z/OS

XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer

GC09-4913-03

z/OS

XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer

GC09-4913-03

Note!
Before using this information and the product it supports, be sure to read the information in “Notices” on page 121.

Fifth Edition (September 2005)

This edition applies to Version 1 Release 7 of z/OS XL C/C++ (5694-A01), Version 1 Release 7 of z/OS.e XL C/C++ (5655-G52), and to all subsequent releases until otherwise indicated in new editions. This edition replaces GC09-4913-02. Make sure that you use the correct edition for the level of the program listed above. Also, ensure that you apply all necessary PTFs for the program.

Order publications through your IBM representative or the IBM branch office serving your location. Publications are not stocked at the address below. You can also browse the books on the World Wide Web by clicking on ″The Library″ link on the z/OS home page. The web address for this page is
`www.ibm.com/servers/eserver/zseries/zos/bkserv`

IBM welcomes your comments. You can send your comments to the following Internet address:
compinfo@ca.ibm.com. Be sure to include your e-mail address if you want a reply.

Include the title and order number of this book, and the page number or topic related to your comment. When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Part 1. Introduction

This part provides answers to some common migration questions.

Note that throughout this document, the short form of a product version and release number (VxRx) is used. For example, this document refers to z/OS® Version 1 Release 4 C/C++ as z/OS V1R4 C/C++. In addition, assume that the modification level of any referenced product is 0 (zero) unless specifically indicated.

Chapter 1. Locating your migration path

This book discusses the implications of migrating applications from each of the supported compilers and libraries listed in Table 2 on page 5 to the z/OS® V1R7 XL C/C++ product. To find the section of the book that applies to your migration, see "How this book is organized" on page 4.

You can use this book to:

- Help determine whether and how you can continue to use existing source code, object code, and load modules
- Become aware of the changes in compiler and run-time behavior that may affect your migration from earlier compilers or from earlier versions of the compiler

Note: In most situations, existing well-written applications can continue to work without modification.

This book does not:

- Discuss all of the enhancements that have been made to the z/OS V1R7 XL C/C++ compiler and z/OS V1R7 Language Environment®.

  Note: For a list of books that provide information about the z/OS V1R7 XL C/C++ compiler and its debugger and utilities, refer to "Bibliography" on page 125.
- Show how to change an existing C program so that it can use C++.

  Note: For a description of some of the differences between C and C++, see *z/OS XL C/C++ Language Reference*.

In this book, references to the products listed in the first column of Table 1 also apply to the products in the second column.

Table 1. Product references

References to these products	Also apply to these products
IBM® SAA® AD/Cycle® Language Environment/370	AD/Cycle C/370™ Language Support Feature
Language Environment (MVS® & VM) R4	IBM Language Environment for MVS & VM
Language Environment (MVS & VM) R5	IBM Language Environment for MVS & VM
Language Environment (MVS & VM) R6	IBM Language Environment for MVS & VM
C/MVS™ V3R1 or V3R2 compiler	C component of the C/C++ for MVS/ESA™ V3R2 or V3R1 compiler
C++/MVS™ V3R2 compiler	C++ component of the C/C++ for MVS/ESA V3R2 compiler

Table 1. Product references (continued)

References to these products	Also apply to these products
C/370 compiler **Note:** You cannot migrate directly from the C/370 V1 compiler to the z/OS XL C compiler. You must first migrate to the C/370 V2 compiler. If you are migrating a program that has been run successfully only with a C/370 run-time environment, contact your service representative.	Any of the following: • The IBM C/370 V1R1 or V1R2 compiler (MVS and VM) compiler and the IBM C/370 V1R1 library • The IBM C/370 V2R1 compiler (MVS and VM and VSE) compiler and the IBM C/370 V2R1 library • The IBM SAA AD/Cycle C/370 V1R0 compiler and the IBM C/370 V2R0 library • The IBM SAA AD/Cycle C/370 V1R2 compiler and the IBM C/370 V2R2 library
Pre-OS/390® C/C++ compilers **Note:** If you are migrating a program that has been run successfully only with a pre-OS/390 C/C++ compiler, contact your service representative.	Any of the following: • AD/Cycle V1R1 or V1R2 compiler • C/MVS V3R1 or V3R2 compiler • AD/Cycle C/370 compilers • IBM C++ for MVS V3R2 compiler and Language Environment R5
OS/390 C/C++ compilers **Note:** If you are migrating a program that has been run successfully only with the OS/390 V1 C/C++ compiler, contact your service representative.	Any of the following: • OS/390 V1R1, V1R2, or V1R3 C/C++ compiler • OS/390 V2R4, V2R5, V2R6, V2R7, V2R8, V2R9, or V2R10 compiler
"Earlier releases" of the z/OS C/C++ compiler **Note:** Service is still available for z/OS C/C++ V1R4 through z/OS C/C++ V1R6.	Any of the following: • z/OS C/C++ V1R1 • z/OS C/C++ V1R2 • z/OS C/C++ V1R3 • z/OS C/C++ V1R4 • z/OS C/C++ V1R5 • z/OS C/C++ V1R6
z/OS	z/OS.e

Note: As of z/OS V1R7, the OS/390 V2R10 C/C++ compiler is no longer shipped with the z/OS product.

How this book is organized

- Part 1 contains some general answers to common migration questions.
- Part 2 describes the considerations for migrating from one of the C/370 compilers.
- Part 3 describes the considerations for migrating from any of the pre-OS/390 releases of C/C++, or any release of z/OS Language Environment.
- Part 4 describes the considerations for migrating from one of the following:
  - OS/390 C compilers
  - z/OS V1R1 C compiler
- Part 5 describes migration issues related to *Programming languages - C/C++ (ISO/IEC 14882:2003(E))*, which documents the C++ Standard.
- Part 6 describes the considerations for migrating from one of the earlier releases of the z/OS C/C++ compiler.

A history of IBM C/C++ compilers and libraries

Table 2 lists the versions of the C and C++ compilers and run-time libraries in the order in which they were first released. You can use this table to help determine which changes described in this book apply to your migration.

Table 2. A history of IBM C/C++ compilers and libraries

Short name	Product number	GA date	Description	Service status
C/370 V1R1	5688-040 5688-039	1988 1988	C/370 V1R1 Compiler C/370 V1R1 Library	Service discontinued
C/370 V1R2	5688-040 5688-039	1989 1989	C/370 V1R2 Compiler C/370 V1R2 Library	Service discontinued
C/370 V2R1	5688-187 5688-188	1991 1991	C/370 V2R1 Compiler C/370 V2R1 Library	
AD V1R1	5688-216	1991	AD/Cycle C/370 V1R1 Compiler (follow-on to C/370 V2R1 compiler)	Service discontinued
LE V1R1	5688-198	1991	Language Environment/370 V1R1 Library(first release of Language Environment/370; follow-on to C/370 V2R1 Library)	Service discontinued
LE V1R2	5688-198	1992	Language Environment/370 V1R2 Library	Service discontinued
AD V1R2	5688-216	1994	AD/Cycle C/370 V1R2 Compiler: • Runs on either LE V1R3 or C/370 V2R2 • Generates code for either LE V1R3 or C/370 V2R2	
LE V1R3	5688-198	1994	Language Environment/370 V1R3 Library, also shipped as part of AD/Cycle C/370 Language Support Feature.	Service discontinued
C/370 V2R2	5688-188	1994	C/370 V2R2 Library (follow-on to the C/370 V2R1 Library; intended to help customers migrate to LE/370).	
C/C++MVS V3R1	5655-121	1995	C/C++ for MVS/ESA V3R1 Compilers, follow-on to AD V1R2 compiler (first release of C++ on MVS).	Service discontinued
LE V1R4	5688-198	1995	LE V1R4 Library for MVS & VM (also shipped as the MVS/ESA SP™ 5.2.0 C/C++ Language Support Feature).	Service discontinued
C/C++/ MVS V3R2	5655-121	1995	C/C++ for MVS/ESA V3R2 Compilers (the successor of the C/C++MVS V3R1 compiler).	
LE V1R5	5688-198	1995	LE V1R5 Library for MVS & VM (also shipped as part of MVS/ESA SP 5.2.2 C/C++ Language Support Feature).	Service discontinued
OS/390 V1R1	5645-001	March 1996	OS/390 V1R1 includes the C/C++ for MVS/ESA V3R2 compilers and the OS/390 V1R1 Language Environment.	Service discontinued
OS/390 V1R2	5645-001	Sept 1996	OS/390 V1R2 C/C++ (follow-on to OS/390 V1R1 C/C++; includes optimization options to improve the execution-time performance of C code). OS/390 V1R2 Language Environment is packaged with OS/390 V1R2.	Service discontinued

Table 2. A history of IBM C/C++ compilers and libraries (continued)

Short name	Product number	GA date	Description	Service status
OS/390 V1R3	5645-001	March 1997	OS/390 V1R3 C/C++ (follow-on to OS/390 V1R2 C/C++; includes optimization options to improve the execution-time performance of C++ code). OS/390 V1R3 Language Environment is packaged with OS/390 V1R3.	Service discontinued
OS/390 V2R4	5647-A01	Sept 1997	OS/390 V2R4 C/C++ (follow-on to OS/390 V1R3 C/C++; includes performance improvements for DLLs, conversion of character string literals, and support for the Program Management Binder). OS/390 V2R4 Language Environment is packaged with OS/390 V2R4.	Service discontinued
OS/390 V2R5	5647-A01	March 1998	OS/390 V2R5 C/C++ (functionally equivalent to OS/390 V2R4 C/C++). OS/390 V2R5 Language Environment is packaged with OS/390 V2R5.	Service discontinued
OS/390 V2R6	5647-A01	Sept 1998	OS/390 V2R6 C/C++ (follow-on to OS/390 V2R4 C/C++; includes support for the IEEE binary floating-point and the long long data types, improvements to the handling and format of packed decimal numbers in C++, and the `TARGET(OSV1R2)` suboption). OS/390 V2R6 Language Environment is packaged with OS/390 V2R6.	Service discontinued
OS/390 V2R7	5647-A01	March 1999	The compiler is functionally equivalent to the OS/390 V2R6 C/C++ compiler. OS/390 V2R7 Language Environment is packaged with OS/390 V2R7.	Service discontinued
OS/390 V2R8	5647-A01	Sept 1999	The compiler is functionally equivalent to the OS/390 V2R6 C/C++ compiler. OS/390 V2R8 Language Environment is packaged with OS/390 V2R8.	Service discontinued
OS/390 V2R9	5647-A01	March 2000	OS/390 V2R9 C/C++ (follow-on to OS/390 V2R6 C/C++). It includes the following: • Compiler options and suboptions: – `CHECKOUT(CAST)` – `COMPRESS` – `CVFT` – `DIGRAPH` (for C) – `IGNERRNO` – `INITAUTO` – `IPA(OBJONLY)` – `PHASEID` – `ROCONST` – `ROSTRING` – `STRICT` – `TARGET` enhancements to suboptions • Directives: – `#pragma leaves` – `#pragma option_override` – `#pragma reachable` OS/390 V2R9 Language Environment is packaged with OS/390 V2R9.	Service discontinued

Table 2. A history of IBM C/C++ compilers and libraries (continued)

Short name	Product number	GA date	Description	Service status
OS/390 V2R10	5647-A01	Sept 2000	OS/390 V2R10 C/C++ (follow-on to OS/390 V2R9 C/C++). It introduced the following: • Compiler options and suboptions: – `COMPACT` – `GOFF` – `IPA(LEVEL(2))` – `XPLINK` • Enhancements to the following compiler options and suboptions: – `SPILL` – `TARGET` • Improvements to: – `#pragma option_override` – Packed decimal optimization in C In addition: • Support for the IBM System Object Model™ (SOM®) was dropped. • OS/390 V2R10 Language Environment was packaged with OS/390 V2R10.	service discontinued
z/OS V1R1	5694-A01	Mar 2001	z/OS V1R1 C/C++ is functionally equivalent to OS/390 V2R10 C/C++.	service discontinued
z/OS V1R2	5694-A01	Oct 2001	z/OS V1R2 C/C++ is fully compliant with *Programming languages - C++ (ISO/IEC 14882:2003(E))*, with support for: • namespaces • type `bool` and associated keywords **bool**, **true**, and **false** • class-member-modifying keywords **mutable** and **explicit** • C++ cast operators `const_cast`, `dynamic_cast`, `reinterpret_cast` and `static_cast` • new template model • Run Time Type Identification (RTTI) • C++ Standard Library, including the Standard Template Library (STL). It also includes the following enhancements: • IBM Open Class® Library new level • Enhanced ASCII and Large File support in C++ Standard I/O Stream Library • `IPA` support for `XPLINK` z/OS V1R2 Language Environment is packaged with z/OS V1R2.	service discontinued
z/OS V1R3	5694-A01	Mar 2002	The compiler is functionally equivalent to the z/OS V1R2 C/C++ compiler. z/OS V1R3 Language Environment is packaged with z/OS V1R3.	Service discontinued
z/OS V1R4	5694-A01	Sept 2002	The compiler is functionally equivalent to the z/OS V1R2 C/C++ compiler. z/OS V1R4 Language Environment is packaged with z/OS V1R4.	

Table 2. A history of IBM C/C++ compilers and libraries (continued)

Short name	Product number	GA date	Description	Service status
z/OS V1R5	5694-A01	Mar 2004	z/OS V1R5 includes: • Optimization level OPT(3), based on Profile Directed Feedback (PDF), as well as other optimization improvements • Loop unrolling control • Debug information format based on Dwarf (The existing debug information format is still supported.) Using IBM Open Class Library for development is no longer supported. z/OS V1R5 Language Environment is packaged with z/OS V1R5.	
z/OS V1R6	5694-A01	Sept 2004	z/OS V1R6 C/C++ supports compilation of 64-bit programs, which is enabled by the `LP64` option. z/OS V1R6 C/C++ introduces the following new compiler suboptions: `ARCH(6)`, `TARGET(zOSV1R6)`, and `TUNE(6)`. It introduces the **__attribute__((aligned(n)))** keyword, which is used in a declaration to specify an alignment for a declared variable. **Note:** For information about this keyword, see http://gcc.gnu.org.onlinedocs. The Standard C++ Library was provided as an `XPLINK` DLL in the previous releases. Support has been added for a non-`XPLINK` DLL version of this library. This can be used in sub-system environments where `XPLINK` is not supported. z/OS V1R6 provides an invocation utility that facilitates portability of applications between AIX® and z/OS C/C++ compilers. If you use the `xlc` utility to invoke the compiler, the following commands will also accept AIX options syntax as well as z/OS options syntax: • `cc` - to compile C programs • `c89` - to compile C programs • `cxx` - to compile C++ programs • `c++` - to compile C++ programs z/OS V1R6 provides new commands and suffixes that support compiler flexibility and code portability. For more information, see Chapter 18, “Compiler invocations,” on page 107.z/OS V1R6 Language Environment is packaged with z/OS V1R6.	

Table 2. A history of IBM C/C++ compilers and libraries (continued)

Short name	Product number	GA date	Description	Service status
z/OS V1R7	5694-A01	Sept 2005	As of z/OS V1R7: • The OS/390 V2R10 compiler is no longer shipped with the z/OS product. As of z/OS V1R7, targeting z/OS V1R3 and earlier releases is no longer supported. The earliest release that can be targeted is zOSV1R4. • A recompile using the `_OPEN_SYS_SOCK_IPV6` feature test macro will expose new definitions in certain run-time library functions. For more information, see "_OPEN_SYS_SOCK_IPV6 feature test macro" on page 112. • A recompile with the `LANGLVL` compiler option on a compiler designed to support C99 may cause compiler error messages to be issued. For information on avoiding these errors, see "C99 with both LANGLVL(LONGLONG) and LANGLVL(EXTENDED)" on page 113 and "LANGLVL(ANSISINIT) and static initialization" on page 89. • There are changes in hexadecimal floating point notation and floating point special values for C99 that can affect the behavior of well-formed applications complying with *Programming languages - C (ISO/IEC 9899:1990)* and earlier versions of the base documents. For more information, see "Floating point support" on page 113. • The `xlc` compiler invocation utility emits default compiler options when you specify the CMDOPTS compiler option. For more information, see "CMDOPTS compiler option and conflict resolution" on page 105 and "Changes that affect xlc invocation" on page 108. • The directive `#pragma unroll()` works only with `for` loops. For more information, see "Application of #pragma unroll()" on page 101. z/OS V1R7 Language Environment is packaged with z/OS V1R7.	

Chapter 2. Common questions about migration

This chapter describes the kind of migration impacts that you may encounter, and the possible solutions.

Will existing Language Environment applications run with z/OS V1R7 Language Environment?

Yes, in nearly all situations, existing well-behaved Language Environment applications can be run with z/OS Language Environment without any modifications. A well-behaved application is one that relies on documented interfaces only.

Example: The *z/OS XL C/C++ Run-Time Library Reference* states that the `remove()` function returns a nonzero return code when a failure occurs. The following code fragments show the correct and incorrect ways to call the `remove()` function and to check the return code:

Incorrect method

```
if (remove("my.file") == -1) {
   call_err();
}
     .
     .
     .
```

Correct method

```
if (remove("my2.file") != 0) {
   call_err();
}
     .
     .
     .
```

As of LE/370 V1R3. the value of the return code from the `remove()` function changed. If an LE/370 V1R2 program was coded incorrectly, and checked for a specific value, as in the first code fragment, a source change is required when the code is migrated. This situation is common when an application relies upon undocumented interfaces. However, if the program was coded correctly, and it did not check for a specific nonzero return code, as in the second fragment, no source changes are required.

Will existing C/370 applications work with z/OS V1R7 Language Environment?

A C/370 application is created using the IBM C/370 Version 1 or Version 2 compiler and library, or the AD/Cycle C/370 V1R2 compiler with the `TARGET(COMPAT)` option and the C/370 V2R2 library. A well-behaved C/370 application, in most situations, works with z/OS Language Environment without any modifications.

Two common migration problems that you may encounter relate to interlanguage calls:

- You must relink applications that contain interlanguage calls between C/370 and Fortran before running them with z/OS Language Environment
- You can run them with z/OS Language Environment only after they are relinked. You cannot continue to run them with the C/370 library.

The same rules apply to applications that contain interlanguage calls between C/370 and COBOL, unless you relink them with the C/370 V2R1 or V2R2 library with the PTF for APAR PN74931 applied. This PTF replaces the C/370 V1R2 link-edit stubs so that they tolerate Language Environment. After your application is relinked using the modified C/370 V1R2 stubs, you can run the application with either the C/370 V2 run-time library or with Language Environment. Refer to "Executable programs with interlanguage calls" on page 18 for more information about COBOL and Fortran interlanguage calls.

Though there are other migration items (described in the following chapters) that may affect your application, these are the most serious ones.

My application does not run — now what?

If your application does not run, it may be either a migration problem, or an error in your program that surfaces as a result of a new design feature in the run-time library. Do the following:

1. Verify the concatenation order of your libraries.

   If you have a load module built with both C/370 library parts and z/OS Language Environment parts, ensure that you are not accidentally initializing your environment using the C-PL/I Common Library rather than z/OS Language Environment. The PDS with the low level qualifier SCEERUN (which belongs to z/OS Language Environment), must be concatenated ahead of the PDS with the low level qualifier SIBMLINK (which belongs to the C-PL/I Common Library).

   Refer to the section "Initialization compatibility" on page 19 for more information.

2. Use environment variables to obtain the "Old Behavior".

   Under z/OS Language Environment, you can use the `ENVAR` run-time option to specify the values of environment variables at execution time. With some environment variables, you can specify the "old behavior" for particular items. The following setting provides you with "old behavior" for the greatest number of items:

   ```
   ENVAR("_EDC_COMPAT=32767")
   ```

   The value assigned to `_EDC_COMPAT` is used as a bit mask. If you assign a value of 32767, the library uses "old behavior" for all of the general compatibility items currently defined by `_EDC_COMPAT`. For more information about `_EDC_COMPAT` and its possible values, refer to the *z/OS XL C/C++ Programming Guide*.

   If `_EDC_COMPAT` solves your migration problem, you can use it with the `ENVAR` run-time option, as shown above, or in a call to `setenv()` either in the CEEBINT High-Level Language exit or in your `main()` program. Using CEEBINT only requires you to relink your application, but adding a call to `setenv()` in the `main()` function requires a recompile and obviously a relink. See the *z/OS XL C/C++ Run-Time Library Reference* for more details about `setenv()`. You may also refer to the *z/OS XL C/C++ Programming Guide* for more details about the `setenv()` function.

3. Relink your application.

   You must relink your C/370 application before running it with z/OS Language Environment, if your application:

   - Contains ILCs between C and Fortran, or between C and COBOL.

     Refer to "Executable programs with interlanguage calls" on page 18 for more information.
   - Is an SPC application that uses the library
   - Contains calls to `ctest()`

When you relink an application with z/OS Language Environment, you ensure that it contains no links to non-z/OS Language Environment interfaces.

4. Review the migration items documented in this book.

   If you find a migration item in this manual that you think may affect your application, use the workaround described in this book. If a relink or a setting of an environment variable is not suggested, you must change your source, and then recompile and relink your application.

5. Look for uninitialized storage.

   In some cases, applications will run with uninitialized storage, because the run-time library may inadvertently clear storage, or because the storage location referenced is set to zero.

   Use the `STORAGE` and `HEAP` run-time options to find uninitialized storage. We recommend `STORAGE(FE,DE,BE)` and `HEAP(16,16,ANY,FREE)` to determine if your application is coded correctly. Any uninitialized pointers will fail at first reference instead of accidentally referencing storage locations at random.

   Note: Your program will run slower with these options specified. Do not use them for production, only development.

6. Look for undocumented interfaces.

   It is possible that your application has dependencies on undocumented interfaces. For example, you may have dependencies on library control blocks, specific errno values, or specific return values. Alter your code to use only documented interfaces, and then recompile and relink.

7. If you followed steps 1 on page 12 through 6, but cannot run your existing load module under z/OS Language Environment, contact your System Programmer to determine whether or not all service has been applied to your system.

   Often, the problem you encounter has already been reported to IBM, and a fix is available.

8. If you have verified with your System Programmer that all service has been applied to your system, ask your Service Representative to open a Problem Management Record (PMR) against the applicable IBM product.

   For information on how to open a PMR, see the `APAR` member in data set `CBC.SCCNDOC`.

I attempt to recompile my application and it fails — why?

The compiler no longer supports some features from previous releases. Some of these changes relate to language standards such as ISO. This book describes these changes, and the alterations you may need to make to your code. For example, you can no longer compile or link a program that uses the IBM Open Class Library; instead, you should use the Standard C++ Library.

The amount of memory required by the compiler sometimes changes from release to release. If you cannot recompile an application that you successfully compiled with a previous release of the compiler, try increasing the region size.

Part 2. From C/370 V2 to z/OS XL C

Note: You cannot migrate directly from the C/370 V1 compiler to the z/OS XL C compiler. You must first migrate to the C/370 V2 compiler. If you are migrating a program that has been run successfully only with a C/370 run-time environment, contact your service representative.

This part discusses the implications of migrating applications to the z/OS XL C compiler when applications were created with one of the compilers and one of the libraries from the following lists:

Compilers:

- The IBM C/370 V2 compiler, 5688-187
- The AD/Cycle C/370 V1R2 compiler with the `TARGET(COMPAT)` compiler option, 5688-216

Libraries:

- The IBM C/370 V2 library, 5688-188, and C-PL/1 Common Library, 5688-082

In this part, z/OS V1R7 may also be referred to as z/OS Language Environment, or Language Environment.

Chapter 3. Application executable program compatibility

This chapter will help application programmers understand the compatibility considerations of application executable programs.

An executable program is the output of the prelink/link or bind process. For more information on the relationship between prelinking, linking, and binding, see the section about prelinking, linking, and binding in *z/OS XL C/C++ User's Guide*. The output of this process is a load module when stored in a PDS and a program object when stored in a PDSE or HFS.

Generally, C/370 executable programs execute successfully with z/OS V1R7 Language Environment without source code changes, recompilation, or relinking. This chapter highlights exceptions and shows how to solve specific problems in compatibility.

Executable program compatibility problems requiring source changes are discussed in Chapter 4, "Source program compatibility," on page 25.

Note: The terms in this section having to do with linking (bind, binding, link, link-edit) refer to the process of creating an executable program from object modules.

Input and output operations

If programs that ran successfully with the C/370 V2R1 library have dependencies on any of the input and output behaviors listed in Chapter 6, "Input and output operations compatibility," on page 39, these programs might require source changes before they can run under z/OS Language Environment.

