

z/OS



C/C++ Compiler and Run-Time Migration Guide

z/OS



C/C++ Compiler and Run-Time Migration Guide

Note!

Before using this information and the product it supports, be sure to read the information in "Notices" on page 81.

First Edition (March 2001)

This edition applies to Version 1 Release 1 Modification 0 of z/OS C/C++ (5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces SC09-2359-05. 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.

Technical changes in the text since the last release of this book are indicated by a vertical line (|) to the left of the change.

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 <http://www.ibm.com/servers/eserver/zseries/zos/bkserv>

IBM welcomes your comments. You can send your comments in any one of the following methods:

- Electronically to the network ID listed below. Be sure to include your entire network address if you want a reply.

Internet: torrcf@ca.ibm.com
IBMLink: [toribm\(torrcf\)](mailto:toribm(torrcf)@ca.ibm.com)

- By FAX, use the following number:

United States and Canada: (416) 448-6161
Other countries: (+1) 416-448-6161

- By mail, to the following address:

IBM Canada Ltd. Laboratory
Information Development
2G/KB7/1150/TOR
1150 Eglinton Avenue East
Toronto, Ontario, Canada M3C 1H7

If you send comments, 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.

© Copyright International Business Machines Corporation 1996, 2001. All rights reserved.

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

Contents

Part 1. Introduction	1
Chapter 1. Locating your Migration Path	3
How This Book Is Organized	3
A History of Compilers and Libraries	4
Chapter 2. Common Questions about Migration	7
Will Existing Language Environment Applications Run with z/OS Language Environment V1R1?	7
Will Existing C/370 Applications Work with z/OS Language Environment V1R1?	7
My Application Does Not Run — Now What?	8
I Attempt to Recompile My Application and It Fails — Why?	9
<hr/>	
Part 2. From C/370 to z/OS V1R1 C/C++	11
Chapter 3. Application Executable Program Compatibility	13
Input and Output Operations	13
Differences Between the C/370 V1 and V2 Compilers	13
Executable Programs That Invoke Debug Tool or dbx.	13
System Programming C Facility (SPC) Executable Programs	13
Executable Programs with Interlanguage Calls	14
Initialization Compatibility	15
IBM C/370 Version 1 and Version 2 Initialization.	15
z/OS Language Environment Initialization	15
z/OS Language Environment Initialization of C/370 Executable Programs	15
Special Considerations: CEEBLI1A and IBMBLI1A	15
Converting Old Executable Programs to New Executable Programs	16
Considerations for Interlanguage Call (ILC) Applications.	17
Chapter 4. Source Program Compatibility	19
Input and Output Operations	19
Differences Between the C/370 V1 and V2 Compilers	19
SIGFPE Exceptions	19
Program Mask Manipulations.	20
The release() Function	20
The realloc() Function	21
Fetched Main Programs	21
User Exits.	21
#line Directive	21
sizeof Operator.	21
System Programming C Applications Built with EDCXSTRX	22
The __librel() Function	22
Library Messages	22
Prefix of perror() and strerror() Messages	22
Compiler Messages and Return Codes	22
_Packed Structures and Unions.	23
Alternate Code Points	23
Chapter 5. C/370 V1 to C/370 V2 Compiler Changes	25
Source Code Incompatibilities	25
Characters	25
The #pragma comment Directive	25

