflush(), wide character I/O [34](#)
 effect on fgetpos(), ftell() and fseek() [35](#)
 seek_cur [35](#)
universal reference time [697](#)
unlink() library function
 using with named pipes [85](#)
 with z/OS UNIX System Services file system files
 [83](#)
unnamed pipes
 creating [76](#)
 example [84](#)
 using [83](#)
UNROLL compiler option [468](#)
unroll pragma [439](#)
unsuffixed numbers
 ILP32 to LP64 migrations [241](#)
updating VSAM records [107](#)
user exit
 for initialization [540](#)
 for termination [539](#), [540](#)
 runtime options [545](#)
 under CICS [543–545](#)
user interface
 ISPF [765](#)
 TSO/E [765](#)
user words [535](#)
user-server stub routines [523](#)

V

V-format records [15](#)
value numbering [451](#)
variable pragma [439](#)
variable-format records [15](#)
variables
 environment [327](#)
 exported [182](#)
 external [432](#)
 global [432](#)
 local [432](#)
 locale [332](#)
 register [433](#)
 static [432](#)

variant characters
 default coding [759](#)
 detail [731](#)
 displaying on workstation or 3270 [759](#)
 keyboard mapping [759](#)
 mapping [759](#)
 mappings [731](#), [732](#)
 use of [731](#)
VB-format records [15](#)
VBS-format records [15](#)
VECTOR compiler option [468](#)
vertical tab escape sequence \v [59](#)
vfwprintf_unlocked() library function [31](#)
vfwprintf() library function [31](#)
vfwscanf_unlocked() library function [30](#)
vfwscanf() library function [30](#)
virtual
 optimizing [427](#)
virtual lookasides
 optimizing [483](#)
VS-format records [15](#)
VSAM (Virtual Storage Access Method)
 __amrc structure [119](#)
 closing a data set [118](#)
 example programs
 KSDS [119](#)
 RRDS [127](#)
 example showing how to access __amrc structure [98](#)
I/O operations
 deleting a record [108](#)
 loading a data set [106](#)
 locating a record [108](#)
 overview [95](#)
 reading a record [105](#)
 repositioning [108](#)
 specifying access mode [102](#)
 summary of binary I/O operations [117](#)
 summary of operations [99](#)
 summary of record I/O operations
 [110](#)
 summary of text I/O operations [116](#)
 updating a record [107](#)
 using fopen() [100](#)
 using freopen() [100](#)
 writing a record [106](#)
I/O stream library [95](#)
keys [98](#)
KSDS example [119](#)
linear data sets [95](#)
naming MVS data sets [100](#)
organization of data sets [95](#)
Record Level Sharing [111](#)
Relative Byte Addresses (RBA) [98](#)
Relative Record Numbers (RRN) [99](#)
return codes [119](#)
RLS [111](#)
RSDS example [128](#)
Transactional VSAM [111](#)
TVS [111](#)
 types and advantages of data sets [97](#)
vwprintf_unlocked() library function [31](#)
vwprintf() library function [31](#)
vwscanf_unlocked() library function [30](#)
vwscanf() library function [30](#)

W

- WARN64 compiler option
 - identifying portability problems [225](#)
- wcsid() library function [642](#)
- wide characters
 - effect of MB_CUR_MAX [29](#)
 - input and output functions [29](#)
 - reading streams and files [30](#)
 - ungetwc() considerations [34](#)
 - writing streams and files [31](#)
- windowing [619](#)
- wprintf_unlocked() library function [32](#)
- wprintf() library function [32](#)
- writable static
 - assembler code [264](#)
 - in reentrant programs [261](#)
- write() library function
 - with pipes [84](#)
 - z/OS UNIX System Services file system I/O [81](#)
- writing
 - binary streams, wide character I/O [33](#)
 - in coded character set IBM-1047 [743](#)
 - multibyte characters [31](#)
 - text streams, wide character I/O [32](#)
 - to memory files [148](#)
 - to OS I/O files [57](#)
 - to terminal files [135](#)
 - to the CELQPIPI MSGRTN file [157](#)
 - to the z/OS Language Environment message file [155](#)
 - to z/OS UNIX System Services file system files [81](#)
 - VSAM data sets [106](#)
- wscanf_unlocked() library function [30](#)
- wscanf() library function [30](#)

X

- X Windows, z/OS TCP/IP [310](#)
- X/Open Socket [293](#)
- X/Open Transport Interface (XTI)
 - concepts [314](#)
 - transport endpoints [314](#)
 - transport providers [315](#)
- XFER, transfer control [597](#)
- xhotc library function [531](#)
- xhotl library function [532](#)
- xhott library function [532](#)
- xhotu library function [533](#)
- XITPTR, CXIT control block [542](#)
- XPLINK
 - and DB2 services, stored procedures [612](#)
 - when to use [457](#)
- xregs library function [533](#)
- xsacc library function [534](#)
- xusr() library function [535](#)
- xusr2() library function [535](#)

Z

- z/OS Basic Skills Documentation [xliv](#)

- z/OS Language
 - Environment
 - message file output [155](#)
- z/OS TCP/IP
 - child process creation restrictions [311](#)
 - configuration file access [311](#)
 - header file restrictions [310](#)
 - interprocess communication [310](#)
 - socket API restrictions [310](#)
- z/OS UNIX file system
 - character special [76](#)
 - closing files [82](#)
 - creating files [75](#)
 - deleting [83](#)
 - directory [76](#)
 - example [88](#), [90](#)
 - FIFO [76](#)
 - file types [75](#)
 - flushing records [82](#)
 - I/O functions, example program [88](#)
 - I/O stream library [75](#)
 - input and output [75](#)
 - link [75](#)
 - naming files [76](#)
 - reading streams and files [80](#)
 - record I/O rules [79](#)
 - regular [75](#)
 - setting positions within files [82](#)
 - writing to streams and files [81](#)
- z/OS UNIX System
 - Services
 - I/O, low-level [88](#)
 - ulimit command [227](#)
- z/OS XL C/C++ integrated CICS translator [588](#)
- z/OS XL C/C++ Programming Guide
 - content, changed [xlvi](#)
 - content, new [xlvi](#)
 - summary of changes [xlvi](#)
- zero-byte records, _EDC_ZERO_RECLEN [18](#), [359](#)
- zFS file systems [448](#)



Product Number: 5655-ZOS

SC14-7315-60