Executable programs that invoke Debug Tool or dbx

When migrating your application from C/370 to z/OS V1R7 Language Environment, you must relink modules that contain calls to `ctest()`. The old library object, @@CTEST, must be replaced as described in "Converting old executable programs to new executable programs" on page 20 and in "Considerations for Interlanguage Call (ILC) applications" on page 21. After you replace the old objects, the new modules are executable under z/OS Language Environment.

System Programming C Facility (SPC) executable programs

There are two types of SPC programs: the ones that still require the run-time library, and the ones that do not.

With z/OS Language Environment, only the SPC executable programs that use the z/OS XL C/C++ run-time library need to be relinked. You can relink applications either from executable programs or from text decks using the z/OS Language Environment text libraries.

- If you relink from text decks, you can use the JCL that originally built the application. However, you must modify it to point to the z/OS Language Environment static or resident library (SCEELKED).
- If you relink from executable programs, you will need to do a CSECT replacement for the appropriate part, such as `EDCXSTRL`, `EDCXENVL`, and `EDCXHOTL`.

Note: If your SPC module has been built with exception handling, the automatic library call is not enabled when you relink, so you must explicitly include the new routine @@SMASK.

Executable programs with interlanguage calls

You must relink C/370 executable programs that contain interlanguage calls (ILCs) to or from COBOL before you can execute them under z/OS Language Environment. Old executable programs that contain ILCs to and from assembler or PL/I language modules do not need to be relinked.

Before you can relink your C/370-COBOL ILC application under the C/370 V2R2 library so that it can run under either the C/370 V2R2 library or Language Environment, you must obtain and apply the PTF for APAR PN74931 for the V2R1 or V1R2 link-edit stubs. This PTF replaces the link-edit stubs so that they tolerate Language Environment. After your application is relinked using the modified V2, you can run the application with either the V2R1 or V2R2 run-time library, or with Language Environment.

Before you can relink your C/370-COBOL ILC application so that it will only run under z/OS Language Environment, you must replace the old library objects @@C2CBL and @@CBL2C, as described in “Converting old executable programs to new executable programs” on page 20 and “Considerations for Interlanguage Call (ILC) applications” on page 21. After you replace the old objects, the new modules will be executable only under z/OS Language Environment.

Fortran-C ILC was not supported prior to Language Environment V1R5 and C/MVS V3R1, for Language Environment-conforming applications. Before you can use Fortran and C ILC routines, you must relink all Fortran-C ILC applications containing pre-Language Environment C or Fortran library routines.

Table 3 outlines when a relink of ILC applications is required, based on languages found in the executable program:

Table 3. Migrations that require relinking

Language	Relink required
Assembler	No
PL/I	No
Fortran	Yes
COBOL	Yes **Note:** If the C/370 ILC application is built (relinked) after the PTF for APAR PN74931 is applied, no relink is required to run under z/OS V1R7 XL C/C++. Otherwise a relink is required.

Note: If you have multiple languages in the executable program, then the sum of the restrictions applies. For example: if you have C, PL/I and Fortran in the executable program, then it should be relinked because Fortran needs to be relinked.

Refer to *z/OS Language Environment Writing Interlanguage Communication Applications* for more information.

Initialization compatibility

Both z/OS V1R7 Language Environment and C/370 modules use static code and dynamic code. Static code sections are emitted or bound with the main program object. Dynamic code sections are loaded and executed by the static component.

The sequence of events during initialization for C/370 modules differs from that for z/OS V1R7 Language Environment modules. The key static code for both C/370 and z/OS Language Environment modules is an object named CEESTART, which controls initialization at execution. Its contents differ between the products, thus there is an old and a new version of CEESTART. The key dynamic code for z/OS Language Environment is CEEBINIT, which is stored in SCEERUN. The key dynamic code for IBM C/370 Version 1 and Version 2 is IBMBLIIA, which is a Common Library part stored in SIBMLINK. The Common Library is used by the C/370 V2 libraries.

Initialization schemes

The tables in this section describe the initialization schemes for the CEESTART and IBMBLIIA modules:

- Table 4 describes the initialization scheme for IBM C/370 Version 1 and Version 2.
- Table 5 describes the initialization scheme for z/OS Language Environment.
- Table 6 describes the z/OS Language Environment initialization scheme for C/370 executable programs.

The following describes the IBM C/370 Version 1 and Version 2 initialization scheme:

Table 4. IBM C/370 Version 1 and Version 2 initialization scheme

Stage	Description
Load	The old CEESTART loads IBMBLIIA.
Initialize	IBMBLIIA initializes the Common Library.
Run	The Common Library runs C/370-specific initialization.
Call	The main program is called.

The following describes the initialization scheme:

Table 5. z/OS Language Environment initialization scheme

Stage	Description
Load	The new CEESTART loads CEEBINIT.
Initialize	CEEBINIT initializes z/OS Language Environment.
Run	z/OS Language Environment C-specific initialization is run.
Call	The main program is called.

Table 6. z/OS Language Environment initialization scheme for C/370 executable programs

Stage	Description
Load	The old CEESTART loads CEEBLIIA (as IBMBLIIA).
Initialize	CEEBLIIA (IBMBLIIA) initializes z/OS Language Environment.
Run	z/OS Language Environment C-specific initialization is run.

Table 6. z/OS Language Environment initialization scheme for C/370 executable programs (continued)

Stage	Description
Call	The main program is called.

In Table 6 on page 19, compatibility with old executable programs depends upon the program's ability to intercept the initialization sequence at the start of the dynamic code and to perform the z/OS Language Environment initialization at that point. This interception is done by providing a part named CEEBLIIA, which has been assigned the alias IBMBLIIA. This provides "initialization compatibility".

Special considerations: CEEBLIIA and IBMBLIIA

The only way to control which environment is initialized for a given old executable program (when CEEBLIIA is assigned the alias of IBMBLIIA) is to correctly arrange the concatenation of libraries.

The version of IBMBLIIA that is found first determines the environment (Language Environment or Common Library) that is initialized.

- If you intend to initialize the Common Library environment, ensure that SIBMLINK is concatenated before SCEERUN.
- If you intend to initialize the z/OS Language Environment environment, ensure that SCEERUN is concatenated before SIBMLINK.

Converting old executable programs to new executable programs

Some sites might have some old executable programs that will require the C/370 Common Library environment unless they have been converted to use z/OS Language Environment. These are incompatible modules that, for example, contain ILCs to COBOL or that use the library function `ctest()` to invoke the Debug Tool.

There are three different methods of converting old modules to new modules, so that they will run under z/OS Language Environment:

- Link from original objects using z/OS Language Environment. EDCSTART and CEEROOTB must be explicitly included.
- Relink the old executable program with z/OS Language Environment using CSECT replacement. EDCSTART and CEEROOTB must be explicitly included.

  Figure 1 on page 21 shows an example of a job that uses this method. The job converts an old executable program to a new executable program by relinking it and explicitly including the z/OS Language Environment CEESTART to replace the old C/370 CEESTART.

  This is a general-purpose job. The comments show the other include statements that are necessary if certain calls are present in the code. Refer to "Considerations for Interlanguage Call (ILC) applications" on page 21 for the specific control statements that are necessary for different kinds of ILCs with COBOL.

```
//Jobcard information
//*
//*******************************************************************************//
//*RELINK C/370 V1 or V2 USER MODULE FOR Language Environment                     *//
//*******************************************************************************//
//*
//*
//LINK     EXEC PGM=HEWL,PARM='RMODE=ANY,AMODE=31,MAP,LIST'
//SYSPRINT DD SYSOUT=*
//SYSLIB   DD DSN=CEE.SCEELKED,DISP=SHR
//SYSLMOD  DD DSN=TSUSER1.A.LOAD,DISP=SHR
//SYSUT1   DD UNIT=VIO,SPACE=(CYL,(10,10))
//SYSLIN   DD *
  INCLUDE SYSLIB(EDCSTART)   ALWAYS NEEDED
  INCLUDE SYSLIB(CEEROOTB)   ALWAYS NEEDED
  INCLUDE SYSLIB(@@CTEST)    NEEDED ONLY IF CTEST CALLS ARE PRESENT
  INCLUDE SYSLIB(@@C2CBL)    NEEDED ONLY IF CALLS ARE MADE TO COBOL
  INCLUDE SYSLIB(@@CBL2C)    NEEDED ONLY IF CALLS ARE MADE FROM COBOL
  INCLUDE SYSLMOD(HELLO)
  ENTRY   CEESTART
  NAME    HELLO(R)
/*
```

Figure 1. Link Job for Converting Executable Programs

- *For modules that have a C `main()`:* Replace the old executable program by recompiling the source (if available). Recompile the source containing the `main()` function with the z/OS V1R7 XL C/C++ compiler, and then relink the objects with z/OS Language Environment. This creates a version of CEESTART for z/OS Language Environment. This is an alternative to explicitly including EDCSTART when linking from objects.

Considerations for Interlanguage Call (ILC) applications

This section lists the linkage editor control statements required to relink modules that contain ILCs between C and COBOL, and C and Fortran. The object modules are compatible with the z/OS Language Environment; however, the ILC linkage between the applications and the library has changed. You must relink these applications using the JCL shown in Figure 1 and the control statements that fit your requirements from the following list. The `INCLUDE SYSLIB(@@CTDLI)` is only necessary if your program will invoke IMS™ facilities using the z/OS XL C library function `ctdli()` and if the z/OS XL C function was called from a COBOL main program.

Control statements for various combinations of ILCs and compiler options are as follows. The modules referenced by SYSLMOD contain the routines to be relinked.

1. C `main()` statically calling COBOL routine B1 or dynamically calling the COBOL routine through the use of fetch(), where B1 was compiled with the RES option. Relink the C module:

   ```
   MODE    AMODE(31),RMODE(ANY)
   INCLUDE SYSLIB(EDCSTART)     ALWAYS NEEDED
   INCLUDE SYSLIB(CEEROOTB)     ALWAYS NEEDED
   INCLUDE SYSLIB(@@C2CBL)      REQUIRED FOR C CALLING COBOL
   INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
   INCLUDE SYSLMOD(SAMP1)
   ENTRY   CEESTART             MAIN ENTRY POINT
   NAME    SAMP1(R)
   ```

2. C `main()` statically calling COBOL routine B2 or dynamically calling the COBOL routine through the use of fetch(), where B2 was compiled with the NORES option. Relink the C module:

```
MODE    AMODE(24),RMODE(24)
INCLUDE SYSLIB(EDCSTART)     ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)     ALWAYS NEEDED
INCLUDE SYSLIB(@@C2CBL)      REQUIRED FOR C CALLING COBOL
INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
INCLUDE SYSLIB(IGZENRI)      REQUIRED FOR COBOL with NORES
INCLUDE SYSLMOD(SAMP2)
ENTRY   CEESTART             MAIN ENTRY POINT
NAME    SAMP2(R)
```

3. C `main()` fetches a C1 function that statically calls a COBOL routine B1 compiled with the RES option. Relink the C module:

```
MODE    AMODE(31),RMODE(ANY)
INCLUDE SYSLIB(EDCSTART)     ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)     ALWAYS NEEDED
INCLUDE SYSLIB(@@C2CBL)      REQUIRED FOR C CALLING COBOL
INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP3)
ENTRY   C1                   ENTRY POINT TO FETCHED ROUTINE
NAME    SAMP3(R)
```

4. C `main()` fetches a C1 function that statically calls a COBOL routine B1 that is compiled with the NORES option. Relink the C module:

```
MODE    AMODE(24),RMODE(24)
INCLUDE SYSLIB(EDCSTART)     ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)     ALWAYS NEEDED
INCLUDE SYSLIB(@@C2CBL)      REQUIRED FOR C CALLING COBOL
INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
INCLUDE SYSLIB(IGZENRI)      REQUIRED FOR COBOL with NORES
INCLUDE SYSLMOD(SAMP4)
ENTRY   C1                   ENTRY POINT TO FETCHED ROUTINE
NAME    SAMP4(R)
```

5. A COBOL main CBLMAIN compiled with the RES option statically or dynamically calls a C1 function. Relink the COBOL module:

```
MODE    AMODE(31),RMODE(ANY)
INCLUDE SYSLIB(EDCSTART)     ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)     ALWAYS NEEDED
INCLUDE SYSLIB(IGZEBST)
INCLUDE SYSLIB(@@CBL2C)      REQUIRED FOR COBOL CALLING C
INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP5)
ENTRY   CBLRTN               COBOL ENTRY POINT
NAME    SAMP5(R)
```

6. A COBOL main CBLMAIN compiled with the NORES option statically or dynamically calls a C1 function. Relink the COBOL module:

```
MODE    AMODE(24),RMODE(24)
INCLUDE SYSLIB(EDCSTART)     ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)     ALWAYS NEEDED
INCLUDE SYSLIB(IGZENRI)
INCLUDE SYSLIB(@@CBL2C)      REQUIRED FOR COBOL CALLING C
INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP6)
ENTRY   CBLRTN               COBOL ENTRY POINT
NAME    SAMP6(R)
```

7. C `main()` calls a Fortran routine. Relink the C module:

```
INCLUDE SYSLIB(EDCSTART)     ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)     ALWAYS NEEDED
INCLUDE SYSLIB(@@CTOF)       REQUIRED FOR C CALLING Fortran
```

```
INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP7)
ENTRY   CEESTART             MAIN ENTRY POINT
NAME    SAMP7(R)
```

8. A Fortran `main()` calls a C function. Relink the C module:

```
INCLUDE SYSLIB(EDCSTART)     ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)     ALWAYS NEEDED
INCLUDE SYSLIB(@@FTOC)       REQUIRED FOR Fortran CALLING C
INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP8)
ENTRY   CEESTART             MAIN ENTRY POINT
NAME    SAMP8(R)
```

For other related Fortran considerations, refer to *z/OS Language Environment Programming Guide*.

Chapter 4. Source program compatibility

This chapter describes the changes that you may have to make to your source code when moving applications to the z/OS V1R7 XL C/C++ product.

Note: You cannot migrate directly from the C/370 V1 compiler to the z/OS XL C compiler. You must first migrate to the C/370 V2 compiler. If you are migrating a program that has been run successfully only with a C/370 run-time environment, contact your service representative.

It considers programs created with one of the following compilers and one of the following libraries.

Compilers:

- The IBM C/370 V2 compiler, 5688-187
- The AD/Cycle C/370 V1R2 compiler with the `TARGET(COMPAT)` compiler option, 5688-216

Libraries:

- The IBM C/370 V2 library, 5688-188, and C-PL/1 Common Library, 5688-082

Chapter 5, “Other migration considerations,” on page 31 has information on run-time options, which may also affect source code compatibility.

Pointer considerations

According to the *ISO C Standard*, pointers to `void` types and pointers to functions are incompatible types. The C/370 V2, AD/Cycle C/370, C/MVS V3, and z/OS XL C compilers perform some type checking, such as in assignments, argument passing on function calls, and function return codes.

Note: If you are not conforming to ISO rules for the use of pointer types, your run-time results may not be as expected, especially when you are using the `OPTIMIZE` compiler option.

Example: With the C/370 V2, the AD/Cycle C/370, and the C/MVS V3 compilers, you could not assign `NULL` to an integer value. The following was not allowed:

```
int i = NULL;
```

With the C/MVS V3R2 and z/OS XL C compilers, you can assign `NULL` pointers to void types if you specify `LANGLVL(COMMONC)` when you compile your program.

Input and output operations

If programs that ran successfully with the C/370 V2R1 library have dependencies on any of the input and output behaviors listed in Chapter 6, “Input and output operations compatibility,” on page 39, these programs might require source changes before they can run under z/OS Language Environment.

Note: You cannot migrate directly from the C/370 V1 compiler to the z/OS XL C compiler. You must first migrate to the C/370 V2 compiler. If you are migrating a program that has been run successfully only with a C/370 run-time environment, contact your service representative.

SIGFPE exceptions

Decimal overflow conditions were masked in the C/370 library before V2R2. The conditions were enabled when the packed decimal data type was introduced in the AD/Cycle C/370 V1R2 compiler, and continue to be enabled with z/OS V1R7 Language Environment. If you have old load modules that accidentally generated decimal overflow conditions, these modules may raise unexpected `SIGFPE` exceptions in the z/OS Language Environment. You cannot migrate such modules to the new library without altering the source.

Note: It is unlikely that such modules are present in a C-only environment. These unexpected exceptions may occur in mixed language modules, particularly those using C and assembler code where the assembler code explicitly manipulates the program mask.

Program mask manipulations

Programs created with the C/370 V1 or V2R1 compiler and library that explicitly manipulated the program mask may require source alteration to execute correctly under z/OS Language Environment. Changes are required if you have one of the following types of programs:

- A C program containing interlanguage calls (ILCs), where the invoked code uses the S/370™ decimal instructions that might generate an unmasked decimal overflow condition, requires modification for migration. There are two methods for migrating the code. The first one is preferred:
  - If the called routine is assembler, save the existing mask, set the new value, and when finished restore the saved mask.
  - Change the C code so that the produced `SIGFPE` signal is ignored in the called code. **Example:** In the following example, the `SIGNAL` calls surround the overflow-producing code. The `SIGFPE` exception signal is ignored, and then reenabled:

```
signal(SIGFPE, SIG_IGN);   /* ignore exceptions */
  ...
  callit():                  /* in called routine */
  ...
  signal(SIGFPE, SIG_DFL);   /* restore default handling */
```

- A C program containing assembler ILCs that explicitly alter the program mask, and do not explicitly save and restore it, also requires modification for migration.

  If user code explicitly alters the state of the program mask, the value before modification must be saved, and the value restored to its former value after the modification. You must ensure that the decimal overflow program mask bit is enabled during the execution of C code. Failure to preserve the mask may result in unpredictable behavior.

These changes also apply in a System Programming C environment, and to Customer Information Control System (CICS®) programs in the handling and management of the PSW mask.

The realloc() function

When the `realloc()` function is used with z/OS Language Environment, a new area is always obtained and the data is copied. This is different from IBM C/370 V2, where, if the new size was equal to or less than the original size, the same area was used.

Programmers may want to ensure that their source code has no dependencies on the behavior of the old version of the `realloc()` function, so that their code is compatible with z/OS Language Environment.

Fetched main programs

IBM C/370 V2 programs that are fetched must be recompiled without a main entry point. Under z/OS Language Environment, any attempt to fetch a main program will fail.

User exits

If both CEEBXITA and IBMBXITA are present in a relinked C/370 module, CEEBXITA will have precedence over IBMBXITA.

Line number control

The AD/Cycle C/370 V1R2 compiler ignored the `#line` directive when either the `EVENTS` or the `TEST` compiler option was in effect. The z/OS XL C compiler never ignores `#line`.

The sizeof operator

Example: The following is an example of how the behavior of `sizeof`, when applied to a function return type, was changed in the C/C++ MVS V3R2 compiler:

```
char foo();
..
s = sizeof foo();
```

If the example is compiled with a compiler prior to C/C++ MVS V3R2, `char` is widened to `int` in the return type, so `sizeof` returns `s = 4`.

If the example is compiled with C/C++ MVS V3R2, or with any OS/390 C/C++ compiler, the size of the original char type is retained. In the above example, `sizeof` returns `s = 1`. The size of the original type of other data types such as `short`, and `float` is also retained.

With the OS/390 V2R4 C/C++ and subsequent compilers, you can use `#pragma wsizeof` or the `WSIZEOF` compiler option to get `sizeof` to return the widened size for function return types if your code has a dependency on this behavior. For more information on `#pragma wsizeof`, see *z/OS XL C/C++ Language Reference*. For more information on the `WSIZEOF` compiler option, see *z/OS XL C/C++ User's Guide*.

System Programming C (SPC) applications built with EDCXSTRX

If you have SPC applications that are built with EDCXSTRX and that use dynamic C library functions, note that the name of the C library function module has changed from EDCXV in C/370 V2 to CEEEV003 in z/OS Language Environment. Change the name from EDCXV to CEEEV003 in the assembler source of your program that loads the library, and reassemble.

The __librel() function

The `__librel()` function is a System/370™ extension to SAA C. It returns the release level of the library that your program is using, in a 32-bit integer. In z/OS Language Environment, it has a field containing a number that represents the library product.

The `__librel()` return value is a 32-bit integer intended to be viewed in hexadecimal format as shown in Table 7. The hexadecimal value is interpreted as 0x*PVRRMMMM*, where:

- The first hex digit *P* represents the product.
- The second hex digit *V* represents the version.
- The third and fourth hex digits *RR* represent the release.
- The fifth through eighth hex digits *MMMM* represent the modification level.

Table 7. The librel return values for supported products

Product	librel value
C/370 V2R2	0x02020000
LE V1R5	0x11050000
OS/390 V1R2	0x21020000
OS/390 V1R3	0x21030000
OS/390 V2R4	0x22040000
OS/390 V2R6	0x22060000
OS/390 V2R7	0x22070000
OS/390 V2R8	0x22080000
OS/390 V2R9	0x22090000
OS/390 V2R10	0x220A0000
z/OS V1R1	0x220A0000
z/OS V1R2	0x41010000
z/OS V1R3	0x41030000
z/OS V1R4	0x41040000
z/OS V1R5	0x41050000
z/OS V1R6	0x41060000
z/OS V1R7	0x41070000

In IBM C/370 V2, the high-order 8 bits were used to return the version number. Now these 8 bits are divided into 2 fields. The first 4 bits contain the product number and the second 4 bits contain the version number.

You must modify programs that use the information returned from `__librel()`. For more information on `__librel()`, see the *z/OS XL C/C++ Run-Time Library Reference*.

Library messages

There are differences in messages between C/370 and z/OS Language Environment. Some run-time messages have been added and some have been deleted; the contents of others have been changed. Any application that is affected by the format or contents of these messages must be updated accordingly. **Do not build dependencies on message contents or message numbers.**

Refer to *z/OS Language Environment Debugging Guide* for details on run-time messages and return codes.

Prefix of perror() and strerror() messages

All `perror()` and `strerror()` messages in C under z/OS Language Environment contain a prefix (in IBM C/370 Version 1 and Version 2 there were no prefixes to these messages). The prefix is EDC*xxxxa*, where *xxxx* is a number (always *5xxx*) and the *a* is either `I`, `E`, or `S`. See *z/OS Language Environment Run-Time Messages* for a list of these messages.

Compiler messages and return codes

There are differences in messages and return codes between the C/370 compilers and the z/OS XL C compiler. Message contents have changed, and return codes for some messages have changed (errors have become warnings, and the other way around). Any application that is affected by message content or return codes must be updated accordingly. **Do not build dependencies on message content, message numbers, or return codes**. See *z/OS XL C/C++ Messages* for a list of messages.

_Packed structures and unions

With the z/OS XL C compiler, you can no longer do the following:

- Assign `_Packed` and non-`_Packed` structures to each other
- Assign `_Packed` and non-`_Packed` unions to each other
- Pass a `_Packed` union or `_Packed` structure as a function parameter if a non-`_Packed` version is expected (or the other way around)

If you attempt to do so, the compiler issues an error message.

Alternate code points

The following alternate code points are not supported by the z/OS XL C compiler:

- X'8B' as alternate code point for X'C0' (the left brace)
- X'9B' as alternate code point for X'D0' (the right brace)

These alternate code points were supported by the C/370 and AD/Cycle C/370 compilers (the `NOLOCALE` option was required if you were using the AD/Cycle C/370 V1R2 compiler).

Chapter 5. Other migration considerations

This chapter provides additional considerations on migrating applications to z/OS V1R7 XL C/C++ that were created with one of the following compilers, and with one of the following libraries.

Compilers:

- The IBM C/370 V2 compiler, 5688-187
- The AD/Cycle C/370 V1R2 compiler with the `TARGET(COMPAT)` compiler option, 5688-216

Libraries:

- The IBM C/370 V1 library, 5688-039, and C-PL/1 Common Library, 5688-082
- The IBM C/370 V2 library, 5688-188, and C-PL/1 Common Library, 5688-082

Changes that affect user JCL, CLISTs, and EXECs

This section describes changes that may affect your JCL, CLISTs and EXECs.

Return codes and messages

Library return codes and messages have been changed, and JCL, CLISTs and EXECs that are affected by them must be changed accordingly (or else the CEEBXITA exit must be customized to emulate the old return codes). IBM C/370 Version 1 and Version 2 return codes were from 0 to 999. However, the z/OS Language Environment return codes have a different range. These return codes are documented in *z/OS Language Environment Debugging Guide*.

Return codes greater than 4095 are returned as modulo 4095 return codes. The return code for an `abort` is now 2000; it was 1000. The return code for an unhandled `SIGFPE`, `SIGILL`, or `SIGSEGV` condition is now 3000; it was 2000.