Structure Declarations	25
Function Argument Compatibility	26
Pointer Considerations	26
Macro Changes	27
Chapter 6. Other Migration Considerations	29
Changes That Affect User JCL, CLISTS, and EXECs	29
Return Codes and Messages	29
Changes in Data Set Names	29
Differences in Standard Streams	29
Passing Command-Line Parameters to a Program	30
SYSMSGs ddname	30
Run-Time Options	30
Ending the Run-Time Options List	30
ISASIZE, ISAINC, STAE/SPIE, LANGUAGE, and REPORT options	30
STACK Default Size	30
STACK parameters	31
HEAP Default Size	31
HEAP Parameters	31
Compile-Time Options	31
DECK Compile-Time Option	31
INLINE Compile-Time Option	31
OPTIMIZE Compile-Time Option	32
SEARCH and LSEARCH Compile-Time Option	32
TEST Compile-Time Option	32
Language Environment Run-Time Options	32
Precedence of Language Environment over C/370 for #pragma runopts	32
System Programming C Facility Applications with #pragma runopts	32
Decimal Exceptions	32
Migration and Coexistence Considerations	33
SIGTERM, SIGINT, SIGUSR1, and SIGUSR2 Exceptions	33
Running Different Versions of the Libraries under CICS	33
CICS Abend Codes and Messages	33
CICS Reason Codes	33
Standard Stream Support under CICS	34
stderr Output under CICS	34
Transient Data Queue Names under CICS	34
HEAP Option Used with the Interface to CICS	34
COBOL Library Routines	35
Passing Control to the Cross System Product	35
Syntax for the CC Command	36
atexit List during abort()	36
Time Functions	36
Direction of Compiler Messages to stderr	36
Compiler Listings	37
Chapter 7. Input and Output Operations Compatibility	39
Opening Files	39
Writing to Files	39
Repositioning within Files	41
Closing and Reopening ASA Files	42
fldata() Return Values	43
Error Handling	43
Miscellaneous	44
VSAM I/O Changes	44
Terminal I/O Changes	44

Part 3. From Pre-OS/390 Releases of C/C++ to z/OS V1R1 C/C++ 47

Chapter 8. Application Executable Program Compatibility	49
Input and Output Operations	49
System Programming C Facility (SPC) Executable Programs	49
Using the LINK Macro to Initiate a main()	49
Inheritance of Run-Time Options with EXEC CICS LINK.	50
STAE/NOSPIE and SPIE/NOSTAE Mapping	50
Class Library Execution Incompatibilities	50
Chapter 9. Source Program Compatibility	51
Input and Output Operations	51
SIGFPE Exceptions	51
Program Mask Manipulations.	51
#line Directive	52
sizeof Operator	52
_Packed Structures and Unions.	53
Alternate Code Points	53
Supporting the ANSI standard	53
LANGVLV(ANSI)	53
Compiler Messages and Return Codes	53
Collection Class Library Source Code Incompatibilities	53
DSECT Utility	54
Chapter 10. Other Migration Considerations	55
Class Library Object Module Incompatibilities.	55
Removal of Database Access Class Library Utility	55
Changes That Affect User JCL, CLISTS, and EXECs	55
CXX Parameter in JCL Procedures	55
SYSMSGs and SYSXMSGs ddnames	55
Compiler Messages and Return Codes	55
Changes in Data Set Names	56
Decimal Exceptions	56
Migration and Coexistence	56
SIGTERM, SIGINT, SIGUSR1, and SIGUSR2 Exceptions	56
Compile-Time Options	56
OPTIMIZE Compile-Time Option	56
IDL Compile-Time Option	57
INLINE Compile-Time Option.	57
SEARCH and LSEARCH Compile-Time Option	57
TEST Compile-Time Option	57
HALT Compile-Time Option	57
Syntax for the CC Command.	58
Time Functions	58
Abnormal Termination Exits	58
Standard Stream Support	59
Direction of Compiler Messages to stderr	59
Array new.	59
Compiler Listings	60
Chapter 11. Input and Output Operations Compatibility	61
Opening Files	61
Writing to Files	61
Repositioning within Files	63
Closing and Reopening ASA Files	64
fdata() Return Values	65

Error Handling	65
Miscellaneous	66
VSAM I/O Changes	66
Terminal I/O Changes	66

Part 4. From OS/390 C/C++ to z/OS V1R1 C/C++ 69

Chapter 12. Changes Between Releases of OS/390 C/C++	71
Compiler	71
Memory Consideration	71
Removal of Model Tool Support	71
Pragma reachable and leaves	71
Reentrant Variables	71
Compiler Options	72
Interprocedural Analysis	72
IPA Object Module Binary Compatibility	72
IPA Link Step Defaults	73
Data Types	73
Floating Point Type to Integer Conversion	73
Long Long Data Type	74
Language Environment	74
Name Conflicts with Run-Time Library Functions	74
Time Functions	76
Direct UCS-2 and UTF-8 Converters	76
Default Option for ABTERMENC Changed to ABEND	76
THREADSTACK Run-Time Option	76
Class Library	76
Removal of SOM [®] Support	76
Removal of Database Access Class Library Utility	76

Part 5. Appendixes. 77

Appendix. Class Library Migration Considerations	79
Notices	81
Programming Interface Information	82
Trademarks	82
Bibliography	85
z/OS	85
z/OS C/C++	85
z/OS Language Environment	85
Assembler	85
COBOL	86
PL/I	86
VS FORTRAN	86
CICS	86
DB2	86
IMS/ESA	87
QMF	87
DFSMS	87
INDEX	89

Part 1. Introduction

This part provides answers to some common migration questions.