Compiler message contents and return codes have changed. You must change JCL, CLISTs, and EXECs that are affected by them. Refer to “Compiler messages and return codes” on page 29 for more information.

Changes in data set names

The names of IBM-supplied data sets may change from one release to another. See the z/OS Program Directory for more information on data set names.

Differences in standard streams

Under z/OS Language Environment there is no longer an automatic association of ddnames `SYSTERM`, `SYSERR`, `SYSPRINT` with `stderr`. Command line redirection of the type `1>&2` is necessary in batch to cause `stderr` and `stdout` to share a device.

In IBM C/370 Version 1 and Version 2, you could override the destination of error messages by redirecting `stderr`. z/OS Language Environment determines the destination of all messages from the `MSGFILE` run-time option. See the section on the `MSGFILE` run-time option in the *z/OS Language Environment Programming Guide* for more information.

Passing command-line parameters to a program

In IBM C/370 Version 1 or Version 2, if an error was detected with the parameters being passed to the main program, the program terminated with a return code of 8 and a message indicating the reason why the program was not run. For example, if there was an error in the redirection parameters, the message would indicate that the program had terminated because of a redirection error.

Under z/OS Language Environment, the same message will be displayed, but the program will also terminate with a 4093 abend, reason code 52 (x'34'). For more information about reason codes see *z/OS Language Environment Debugging Guide*.

SYSMSGS ddname

The method of specifying the language for compiler messages has changed. Instead of specifying a messages data set for the `SYSMSGS` ddname, you must now use the `NATLANG` run-time option. If you specify a data set for the `SYSMSGS` ddname, it will be ignored.

Note: For information about the `NATLANG` run-time option, see the *z/OS Language Environment Customization* and the *z/OS Language Environment Programming Reference*.

CBCI and CBCXI procedures

As of z/OS V1R5, the CBCI and CBCXI procedures contain the variable CLBPRFX. If you have any JCL that uses these procedures, you must either customize these procedures (for example, at installation time) or modify your JCL to provide a value for CLBPRFX.

Run-time options

This section describes changes that may affect your run-time options.

Ending the run-time options list

In C/370 V1 and V2, when passing only run-time options to a C/370 program, you did not have to end the arguments with a slash (/). With z/OS Language Environment, you must end the arguments with a slash.

With z/OS Language Environment, if you have no run-time options and the input arguments passed to `main()` contains a slash, you must prefix the arguments with a slash. JCL, CLISTs, and EXECs that are affected by the slash must be changed accordingly.

ISASIZE, ISAINC, STAE/SPIE, LANGUAGE, and REPORT options

Use the z/OS Language Environment equivalent for the IBM C/370 Version 1 and Version 2 run-time options on the command line and in `#pragma runopts`.

ISASIZE/ISAINC	becomes	`STACK`
LANGUAGE	becomes	`NATLANG`
REPORT	becomes	`RPTSTG`
SPIE/STAE	becomes	`TRAP`

The C/370 run-time options are mapped to z/OS Language Environment equivalents. However, if you do not use the z/OS Language Environment options,

during execution you will get a warning message which cannot be suppressed. JCL, CLISTs and EXECs that are affected by these differences must be changed accordingly.

STACK default size

The default size and increment for the STACK run-time option have changed. If you have not indicated the size and increment, STACK will be allocated differently when your program is running under z/OS Language Environment. The defaults in IBM C/370 Version 1 and Version 2 were 0K size and 0K increment. The defaults under z/OS Language Environment without CICS are 128K size, 128K increment, and BELOW, and with CICS are 4K size, 4080 increment, and ANYWHERE. With CICS the default location has changed to ANYWHERE.

To summarize, in z/OS Language Environment, the IBM-supplied defaults are STACK(128K,128K,BELOW,KEEP) without CICS and STACK(4K,4080,ANYWHERE,KEEP) with CICS.

STACK parameters

The parameters for the STACK run-time option are all positional in z/OS Language Environment; in IBM C/370 Version 1 and Version 2, only the first two were. The keyword parameter could be specified if the first two were omitted. Now, to specify only ANYWHERE you must enter: STACK(,,ANYWHERE).

HEAP default size

The default size and increment for the HEAP run-time option have changed. If you have not indicated the size and increment, HEAP will be allocated differently when running under z/OS Language Environment. The defaults in IBM C/370 Version 1 and Version 2 were 4K size and 4K increment. The defaults under z/OS Language Environment without CICS are 32K size and 32K increment and with CICS are 4K size and 4080 increment.

Two new parameters have been added, initsz24 and incrsz24. They determine how much of the heap is allocated and incremented below the 16M line.

For information about these parameters, see the *z/OS Language Environment Programming Reference*.

To summarize, under z/OS Language Environment, the IBM-supplied defaults are HEAP(32K,32K,ANYWHERE,KEEP,8K,4K) without CICS and HEAP(4K,4080,ANYWHERE,KEEP,4K,4080) with CICS.

HEAP parameters

In IBM C/370 Version 1 and Version 2, the first two of the four parameters for the HEAP option were positional. The keyword parameters could be specified if the first two were omitted. Under z/OS Language Environment, all parameters are positional. To specify only KEEP, you must enter HEAP(,,,KEEP).

Compiler options

This section describes changes that may affect your compiler options.

DECK compiler option

In IBM C/370 V1, the DECK compiler option directed the object module to the data set associated with SYSLIN. With the z/OS XL C compiler, as with the AD/Cycle C/370 and IBM C/370 V2 compilers, the object module is directed to the data set associated with SYSPUNCH.

As of z/OS V1R2, the DECK compiler option is no longer supported. The replacement for DECK functionality that routes output to DD:SYSPUNCH, is to use OBJECT(DD:SYSPUNCH).

HWOPTS compiler option

In IBM AD/Cycle C/370 V1, the HWOPTS compiler option directed the compiler to generate code to take advantage of different hardware. As of z/OS V1R2, the HWOPTS compiler option is no longer supported. The replacement for it is the ARCHITECTURE option.

INLINE compiler option

The defaults for the INLINE compiler option have changed. In the past, the default for the threshold suboption was 250 ACUs (Abstract Code Units). With the C/MVS V3 and z/OS XL C compilers, the default is 100 ACUs.

OMVS compiler option

In IBM AD/Cycle C/370 V1, the OMVS compiler option directed the compiler to use the POSIX.2 standard rules for searching for files specified with #include directives. As of z/OS V1R2, the OMVS compiler option is no longer supported. The replacement for it is the OE option.

OPTIMIZE compiler option

In the C/370 V2R1 and subsequent compilers, OPTIMIZE mapped to OPT(1).

As of OS/390 V2R6, the C compiler maps both OPTIMIZE and OPT(1) to OPT(2).

SEARCH and LSEARCH compiler options

The include file search process has changed. Prior to the C/MVS V3R2 compiler, if you used the LSEARCH option more than once, the compiler would only search the libraries specified for the last LSEARCH option. The z/OS XL C/C++ compiler searches all of the libraries specified for all of the LSEARCH options, from the point of the last NOLSEARCH option.

Similarly, if you specify the SEARCH option more than once, the z/OS XL C/C++ compiler searches all of the libraries specified for all of the SEARCH options, from the point of the last NOSEARCH option. Previously, only the libraries specified for the last SEARCH option were searched.

TEST compiler option

As of the OS/390 C/C++ compilers, the default for the PATH suboption of the TEST option has changed from NOPATH to PATH. Also, the INLINE option is ignored when the TEST option is in effect at OPT(0), but the INLINE option is no longer ignored if OPT(1), OPT(2), or OPT(3) is in effect.

As of C/C++ MVS V3R2, a restriction applies to the TEST compiler option if you are using the z/OS XL C/C++ compiler. Now, the maximum number of lines in a single

source file cannot exceed 131,072. If you exceed this limit, the results from the Debug Tool and z/OS Language Environment Dump Services are undefined.

Language Environment run-time options

If occurrences of `ISASIZE/ISAINC`, `STAE/SPIE`, `LANGUAGE`, and `REPORT runopts` are specified by `#pragma runopts` in your source code, you may want to change them to the z/OS Language Environment equivalent before recompiling. These options are mapped to the z/OS Language Environment equivalent, but if you do not change them, you will get a warning or informational message during compilation.

Changes to putenv()

As of z/OS V1R5, the C/C++ function `putenv()` changed to place the string passed to `putenv()` directly into the array of environment variables. This behavior assures compliance with the POSIX standard. Before the change, the storage used to define the environment variable passed into `putenv()` was not added to the array of environment variables. Instead, the system copied the string into system allocated storage. To restore the previous behavior of `putenv()`, set environment variable `_EDC_PUTENV_COPY` to `YES`.

For additional information on `putenv()` and `_EDC_PUTENV_COPY`, see *z/OS XL C/C++ Run-Time Library Reference*. You may also refer to *z/OS XL C/C++ Programming Guide*, for information on `putenv()` and `_EDC_PUTENV_COPY`.

Precedence of Language Environment over C/370 settings for #pragma runopts directive

If you link together C/370 and z/OS Language Environment object modules, and both modules contain `#pragma runopts`, the `#pragma runopts` settings in the Language Environment object module will take precedence.

System Programming C (SPC) Facility applications with #pragma runopts

If you code a program for use in the SPC environment and you use `#pragma runopts` to specify the heap or stack directives, the z/OS XL C compiler will expand these directives according to the z/OS Language Environment defaults and rules. Thus, the program may behave differently under z/OS Language Environment.

Decimal exceptions

z/OS Language Environment provides support for the packed decimal overflow exception using native S/390® hardware enablement (as did the C/370 V2R2 library).

The value of the program mask in the program status word (PSW) is 4 (decimal overflow enabled).

Migration and coexistence considerations

The following points identify migration and coexistence considerations for user applications:

- CICS programs running under z/OS Language Environment are enabled for decimal exceptions.

- The C packed decimal support routines are not supported in an environment that exploits asynchronous events.

SIGTERM, SIGINT, SIGUSR1, and SIGUSR2 exceptions

There are changes to application/program behavior for `SIGTERM`, `SIGINT`, `SIGUSR1`, and `SIGUSR2` exceptions from C/370 V2.

The differences or incompatibilities are:

- The defaults for the `SIGINT`, `SIGTERM`, `SIGUSR1`, and `SIGUSR2` signals changed in LE/370 Release 3, from what they were in C/370 V1 and V2 and LE/370 V1R1 and V2R2. These changes were carried into z/OS V1R7 Language Environment. In the C/370 library and LE/370 V1R1 and V1R2, the defaults for `SIGINT`, `SIGUSR1`, and `SIGUSR2` were to ignore the signals. As of LE/370 V1R3, the defaults are to terminate the program and return a return code of 3000. For `SIGTERM`, the default has always been to terminate the program, but the return code is now 3000 whereas before it was 0.
- Applications that terminate abnormally will **not** drive the `atexit` list.

Running different versions of the libraries under CICS

You cannot run two different versions of the C/370 run-time libraries within one CICS region.

Sometimes a C/370 Version 2 CICS interface (EDCCICS) and the z/OS Language Environment CICS interface can be present in a CICS system through CEDA/PPT definitions and inclusion of modules in the APF STEPLIB. Even if both versions are present, the z/OS Language Environment version will be initialized by CICS when the region is initialized.

CICS abend codes and messages

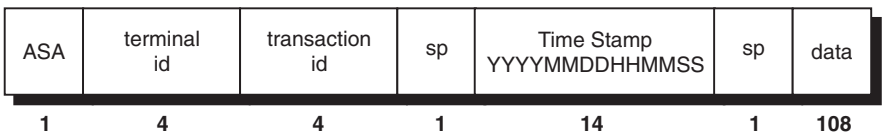
Abend codes such as ACC2 that were used by IBM C/370 Version 1 and Version 2 under CICS are not issued under z/OS Language Environment. An equivalent z/OS Language Environment abend code is issued instead; for example, 4nnn.

CICS reason codes

Reason codes that appeared in the CICS message console log have been changed. The new ones are documented in the *z/OS Language Environment Debugging Guide*.

Standard stream support under CICS

Under CICS, with z/OS Language Environment, records sent to the transient data queues associated with `stdout` and `stderr` with default settings take the form of a message as follows:



where:

ASA is the carriage-control character

terminal id	is a 4-character terminal identifier
transaction id	is a 4-character transaction identifier
sp	is a space
Time Stamp	is the date and time displayed in the format YYYYMMDDHHMMSS
data	is the data sent to the standard streams `stdout` and `stderr`.

C/370 V2 used a different format.

stderr output under CICS

Output from `stderr` is sent to the CICS transient data queue, CESE. CESE is also used by z/OS Language Environment for run-time error messages, dumps, and storage reports. If you previously used this file exclusively for C/370 `stderr` output, you should note that the output may be different.

Transient data queue names under CICS

Transient data queue names are mapped as follows under z/OS Language Environment:

Old name	New name
CCSI	CESI
CCSO	CESO
CCSE	CESE

HEAP option used with the interface to CICS

In C/370 V1R2 and V2, the location of heap storage under CICS was primarily determined by the residence mode (RMODE) of the program.The logic for determining the location of heap was as follows:

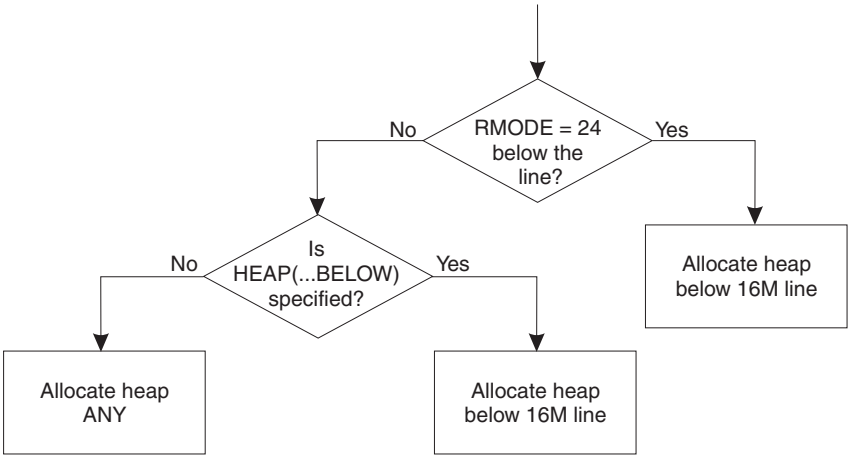


Figure 2. Heap location logic

With z/OS Language Environment, the location of heap storage is determined only by the HEAP(...ANYWHERE|BELOW) options. RMODE does not affect where the heap is allocated. Where the location of heap storage is important, you may want to change source accordingly.

COBOL library routines

All of the language libraries in z/OS Language Environment are packaged as a single unit in SCEERUN. Because of this packaging, for C-only applications, z/OS V1R7 Language Environment

Chapter 6. Input and output operations compatibility

Changes were made to input and output support in the C/370 V2R2 and LE/370 V1R3 libraries. These changes also apply to z/OS V1R7 Language Environment. If your programs performed input and output operations with the C/370 V2R1 library, you should read the changes listed in this section.

References in this chapter to previous releases or previous behavior apply to the products listed above.

You will generally be able to migrate "well-behaved" programs: programs that do not rely on undocumented behavior, restrictions, or invalid behaviors of previous releases. For example, if library documentation only specified that a return code was a negative value, and your code relies on that value being `-3`, your code is not well-behaved and is relying on undocumented behavior.

Another example of a program that is not well-behaved is one that specifies `recfm=F` for a terminal file and depends on Language Environment to ignore this parameter, as it did previously.

However, you may still need to change even well-behaved code under circumstances described in the following section.

Opening files

- When you call the `fopen()` or `freopen()` library function, you can specify each parameter only once. If you specify any keyword parameter in the *mode* string more than once, the function call fails. Previously, you could specify more than one instance of a parameter.
- The library no longer supports uppercase open modes on calls to `fopen()` or `freopen()`. You must specify, for example, `rb` instead of `RB`, to conform to the ANSI/ISO standard.
- You cannot open a `non-HFS` file more than once for a write operation. Previous releases allowed you, in some cases, to open a file for write more than once. For example, you could open a file by its data set name and then again by its ddname. This is no longer possible for `non-HFS` files, and is not supported.
- Previously, `fopen()` allowed spaces and commas as delimiters for mode string parameters. Only commas are allowed now.
- If you are using PDSs or PDSEs, you cannot specify any spaces before the member name.

Writing to files

- Write operations to files opened in binary mode are no longer deferred. Previously, the library did not write a block that held *nn* bytes out to the system until the user wrote *nn*+1 bytes to the block. The z/OS Language Environment library follows the rules for full buffering, described in *z/OS XL C/C++ Programming Guide*, and writes data as soon as the block is full. The *nn* bytes are still written to the file, the only difference is in the timing of when it is done.
- For non-terminal files, the backspace character (`'\b'`) is now placed into files as is. Previously, it backed up the file position to the beginning of the line.
- For all text I/O, truncation for `fwrite()` is now handled the same way that it is handled for `puts()` and `fputs()`. If you write more data than a record can hold,

and your output data contains any of the terminating control characters, '\n' or '\r' (or '\f', if you are using ASA), the library still truncates extra data; however, recognizing that the text line is complete, the library writes subsequent data to the next record boundary. Previously, fwrite() stopped immediately after the library began truncating data, so that you had to add a control character before writing any more data.

- You can now partially update a record in a file opened with type=record. Previous libraries returned an error if you tried to make a partial update to a record. Now, a record is updated up to the number of characters you specify, and the remaining characters are untouched. The next update is to the next record.
- z/OS Language Environment blocks files more efficiently than some previous libraries did. Applications that depend on the creation of short blocks may fail.
- The behavior of ASA files when you close them has changed. In previous releases, this is what happened:

Written to file	Read from file after fclose(), fopen()
abc\n\n\n	abc\n\n\n\n
abc\n\n	abc\n\n\n
abc\n	abc\n

In this release, you read from the file what you wrote to it. For example:

Written to file	Read from file after fclose(), fopen()
abc\n\n\n	abc\n\n\n
abc\n\n	abc\n\n
abc\n	abc\n

In previous products, writing a single new-line character to a new file created an empty file under MVS. z/OS Language Environment treats a single new-line character written to a new file as a special case, because it is the last new-line character of the file. The library writes a single blank to the file. When you read this file, you see two new-line characters instead of one. You also see two new-line characters on a read if you have written two new-line characters to the file.

The behavior of appending to ASA files has also changed. The following table shows what you get from an ASA file when you:

1. Open an ASA file for write.
2. Write abc.
3. Close the file.
4. Append xyz to the ASA file.
5. Open the same ASA file for read.

Table 8. Appending to ASA files

abc Written to file, fclose() then append xyz	What you read from file after fclose(), fopen()	
	Previous release	New release
abc ==> xyz	\nabc\nxyz\n	same as previous release
abc ==> \nxyz	\nabc\nxyz\n	\nabc\n\nxyz\n
abc ==> \rxyz	\nabc\rxyz\n	\nabc\n\rxyz\n
abc\n ==> xyz	\nabc\nxyz\n	same as previous release
abc\n ==> \nxyz	\nabc\nxyz\n	\nabc\n\nxyz\n

Table 8. Appending to ASA files (continued)

abc Written to file, fclose() then append xyz	What you read from file after fclose(), fopen()	
	Previous release	New release
abc\n ==> \rxyz	\nabc\rxyz\n	\nabc\n\rxyz\n
abc\n\n ==> xyz	\nabc\n\n\nxyz\n	\nabc\n\nxyz\n
abc\n\n ==> \nxyz	\nabc\n\n\nxyz\n	same as previous release
abc\n\n ==> \rxyz	\nabc\n\n\rxyz\n	same as previous release

- The behavior of DBCS strings has changed.
  1. I/O now checks the value of MB_CUR_MAX to determine whether to interpret DBCS characters within a file.
  2. When MB_CUR_MAX is 4, you can no longer place control characters in the middle of output DBCS strings for interpretation. Control characters within DBCS strings are treated as DBCS data. This is true for terminals as well. Previous products split the DBCS string at the '\n' (new-line) control character position by adding an SI (Shift In) control character at the new-line position, displaying the line on the terminal, and then adding an SO (Shift Out) control character before the data following the new-line character. If MB_CUR_MAX is 1, the library interprets control characters within any string, but does not interpret DBCS strings. SO and SI characters are treated as ordinary characters.
  3. When you are writing DBCS data to text files, if there are multiple SO (Shift Out) control-character write operations with no intervening SI (Shift In) control character, the library discards the SO characters, and marks that a truncation error has occurred. Previous products allowed multiple SO control-character write operations with no intervening SI control character without issuing an error condition.
  4. When you are writing DBCS data to text files and specify an odd number of DBCS bytes before an SI control character, the last DBCS character is padded with a X'FE' byte. If a SIGIOERR handler exists, it is triggered. Previous products allowed incorrectly placed SI control-character write operations to complete without any indication of an error.
  5. Now, when an SO has been issued to indicate the beginning of a DBCS string within a text file, the DBCS must terminate within the record. The record will have both an SO and an SI.

Repositioning within files

- The behavior of fgetpos(), fseek() and fflush() following a call to ungetc() has changed. Previously, these functions have all ignored characters pushed back by ungetc() and have considered the file to be at the position where the first ungetc() character was pushed back. Also, ftell() acknowledged characters pushed back by ungetc() by backing up one position if there was a character pushed back. Now:
  - fgetpos() behaves just as ftell()does.
  - When a seek from the current position (SEEK_CUR) is performed, fseek() accounts for any ungetc() character before moving, using the user-supplied offset.
  - fflush() moves the position back one character for every character that was pushed back.

If you have applications that depend on the previous behavior of `fgetpos()`, `fseek()`, or `fflush()`, you may use the new `_EDC_COMPAT` environment variable so that source code need not change to compensate for the new behavior. `_EDC_COMPAT` is described in *z/OS XL C/C++ Programming Guide*.

- For OS I/O to and from files opened in text mode, the `ftell()` encoding system now supports higher blocking factors for smaller block sizes. In general, you should not rely on `ftell()` values generated by code you developed using previous releases of the library. You can try `ftell()` values taken in previous releases for files opened in text or binary format if you set the environment variable `_EDC_COMPAT` before you call `fopen()` or `freopen()`. Do not rely on `ftell()` values saved across program boundaries. `_EDC_COMPAT` is described in *z/OS XL C/C++ Programming Guide*.
- For record I/O, `ftell()` now returns the relative record number instead of an encoded offset from the beginning of the file. You can supply the relative record number without acquiring it from `ftell()`. You cannot use old `ftell()` values for record I/O, regardless of the setting of `_EDC_COMPAT`. `_EDC_COMPAT` is described in *z/OS XL C/C++ Programming Guide*.
- If you have used `ungetc()` to move the file pointer to a position before the beginning of the file, calls to `ftell()` and `fgetpos()` now fail. Previously, `ftell()` returned the value `0` for such calls, but set `errno` to a non-zero value. Previously, `fgetpos()` did not account for `ungetc()` calls. See *z/OS XL C/C++ Programming Guide* for information on how to change `fgetpos()` behavior by using `_EDC_COMPAT`.

  For example, suppose that you are at relative position 1 in the file and `ungetc()` is performed twice. `ftell()` and `fgetpos()` will now report the relative position -1, which is before the start of the file, causing both `ftell()` and `fgetpos()` to fail.
- After you have called `ftell()`, calls to `setbuf()` or `setvbuf()` may fail. Applications should never call I/O functions between calls to `fopen()` or `freopen()` and calls to the functions that control buffering.

Closing and reopening ASA files

The behavior of ASA files when you close and reopen them is now consistent:

Table 9. Closing and reopening ASA files

	Physical record after close					
Written to file	**Previous behavior**			**New behavior**		
`abc`	Char	`abc`	(1)	same as previous release		
	Hex	`4888` `0123`	(1)			
`abc\n`	Char	`abc`	(1)	same as previous release		
	Hex	`4888` `0123`	(1)			
`abc\n\n`	Char	`abc` `0`	(1) (2)	Char	`abc`	(1) (2)

	Hex	`4888`
		`0123`
		`F`
		`0`
			(1)
			
			(2)
				Hex	`4888`
					`0123`
					`4`
					`0`
						(1)
						
						(2)

Table 9. Closing and reopening ASA files (continued)

Written to file	Physical record after close					
	Previous behavior			**New behavior**		
`abc\n\n\n`	Char	`abc` `-`	(1) (2)	Char	`abc`	(1) (2)

	Hex	`4888`
		`0123`
		`6`
		`0`
			(1)
			(2)
				Hex	`4888`
					`0123`
					`4`
					`0`
						(1)
						(2)

`abc\r`	Char	`abc` `+`	(1) (2)	same as previous release		

	Hex	`4888`
		`0123`
		`4`
		`E`
			(1)
			(2)			

`abc\f`	Char	`abc` `1`	(1) (2)	same as previous release		

	Hex	`4888`
		`0123`
		`F`
		`1`
			(1)
			(2)			

Values returned by the fldata() function

There are minor changes to the values that the `fldata()` library function returns. It may now return more specific information in some fields. For more information on `fldata()`, see the "Input and Output" section in *z/OS XL C/C++ Programming Guide*.

Error handling

The general return code for errors is now `EOF`. In previous products, some I/O functions returned `1` as an error code to indicate failure. This caused some confusion because `1` is a possible `errno` value as well as a return code. `EOF` is not a valid `errno` value.

Programs that rely on specific values of `errno` may not run as expected because certain `errno` values have changed. As of OS/390, V1R5 Language Environment, error messages have the format EDC5xxx. You can find the error message information for a particular `errno` value by applying the `errno` value to EDC5xxx (for example, 021 becomes EDC5021), and looking up the EDC5xxx message in *z/OS Language Environment Debugging Guide*.

Miscellaneous

OS/390

- The inheritance model for standard streams now supports repositioning. Previously, if you opened `stdout` or `stderr` in update mode, and then called another C program by using the ANSI-style `system()` function, the program that you called inherited the standard streams, but moved the file position for `stdout` or `stderr` to the end of the file. Now, the library does not move the file position to the end of the file. For text files, the position is moved only to the nearest record boundary not before the current position. This is consistent with the way `stdin` behaves for text files.

- The values for L_tmpnam and FILENAME_MAX have been changed:

Constant	Old values	New values
L_tmpnam	47	1024
FILENAME_MAX	57	1024

- The names produced by the tmpnam() library function are now different. Any code that depends on the internal structure of these names may fail.

VSAM I/O changes

- The library no longer appends an index key when you read from an RRDS file opened in text or binary mode.
- RRDS files opened in text or binary mode no longer support setting the access direction to BWD.

Terminal I/O changes

- The library will now use the actual recfm and lrecl specified in the fopen() or freopen() call that opens a terminal file. Incomplete new records in fixed binary and record files are padded with blank characters until they are full, and the __recfmF flag is set in the fldata() structure.

  Previously, MVS terminals unconditionally set recfm=U. Terminal I/O did not support opening files in fixed format.
- The use of an LRECL value in the fopen() or freopen() call that opens a file sets the record length to the value specified.

  Previous releases unconditionally set the record length to the default values.
- The use of a RECFM value in the fopen() or freopen() call that opens a file sets the record format to the value specified.

  Previous releases unconditionally set the record format to the default values.
- For input text terminals, an input record now has an implicit logical record boundary at LRECL if the size of the record exceeds LRECL. The character data in excess of LRECL is discarded, and a '\n' (new-line) character is added at the end of the record boundary. You can now explicitly set the record length of a file as a parameter on the fopen() call.

  The old behavior was to allow input text records to span multiple LRECL blocks.
- Binary and record input terminals now flag an end-of-file condition with an empty input record. You can clear the EOF condition by using the rewind() or clearerr() library function.

  Previous products did not allow these terminal types to signal an end-of-file condition.
- When an input terminal requires input from the system, all output terminals with unwritten data are flushed in a way that groups the data from the different open terminals together, each separated from the other with a single blank character.

  The old behavior is equivalent to the new behavior, except that two blank characters separate the data from each output terminal.

Part 3. From pre-OS/390 releases of C/C++ to z/OS V1R7 XL C/C++

This part discusses the implications of migrating to the z/OS V1R7 XL C/C++ product when applications were created with one of the following compilers and one of the following libraries:

Compilers:

- AD/Cycle C/370 V1R2 compiler without the `TARGET(COMPAT)` compiler option (refer to Part 2 when the `TARGET(COMPAT)` option is specified), 5688-216
- IBM C/C++ for MVS/ESA V3R1 compiler, 5655-121
- IBM C/C++ for MVS/ESA V3R2 compiler, 5655-121
- IBM OS/390 C/C++ V1R1 compiler, 5645-001

Libraries:

- IBM SAA AD/Cycle Language Environment/370 V1R1, 5688-198
- IBM SAA AD/Cycle Language Environment/370 V1R2, 5688-198
- IBM SAA AD/Cycle Language Environment/370 V1R3, 5688-198
- IBM Language Environment for MVS & VM
- AD/Cycle C/370 Language Support Feature of MVS/ESA SP V5R1, 5655-068 and 5655-069
- C/C++ Language Feature of MVS/ESA SP V5R2, 5655-068 and 5655-069
- IBM OS/390 V1R1 Language Environment, 5645-001

Notes:

1. The OS/390 V1R1 compiler and library are equivalent to the final MVS/ESA compiler and library.
2. As of z/OS V1R2, the C++ compiler supports *Programming languages - C++ (ISO/IEC 14882:2003(E))*. There are some changes in compiler behavior that are incompatible with previous C++ compilers. Refer to Chapter 14, “Migrating to the currently supported Standard C++,” on page 87 for more details.

This part does not discuss converting a C program to C++. The only C++ compiler migration considerations covered are those between different versions of the C++ component of the IBM C/C++ for MVS/ESA compilers and the z/OS V1R7 XL C/C++ compiler.

In this part, references to the products in the first column of the following table also apply to the products in the second column.

Table 10. Product references

References to these products	Also apply to these products
LE/370 V1R3	MVS/ESA SP V5R1AD/Cycle C/370 Language Support Feature
Language Environment R4	C/C++ Language Feature of MVS/ESA SP V5R2
Language Environment R5	C/C++ Language Feature of MVS/ESA SP V5R2M2

Table 10. Product references (continued)

References to these products	Also apply to these products
OS/390 V1R1	IBM C/C++ for MVS/ESA V3R2 compiler and IBM Language Environment for MVS and VM V1R5

Chapter 7. Application executable program compatibility

Generally, C/370 executable programs execute successfully with z/OS V1R7 Language Environment without changing source code, or recompiling or relinking programs.

This chapter:

- Highlights exceptions.
- Helps application programmers understand and resolve the compatibility issues that might occur when they migrate application executable programs from a pre-OS/390 release of C/C++ to z/OS V1R7 XL C/C++.

Executable program compatibility problems that require source changes are discussed in Chapter 8, “Source program compatibility,” on page 51.

Notes:

1. An executable program is the output of the prelink/link or bind process. For more information on the relationship between prelinking, linking, and binding, see the section about prelinking, linking, and binding in *z/OS XL C/C++ User's Guide*.
2. The terms in this section having to do with linking (bind, binding, link, link-edit) refer to the process of creating an executable program from object modules.
3. The output of a prelinking, linking, or binding process depends on where the executable programs are stored:
   - When the programs are stored in a PDS, the output is a *load module*.
   - When the programs are stored in a PDSE or HFS, the output is a *program object*.

Input and output operations

If programs that ran successfully with the C/370 V2R1 library have dependencies on any of the input and output behaviors listed in Chapter 6, “Input and output operations compatibility,” on page 39, these programs might require source changes before they can run under z/OS Language Environment.

System Programming C Facility (SPC) executable programs

If you have an LE/370 V1R1 or V1R2 SPC application that was built with exception handling, you must do the following:

- Relink it with z/OS V1R7 Language Environment, using the SCEESPC data set.
- Explicitly include the new routine @@SMASK because an automatic library call is not enabled when you relink.

Note: When you use the EDCXERR, EDCXABRT and EDCXHDLR options to link object modules, you are building exception handling into the application.

Inheritance of run-time options

As programming issues with the compiler are resolved, there have been changes in the inheritance of run-time options.

Availability of standard streams and memory files with the LINK macro

With LE/370 V1R1MO, when the LINK macro was used to initiate one C `main()` from another, any run-time options specified in calling a child `main()` were ignored. The parent run-time options were inherited. The conditions left unhandled in the child were propagated to the parent.

As of LE/370 V1R1M1, run-time options are no longer propagated.

With LE/370 V1R1MO, using the LINK macro to initiate a child `main()` restricted you from using both standard streams and memory files in the child.As of LE/370 V1R1M1, these restrictions no longer apply. The parent's standard streams and memory files are shared by the child.

Heap or stack shortages with the EXEC CICS LINK command

With LE/370 V1R1, run-time options were inherited from an ancestor whenever an EXEC CICS LINK command was used. Application developers who used `STACK` and `HEAP` to tune CICS C applications had to take particular note of this because a large heap or stack size specified in the first run unit of a transaction chain of run units could cause shortages when it was allocated for each unit.

As of later releases of Language Environment, run-time options are no longer inherited.

STAE and SPIE option mappings to TRAP suboptions

`STAE` and `SPIE` options have been replaced with the `TRAP` option. We recommend that you use the `TRAP` option, not `STAE` and `SPIE`. However, for ease of migration, the `STAE` and `SPIE` options are supported *as long as the `TRAP` option is not explicitly specified*.

Table 11. STAE and SPIE option mappings to TRAP suboptions

If the following legacy options are in effect . . .	The result is that of the following TRAP suboptions
`STAE` and `SPIE`	`TRAP(ON,SPIE)`
`NOSTAE` and `NOSPIE`	`TRAP(OFF)`
`STAE` and `NOSPIE`	`TRAP(OFF)`
`NOSTAE` and `SPIE`	`TRAP(OFF)`

Class library execution incompatibilities

There are execution incompatibilities for applications that use the following class libraries:

- Collection Class Library from C++ for MVS/ESA V3R1M0, V3R1M1 or V3R2M0
- Application Support Class Library from C++ for MVS/ESA V3R1M0, V3R1M1 or V3R2M0

You can no longer recompile and relink applications that use these class libraries.

As of z/OS V1R5, development with the IBM Open Class Library (IOC) is not supported. You can no longer compile and link applications that use IOC classes. This includes all the classes, templates, and facilities that are described in *IBM*

Open Class Library Reference with the two exceptions noted below. Run-time support is provided for existing applications that use IOC, but this support will be removed in a future release.

The following classes are still supported for application development:

- UNIX® System Laboratories (USL) I/O Stream Library
- USL Complex Mathematics Library

As of z/OS V1R5, the name of the element that provides this application development support has changed from IBM Open Class Library to Run-Time Library Extensions. The directory path for the header file has changed from `/usr/lpp/ioclib` to `/usr/lpp/cbclib`.

Although support for these classes is not being removed at this time, it is recommended that you migrate to the Standard C++ iostream and complex classes. This is especially important if you are migrating other IOC streaming classes to Standard C++ Library streaming classes, because combining USL and Standard C++ Library streams in one application is not recommended. For more information about these classes, see *C/C++ Legacy Class Libraries Reference*.

For information about migrating away from these classes, see *IBM Open Class Library Transition Guide*.

Chapter 8. Source program compatibility

In general, you can use source programs with the z/OS V1R7 XL C/C++ product without modification, if they were created with one of the following:

- AD/Cycle C/370 compiler running with Language Environment V1R2 or later
- C/C++ for MVS V3R1 or V3R2 compiler programs running with Language Environment V1R4 or later

This chapter highlights the exceptions and shows how to solve specific problems in compatibility.

Chapter 9, “Other migration considerations,” on page 55 has information on run-time options, which may also affect source code compatibility.

Input and output operations

If programs that ran successfully with the C/370 V2R1 library have dependencies on any of the input and output behaviors listed in Chapter 6, “Input and output operations compatibility,” on page 39, these programs might require source changes before they can run under z/OS Language Environment.

SIGFPE exceptions

Decimal overflow conditions were masked in V1R1 and V1R2 of LE/370. These conditions were enabled when the packed decimal data type was introduced in the AD/Cycle C/370 V1R2 compiler, and continue to be enabled with z/OS V1R7 Language Environment.

If you have old load modules that accidentally generated decimal overflow conditions, they may behave differently with z/OS V1R7 Language Environment by raising unexpected SIGFPE exceptions. Without source alteration, such modules cannot be migrated to the new library, and are unsupported. It is unlikely that such modules will occur in a C-only environment. These unexpected exceptions may occur in mixed language modules, particularly those using C and assembler code where the assembler code explicitly manipulates the program mask.

Program mask manipulations

Programs created with LE/370 V1R1 or V1R2 that explicitly manipulated the program mask may require source alteration to execute correctly under z/OS V1R7 Language Environment. Changes are required if you have one of the following types of programs:

- A C program containing assembler interlanguage calls (ILC), in which the invoked code uses the S/370 decimal instructions that might generate an unmasked decimal overflow condition, requires modification for migration. There are two methods for migrating the code. The first one is preferred:
  - *Preferred:* Modify the assembler code to save the existing mask, set the new value, and when finished, restore the saved mask.
  - Change the C code so that the produced `SIGFPE` signal is ignored in the called code. **Example:** In the following example, the `SIGNAL` calls surround the overflow-producing code. The `SIGFPE` exception signal is ignored, and then reenabled:

```
signal(SIGFPE, SIG_IGN);   /* ignore exceptions */
...
callit():                  /* in called routine */
...
signal(SIGFPE, SIG_DFL);   /* restore default handling */
```

- A C program containing assembler ILCs that explicitly alter the program mask, and do not explicitly save and restore it, also requires modification for migration.

  If user code explicitly alters the state of the program mask, the value before modification must be saved, and restored to its former value after the modification. You must ensure that the decimal overflow program mask bit is enabled during the execution of C code. Failure to preserve the mask may result in unpredictable behavior.

These changes also apply in a System Programming C environment, and to Customer Information Control System (CICS) programs in the handling and management of the PSW mask.

Line number control

The AD/Cycle C/370 and C/MVS V3R1 compilers ignored `#line` when either the `EVENTS` or the `TEST` compiler option was in effect.

As of C/MVS V3R2, the compiler does not ignore `#line`.

Note: For information about using `#line`, refer to *z/OS XL C/C++ Language Reference*

Function return type sizes

Example: The following is an example of how the behavior of `sizeof`, when applied to a function return type, was changed in the C/C++ MVS V3R2 compiler:

```
char foo();
..
s = sizeof foo();
```

If the example is compiled with a compiler prior to C/C++ MVS V3R2, `char` is widened to `int` in the return type, so `sizeof` returns `s = 4`.

If the example is compiled with C/C++ MVS V3R2, OS/390 C/C++ compiler, or z/OS XL C/C++ compiler, the size of the original type is retained. In the above example, `sizeof` returns `s = 1`. The size of the original type of other data types such as `short` and `float` is also retained.

If you are using OS/390 V2R4 C/C++ and subsequent compilers, and you require `sizeof` to return the widened size for function return types, then you must use `#pragma wsizeof` together with the `WSIZEOF` compiler option. For more information on `#pragma wsizeof`, see the *z/OS XL C/C++ Language Reference*. For more information on the `WSIZEOF` compiler option, see the *z/OS XL C/C++ User's Guide*.

_Packed structures and unions

If you are migrating from an AD/Cycle C/370 compiler to the z/OS XL C compiler, you can no longer do the following:

- assign `_Packed` and non-`_Packed` structures to each other
- assign `_Packed` and non-`_Packed` unions to each other

- pass a `_Packed` union or `_Packed` structure as a function parameter if a non-`_Packed` version is expected (or the other way around)

If you attempt to do so, the compiler issues an error message.

Alternate code points

The following alternate code points are not supported by the z/OS XL C/C++ compilers:

- X'8B' as alternate code point for X'C0' (the left brace)
- X'9B' as alternate code point for X'D0' (the right brace)

These alternate code points were supported by the C/370 and AD/Cycle C/370 compilers (the `NOLOCALE` option was required if you were using the AD/Cycle C/370 V1R2 compiler).

Support of Standard C++

As of z/OS V1R7, the C++ compiler supports *Programming languages - C++ (ISO/IEC 14882:2003(E))*, which documents the currently supported C++ standard. For more information, see Chapter 14, “Migrating to the currently supported Standard C++,” on page 87.

LANGLVL(ANSI) changes

As of the C/C++ MVS V3R2 compiler, if you specify `LANGLVL(ANSI)`, the compiler recognizes `char`, `unsigned char`, and `signed char` as three distinct types.

As of z/OS V1R2 C++, if you specify `LANGLVL(ANSI)`, the compiler will conform to the currently supported Standard C++. See Chapter 14, “Migrating to the currently supported Standard C++,” on page 87 for details.

Compiler messages and return codes

There are differences in messages and return codes between different versions of the compiler. Message contents have changed, and return codes for some messages have changed (some errors have become warning, and in very rare situations, some warnings have become errors). You must update accordingly any application that is affected by message contents or return codes. **Do not build dependencies on message content, message numbers, or return codes**. Refer to *z/OS XL C/C++ Messages* for a list of compiler messages.

Class library source code incompatibilities

There are source code incompatibilities for applications that use the following class libraries:

- Collection Class Library from C++ for MVS/ESA V3R1M0, V3R1M1 or V3R2M0
- Application Support Class Library from C++ for MVS/ESA V3R1M0, V3R1M1 or V3R2M0

You can no longer recompile and relink applications that use these class libraries.

As of z/OS V1R5, development with the IBM Open Class Library (IOC) is not supported. You can no longer compile and link applications that use IOC classes. This includes all the clases, templates, and facilities that are described in *IBM Open*

Class Library Reference with the two exceptions noted below. Run-time support is provided for existing applications that use IOC, but this support will be removed in a future release.

The following classes are still supported for application development:

- UNIX System Laboratories (USL) I/O Stream Library
- USL Complex Mathematics Library

As of z/OS V1R5, the name of the element that provides this application development support has changed from IBM Open Class Library to Run-Time Library Extensions. The directory path for the header file has changed from `/usr/lpp/ioclib` to `/usr/lpp/cbclib`.

Although support for these classes is not being removed at this time, it is recommended that you migrate to the Standard C++ iostream and complex classes. This is especially important if you are migrating other IOC streaming classes to Standard C++ Library streaming classes, because combining USL and Standard C++ Library streams in one application is not recommended. For more information about these classes, see *C/C++ Legacy Class Libraries Reference*.

For information about migrating away from these classes, see *IBM Open Class Library Transition Guide*.

DSECT utility and packed structures

Header files generated by the DSECT utility now use `#pragma pack` rather than `_Packed` for packed structures. In rare cases, you may have to modify and recompile your code.

Chapter 9. Other migration considerations

This chapter provides additional considerations on migrating applications from the compilers and libraries listed in Part 3, “From pre-OS/390 releases of C/C++ to z/OS V1R7 XL C/C++,” on page 45 to the z/OS V1R7 XL C/C++ feature.

Removal of Database Access Class Library utility

As of OS/390 V2R4 C/C++, the Database Access Class Library utility is no longer available.

Changes that affect user JCL, CLISTs, and EXECs

This section describes changes that may affect your JCL, CLISTs, and EXECs.

CXX parameter in JCL procedures

With C++ for MVS/ESA V3R2, OS/390®, and z/OS C++ compilers, the CBCC, CBCCL, and CBCCLG procedures, which compile C++ code, now include parameter CXX. You must include this parameter if you have written your own JCL to compile a C++ program. Otherwise, you invoke the C compiler.

When you pass options to the compiler, you must specify parameter CXX. You must use the following format to specify options:

```
run-time options/CXX compiler options
```

Examples of specifying class library header files at compile time

In z/OS V1R1 and earlier compilers, using the following JCL on the compile step would work, although it was not recommended:

```
//SYSLIB DD DSN=CEE.SCEEH.H,DISP=SHR
//       DD DSN=CEE.SCEEH.SYS.H,DISP=SHR
//       DD DSN=CBC.SCLBH.H,DISP=SHR
```

As of z/OS V1R2, the logical record length for the SCLBH data sets has been increased from 80 bytes to 120 bytes. Due to this change, the SYSLIB job card shown above no longer works, and must be removed from your JCL. In its place, you must use the `SEARCH` compiler option.

Example:

```
SEARCH(//'CEE.SCEEH.+',//'CBC.SCLBH.+')
```

Using the `SEARCH` compiler option instead of a SYSLIB concatenation allows the C++ compiler to search for files based on both the file name and file type.

SYSMSGS and SYSXMSGS ddnames

With the C/C++ for for MVS/ESA V3R2, OS/390, and z/OS XL C/C++ compilers, the method of specifying the language for compiler messages has changed. At compile time, instead of specifying message data sets on the `SYSMSGS` and `SYSXMSGS` ddnames, you must now use the `NATLANG` run-time option. If you specify data sets for these ddnames, they are ignored.

Note: For information about the `NATLANG` run-time option, see the *z/OS Language Environment Customization* and the *z/OS Language Environment Programming Reference*.

Changes in data set names

The names of IBM-supplied data sets may change from one release to another. See the z/OS Program Directory for more information on data set names.

CBCI and CBCXI procedures

As of z/OS V1R5, the CBCI and CBCXI procedures contain the variable CLBPRFX. If you have any JCL that uses these procedures, you must either customize these procedures (for example, at installation time) or modify your JCL to provide a value for CLBPRFX.

Decimal exceptions

z/OS Language Environment provides support for the packed decimal overflow exception using native S/390 hardware enablement, as did LE/370 V1R3, Language Environment V1R4, and Language Environment V1R5.

The value of the program mask in the program status word (PSW) is 4 (decimal overflow enabled).

Migration and coexistence

The following points identify migration and coexistence considerations for user applications:

- As of LE/370 V1R3, CICS programs were enabled for decimal exceptions.
- The C packed decimal support routines are not supported in an environment that exploits asynchronous events.

SIGTERM, SIGINT, SIGUSR1, and SIGUSR2 exceptions

As of LE/370 V1R3, there were changes to application/program behavior for `SIGTERM`, `SIGINT`, `SIGUSR1`, and `SIGUSR2` exceptions from previous releases of the LE/370 product. These changes in behavior carried over into the z/OS V1R7 Language Environment product.

The differences or incompatibilities are:

- The defaults for the `SIGINT`, `SIGTERM`, `SIGUSR1`, and `SIGUSR2` signals changed in LE/370 V1R3, from what they were in C/370 V1R1 and V1R2 and LE/370 V1R1 and V1R2. In the C/370 library and LE/370 V1R1 and V1R2, the defaults for `SIGINT`, `SIGUSR1`, and `SIGUSR2` were to ignore the signals. As of LE/370 V1R3, the default is to terminate the program and return a return code of 3000. For `SIGTERM`, the default has always been to terminate the program, but the return code is now 3000 whereas before it was 0.
- Applications that terminate abnormally will **not** drive the `atexit` list.

Compiler options

This section describes changes that may affect your compiler options.

DECK compiler option

In IBM C/370 V1, the `DECK` compiler option directed the object module to the data set associated with `SYSLIN`. With the OS/390 C and the z/OS V1R1 C compiler, as with the AD/Cycle C/370 and IBM C/370 V2 compilers, the object module is directed to the data set associated with `SYSPUNCH`.

As of z/OS V1R2, the DECK compiler option is no longer supported. The replacement for DECK functionality that routes output to DD:SYSPUNCH, is to use OBJECT(DD:SYSPUNCH).

ENUM compiler option

z/OS V1R2 introduced the ENUM option as a means for controlling the size of enumeration types. The default setting, ENUM(SMALL), provides the same behavior in previous releases of the compiler. If you want to use the ENUM option, it is recommended that the same setting be used for the whole application; otherwise, you may find inconsistencies when the same enumeration type is declared in different compilation units. Use the #pragma enum, if necessary, to control the size of individual enumeration types (especially in common header files).

HALT compiler option

As of C++ for MVS/ESA V3R2, the C++ compilers do not accept 33 as a valid parameter for the HALT compiler option.

HWOPTS compiler option

In IBM AD/Cycle C/370 V1, the HWOPTS compiler option directed the compiler to generate code to take advantage of different hardware. As of z/OS V1R2, the HWOPTS compiler option is no longer supported. The replacement for it is the ARCHITECTURE option.

INFO compiler option

As of z/OS V1R2, the INFO option default has been changed from NOINFO to INFO(LAN) for C++.

As of z/OS V1R6, the INFO option is supported by the C compiler.

Note: The CHECKOUT C compiler option will continue to be supported for backward compatibility only.

INLINE compiler option

For C, the default for the INLINE compiler option was changed to 100 ACUs (Abstract Code Units) in the MVS/ESA V3R1 compiler. Hence, with the C for MVS/ESA V3 and the z/OS XL C compilers, the default is 100 ACUs. In the past, the default was 250 ACUs.

For C++, the z/OS V1R1 and earlier compilers did not accept the INLINE option but did perform inlining at OPT with a fixed value of 100 for the threshold and 2000 for the limit. As of z/OS V1R2, the C++ compiler accepts the INLINE option, with defaults of 100 and 1000 for the threshold and limit, respectively. As a result of this change, code that used to be inlined may no longer be inlined due to the decrease in the limit from 2000 to 1000 ACUs.

LANGLVL(COMPAT) compiler option

In IBM C++ for MVS/ESA V3, the LANGLVL(COMPAT) option directed the compiler to generate code that is compatible with older levels of C++. As of z/OS V1R2, the LANGLVL(COMPAT) compiler option is no longer supported.

OMVS compiler option

In IBM AD/Cycle C/370 V1, the OMVS compiler option directed the compiler to use the POSIX.2 standard rules for searching for files specified with #include directives. As of z/OS V1R2, the OMVS compiler option is no longer supported. The replacement for it is the OE option.

OPTIMIZE compiler option

In the AD/Cycle C/370 compilers:

- OPT(0) was mapped to NOOPT
- OPT and OPT(1) were mapped to OPT(1)
- OPT(2) was mapped to OPT(2)
- OPT(3) was added to complete the list of opt levels

In the C/C++ for MVS/ESA V3 compilers, and the OS/390 V1R1 compiler:

- OPT(0) was mapped to NOOPT
- OPT, OPT(1) and OPT(2) were mapped to OPT

In the OS/390 V2R6 C/C++ compiler:

- OPT(0) was maped to NOOPT
- OPT, OPT(1) and OPT(2) were mapped to OPT(2).

While the OPT level mapping for the C/C++ for MVS/ESA V3 and OS/390 V2R6 compilers is the same, the optimization is different. The underlying compiler technology within these compilers has changed significantly.

SEARCH and LSEARCH compiler options

The include file search process has changed. Prior to the C for MVS/ESA V3R2 compiler, if you used the LSEARCH option more than once, the compiler searched only the libraries specified for the last LSEARCH option. The z/OS XL C/C++ compiler searches all of the libraries specified for all of the LSEARCH options, from the point of the last NOLSEARCH option.

Similarly, if you specify the z/OS XL C/C++ SEARCH option more than once, the z/OS C/C++ compiler searches all of the libraries specified for all of the SEARCH options, from the point of the last NOSEARCH option. Previously, only the libraries specified for the last SEARCH option were searched.

SRCMSG compiler option

In IBM C++ for MVS/ESA V3, the SRCMSG option directed the compiler to add the corresponding source code lines to the diagnostic messages that are written to stderr. As of z/OS V1R2, the SRCMSG compiler option is no longer supported.

SYSLIB, USERLIB, SYSPATH and USERPATH compiler options

In IBM C/C++ for MVS/ESA V3, the SYSLIB, USERLIB, SYSPATH and USERPATH compiler options directed the compiler to specified include files. As of z/OS V1R2, these compiler options are no longer supported. Instead, you use the SEARCH and LSEARCH options to find include files.

TEST compiler option

As of the OS/390 C/C++ compilers, the default for the PATH suboption of the TEST option has changed from NOPATH to PATH. Also, the INLINE option is ignored when the TEST option is in effect at OPT(0), but the INLINE option is no longer ignored if OPT(1), OPT(2), or OPT(3) is in effect.

As of C/C++ MVS V3R2, a restriction applies to the TEST compiler option. Now, the maximum number of lines in a single source file cannot exceed 131,072. If you exceed this limit, the results from the Debug Tool and z/OS Language Environment Dump Services are undefined.

As of z/OS V1R6, when using the c89/c++ utility, the -g flag has changed from specifying the TEST option to DEBUG(FORMAT(DWARF)).

Note: For 64–bit environments only, you can use the new environment variable {_DEBUG_FORMAT} to determine the debug format (DWARF or ISD) to which the –g flag option is translated. For information about this new environment variable and the c89/c++ utility, see *z/OS XL C/C++ User's Guide*.

Changes to putenv()

As of z/OS V1R5, the C/C++ function putenv() changed to place the string passed to putenv() directly into the array of environment variables. This behavior assures compliance with the POSIX standard. Before the change, the storage used to define the environment variable passed into putenv() was not added to the array of environment variables. Instead, the system copied the string into system allocated storage. To restore the previous behavior of putenv(), set environment variable _EDC_PUTENV_COPY to YES.

For additional information on putenv() and _EDC_PUTENV_COPY, see *z/OS XL C/C++ Run-Time Library Reference*. You may also refer to *z/OS XL C/C++ Programming Guide*, for information on putenv() and _EDC_PUTENV_COPY.

Length of external variable names

As of z/OS V1R3, external names (such as entry points and external references) can be up to 32,767 bytes long.

The z/OS V1R2 binder imposes a limit of 1024 characters for the length of external names. Both the OS/390 C++ compiler and z/OS C++ compiler may sometimes generate mangled names that are longer than this limit. This could occur more often when using the Standard Template Library with the z/OS V1R2 C++ compiler. Should you encounter this problem:

- Reduce the length of the C++ class names.
- Use #pragma map to map the long name to a short one.
- For NOXPLINK applications, use the prelinker.

Syntax for the CC command

With the C/C++ for MVS/ESA V3R2, OS/390, and z/OS XL C/C++ compilers, you can use a new syntax to invoke the CC command.

At customization time, your system programmer can customize the CC EXEC to accept only the old syntax (the one supported by compilers before C/C++ for MVS/ESA V3R2), only the new syntax, or both syntaxes.

You should customize the CC EXEC to accept only the new syntax, because the old syntax may not be supported in the future.

If you customize the CC EXEC to accept both the old and new syntaxes, you must invoke it using either the old or the new syntax, not a mixture of both. Be aware that the old syntax does not support Hierarchical File System (HFS) files.

Refer to the z/OS Program Directory for more information about installation and customization, and to the *z/OS XL C/C++ User's Guide* for more information about compiler options.

Time functions

You should customize your locale information. Otherwise, in rare cases, you may encounter errors. In a POSIX application, you can supply time zone and alternative time (for example, daylight) information with the `TZ` environment variable. In a non-POSIX application, you can supply this information with the `_TZ` environment variable. If no `TZ` environment variable is defined for a POSIX application or no `_TZ` environment variable is defined for a non-POSIX application, any customized information provided by the `LC_TOD` locale category is used. By setting the `TZ` environment variable for a POSIX application, or the `_TZ` environment variable for a non-POSIX application, or by providing customized time zone or daylight information in an `LC_TOD` locale category, you allow the time functions to preserve both time and date, correctly adjusting for alternative time on a given date.

Refer to the *z/OS XL C/C++ Programming Guide* for more information about both environment variables and customizing a locale.

Abnormal termination exits

The abnormal termination exits CEEBDATX (for batch) and CEECDATX (for CICS) are now automatically linked at install time for z/OS Language Environment; the sample exit is no longer required. These exits were only available in the sample library in LE/370 V1R3. They allow you to automatically produce a system dump (with abend code 4039), when abnormal termination occurs. In previous releases of Language Environment, only a Language Environment-formatted dump was generated (which continues to be produced under z/OS V1R7 Language Environment).

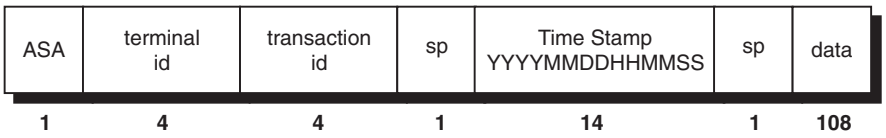
For a non-CICS application, you can trigger the dump by ensuring that SYSUDUMP is defined in the GO step of the JCL that you are using (for example, by including the statement SYSUDUMP DD SYSOUT=*). If SYSUDUMP is not included in your JCL, or is defined as DUMMY, the dump will be suppressed. As of C/C++ for MVS/ESA V3R1, the standard JCL procedures shipped with the compiler do not include SYSUDUMP.

In a CICS environment, you automatically receive the default transaction dump unless you disable it by using the CEMT transaction, and by specifying the dumpcode '4039'.

You can also modify CEEBDATX and CEECDATX to suppress the dumps. The exits are available in the z/OS V1R7 Language Environment.

Standard stream support

Under CICS, with z/OS Language Environment, records sent to the transient data queues associated with `stdout` and `stderr` with default settings take the form of a message as follows:



where:

ASA is the carriage-control character

terminal id is a 4 character terminal identifier

transaction id is a 4 character transaction identifier

sp is a space

Time Stamp is the date and time displayed in the format YYYYMMDDHHMMSS

data is the data sent to the standard streams `stdout` and `stderr`

This format was associated with `stderr` for all releases of Language Environment. However, it has only been used for `stdout` since LE/370 Release 3; therefore, you should be aware of this change if you are migrating to z/OS V1R7 Language Environment.

Direction of compiler messages to stderr

All messages produced by the C/C++ for MVS/ESA V3R2 and z/OS XL C++ compilers are sent to `stderr`. In the past, some messages were sent to `stdout`.

Array new

In the C++ for MVS/ESA V3R1 compiler, the array version of `new` was not initially supported. It is supported in a PTF (APAR PN72107) available for the C++ for MVS/ESA V3R1 compiler, and it is also supported in the C++ for MVS/ESA V3R2 and later C++ compilers.

Example: If you are migrating from the base C/C++ for MVS/ESA V3R1 compiler to z/OS V1R7 XL C/C++, and you have written your own global `new` operator, it is no longer called when you create an array object:

```
void*   operator new (MyClass *, size_t sz) {
  g_new_count++;
  return MyMalloc(sz);
}

main() {
    X new_array[10]; // the global new operator
                     // shown above will not be called if the fix for
                     // APAR PN72107 or the V3R2
                     // compiler is installed
}
```

You have to add an overloaded operator to `new[]` if you require this for arrays.

Compiler listings

As of OS/390 C/C++ V2 R6, `OPT(1)` maps to `OPT(2)`. The compiler listing no longer contains the part of the pseudo-assembler listing that was associated with `OPT(1)`. Listing formats, especially the pseudo-assembler parts, will continue to change from release to release. **Do not build dependencies on the structure or content of listings.** For information about C listings or the C++ listings for the current release, refer to *z/OS XL C/C++ User's Guide*.

Chapter 10. Input and output operations compatibility

Changes were made to input and output support in the C/370 V2R2 and LE/370 V1R3 libraries. These changes also apply to z/OS V1R7 Language Environment. You should read the changes listed in this section if your programs performed input and output operations with the following products:

- LE/370 V1R1
- LE/370 V1R2

References in this chapter to previous releases or previous behavior apply to the products listed above.

You will generally be able to migrate "well-behaved" programs: programs that do not rely on undocumented behavior, restrictions, or invalid behaviors of previous releases. For example, if library documentation only specified that a return code was a negative value, and your code relies on that value being `-3`, your code is not well-behaved and is relying on undocumented behavior.

Another example of a program that is not well-behaved is one that specifies `recfm=F` for a terminal file and depends on Language Environment to ignore this parameter, as it did previously.

However, you may still need to change even well-behaved code under circumstances described in the following section.

Opening files

- When you call the `fopen()` or `freopen()` library function, you can specify each parameter only once. If you specify any keyword parameter in the *mode* string more than once, the function call fails. Previously, you could specify more than one instance of a parameter.
- The library no longer supports uppercase open modes on calls to `fopen()` or `freopen()`. You must specify, for example, `rb` instead of `RB`, to conform to the ANSI/ISO standard.
- You cannot open a non-HFS file more than once for a write operation. Previous releases allowed you, in some cases, to open a file for write more than once. For example, you could open a file by its data set name and then again by its ddname. This is no longer possible for non-HFS files, and is not supported.
- Previously, `fopen()` allowed spaces and commas as delimiters for mode string parameters. Only commas are allowed now.
- If you are using a PDS or a PDSE, you cannot specify any spaces before the member name.

Writing to files

- Write operations to files opened in binary mode are no longer deferred. Previously, the library did not write a block that held *nn* bytes out to the system until the user wrote *nn*+1 bytes to the block. The z/OS Language Environment library follows the rules for full buffering, described in *z/OS XL C/C++ Programming Guide*, and writes data as soon as the block is full. The *nn* bytes are still written to the file, the only difference is in the timing of when it is done.
- For non-terminal files, the backspace character (`'\b'`) is now placed into files as is. Previously, it backed up the file position to the beginning of the line.

- For all text I/O, truncation for `fwrite()` is now handled the same way that it is handled for `puts()` and `fputs()`. If you write more data than a record can hold, and your output data contains any of the terminating control characters, `'\n'` or `'\r'` (or `'\f'`, if you are using ASA), the library still truncates extra data; however, recognizing that the text line is complete, the library writes subsequent data to the next record boundary. Previously, `fwrite()` stopped immediately after the library began truncating data, so that you had to add a control character before writing any more data.
- You can now partially update a record in a file opened with `type=record`. Previous libraries returned an error if you tried to make a partial update to a record. Now, a record is updated up to the number of characters you specify, and the remaining characters are untouched. The next update is to the next record.
- z/OS Language Environment blocks files more efficiently than some previous libraries did. Applications that depend on the creation of short blocks may fail.
- The behavior of ASA files when you close them has changed. In previous releases, this is what happened:

Written to file	Read from file after fclose(), fopen()
abc\n\n\n	abc\n\n\n\n
abc\n\n	abc\n\n\n
abc\n	abc\n

In this release, you read from the file what you wrote to it. For example:

Written to file	Read from file after fclose(), fopen()
abc\n\n\n	abc\n\n\n
abc\n\n	abc\n\n
abc\n	abc\n

In previous products, writing a single new-line character to a new file created an empty file under MVS. z/OS Language Environment treats a single new-line characters written to a new file as a special case, because it is the last new-line character of the file. The library writes a single blank to the file. When you read this file, you see two new-line characters instead of one. You also see two new-line characters on a read if you have written two new-line characters to the file.

The behavior of appending to ASA files has also changed. The following table shows what you get from an ASA file when you:

1. Open an ASA file for write.
2. Write `abc`.
3. Close the file.
4. Append `xyz` to the ASA file.
5. Open the same ASA file for read.

Table 12. Appending to ASA files

abc Written to file, `fclose()` then append `xyz`	**What you read from file after `fclose()`, `fopen()`**	
	Previous release	**New release**
abc ==> xyz	\nabc\nxyz\n	same as previous release
abc ==> \nxyz	\nabc\nxyz\n	\nabc\n\nxyz\n
abc ==> \rxyz	\nabc\rxyz\n	\nabc\n\rxyz\n

Table 12. Appending to ASA files (continued)

abc Written to file, fclose() then append xyz	What you read from file after fclose(), fopen()	
	Previous release	New release
abc\n ==> xyz	\nabc\nxyz\n	same as previous release
abc\n ==> \nxyz	\nabc\nxyz\n	\nabc\n\nxyz\n
abc\n ==> \rxyz	\nabc\rxyz\n	\nabc\n\rxyz\n
abc\n\n ==> xyz	\nabc\n\n\nxyz\n	\nabc\n\nxyz\n
abc\n\n ==> \nxyz	\nabc\n\n\nxyz\n	same as previous release
abc\n\n ==> \rxyz	\nabc\n\n\rxyz\n	same as previous release

- The behavior of DBCS strings has changed.
  1. I/O now checks the value of MB_CUR_MAX to determine whether to interpret DBCS characters within a file.
  2. When MB_CUR_MAX is 4, you can no longer place control characters in the middle of output DBCS strings for interpretation. Control characters within DBCS strings are treated as DBCS data. This is true for terminals as well. Previous products split the DBCS string at the '\n' (new-line) control character position by adding an SI (Shift In) control character at the new-line position, displaying the line on the terminal, and then adding an SO (Shift Out) control character before the data following the new-line character. If MB_CUR_MAX is 1, the library interprets control characters within any string, but does not interpret DBCS strings. SO and SI characters are treated as ordinary characters.
  3. When you are writing DBCS data to text files, if there are multiple SO (Shift Out) control-character write operations with no intervening SI (Shift In) control character, the library discards the SO characters, and marks that a truncation error has occurred. Previous products allowed multiple SO control-character write operations with no intervening SI control character without issuing an error condition.
  4. When you are writing DBCS data to text files and specify an odd number of DBCS bytes before an SI control character, the last DBCS character is padded with a X'FE' byte. If a SIGIOERR handler exists, it is triggered. Previous products allowed incorrectly placed SI control-character write operations to complete without any indication of an error.
  5. Now, when an SO has been issued to indicate the beginning of a DBCS string within a text file, the DBCS must terminate within the record. The record will have both an SO and an SI.

Repositioning within files

- The behavior of fgetpos(), fseek() and fflush() following a call to ungetc() has changed. Previously, these functions have all ignored characters pushed back by ungetc() and have considered the file to be at the position where the first ungetc() character was pushed back. Also, ftell() acknowledged characters pushed back by ungetc() by backing up one position if there was a character pushed back. Now,
  - fgetpos() behaves just as ftell()does
  - When a seek from the current position (SEEK_CUR) is performed, fseek() accounts for any ungetc() character before moving, using the user-supplied offset

  - `fflush()` moves the position back one character for every character that was pushed back.

  If you have applications that depend on the previous behavior of `fgetpos()`, `fseek()`, or `fflush()`, you may use the new `_EDC_COMPAT` environment variable so thatsource code need not change to compensate for the new behavior. `_EDC_COMPAT` is described in *z/OS XL C/C++ Programming Guide*.

- For OS I/O to and from files opened in text mode, the `ftell()` encoding system now supports higher blocking factors for smaller block sizes. In general, you should not rely on `ftell()` values generated by code you developed using previous releases of the library. You can try `ftell()` values taken in previous releases for files opened in text or binary format if you set the environment variable `_EDC_COMPAT` before you call `fopen()` or `freopen()`. Do not rely on `ftell()` values saved across program boundaries. `_EDC_COMPAT` is described in *z/OS XL C/C++ Programming Guide*.
- For record I/O, `ftell()` now returns the relative record number instead of an encoded offset from the beginning of the file. You can supply the relative record number without acquiring it from `ftell()`. You cannot use old `ftell()` values for record I/O, regardless of the setting of `_EDC_COMPAT`. `_EDC_COMPAT` is described in *z/OS XL C/C++ Programming Guide* .
- If you have used `ungetc()` to move the file pointer to a position before the beginning of the file, calls to `ftell()` and `fgetpos()` now fail. Previously, `ftell()` returned the value `0` for such calls, but set `errno` to a non-zero value. Previously, `fgetpos()` did not account for `ungetc()` calls. See *z/OS XL C/C++ Programming Guide* for information on how to change `fgetpos()` behavior by using _EDC_COMPAT.

  For example, suppose that you are at relative position 1 in the file and `ungetc()` is performed twice. `ftell()` and `fgetpos()` will now report the relative position -1, which is before the start of the file, causing both `ftell()` and `fgetpos()` to fail.
- After you have called `ftell()`, calls to `setbuf()` or `setvbuf()` may fail. Applications should never call I/O functions between calls to `fopen()` or `freopen()` and calls to the functions that control buffering.

Closing and reopening ASA files

The behavior of ASA files when you close and reopen them is now consistent:

Table 13. Closing and reopening ASA files

	Physical record after close					
Written to file	**Previous behavior**			**New behavior**		
`abc`	Char	`abc`	(1)	same as previous release		
	Hex	`4888` `0123`	(1)			
`abc\n`	Char	`abc`	(1)	same as previous release		
	Hex	`4888` `0123`	(1)			
`abc\n\n`	Char	`abc` `0`	(1) (2)	Char	`abc`	(1) (2)

	Hex	`4888`
		`0123`
		`F`
		`0`
			(1)
			
			(2)
				Hex	`4888`
					`0123`
					`4`
					`0`
						(1)
						
						(2)

Table 13. Closing and reopening ASA files (continued)

	Physical record after close					
Written to file	**Previous behavior**			**New behavior**		
`abc\n\n\n`	Char	`abc` -	(1) (2)	Char	`abc`	(1) (2)

	Hex	`4888`
		`0123`
		`6`
		`0`
			(1)
			(2)
				Hex	`4888`
					`0123`
					`4`
					`0`
						(1)
						(2)

`abc\r`	Char	`abc` +	(1) (2)	same as previous release		

	Hex	`4888`
		`0123`
		`4`
		`E`
			(1)
			(2)			

`abc\f`	Char	`abc` `1`	(1) (2)	same as previous release		

	Hex	`4888`
		`0123`
		`F`
		`1`
			(1)
			(2)			

fldata() return values

There are minor changes to the values that the `fldata()` library function returns. It may now return more specific information in some fields. For more information on `fldata()`, see the “Input and Output” section in *z/OS XL C/C++ Programming Guide*.

Error handling

The general return code for errors is now `EOF`. In previous products, some I/O functions returned 1 as an error code to indicate failure. This caused some confusion, as 1 is a possible `errno` value as well as a return code. `EOF` is not a valid `errno` value.

Programs that rely on specific values of `errno` may not run as expected, because certain `errno` values have changed. As of OS/390 Language Environment V1R5, error messages have the format EDC5xxx. You can find the error message information for a particular `errno` value by applying the `errno` value to EDC5xxx (for example, 021 becomes EDC5021), and looking up the EDC5xxx message in *z/OS Language Environment Debugging Guide*.

Miscellaneous

- The inheritance model for standard streams now supports repositioning. Previously, if you opened `stdout` or `stderr` in update mode, and then called another C program by using the ANSI-style `system()` function, the program that you called inherited the standard streams, but moved the file position for `stdout` or `stderr` to the end of the file. Now, the library does not move the file position to the end of the file. For text files, the position is moved only to the nearest record boundary not before the current position. This is consistent with the way `stdin` behaves for text files.

- The values for L_tmpnam and FILENAME_MAX have been changed:

Constant	Old values	New values
L_tmpnam	47	1024
FILENAME_MAX	57	1024

- The names produced by the tmpnam() library function are now different. Any code that depends on the internal structure of these names may fail.

VSAM I/O changes

- The library no longer appends an index key when you read from an RRDS file opened in text or binary mode.
- RRDS files opened in text or binary mode no longer support setting the access direction to BWD.

Terminal I/O changes

- The library will now use the actual recfm and lrecl specified in the fopen() or freopen() call that opens a terminal file. Incomplete new records in fixed binary and record files are padded with blank characters until they are full, and the __recfmF flag is set in the fldata() structure.

  Previously, MVS terminals unconditionally set recfm=U. Terminal I/O did not support opening files in fixed format.
- The use of an LRECL value in the fopen() or freopen() call that opens a file sets the record length to the value specified.

  Previous releases unconditionally set the record length to the default values.
- The use of a RECFM value in the fopen() or freopen() call that opens a file sets the record format to the value specified.

  Previous releases unconditionally set the record format to the default values.
- For input text terminals, an input record now has an implicit logical record boundary at LRECL if the size of the record exceeds LRECL. The character data in excess of LRECL is discarded, and a '\n' (new-line) character is added at the end of the record boundary. You can now explicitly set the record length of a file as a parameter on the fopen() call.

  The old behavior was to allow input text records to span multiple LRECL blocks.
- Binary and record input terminals now flag an end-of-file condition with an empty input record. You can clear the EOF condition by using the rewind() or clearerr() library function.

  Previous products did not allow these terminal types to signal an end-of-file condition.
- When an input terminal requires input from the system, all output terminals with unwritten data are flushed in a way that groups the data from the different open terminals together, each separated from the other with a single blank character.

  The old behavior is equivalent to the new behavior, except that two blank characters separate the data from each output terminal.

Part 4. From OS/390 C/C++ to z/OS V1R7 XL C/C++

This part discusses the implications of migrating applications that were created with one of the following compilers and one of the following libraries to the z/OS V1R7 XL C/C++ product.

Notes:

1. As of z/OS V1R7 XL C/C++, OS/390 V2R10 compiler is no longer shipped with the z/OS product. The OS/390 V2R10 compiler is equivalent to the z/OS V1R1 compiler.
2. The OS/390 V1R1 compiler and library were equivalent to the final MVS/ESA compiler and library, and are described in Part 3 of this book.

Compilers:

- IBM OS/390 C/C++ V1R2 compiler, 5645-001
- IBM OS/390 C/C++ V1R3 compiler, 5645-001
- IBM OS/390 C/C++ V2R4 (or V2R5) compiler, 5647-A01
- IBM OS/390 C/C++ V2R6 (or V2R7 or V2R8) compiler, 5647-A01
- IBM OS/390 C/C++ V2R9 compiler, 5647-A01
- IBM OS/390 C/C++ V2R10 compiler, 5647-A01
- IBM z/OS C/C++ V1R1 compiler, 5694-A01

Libraries:

- IBM OS/390 V1R2 Language Environment, 5645-001
- IBM OS/390 V1R3 Language Environment, 5645-001
- IBM OS/390 V1R4 Language Environment, 5647-A01
- IBM OS/390 V1R5 Language Environment, 5647-A01
- IBM OS/390 V1R6 Language Environment, 5647-A01
- IBM OS/390 V1R7 Language Environment, 5647-A01
- IBM OS/390 V1R8 Language Environment, 5647-A01
- IBM OS/390 V1R9 Language Environment, 5647-A01
- IBM OS/390 V1R10 Language Environment, 5647-A01
- IBM z/OS V1R1 Language Environment, 5694-A01

Chapter 11. Compiler changes between OS/390 C/C++ and z/OS V1R7 XL C/C++

This chapter describes the compiler changes that you may encounter if you are migrating from a previous release of OS/390 C/C++ or z/OS V1R1 C/C++ to z/OS V1R7 XL C/C++. Also refer to Chapter 14, "Migrating to the currently supported Standard C++," on page 87 for details on support of *Programming languages - C++ (ISO/IEC 14882:2003(E))*, which documents the currently supported Standard C++.

Compiler changes

Potential impact on memory requirements

Memory requirements for compilation may increase for successive releases as new logic is added. If you cannot recompile an application that you successfully compiled with a previous release of the compiler, try increasing the region size.

Removal of Model Tool support

As of OS/390 V2R10, the Model Tool is no longer available.

1998 Standard C++ support

As of z/OS V1R2, the C++ compiler supports *Programming languages - C++ (ISO/IEC 14882:1998(E))*. See Chapter 14, "Migrating to the currently supported Standard C++," on page 87 for details.

Addition of the #pragma reachable and #pragma leaves directives

These pragmas help the optimizer in moving code around the function call site when exploring opportunities for optimization. Since the addition of these pragmas in OS/390 V2R9, the optimizer is more aggressive.

Functions that exhibit the leave and reachable properties must be identified by these pragmas. The C run-time library functions `setjmp` and `longjmp` (and related functions such as `sigsetjmp`, `siglongjmp`) are such functions. If your version of `setjmp.h` does not include these pragmas, you should add them to your program code as follows:

```
#pragma leaves (longjmp, _longjmp, siglongjmp)
#pragma reachable (setjmp, _setjmp, sigsetjmp)
```

Alternatively, if the functions refer to the C run-time library provided by the system (or another library that strictly conforms to the C standard), you can turn on the `LIBANSI` option.

For more information on using `#pragma reachable` and `#pragma leaves` directives, refer to *z/OS XL C/C++ Language Reference*.

Reentrant variables when the compiler option is NORENT

In previous releases of the compiler, `#pragma variable (`*`name`*`, RENT)` had no effect if the compiler option was `NORENT`. As of OS/390 V2R9, a variable can be reentrant even if the compiler option is `NORENT`.

This change may cause some programs that compiled and linked successfully in previous releases to fail during link-edit in the current release. This applies if *all* of the following are true:

- The program is written in C and compiled with the NORENT option
- At least one variable is reentrant
- The program is compiled and linked with the output directed to a PDS and the prelinker was NOT used. JCL procedures that may have been used to do this in previous releases are: EDCCL, EDCCLG, EDCL, and EDCLG (not all of these procedures are available with the z/OS V1R7 compiler).

In previous releases, #pragma variable (*name*, NORENT) was ignored for static variables. As of OS/390 V2R10, this pragma is accepted if the ROCONST option is turned on, and the variable is const qualified and not initialized with an address.

Compiler options

Changes in default settings

ARCHITECTURE compiler option: As of z/OS V1R6, the default value of the ARCHITECTURE compiler option is 5.

In OS/390 V2R10 to z/OS V1R5 releases, the default value of the ARCHITECTURE compiler option is 2. In OS/390 V2R9 and previous releases, the default value of the ARCHITECTURE compiler option is 0.

CHECKOUT(CAST) compiler suboption: This suboption instructs the C compiler to check the source code for pointer casting that might affect optimization, that is, those castings that violate the ansi-aliasing rule. (Refer to *z/OS XL C/C++ User's Guide* for more details about ANSIALIAS option.) Prior to z/OS V1R2, the compiler issued a WARNING message whenever this condition was detected. As of z/OS V1R2, this message is INFORMATIONAL. If you wish to be alerted by the compiler that this message has been issued, you can use the HALTONMSG compiler option. The HALTONMSG option causes the compiler to stop after source code analysis, skip the code generation, and return with a return code of 12.

DIGRAPH compiler option: As of z/OS V1R2, the DIGRAPH option default for C and C++ has been changed from NODIGRAPH to DIGRAPH.

INFO compiler option: As of z/OS V1R2, the INFO option default has been changed from NOINFO to INFO(LAN) for C++.

INLINE compiler option: For C++, the z/OS V1R1 and earlier compilers did not allow you to change the inlining threshold. These compilers performed inlining at OPT with a fixed value of 100 for the threshold and 2000 for the limit.

As of z/OS V1R2, the C++ compiler accepts the INLINE option, with defaults of 100 and 1000 for the threshold and limit, respectively. As a result of this change, code that used to be inlined may no longer be inlined due to the decrease in the limit from 2000 to 1000 ACUs (Abstract Code Units).

OPTIMIZE compiler option: In the OS/390 V1R2, V1R3, V2R4, and V2R5 C/C++ compilers:

- OPT(0) mapped to NOOPT
- OPT and OPT(1) mapped to OPT(1)
- OPT(2) mapped to OPT(2)

As of OS/390 V2R6:

- OPT(0) maps to NOOPT
- OPT, OPT(1) and OPT(2) map to OPT(2)

ROSTRING compiler option: As of z/OS V1R2, the ROSTRING option default for C is changed from NOROSTRING to ROSTRING. The default for C++ has always been ROSTRING.

ROSTRING informs the compiler that string literals are read-only, thus allowing more freedom for the compiler to handle string literals. If you are not sure whether your program modifies string literals or not, specify the NOROSTRING compiler option.

ROCONST compiler option: As of z/OS V1R2, the ROCONST option default for C++ is changed from NOROCONST to ROCONST. The default for C remains NOROCONST.

STATICINLINE compiler option: As of z/OS V1R2, the compiler supports the STATICINLINE compiler option and the default is NOSTATICINLINE. Specify STATICINLINE for compatibility with C++ compilers provided by previous versions of the compiler. For more information about STATICINLINE, refer to *z/OS XL C/C++ User's Guide*.

TARGET compiler option: The TARGET compiler option allows you to compile an application using the current compiler, and then link and run the application on a lower level system. From release to release, the compiler supports a changing list of TARGET suboptions. As newer releases are added to the list, older releases are removed. Because of enhancements and optimizations added to the compiler in a new release, the same TARGET setting may not generate exactly the same code from one release to the next.

As of z/OS V1R7, targeting z/OS V1R3 and earlier releases is no longer supported. The earliest release that can be targeted is zOSV1R4.

New compiler option that may affect source code

ENUM compiler option: z/OS V1R2 introduced the ENUM option as a means for controlling the size of enumeration types. The default setting, ENUM(SMALL), provides the same behavior that occurred in previous releases of the compiler.

If you want to use the ENUM option, it is recommended that the same setting be used for the whole application; otherwise, you may find inconsistencies when the same enumeration type is declared in different compilation units. Use the #pragma enum, if necessary, to control the size of individual enumeration types (especially in common header files).

Compiler options that are no longer supported

As of z/OS V1R2, the following compiler options are no longer supported:

- DECK

  The replacement for DECK functionality that routes output to DD:SYSPUNCH is to use OBJECT(DD:SYSPUNCH).
- GENPCH
- HWOPTS

  The replacement for HWOPTS is ARCHITECTURE.
- LANGLVL(COMPAT)
- OMVS

The replacement for OMVS is OE.

- SRCMSG
- SYSLIB

  The replacement for SYSLIB is SEARCH.
- SYSPATH

  The replacement for SYSPATH is SEARCH.
- USEPCH
- USERLIB

  The replacement for USERLIB is LSEARCH.
- USERPATH

  The replacement for USERPATH is LSEARCH.

As of OS/390 V2R10, the following SOM-related compiler options are no longer supported:

- SOM | NOSOM
- SOMEinit | NOSOMEinit
- SOMGs | NOSOMGs
- SOMRo | NOSOMRo
- SOMVolattr | NOSOMVolattr
- XSominc | NOXSominc

As of OS/390 V2R4, the IDL compiler option is no longer available. If you continue to require IDL for your applications, new IDL or IDL modifications must be coded by hand. You can then use the IDL compiler to generate your C/C++ source code.

Compiler messages and return codes

From release to release, message contents often change and severity levels may also change (for example, an error becoming a warning). These changes to the severity level may affect the return code of the compilation.

You must update any application that is affected by message contents or return codes. **Do not build dependencies on message content, message numbers, or return codes**. See *z/OS XL C/C++ Messages* for a list of compiler messages.

Changes in data set names

The names of IBM-supplied data sets may change from one release to another. See the *z/OS Program Directory* for more information on data set names.

Compiler listings

As of OS/390 V2R6 C/C++, OPT(1) maps to OPT(2). The compiler listing no longer contains the part of the pseudo-assembler listing that was associated with OPT(1). Listing formats, especially the pseudo-assembler parts, will continue to change from release to release. **Do not build dependencies on the structure or content of listings.** For information about C listings or the C++ listings for the current release, refer to *z/OS XL C/C++ User's Guide*.

Changes that affect c89 invocation

As of z/OS V1R6, the –g flag option is no longer translated to the TEST compiler option.

Note: The TEST and GONUMBER options remain unchanged, but work only with 32-bit compiles.

A new environment variable _DEBUG_FORMAT has been introduced to enable users to request the old translation of the -g flag option for 32-bit compiles:

- If _DEBUG_FORMAT equals DWARF (the default), -g is translated to DEBUG(FORMAT(DWARF)).
- If _DEBUG_FORMAT equals ISD, then -g is translated to TEST (the old translation).

For non-DLL C++ compiles, a dummy definition side file will be allocated to prevent the binder from issuing a warning message. If you do want the binder to issue a warning message, when an export symbol is encountered, specify the DLL=NO option for the link-editing phase.

For more information, see the c89 utility information in *z/OS XL C/C++ User's Guide*.

Changes that affect user JCL

Examples of specifying class library header files at compile time

In z/OS V1R1 C/C++, OS/390 V2R10, and earlier compilers, using the following JCL on the compile step would work, although it is *not recommended:*

```
//SYSLIB DD DSN=CEE.SCEEH.H,DISP=SHR
//       DD DSN=CEE.SCEEH.SYS.H,DISP=SHR
//       DD DSN=CBC.SCLBH.H,DISP=SHR
```

As of z/OS V1R2, the record size for the SCLBH data sets have been increased from LRECL=80 to LRECL=120. Due to this change, the SYSLIB shown above will no longer work, and must be removed from your JCL. The replacement for this is the SEARCH compiler option, as in the following example:

```
SEARCH(//'CEE.SCEEH.+',//'CBC.SCLBH.+')
```

Using the SEARCH compiler option instead of a SYSLIB concatenation allows the C++ compiler to search for files based on both the file name and file type.

CBCI and CBCXI procedures

As of z/OS V1R5, the CBCI and CBCXI procedures contain the variable CLBPRFX. If you have any JCL that uses these procedures, you must either customize these procedures (for example, at installation time) or modify your JCL to provide a value for CLBPRFX.

Changes that affect Interprocedural Analysis

Note: For detailed information about using IPA Link step, refer to the *z/OS XL C/C++ User's Guide*.

IPA object module binary compatibility

Release-to-release binary compatibility is maintained by the z/OS XL C/C++ IPA Compile and IPA Link as follows:

- An object file produced by an IPA Compile which contains IPA Object or combined IPA and conventional object information can be used as input to the IPA Link of the same or later Version/Release of the compiler.

- An object file produced by an IPA Compile which contains IPA Object or combined IPA and conventional object information cannot be used as input by the IPA Link of an earlier Version/Release of the compiler. If this is attempted, an error diagnostic message will be issued by the IPA Link.
- Note that if the IPA object is reproduced by a later IPA Compile, additional optimizations may be performed and the resulting application program may perform better.

Exception: The IPA object files produced by the OS/390 V1R2 C IPA Compile. These must by recompiled from the program source using an OS/390 V1R3 or later compiler before attempting to process them with the z/OS V1R7 XL C/C++ IPA Link.

IPA Link Step defaults

As of OS/390 V1R3, the following IPA Link Step defaults changed:

- The default optimization level is `OPT(1)`
- The default is `INLINE`, unless `NOOPT`, `OPT(0)` or `NOINLINE` is specified.
- The default inlining threshold is now 1000 ACUs (Abstract Code Units). With OS/390 C/C++ V1R2, the threshold was 100 ACUs.
- The default expansion threshold is now 8000 ACUs. With OS/390 C/C++ V1R2, the threshold was 1000 ACUs.

As of OS/390 V2R6, the default optimization level for the IPA Link step is `OPT(2)`.

Changes that affect data type support

Effect of ARCH level on conversion from floating point to integer type

Consider the following piece of code where a floating point type is converted to a signed integer type:

```
double x;
int i;
/* ... */

i = x; /* overflow if x is too large */
/* value of i undefined */
```

When the conversion causes an overflow (that is, the floating type value is larger than `INT_MAX`), the behavior is undefined according to the C Standard.

The actual result depends on the ARCHITECTURE level (the `ARCH` option), which determines the machine instruction used to do the conversion. For example, there are input values that would result in a large negative value for `ARCH(2)` and below, while the same input would result in a large positive value for `ARCH(3)` and above.

If overflow processing is important to the program, it should be checked explicitly. For example:

```
double x;
int i;
if (x < (double) INT_MAX)
    i = x;
else {
    /* overflow */
}
```

Compiler-defined _LONG_LONG macro

The long long data type is supported as a native data type when the LANGLVL(LONGLONG) option is turned on. This option is turned on by default by the compiler option LANGLVL(EXTENDED). The _LONG_LONG macro is predefined for all language levels other than ANSI.

As of z/OS V1R6, when LANGLVL(LONGLONG) is turned on, the _LONG_LONG macro is defined by the compiler.

Attention: If you have defined your own _LONG_LONG macro in previous compiler releases, you must remove this user-defined macro before compiling your program.

Chapter 12. Language Environment changes between OS/390 C/C++ and z/OS V1R7 XL C/C++

This chapter describes the Language Environment elements that may impact your migration from a release of OS/390 C/C++ or z/OS V1R1 C/C++ to z/OS V1R7 XL C/C++.

Name conflicts with run-time library functions

When taking code previously compiled and link-edited on a system earlier than OS/390 V2R4, and moving to a system at OS/390 V2R4 or later, you might have a problem with name conflicts if both the following are true:

1. You created functions with the same name as library functions.
2. When linking your application you included the IBM supplied Language Environment link library before the files that contain your function definitions.

Previous releases of the OS/390 C/C++ run-time headers used the #pragma map directive to convert many function names into identifiers prefixed with "@@". For example, if you included `fcntl.h` in your source, a reference to `open()` in your source code resulted in an external name `@@OPEN` in the object code. As of OS/390 V2R4 many pragma maps have been eliminated. If you created functions with the same name as library functions, you must ensure that the file containing your version of the function precedes the IBM supplied Language Environment link library in the search order when linking your application. If you have object modules containing identifiers like `OPEN` that you want resolved to your version of `open()`, you may need to alter your JCL to ensure that your version precedes the IBM supplied Language Environment link library in the search order.

Also, if you have multiple, interdependent modules that rely on the name mapping present in prior releases, you cannot recompile one without recompiling the others. For example, module A includes `fcntl.h` and calls `open()` resulting in a reference to `@@OPEN` in the object code. Module B implements your version of `open()` and also includes `fcntl.h`, so that the external name of the called function is mapped to `@@OPEN`. You must recompile both modules.

Table 14 lists the functions that had pragma maps deleted in OS/390 V2R4.

Table 14. Functions that had pragma maps deleted

____loc1()	__atoe()	__atoe_l()	__cnvblk()	__dlght()
__etoa()	__etoa_l()	__gderr()	__getipc()	__ipdbcs()
__ipdspx()	__iphost()	__ipmsgc()	__ipnode()	__iptcpn()
__opargf()	__operrf()	__opindf()	__opoptf()	__sigerr()
__sigign()	__sigpro()	__tzone()	__wsinit()	_longjmp()
_setjmp()	_tolower()	_toupper()	accept()	access()
alarm()	a64l()	basename()	bcmp()	bcopy()
bind()	brk()	bzero()	catclose()	catgets()
catopen()	cclass()	chaudit()	chdir()	chmod()
chown()	chroot()	clearenv()	clearenv()	close()
closedir()	closelog()	clrmemf()	confstr()	connect()
creat()	crypt()	ctdli()	ctdli()	ctermid()
ctermid()	cuserid()	cuserid()	dirname()	drand48()
dup()	dup2()	dynalloc()	dynfree()	ecvt()
encrypt()	endgrent()	endpwent()	erand48()	execl()

Table 14. Functions that had pragma maps deleted (continued)

execle()	execlp()	execv()	execve()	execvp()
fattach()	fchaudit()	fchdir()	fchmod()	fcntl()
fcvt()	fdelrec()	fdetach()	fetch()	fetchep()
ffs()	fileno()	fldata()	flocate()	fmtmsg()
fnmatch()	fork()	fstat()	fstatvfs()	ftime()
ftok()	ftw()	fupdate()	gcsp()	gcvt()
getcwd()	getdate()	getegid()	geteuid()	getgid()
getgrent()	getgrgid()	getgrnam()	getmsg()	getopt()
getopt()	getpass()	getpgid()	getpgrp()	getpid()
getpmsg()	getppid()	getpwent()	getpwnam()	getpwuid()
getsid()	getsyntx()	getuid()	getutxid()	getw()
getwd()	glob()	globfree()	grantpt()	hcreate()
hdestroy()	hsearch()	iconv()	index()	insque()
ioctl()	ioctl()	isatty()	isnan()	jrand48()
kill()	killpg()	lchown()	lcong48()	lfind()
link()	listen()	lockf()	lrand48()	lsearch()
lseek()	lstat()	l64a()	maxcoll()	maxdesc()
memccpy()	mkdir()	mkfifo()	mkstemp()	mktemp()
mmap()	mount()	mprotect()	mrand48()	msgctl()
msgget()	msgrcv()	msgsnd()	msgxrcv()	msync()
munmap()	nftw()	nice()	nlist()	nrand48()
open()	opendir()	openlog()	pathconf()	pause()
pclose()	pipe()	poll()	popen()	ptsname()
putenv()	putmsg()	putpmsg()	putw()	random()
re_comp()	re_exec()	read()	readdir()	readv()
realpath()	recv()	recvfrom()	regcmp()	regcomp()
regerror()	regex()	regexec()	regfree()	release()
remque()	rexec()	rindex()	rmdir()	sbrk()
scalb()	seed48()	seekdir()	semctl()	semget()
semop()	send()	sendto()	setegid()	setenv()
setenv()	seteuid()	setgid()	setgrent()	setkey()
setpeer()	setpgid()	setpgrp()	setpwent()	setregid()
setreuid()	setsid()	setstate()	setuid()	shmat()
shmctl()	shmdt()	shmget()	shutdown()	sighold()
sigpause()	sigrelse()	sigset()	sigstack()	sigwait()
sleep()	socket()	spawn()	spawnp()	srandom()
srand48()	stat()	statvfs()	strdup()	strfmon()
strptime()	svc99()	swab()	sync()	sysconf()
syslog()	t_accept()	t_alloc()	t_bind()	t_close()
t_error()	t_free()	t_listen()	t_look()	t_open()
t_rcv()	t_rcvdis()	t_rcvrel()	t_snd()	t_snddis()
t_sndrel()	t_sync()	t_unbind()	tcdrain()	tcflow()
tcflush()	tcgetsid()	tdelete()	telldir()	tempnam()
tfind()	times()	tinit()	truncate()	tsearch()
tsetsubt()	tsyncro()	tterm()	ttyname()	ttyslot()
twalk()	tzset()	ualarm()	ulimit()	umask()
umount()	uname()	unlink()	unlockpt()	usleep()
utime()	utimes()	utimes()	valloc()	vfork()
w_ioctl()	w_statfs()	wait()	waitid()	waitpid()
wait3()	wordexp()	wordfree()	write()	writev()

Time functions

You should customize your locale information. Otherwise, in rare cases, you may encounter errors. In a POSIX application, you can supply time zone and alternative time (for example, daylight) information with the TZ environment variable. In a non-POSIX application, you can supply this information with the _TZ environment variable. If no TZ environment variable is defined for a POSIX application or no _TZ environment variable is defined for a non-POSIX application, any customized information provided by the LC_TOD locale category is used. By setting the TZ environment variable for a POSIX application, or the _TZ environment variable for a non-POSIX application, or by providing customized time zone or daylight information in an LC_TOD locale category, you allow the time functions to preserve both time and date, correctly adjusting for alternative time on a given date.

Refer to *z/OS XL C/C++ Programming Guide* for more information about both environment variables and customizing a locale.

Direct UCS-2 and UTF-8 converters

OS/390 V2R9 added new UCS-2 and UTF-8 converters. These are direct conversions that no longer use the tables built by the uconvdef utility processing of UCMAPS. If you have modified UCMAPS, UCS-2 and UTF-8 converters will no longer use those modified UCMAPS. If you still need to use the modifications that you made to UCMAPS, you will now need to set the _ICONV_UCS2 environment variable to "0". Refer to *z/OS XL C/C++ Programming Guide* for more information about the _ICONV_UCS2 environment variable.

Default option for ABTERMENC changed to ABEND

As of OS/390 V2R9, the default option for ABTERMENC is ABEND instead of RETCODE. If you are expecting the default behavior of ABTERMENC to be RETCODE, you **must** change the setting in CEEDOPT (CEECOPT for CICS). Refer to *z/OS Language Environment Customization* for details on changing CEEDOPT and CEECOPT.

THREADSTACK run-time option

As of OS/390 V2R10 Language Environment, the new THREADSTACK run-time option replaces the NONIPTSTACK and NONONIPTSTACK options. The old options will still be accepted, but an information message will be issued, telling the user to switch to the new THREADSTACK option. The old options do not have support for specifying the initial and increment sizes of the new XPLINK downward growing stack. Refer to *z/OS Language Environment Customization* for more information on the THREADSTACK run-time option.

Changes to putenv()

As of z/OS V1R5, the C/C++ function putenv() changed to place the string passed to putenv() directly into the array of environment variables. This behavior assures compliance with the POSIX standard. Before the change, the storage used to define the environment variable passed into putenv() was not added to the array of environment variables. Instead, the system copied the string into system allocated storage. To restore the previous behavior of putenv(), set environment variable _EDC_PUTENV_COPY to YES.

For additional information on putenv() and _EDC_PUTENV_COPY, see *z/OS XL C/C++ Run-Time Library Reference*. You may also refer to *z/OS XL C/C++ Programming*

Guide, for information on `putenv()` and `_EDC_PUTENV_COPY`.

Chapter 13. Class library changes between OS/390 C/C++ and z/OS V1R7 XL C/C++

This chapter describes the changes that you may have to make if you are using class libraries and migrating to z/OS V1R7 XL C/C++ from a release of OS/390 C/C++ or z/OS V1R1 C/C++XL C/C++. Also refer to Chapter 14, “Migrating to the currently supported Standard C++,” on page 87 for details on support of *Programming languages - C++ (ISO/IEC 14882:2003(E))*, which documents the currently supported Standard C++.

IBM Open Class Library

As of z/OS V1R5, development with the IBM Open Class Library (IOC) is not supported. You can no longer compile and link applications that use IOC classes. This includes all the clases, templates, and facilities that are described in *IBM Open Class Library Reference* with the two exceptions noted below. Run-time support is provided for existing applications that use IOC, but this support will be removed in a future release.

The following classes are still supported for application development:

- UNIX System Laboratories (USL) I/O Stream Library
- USL Complex Mathematics Library

As of z/OS V1R5, the name of the element that provides this application development support has changed from IBM Open Class Library to Run-Time Library Extensions. The directory path for the header file has changed from `/usr/lpp/ioclib` to `/usr/lpp/cbclib`.

Although support for these classes is not being removed at this time, it is recommended that you migrate to the Standard C++ iostream and complex classes. This is especially important if you are migrating other IOC streaming classes to Standard C++ Library streaming classes, because combining USL and Standard C++ Library streams in one application is not recommended. For more information about these classes, see *C/C++ Legacy Class Libraries Reference*.

For information about migrating away from these classes, see *IBM Open Class Library Transition Guide*.

Migrating from USL I/O Stream Library to Standard C++ I/O Stream Library

The values for some enumerations differ slightly between the USL and Standard C++ I/O Stream libraries. This may cause problems when migrating to the Standard C++ I/O Stream Library.

The following flags have been added:

- flags for controlling formatting: `boolalpha`, `adjustfield`, `basefield`, `floatfield`

The following flags have been removed:

- flags for controlling formatting: `stdio`
- flags for controlling the open mode: `nocreate`, `noreplace`, `bin`
- flags for controlling the io state: `hardfail`

There may be other small differences.

Mixing the C++ Standard I/O Stream Library, USL I/O Stream Library, and C I/O

While it is possible to mix the Standard C++ I/O Stream Library, the USL I/O Stream Library, and C I/O, it is not recommended. The USL I/O Stream Library uses a separate buffer so you would need to flush the buffer after each call to `cout` by either setting `ios::unitbuf` or calling `sync_with_stdio()`. You should avoid switching between the I/O Stream Library formatted extraction functions and C stdio library functions whenever possible, and you should also avoid switching between versions of the I/O Stream Libraries. For more information, see *z/OS XL C/C++ Programming Guide* and *C/C++ Legacy Class Libraries Reference*.

Removal of SOM support

As of OS/390 V2R10, the IBM System Object Model (SOM) is no longer supported in the C++ compiler.

Removal of Database Access Class Library utility

As of OS/390 V2R4, the Database Access Class Library utility is no longer available.

Part 5. ISO C/C++ Standard migration issues

This part discusses the implications of migrating applications that were created with C/C++ compilers that are not compliant with *Programming languages - C++ (ISO/IEC 14882:2003(E))*, which documents the currently supported Standard C++.

As of z/OS V1R2, the z/OS C++ compiler was compliant with *Programming languages - C++ (ISO/IEC 14882:1998(E))*.

As of z/OS V1R7:

- z/OS XL C/C++ is compliant with the 2003 standard.
- OS/390 V2R10 compiler is no longer shipped with the z/OS product.

As of z/OS V1R7 XL C/C++,

Note: You can identify the ISO Standard level that is supported by the compiler by checking the standard macro __**cplusplus** and its value, which remains unchanged from V1R6. This macro has the value 199711. If you are compiling a C ++ translation unit, the name _ _`cplusplus` is defined to the value 199711L .

Chapter 14. Migrating to the currently supported Standard C++

This chapter discusses the implications of migrating applications that were compiled using a compiler that is not compliant with any ISO/IEC C++ standard.

These compilers include:

- Any release earlier than the z/OS V1R2 C/C++ compiler
- The OS/390 V2R10 C/C++ compiler that was reshipped with z/OS V1R2, V1R3, V1R4, V1R5, V1R6 C/C++

Note: As of V1R7, the OS/390 V2R10 C/C++ compiler is no longer shipped with the z/OS product.

Code that compiles without errors in earlier C++ compilers may produce warnings or error messages in the z/OS XL V1R7 C++ compiler. This could be due either to changes in the language or to differences in the compiler behavior. Language elements that may affect your code are shown in the topics “Changes in language features to comply with the currently supported Standard C++” on page 89 and “Language features that comply with the currently supported Standard C++” on page 91. The topic“Errors due to changes in compiler behavior” on page 92 may also be applicable to the output from your compiler.

Choosing an approach based on your migration objectives

Table 15 shows the different migration scenarios and the recommended approach for each.

Table 15. Migration objectives and approaches

Is code compliant with 1998 ISO Standard C++?	**Migration objectives**	**Action**
Yes (ported or new).	Remain compliant with Standard C++	Use LANGLVL(ANSI)
No	Exploit Standard C++ language features, even if codebases must be modified	Use the following compiler options and suboptions to aid the migration process: • LANGLVL(COMPAT92) • LANGLVL() suboptions to control individual language features **Note:** See Table 16 on page 88.
	• Ignore Standard C++ language features • Avoid modifying codebases	Use LANGLVL(COMPAT92) to tolerate language incompatibilities

Compiler options for compatibility with earlier C/C++ compilers

To make your application conform to the C++ Standard, you may need to change your existing source code. You can use the compiler options and suboptions listed in Table 16 on page 88 to break up the changes into smaller steps. (For details, see *z/OS XL C/C++ User's Guide*.)

You can also use the following predefined option groups:

LANGLVL(COMPAT92)
Use this option group if your code compiles with a previous compiler and you want to move to z/OS V1R7 XL C/C++ with minimal changes. This group is the closest you can get to the old behavior of the previous compilers.

LANGLVL(STRICT98) or LANGLVL(ANSI)
These two groups are identical. Use one of them when you compile new or ported code that is C++ Standard compliant.

LANGLVL(EXTENDED)
This option group indicates all language constructs available with z/OS XL C/C++. This enables extensions to the C++ Standard.

The following table lists the options and settings that are included in each group.

Note: Except for TMPLPARSE, all settings have a value of either **On** (meaning the suboption is enabled) or **Off** (meaning the suboption is not enabled).

Table 16. Compiler options and suboptions for compatibility with previous compilers

Option	Group		
	compat92	**strict98 | ansi**	**extended**
KEYWORD(bool) | NOKEYWORD(bool)	Off	On	On
KEYWORD(explicit) | NOKEYWORD(explicit)	Off	On	On
KEYWORD(export) | NOKEYWORD(export)	Off	On	On
KEYWORD(false) | NOKEYWORD(false)	Off	On	On
KEYWORD(mutable) | NOKEYWORD(mutable)	Off	On	On
KEYWORD(namespace) | NOKEYWORD(namespace)	Off	On	On
KEYWORD(true) | NOKEYWORD(true)	Off	On	On
KEYWORD(typename) | NOKEYWORD(typename)	Off	On	On
KEYWORD(using) | NOKEYWORD(using)	Off	On	On
LANGLVL(ANONSTRUCT | NOANONSTRUCT)	Off	Off	On
LANGLVL(ANONUNION | NOANONUNION)	On	Off	On
LANGLVL(ANSIFOR | NOANSIFOR)	Off	On	On
LANGLVL(ANSISINIT | NOANSISINIT)	Off	On	On
LANGLVL(ILLPTOMEM | NOILLPTOMEM)	On	Off	On
LANGLVL(IMPLICITINT | NOIMPLICITINT)	On	Off	On
LANGLVL(LIBEXT | NOLIBEXT)	On	Off	On
LANGLVL(LONGLONG | NOLONGLONG)	On	Off	On
LANGLVL(OFFSETNONPOD | OFFSETNONPOD)	On	Off	On
LANGLVL(OLDDIGRAPH | OLDDIGRAPH)	Off	On	Off
LANGLVL(OLDFRIEND | NOOLDFRIEND)	On	Off	On
LANGLVL(OLDMATH | NOOLDMATH)	On	Off	Off
LANGLVL(OLDTEMPACC | NOOLDTEMPACC)	On	Off	On
LANGLVL(OLDTMPLALIGN | NOOLDTMPLALIGN)	On	Off	Off
LANGLVL(OLDTMPLSPEC | NOOLDTMPLSPEC)	On	Off	On
LANGLVL(TRAILENUM | NOTRAILENUM)	On	Off	On
LANGLVL(TYPEDEFCLASS | TYPEDEFCLASS)	On	Off	On

Table 16. Compiler options and suboptions for compatibility with previous compilers (continued)

Option	Group		
	compat92	**strict98 | ansi**	**extended**
LANGLVL(ZEROEXTARRAY | NOZEROEXTARRAY)	Off	Off	On
RTTI | NORTTI	Off	On	On
TMPLPARSE(NO | ERROR | WARN)	NO	WARN	NO

Changes in language features to comply with the currently supported Standard C++

Refer to the *z/OS XL C/C++ Language Reference* for details.

LANGLVL(ANSISINIT) and static initialization

As of z/OS V1R5, you can use the LANGLVL(NOANSISINIT) option to maintain the same order of destruction for statically initialized objects whenever you compile programs that had previously been compiled with z/OS V1R1 and earlier C/C++ compilers.

As of z/OS V1R2 (when the compiler became fully compliant with *Programming languages - C++ (ISO/IEC 14882:2003(E))*, DLLs built by the compiler run object destructors differently from those created with the earlier C/C++ compilers.

Table 17. Destruction of statically initialized objects before and after compliance with ISO/IEC 14882:2003(E)

z/OS V1R1 and earlier C/C++ compilers	z/OS V1R2 and later compilers
Destructor calls are run as the last thing on the `atexit` list, as part of the termination code.	If an object is created with the C++ standard way of initializing (LANGLVL(ANSISINIT)): • Destructor calls for objects created by z/OS V1R2 and later compilers are added to the `atexit` list. This list will then be run before the `atexit` entry for the termination code. • Any DLL built with z/OS V1R2 and later compilers will have the destructors for the global objects run in the wrong order relative to other DLLs or main program that were built with z/OS V1R1 and earlier C/C++ compilers.

For-loop scoping

In Standard C++, the scope of a variable in a `for-loop` initializer declaration is to the end of the loop body. The scope of such variables in the z/OS V1R1 compiler and earlier C/C++ compilers, is to the end of the lexical block containing the `for-loop`. For example:

```
int i=0;

void f()
{
    for(int i=0; i<10; i++)
    {
       if(...) break;
```

```
    }
    if(i==10) { ... }      // 1
    ...
}
```

1

- As of z/OS V1R2, this means that `i` is declared in file scope (::).
- In z/OS V1R1 and earlier C/C++ compilers, this means that the `i` is declared in the `for-loop`. As of z/OS V1R5, to maintain this context use `LANGLVL(NOANSIFOR)` option.

Implicit int and type declarations

The use of an implicit `int` in a declaration is no longer valid in Standard C++, as shown in the following example:

```
const i;   // previously meant const int i
main() { } // previously returned int
```

Hence, as of z/OS V1R2, the following code is no longer valid:

```
inline f() {
    return 0;
}
```

To comply with the standard, specify the type of every function and variable. Use the `LANGLVL(IMPLICITINT)` option to compile code containing implicit `int`s.

Changes to friend declarations

As of the z/OS V1R2 C++ compiler, a class named as a `friend` is not visible until introduced into scope by another declaration:

```
class C {
    friend class D;
};
D* p;  // error, D not in scope
```

Friend class declarations must always be elaborated.

```
friend class C; // need class keyword
```

To allow friend declarations without elaborated class names, use `LANGLVL(OLDFRIEND)` option.

Exception handling and cv-qualification

As of z/OS V1R2:

- A temporary copy is thrown rather than the actual object itself.
- The cv-qualification in the catch clause is not considered when the type caught is the same (possibly cv-qualified) type as that thrown or a reference to the same (possibly cv-qualified) type.

  Note: *cv* is short form for *const/volatile*.
- New casts also throw exceptions.

This is not the case in z/OS V1R1 and earlier C/C++ compilers. As of z/OS V1R5, there is no available option to enable the old behavior.

Language features that comply with the currently supported Standard C++

Keywords

As of z/OS V1R2, the following names are reserved as keywords and cannot be used for naming identifiers:

- `bool`
- `explicit`
- `export`
- `false`
- `mutable`
- `namespace`
- `true`
- `typename`
- `using`

If you are compiling old code that uses a keyword (for example, `typename`) as an identifier, you can either remove your definition, or use the `NOKEYWORD(typename)` option.

Namespaces and macro definitions

Namespaces are not supported in the z/OS V1R1 and earlier C/C++ compilers. Code that has been ported to pre-z/OS V1R2 from platforms that support namespaces, may have implemented a workaround by defining `namespace` as a macro to nothing.

Example:

```
#define std
#define using
#define namespace
```

As of the z/OS V1R5 C/C++ compiler, you need to undefine the macro before you can compile such code.

The bool type and returned values

The `bool` type is not supported in the OS/390 V2R9 compiler and earlier C/C++ compilers. Relational operators returned `int`.

As of z/OS V1R5, relational operators return `bool` instead of `int`. To disable this keyword use the `NOKEYWORD(bool)` option.

The mutable keyword and macro definitions

As of z/OS V1R2, the `mutable` keyword allows a class data member to be modified even though it is the data member of a `const` object.

The `mutable` keyword is not supported in the z/OS V1R1 compiler and earlier C/C++ compilers. Code ported to these compilers from other platforms might have implemented a workaround by defining `mutable` as a macro to nothing:

```
#define mutable
```

As of z/OS V1R2, you need to undefine the macro before you can compile such code.

As of z/OS V1R5, you can use the NOKEYWORD(mutable) option to disable this keyword.

Wide character definitions (wchar_t)

As of z/OS V1R5, the C/C++ compiler defines wchar_t as a simple type. The z/OS V1R1 compiler and earlier C/C++ compilers required that wide characters be defined as typedef.

The explicit keyword

The z/OS V1R1 compiler and earlier C/C++ compilers did not support the explicit keyword.

The purpose of this keyword is to make what would otherwise be a conversion constructor into a normal constructor:

Example:

```
class C {
    explicit C(int);
};
C c(1);   // ok
C d = 1;  // error, no conversion constructor
```

As of z/OS V1R5, you can use the NOKEYWORD(explicit) option to disable this keyword.

C++ cast operators

z/OS V1R2 introduced new cast operators: const_cast, dynamic_cast, reinterpret_cast and static_cast. These were not supported in the z/OS V1R1 compiler and earlier C/C++ compilers.

Changes to digraphs in the C++ Language

Programming languages - C++ (ISO/IEC 14882:2003(E)) now defines and, bitor, or, xor, compl, bitand, and_eq, or_eq, xor_eq, not, and not_eq as alternate tokens for &&, |, ||, ^, ~, &, &=, |=, ^=, ! and !=.

As of z/OS V1R2, a program that uses any of these alternate tokens as variable, function, or type names, must be compiled with the NODIGRAPH option to suppress the parsing of these tokens as digraphs.

Errors due to changes in compiler behavior

This section describes coding that compiles without errors in the z/OS V1R1 compiler and earlier C/C++ compilers but produces errors or warnings in the z/OS V1R7 compiler. For more details on compiler messages, refer to *z/OS XL C/C++ Messages*.

Access-checking errors

Example:

```
class A {
    class B {
        void f(A::B);
```

```
        // A::B is private and cannot be accessed from B
        // void f(B); <--this is the appropriate change which
        // works for both compilers.
    };
};
```

The following code would result in the error `CCN5413:"A::B" is already declared with a different access:`

```
class A {
  public:
    class B;
    const B& foo();
  private:
    class B {};
};
```

This can be solved by either moving the definition of class B to the public part of class A (before the declaration of `foo()`) or moving the declaration of the member function foo to the private of class A (after the class B definition).

Type definition errors

This code will generate error `CCN5193: A typedef name cannot be used in this context`. Do not use the typedef-name; instead, use the name of the class:

```
class A { };
typedef A B;
class C {
    friend class B; // Should be friend class A;
};
```

Errors caused by ambiguous overloads

Programming languages - C (ISO/IEC 9899:2003) introduced error messages for standard floating point and long double overloads of the standard math functions.

As of z/OS V1R5, compiling the following code example will produce an error message.

Code example:

```
#include <math.h>
int main()
{
    float a = 137;
    float b;
    b = pow(a, 2.0);        // The call to "pow" has no best match.
    return 0;
}
```

Error message: `CCN5219: The call to "pow" has no best match`

Solutions: You can avoid the error if you do either of the following:

- Use the `LANGLVL(OLDMATH)` option, which removes the float and long double overloads.
- Cast `pow` arguments. For example, casting `2.0` to be of type `float` solves the problem:

  ```
  b = pow(a, (float)2.0);
  ```

Errors caused by user-defined conversions

Code example:

```
//e.C
struct C {};
struct A {
    A();
    A(const C &);
    A(const A &);
};
struct B {
    operator A() const { A a ; return a;};
    operator C() const { C c ; return c;};
};
void f(A x) {};
int main(){
    B b;
    f((A)b);
    // The call matches two constructors for A instead of calling operator A()
    return 0;
}
```

Error messages:

```
CCN5216: An expression of type "B" cannot be converted to "A".
CCN5219: The call to "A::A" has no best match.
CCN6228: Argument number 1 is an lvalue of type "B".
CCN6202: No candidate is better than "A::A(const A&)".
CCN6231: The conversion from argument number 1 to "const A &" uses the
user-defined conversion "B::operator A() const" followed by an
lvalue-to-rvalue transformation.
CCN6202: No candidate is better than "A::A(const C &)".
CCN6231: The conversion from argument number 1 to "const C &" uses the
user-defined conversion "B::operator C() const ".
```

Potential solutions:

- Changing `f((A)b)` to the explicit call `f(b.operator A())`
- Removing the constructor `A(const C &)`
- Adding a constructor `A(B)`
- Removing either operator `A()` or operator `C()`

Note: The solution you choose depends on your access to classes A, B, and C.

Syntax errors with new

The z/OS V1R1 compiler and earlier C/C++ compilers treated the following two statements as semantically equivalent:

```
new (int *) [1];
new int* [1];
```

The first statement is syntactically incorrect even in older versions of the C++ Standard. However, previous versions of C++ accepted it. This inconsistency with the language standard was corrected in the z/OS V1R2 compiler; the first statement will now produce a compilation error.

Changes in template compilations

If your code makes use of templates, you will be affected by various changes that were introduced in z/OS V1R2.

Name resolution

The z/OS V1R1 C++ compiler and earlier C/C++ compilers do not parse or otherwise process a class or function template definition until it has been

determined that an instantiation of that template is required. Template definitions for which no instantiation is required are never parsed by the z/OS V1R1 C++ compiler and earlier C/C++ compilers.

By default, as of z/OS V1R2, the C++ compiler processes class and function template definitions in two phases:

- When the template definition is encountered by the compiler, the definition is parsed. Names that are used in the template definition and that are not dependent on the template parameters are resolved at this time.
- When it is determined that a specific instantiation of the template is required, names that are dependent on the template parameters are resolved and an implicit specialization is instantiated.

To approximate the behavior of the z/OS V1R1 C++ compiler and earlier C/C++ compilers, users of the C++ compiler, as of z/OS V1R5, may use the `TMPLPARSE(NO)` option to override this default behavior. When the `TMPLPARSE(NO)` option is in effect, the first phase described above is delayed until it is determined that an instantiation is required. Template definitions for which no instantiation is required are not parsed. The `TMPLPARSE(NO)` option does not eliminate the distinction between the two phases.

An unqualified name that is not found by name lookup and not indicated to be a type by the `typename` keyword, is assumed to not name a type.

Unqualified name lookup does not consider template-dependent base classes.

Example: As of z/OS V1R2, template-dependent base classes are not searched during name resolution:

```
int *t=0;
template <class T> struct Base {
    U t;
};
template <class T> class C : public Base<T> {
    T f() {
        return t;  // refers to global int *t
    }
};
```

Example: The keyword `typename` must be used to mark a qualified dependent name as a type. The following example illustrates this:

```
template <class T> struct A
{
    typedef int X;
};
template <class T> struct B:A <T>
{
    T::Y b1;      //error Y is not a type
    A <T>::X b2;  // error X is not a type
    void foo(X);  // error X is not a type
};
```

The errors can be fixed by changing the definition of B to:

```
template <class T> struct B : A <T>
{
    typename T::Y b1;
        // keyword "typename" tells parser Y is a type
    typename A<T>::X b2;
```

```
        // keyword "typename" tells parser X is a type
    void foo(typename A<T>::X);
        // keyword "typename" tells parser X is a type
};
```

Example of template keyword

As of z/OS V1R2, the `template` keyword is used to indicate templates in qualifiers. For example:

```
struct A {
    Template<class T> T f(T t) { return t;}
};
template <class T> class C {
    void g(T* a) {
    // The following would become ambiguous without
    // the keyword template
    int i = a->template f<int>(1);
}
C<A> c;
```

Template specialization

As of z/OS V1R2, template specializations must be preceded with the string `template<>`. For example:

```
template <class T> class C {};
template <> C<int> { int i; };
```

Explicit call to destructor of scalar type

This problem is not template-specific, but usually occurs in templates.

Example:

```
typedef int INT;
INT *p;
// ...
p->INT::~INT(); // ok in z/OS V1R5 C++
```

The z/OS V1R1 compiler and earlier C/C++ compilers give a warning to the explicit destructor call. You can safely ignore this warning.

Friend declarations in templates

Since z/OS V1R2, `friend` declarations in templates may not have the same meaning as with earlier C/C++ compilers. For example, the following code will generate a warning message:

```
struct A {} a;
template <class T> struct S;
template <class T> void f(T&, S<T>&) {}
template <class T> A& operator << (A&, S<T>&) { return a; }
template <class T> struct S
{
    friend void f (T&, S&); // no explicit arguments
    friend A& operator <<(A&, S&); // no explicit arguments
};
```

To migrate this code, the `friend` declarations should be changed to include explicit template arguments:

```
template <class T> struct S
{
    friend void f<T> (T&, S&); // explicit argument T
    friend A& operator << <T>(A&, S&); // explicit argument T
};
```

Without the explicit arguments, the `friend` declarations will introduce non-template functions `'f(int&, S&)'` and `'operator <<(A&, S&)'` into global scope and these non-template functions (which have no corresponding definition) will be the friends of `S`.

With the template argument added explicitly, an instantiation of `S`, such as `S<int>`, will make the template instantiations `f<int>(int&, S<int>&)` and `operator << <int> (A&, S<int>&)`, friends of `S`.

The z/OS V1R1 compiler and earlier C/C++ compilers would not accept explicit template arguments on `friend` declarations. If you wish to maintain compatibility with earlier C/C++ compilers, the explicit template arguments should be added with the use of a macro.

Friend declarations in class member lists

A friend declaration in a class member list grants, to the nominated friend function or class, access to the private and protected members of the enclosing class. In z/OS V1R1 and earlier C/C++ compilers, friend declarations introduce the name of a nominated friend function to the scope that encloses the class containing the friend declaration. As of z/OS V1R2, friend declarations do not introduce the name of a nominated friend function to the scope that encloses the class containing the friend declaration.

In the example source file below, the function name `lib_func1` is not known to the z/OS V1R6 C++ compiler at the point at which it is called in the function `f`. This source file will not compile successfully.

```
// g.C
// ---
class A {
    friend int lib_func1(int); // This function is from a library.
};
int f(){
    return lib_func1(1);
}
```

The example will compile successfully if the following declaration is added to the file in the global namespace scope at some point prior to the definition of the function named `f`.

```
int lib_func1(int);
```

Inlined virtual functions in a class

Whenever a virtual function exists in a class, the compiler generates a virtual function table for the class and stores a pointer to the table. For any class that has at least one virtual function that is not defined as inline, the compiler can generate the virtual function table in the same module as the definition of the first non-inlined virtual function. Only one copy of the virtual function table for a class will exist.

However, when all virtual functions for a class are inlined, the compiler has insufficient information to generate a unique virtual function table and, instead, generates a virtual function table in each module that uses the class.

As of z/OS V1R2, the virtual function table is visible to the binder. Therefore, in the situation where a class has inlined virtual functions, the binder detects the virtual function tables in more than one module and generates duplicate object warnings.

Part 6. From earlier releases of z/OS C/C++ to z/OS V1R7 XL C/C++

This part discusses the implications of migrating applications that were created with one of the following compilers and one of the following libraries to the z/OS V1R7 XL C/C++ product.

Compilers:

- IBM z/OS V1R2 C/C++ compiler, 5694-A01
- IBM z/OS V1R3 C/C++ compiler, 5694-A01
- IBM z/OS V1R4 C/C++ compiler, 5694-A01
- IBM z/OS V1R5 C/C++ compiler, 5694-A01
- IBM z/OS V1R6 C/C++ compiler, 5694-A01
- IBM z/OS.e V1R3 C/C++ compiler, 5655-G52
- IBM z/OS.e V1R4 C/C++ compiler, 5655-G52
- IBM z/OS.e V1R5 C/C++ compiler, 5655-G52
- IBM z/OS.e V1R6 C/C++ compiler, 5655-G52

Libraries:

- IBM z/OS V1R2 Language Environment, 5694-A01
- IBM z/OS V1R3 Language Environment, 5694-A01
- IBM z/OS V1R4 Language Environment, 5694-A01
- IBM z/OS V1R5 Language Environment, 5694-A01
- IBM z/OS V1R6 Language Environment, 5694-A01
- IBM z/OS.e V1R4 Language Environment, 5655-G52
- IBM z/OS.e V1R5 Language Environment, 5655-G52
- IBM z/OS.e V1R6 Language Environment, 5655-G52

Notes:

1. The z/OS V1R3 and V1R4 compilers are equivalent to the z/OS V1R2 compiler.
2. The z/OS V1R1 compiler and library are equivalent to the OS/390 V2R10 compiler and library, and are described in Part 4 of this book.
3. To aid in migration, the OS/390 V2R10 C/C++ compiler was shipped as part of z/OS V1R2, V1R3, V1R4, V1R5, and V1R6 C/C++. For information about migrating applications that were compiled using the OS/390 V2R10 C/C++ compiler (even if the operating system level was z/OS V1R1, V1R2, V1R3, V1R4, V1R5, or V1R6), see Part 4, “From OS/390 C/C++ to z/OS V1R7 XL C/C++,” on page 69 and Part 5, “ISO C/C++ Standard migration issues,” on page 85. As of V1R7, OS/390 V2R10 compiler is no longer shipped with the z/OS product.
4. The z/OS.e compilers and libraries are functionally equivalent to the corresponding z/OS compilers and libraries.

Chapter 15. Source program compatibility

In general, you can use source programs with the z/OS V1R7 XL C/C++ product without modification, if they were created with one of the earlier versions of the z/OS C/C++ compiler.

This chapter highlights the exceptions.

Support of Standard C++

As of z/OS V1R7, the C++ compiler supports *Programming languages - C++ (ISO/IEC 14882:2003(E))*, which documents the currently supported C++ standard. For more information, see Chapter 14, “Migrating to the currently supported Standard C++,” on page 87.

Application of #pragma unroll()

As of z/OS V1R7 XL C/C++, `#pragma unroll()` works only with `for` loops.

If your code specifies `#pragma unroll()` prior to a `while` or a `do` loop, the compiler ignores the pragma directive and generates a warning message.

For detailed information about unrolling loops, refer to:

- *z/OS XL C/C++ Language Reference*
- *z/OS XL C/C++ Programming Guide*
- *z/OS XL C/C++ User's Guide*

Chapter 16. Changes that affect c89 invocation

As of z/OS V1R6, the –g flag option is no longer translated to the TEST compiler option.

Note: The TEST and GONUMBER options remain unchanged, but work only with 32-bit compiles.

A new environment variable _DEBUG_FORMAT has been introduced to enable users to request the old translation of the -g flag option for 32-bit compiles:

- If _DEBUG_FORMAT equals DWARF (the default), -g is translated to DEBUG(FORMAT(DWARF)).
- If _DEBUG_FORMAT equals ISD, then -g is translated to TEST (the old translation).

For non-DLL C++ compiles, a dummy definition side file will be allocated to prevent the binder from issuing a warning message. If you do want the binder to issue a warning message, when an export symbol is encountered, specify the DLL=NO option for the link-editing phase.

For more information, see the c89 utility information in *z/OS XL C/C++ User's Guide*.

Chapter 17. Compiler changes

Compiler options

Compiler options with default setting changes

None for this release.

New compiler option that may affect existing programs

None for this release.

Compiler options that are no longer supported

None for this release.

CMDOPTS compiler option and conflict resolution

As of z/OS V1R7:

- Default options specified in the configuration file have the same weight as if they were specified on the command line. The C/C++ compiler cannot distinguish between an option specified in the configuration file and an option specified on the command line.
- Any conflict between options and pragmas is resolved in favor of the option.
- The C/C++ compiler no longer requires that default options be specified in the configuration file.

If you customize your xlc configuration file using the sample default configuration file shipped in z/OS V1R7 XL C/C++ compiler, you might experience a change in behavior because the defaults for supported `xlc` commands are no longer specified on the options attribute in the configuration file. Instead, the xlc utility emits the defaults as suboptions of the CMDOPTS compiler option. This may cause a change in behavior because the z/OS V1R7 XL C/C++ compiler resolves conflicts between options and pragmas differently, depending on whether options are specified as suboptions of the CMDOPTS option or explicitly on the command line and in the options attributes.

TARGET compiler option

The `TARGET` compiler option allows you to compile an application using the current compiler, and then link and run the application on a lower level system. From release to release, the compiler supports a changing list of `TARGET` suboptions. As newer releases are added to the list, older releases are removed. Because of enhancements and optimizations added to the compiler in a new release, the same `TARGET` setting may not generate exactly the same code from one release to the next.

In z/OS V1R6, the following release suboptions were supported: zOSV1R2, zOSV1R3, zOSV1R4, zOSV1R5, and zOSV1R6.

As of z/OS V1R7, targeting z/OS V1R3 and earlier releases is no longer supported. The earliest release that can be targeted is zOSV1R4.

Compiler messages and return codes

From release to release, message contents often change and severity levels may also change (for example, an error becoming a warning). These changes to the severity level may affect the return code of the compilation.

You must update any application that is affected by message contents or return codes. **Do not build dependencies on message content, message numbers, or return codes**. See *z/OS XL C/C++ Messages* for a list of compiler messages.

Compiler listings

Do not build dependencies on the structure or content of listings. For information about C listings or the C++ listings for the current release, refer to *z/OS XL C/C++ User's Guide*.

64-bit compiles and line number information

64-bit compiles do not support the `GONUMBER` compiler option and line number information is not available within 64-bit compiled objects. The Language Environment traceback tool and Language Environment dump services also do not produce line number information in the traceback for 64-bit.

Note: For 32-bit compiles, line number information is still generated and available for use with debugging.

Chapter 18. Compiler invocations

As of z/OS V1R6, compiler invocation is supported by two different utilities:

- c89
- xlc

z/OS V1R6 introduced the following commands:

- `xlc` command to compile a C program
- `xlC` and `xlc++` commands to compile a C++ program

z/OS V1R6 introduced the following command suffixes:

- `_x` suffix to compile the program with XPLINK
- `_64` suffix to compile the program under LP64

The utility you want to use depends on:

- Whether you need to port code between z/OS and AIX.
- How you want to set up your build environment.

For example, use the command `c89_x` to compile an ANSI-compliant program with XPLINK.

Note: For information about how to use these commands and suffixes, see *z/OS XL C/C++ User's Guide*.

Table 18. Differences between the x89 and xlc compiler invocation utilities

	c89 utility	**xlc utility**
Command support	No support for AIX options syntax	The `cc`, `c89`, `cxx`, and `c++` commands accept AIX C/C++ as well as z/OS C/C++ options syntax.
Environment setup	Determined by configuration file	Determined by environment variables

Changes that affect c89 invocation

As of z/OS V1R6, the `–g` flag option is no longer translated to the `TEST` compiler option.

Note: The `TEST` and `GONUMBER` options remain unchanged, but work only with 32-bit compiles.

A new environment variable `_DEBUG_FORMAT` has been introduced to enable users to request the old translation of the `-g` flag option for 32-bit compiles:

- If `_DEBUG_FORMAT` equals `DWARF` (the default), `-g` is translated to `DEBUG(FORMAT(DWARF))`.
- If `_DEBUG_FORMAT` equals `ISD`, then `-g` is translated to `TEST` (the old translation).

For non-DLL C++ compiles, a dummy definition side file will be allocated to prevent the binder from issuing a warning message. If you do want the binder to issue a warning message, when an export symbol is encountered, specify the `DLL=NO` option for the link-editing phase.

For more information, see the c89 utility information in *z/OS XL C/C++ User's Guide*.

Changes that affect xlc invocation

If you customize your xlc configuration file using the sample default configuration file shipped in z/OS V1R7 XL C/C++ compiler, you might experience a change in behavior because the defaults for supported xlc commands are no longer specified on the options attribute in the configuration file. Instead, the xlc utility emits the defaults as suboptions of the CMDOPTS compiler option. This may cause a change in behavior because the z/OS V1R7 XL C/C++ compiler resolves conflicts between options and pragmas differently, depending on whether options are specified as suboptions of the CMDOPTS option or explicitly on the command line and in the options attributes.

Chapter 19. Changes that affect user JCL

CBCI and CBCXI procedures

As of z/OS V1R5, the CBCI and CBCXI procedures contain the variable CLBPRFX. If you have any JCL that uses these procedures, you must either customize these procedures (for example, at installation time) or modify your JCL to provide a value for CLBPRFX.

Chapter 20. Language Environment changes

Changes to enum types in system header files

As of z/OS V1R7 XL C/C++, selected enumerated (enum) type declarations in system header files are protected to avoid potential execution errors. This allows you to specify the ENUMSIZE() compiler option with a value other than SMALL without risking incorrect mapping of enum data types (for example, if they were used inside of a structure).

With earlier versions of the compiler, if you specified ENUMSIZE() with a value other than SMALL, data that was declared with certain enum types could be incorrectly mapped. In some instances, the header files in the library referenced the types (such as `__device_t` in the typedef `fldata_t`), which resulted in a potential inconsistency between the mapping seen during application execution and that declared in the library (which is built with the default ENUMSIZE(SMALL)).

Even when you specify ENUMSIZE() with a value other than SMALL, the enumerations listed in Table 19will always be ENUMSIZE(SMALL).

Table 19. Protected enumeration type declatations

Header file	Enumerations
stdio.h	__device_t
search.h	ACTION VISIT
sys/uio.h	uio_rw
sys/wait.h	idtype_t
_Ccsid.h	__csType
__ledebug.h	asfAmodeType asfCallbackResult
yvals.h	_Mux

Changes to putenv()

As of z/OS V1R5, the C/C++ function `putenv()` changed to place the string passed to `putenv()` directly into the array of environment variables. This behavior assures compliance with the POSIX standard. Before the change, the storage used to define the environment variable passed into `putenv()` was not added to the array of environment variables. Instead, the system copied the string into system allocated storage. To restore the previous behavior of `putenv()`, set environment variable `_EDC_PUTENV_COPY` to `YES`.

This change was implemented in z/OS V1R2 with APAR PQ61928 applied. If you have this APAR installed on your system, the change is already valid.

For additional information on `putenv()` and `_EDC_PUTENV_COPY`, see *z/OS XL C/C++ Run-Time Library Reference*. You may also refer to *z/OS XL C/C++ Programming Guide*, for information on `putenv()` and `_EDC_PUTENV_COPY`.

Base locale default currency change

Before z/OS V1R6, the default currency for EEC was set to local currency in the LC_MONETARY category of the locale. If the user wanted to set Euro as currency, the @euro locales would need to be set using `setlocale()`.

As of z/OS V1R6 the LC_MONETARY information in the base locale is now set to use the Euro. If you set the base locale, you will now have the Euro as the default currency. If you want your applications to continue using the old (local) currency, you will now need to issue `setlocale()` with the new @preeuro locale as the parameter.

Behavior of the current @euro locales has not changed.

Movement of LOCALDEF utilities

As of z/OS V1R6, the following LOCALDEF utilities have been moved to new data sets.

Utility	From C/C++ Data Set	To Language Environment Data Set
LOCALDEF	CBC.SCCNUTL	CEE.SCEECLST
EDCLDEF	CBC.SCCNPRC	CEE.SCEEPROC
EDCXLDEF	CEE.SCCNPRC	CEE.SCEEPROC
CCNELDEF	CBC.SCCNCMP	CEE.SCEERUN2
CCNLMSGS	CBC.SCCNCMP	CEE.SCEERUN2

If you use the MVS batch or TSO localedef (LOCALDEF) utility interfaces, you may need to do the following:

- Add or replace the Language Environment procedures library (CEE.SCEEPROC) where you currently have the C/C++ procedures library (CBC.SCCNPRC).
- Add or replace the Language Environment clist/exec library (CEE.SCEECLST) where you currently have the C/C++ clist/exec library (CBC.SCCNUTL). In addition, you may need to customize the Language Environment customization member (CEE.SCEECLST(CEE.CEL4CUST)) in addition to customizing the C/C++ customization member (CBC.SCCNUTL(CBC.CCNCCUST)).
- Add the Language Environment library CEE.SCEERUN2 (in addition to CEE.SCEERUN) where you currently have the C/C++ library CBC.SCCNCMP.

_OPEN_SYS_SOCK_IPV6 feature test macro

As of z/OS V1R7, a recompile using the `_OPEN_SYS_SOCK_IPV6` feature test macro will expose new definitions in <netinet/ip6.h> and <netinet/icmp6.h>, and the following new functions in <netinet/in.h>:

inet6_opt_append()
inet6_opt_find()
inet6_opt_finish()
inet6_opt_get_val()
inet6_opt_init()
inet6_opt_next()
inet6_opt_set_val()
inet6_rth_add()
inet6_rth_getaddr()
inet6_rth_init()
inet6_rth_reverse()
inet6_rth_segments()
inet6_rth_space()

C99 with both LANGLVL(LONGLONG) and LANGLVL(EXTENDED)

C99 incorporates the `long long` data type as standard. Applications using `long long` support and recompiling at z/OS V1R7 may experience problems when:

- Using a compiler designed to support C99
- Not asking for extended features

If an application currently uses the `LANGLVL(LONGLONG)` compiler option to get at the `long long` data type, and also uses certain non-standard `long long` macros, recompiling at z/OS V1R7 may cause compiler error messages to be issued, since these nonstandard definitions are hidden with this combination of compiler and `LANGLVL` option.

If an application currently uses `LANGLVL(EXTENDED)`, then the nonstandard definitions will continue to be exposed since extended features are requested. For those applications that want to use a compiler designed to support C99, but do not want extended features, change the source code to use the C99 standard `long long` macros, as shown in Table 20.

Table 20. C99 standard macros to replace non-standard long long macros that cause z/OS V1R7 errors

Non-standard `long long` macros	**C99 standard `long long` macros**
LONGLONG_MIN	LLONG_MIN
LONGLONG_MAX	LLONG_MAX
ULONGLONG_MAX	ULLONG_MAX

The definitions in Table 20 are commonly used with the following functions:

- `llabs()`
- the following `long long` numeric conversion functions
  - `strtoll()`
  - `strtoull()`
  - `wcstoll()`
  - `wcstoull()`

Note: If you use the `TARGET` compiler option with any suboption other than the zOSV1R7 suboption, you cannot use the new C99 standard macros.

Floating point support

There are changes in hexadecimal floating point notation and floating point special values for C99.

Hexadecimal floating point notation

Changes in support of hexadecimal floating point notation in the numeric conversion functions introduced in *Programming languages - C (ISO/IEC 9899:1999)* can alter the behavior of well-formed applications that comply with the *Programming languages - C (ISO/IEC 9899:1990)* standard and earlier versions of the base documents. One such example would be:

```
int what_kind_of_number (char *s){
    char *endp; *EXP = "p+0"
    double d;
    long l;
```

```
    d = strtod(s,&endp);
    if (s != endp && *endp == `\0')
        printf("It is a float with value %g\n", d);
    else{
        l = strtol(s,&endp,0);
        if (s != endp && (strcmp(endp,EXP)== 0))
            printf("It is an integer with value %ld\n", l);
        else
            return 1;
    }
    return 0;
}
```

If the function is called with: `what_kind_of_number ("0xAp+0")`, a ISO/IEC 9899:1990 standard-compliant library will result in the function printing: `It is an integer with value 10`. As of *Programming languages - C (ISO/IEC 9899:1999)*, which documents the C99 standard, the result is: `It is a float with value 10`. The change in behavior is due to the inclusion of floating-point numbers in hexadecimal notation without requiring that either a decimal point or the binary exponent be present.

Floating point special values

The numeric conversion functions accept the following special values at all times:

- ±inf or ±INF
- ±nanq or ±nanq(n-char-sequence), and ±NANQ or ±NANQ(n-char-sequence)
- ±nans or ±nans(n-char-sequence), and ±NANS or ±NANS(n-char-sequence)
- ±nan or ±nan(n-char-sequence), and ±NAN or ±NAN(n-char-sequence)

Restriction:

- The z/OS XL C/C++ compiler and z/OS Language Environment C/C++ Run-Time Library do not include `_Imaginary` or formal support of the IEC 60559 floating point as described in Annex F and Annex G of the C99 standard.

Chapter 21. Class library changes

If you are using class libraries, this chapter describes the changes that you may have to make if you are migrating from a previous release of z/OS C/C++ to z/OS V1R7 XL C/C++. Also refer to Chapter 14, “Migrating to the currently supported Standard C++,” on page 87 for details on ISO Standard C++ support.

Removal of IBM Open Class Library

As of z/OS V1R5, development with the IBM Open Class Library (IOC) is not supported. You can no longer compile and link applications that use IOC classes. This includes all the clases, templates, and facilities that are described in *IBM Open Class Library Reference* with the two exceptions noted below. Run-time support is provided for existing applications that use IOC, but this support will be removed in a future release.

The following classes are still supported for application development:

- UNIX System Laboratories (USL) I/O Stream Library
- USL Complex Mathematics Library

As of z/OS V1R5, the name of the element that provides this application development support has changed from IBM Open Class Library to Run-Time Library Extensions. The directory path for the header file has changed from `/usr/lpp/ioclib` to `/usr/lpp/cbclib`.

Although support for these classes is not being removed at this time, it is recommended that you migrate to the Standard C++ iostream and complex classes. This is especially important if you are migrating other IOC streaming classes to Standard C++ Library streaming classes, because combining USL and Standard C++ Library streams in one application is not recommended. For more information about these classes, see *C/C++ Legacy Class Libraries Reference*.

For information about migrating away from these classes, see *IBM Open Class Library Transition Guide*.

Migrating from USL I/O Stream Library to Standard C++ I/O Stream Library

The values for some enumerations differ slightly between the USL and Standard C++ I/O Stream libraries. This may cause problems when migrating to the Standard C++ I/O Stream Library.

The following flags have been added:

- flags for controlling formatting: `boolalpha`, `adjustfield`, `basefield`, `floatfield`

The following flags have been removed:

- flags for controlling formatting: `stdio`
- flags for controlling the open mode: `nocreate`, `noreplace`, `bin`
- flags for controlling the io state: `hardfail`

There may be other small differences.

Mixing the C++ Standard I/O Stream Library, USL I/O Stream Library, and C I/O

While it is possible to mix the Standard C++ I/O Stream Library, the USL I/O Stream Library, and C I/O, it is not recommended. The USL I/O Stream Library uses a separate buffer so you would need to flush the buffer after each call to `cout` by either setting `ios::unitbuf` or calling `sync_with_stdio()`. You should avoid switching between the I/O Stream Library formatted extraction functions and C stdio library functions whenever possible, and you should also avoid switching between versions of the I/O Stream Libraries. For more information, see *z/OS XL C/C++ Programming Guide* and *C/C++ Legacy Class Libraries Reference*.

Part 7. Appendixes

Appendix. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

`www.ibm.com/servers/eserver/zseries/zos/bkserv/`

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

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

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This publication documents *intended* Programming Interfaces that allow the customer to write z/OS or z/OS.e XL C/C++ programs.

Trademarks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries or both:

AD/Cycle	AIX	C/370
C/MVS	C++/MVS	CICS
IBM	IMS	Language Environment
MVS	MVS/ESA	Open Class
OS/390	S/370	S/390
SAA	SOM	SP
System Object Model	z/OS	

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

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

Bibliography

This bibliography lists the publications for IBM products that are related to the z/OS XL C/C++ product. 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*, SA22-7500, for a complete list of publications belonging to the z/OS product.

Related publications not listed in this section can be found on the *IBM Online Library Omnibus Edition MVS Collection*, SK2T-0710, the *z/OS Collection*, SK3T-4269, or on a tape available with z/OS.

z/OS

- *z/OS Introduction and Release Guide*, GA22-7502
- *z/OS and z/OS.e Planning for Installation*, GA22-7504
- *z/OS Summary of Message and Interface Changes*, SA22-7505
- *z/OS Information Roadmap*, SA22-7500
- *z/OS Licensed Program Specifications*, GA22-7503
- *z/OS Migration*, GA22-7499
- *z/OS Program Directory*, GI10-0670

z/OS XL C/C++

- *z/OS XL C/C++ Programming Guide*, SC09-4765
- *z/OS XL C/C++ User's Guide*, SC09-4767
- *z/OS XL C/C++ Language Reference*, SC09-4815
- *z/OS XL C/C++ Messages*, GC09-4819
- *z/OS XL C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C Curses*, SA22-7820
- *z/OS XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer*, GC09-4913
- *IBM Open Class Library Transition Guide*, SC09-4948
- *Standard C++ Library Reference*, SC09-4949

z/OS Run-Time Library Extensions

- *C/C++ Legacy Class Libraries Reference*, SC09-7652
- *z/OS Common Debug Architecture User's Guide*, SC09-7653
- *z/OS Common Debug Architecture Library Reference*, SC09-7654
- *DWARF/ELF Extensions Library Reference*, SC09-7655

Debug Tool

- *Debug Tool* documentation, which is available at: www.ibm.com/software/awdtools/debugtool/library/

z/OS Language Environment

- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Debugging Guide*, GA22-7560
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Programming Reference*, SA22-7562
- *z/OS Language Environment Run-Time Application Migration Guide*, GA22-7565
- *z/OS Language Environment Writing Interlanguage Communication Applications*, SA22-7563
- *z/OS Language Environment Run-Time Messages*, SA22-7566

Assembler

- *HLASM Language Reference*, SC26-4940
- *HLASM Programmer's Guide*, SC26-4941

COBOL

- *COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*, GC26-4764
- *COBOL for OS/390 & VM Programming Guide*, SC26-9049
- *COBOL for OS/390 & VM Language Reference*, SC26-9046
- *COBOL for OS/390 & VM Diagnosis Guide*, GC26-9047
- *COBOL for OS/390 & VM Licensed Program Specifications*, GC26-9044
- *COBOL for OS/390 & VM Customization under OS/390*, GC26-9045
- *COBOL Millenium Language Extensions Guide*, GC26-9266

PL/I

- *VisualAge PL/I Language Reference*, SC26-9476
- *PL/I for MVS & VM Language Reference*, SC26-3114
- *PL/I for MVS & VM Programming Guide*, SC26-3113
- *PL/I for MVS & VM Compiler and Run-Time Migration Guide*, SC26-3118

VS FORTRAN

- *Language and Library Reference*, SC26-4221
- *Programming Guide*, SC26-4222

CICS Transaction Server for z/OS

- *CICS Application Programming Guide*, SC34-6231
- *CICS Application Programming Reference*, SC34-6232
- *CICS Distributed Transaction Programming Guide*, SC34-6236
- *CICS Front End Programming Interface User's Guide*, SC34-6234
- *CICS Messages and Codes*, GC34-6241
- *CICS Resource Definition Guide*, SC34-6228
- *CICS System Definition Guide*, SC34-6226
- *CICS System Programming Reference*, SC34-6233
- *CICS User's Handbook*, SC34-6240

- *CICS Family: Client/Server Programming*, SC33-1435
- *CICS Transaction Server for z/OS Migration from CICS/ESA Version 4.1*, GC34-6219
- *CICS Transaction Server for z/OS Release Guide*, GC34-6218
- *CICS Transaction Server for z/OS Installation Guide*, GC34-6224

DB2

- *DB2 Administration Guide*, SC18-7413
- *DB2 Application Programming and SQL Guide*, SC18-7415
- *DB2 ODBC Guide and Reference*, SC18-7423
- *DB2 Command Reference*, SC18-7416
- *DB2 Data Sharing: Planning and Administration*, SC18-7417
- *DB2 Installation Guide*, GC18-7418
- *DB2 Messages and Codes*, GC18-7422
- *DB2 Reference for Remote DRDA Requesters and Servers*, SC18-7424
- *DB2 SQL Reference*, SC18-7426
- *DB2 Utility Guide and Reference*, SC18-7427

IMS/ESA®

- *IMS Version 8: Application Programming: Design Guide*, SC27-1287
- *IMS Version 8: Application Programming: Transaction Manager*, SC27-1289
- *IMS Version 8: Application Programming: Database Manager*, SC27-1286
- *IMS Version 8: Application Programming: EXEC DLI Commands for CICS and IMS Version 8:*, SC27-1288

MVS

- *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643
- *z/OS MVS Program Management: Advanced Facilities*, SA22-7644

QMF

- *Introducing QMF*, GC26-9576
- *Using QMF*, SC26-9578
- *Developing QMF Applications*, SC26-9579
- *Reference*, SC26-9577
- *Installing and Managing QMF on MVS*, SC26-9575
- *Messages and Codes*, SC26-9580

DFSMS

- *z/OS DFSMS Introduction*, SC26-7397
- *z/OS DFSMS Managing Catalogs*, SC26-7409
- *z/OS DFSMS Using Data Sets*, SC26-7410
- *z/OS DFSMS Macro Instructions for Data Sets*, SC26-7408
- *z/OS DFSMS Access Method Services for Catalogs*, SC26-7394

INDEX

Special characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Program Number: 5694–A01 and 5655–G52

Printed in the United States of America

GC09-4913-03