Note that throughout this book, the short form of a product's version and release (VxRx) is used. For example, this book refers to *OS/390 Version 2 Release 4 C/C++* as *OS/390 V2R4 C/C++*. In addition, assume that the modification level of any referenced product is 0 (zero) unless specifically indicated. For example, this book refers to *AD/Cycle C/370 Version 1 Release 1 Modification 1* as *AD/Cycle C/370 V1R1M1*.

Chapter 1. Locating your Migration Path

This book discusses the implications of migrating applications from each of the compilers and libraries listed in Table 2 on page 4 to the z/OS V1R1 C/C++ product. To find the section of the book that applies to your migration, see “How This Book Is Organized”.

Use this book to help determine what must be done to continue to use existing source code, object code, and load modules, and to be aware of differences in behavior between products that may affect your migration. 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 V1R1 C/C++ compiler and z/OS Language Environment V1R1. This book does not show how to change an existing C program so that it can use C++. For a list of books that provide information about the z/OS V1R1 C/C++ compiler and its class libraries, debugger, and utilities, refer to “Bibliography” on page 85. For a description of some of the differences between C and C++, see *z/OS 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
LE/370 R3	MVS/ESA™ SP V5R1 OpenEdition®, AD/Cycle® C/370™ Language Support Feature
Language Environment® R4	C/C++ Language Feature of MVS/ESA SP V5R2M0
Language Environment R5	C/C++ Language Feature of MVS/ESA SP V5R2M2
C/MVS™ V3R2 compiler	C component of the C/C++ for MVS/ESA V3R2 compiler
C++/MVS V3R2 compiler	C++ component of the C/C++ for MVS/ESA V3R2 compiler
OS/390 V1R1	IBM® C/C++ for MVS™ V3R2 compiler and Language Environment R5

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 following:
 - The IBM C/370 V1 or V2R1 compiler and the IBM C/370 V1 or V2 library
 - The IBM SAA AD/Cycle C/370 V1R2 compiler and the IBM C/370 V2R2 library
- Part 3 describes the considerations for migrating from one of the following compilers, and any release of z/OS Language Environment:
 - The AD/Cycle C/370 compilers
 - The MVS C/C++ V3 compilers
 - The OS/390 V1R1 C/C++ compiler

Introduction

- Part 4 describes the considerations for migrating from one of the following:
 - OS/390 V1R2 C/C++
 - OS/390 V1R3 C/C++
 - OS/390 V2R4 C/C++
 - OS/390 V2R5 C/C++
 - OS/390 V2R6 C/C++
 - OS/390 V2R7 C/C++
 - OS/390 V2R8 C/C++
 - OS/390 V2R9 C/C++

A History of 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. Use this table to help determine which changes described in this book apply to your migration.

Table 2. A History of Compilers and Libraries

Short Name	Product Number	GA Date	Description	Service Status
C/370 V1R1	5688-040	1988	C/370 V1R1 Compiler	end of service
	5688-039	1988	C/370 V1R1 Library	
C/370 V1R2	5688-040	1989	C/370 V1R2 Compiler	end of service
	5688-039	1989	C/370 V1R2 Library	
C/370 V2R1	5688-187	1991	C/370 V2R1 Compiler	
	5688-188	1991	C/370 V2R1 Library	
AD V1R1	5688-216	1991	AD/Cycle C/370 V1R1 Compiler, follow-on to C/370 V2R1 Compiler.	end of service
LE V1R1	5688-198	1991	LE/370 V1R1 Library, first release of Language Environment/370; follow-on to C/370 V2R1 Library.	end of service
LE V1R2	5688-198	1992	LE/370 V1R2 Library	end of service
AD V1R2	5688-216	1994	AD/Cycle C/370 V1R2 Compiler: <ul style="list-style-type: none"> • Runs on either LE V1R3 or C/370 V2R2 • Generates code for either LE V1R3 or C/370 V2R2 	
LE V1R3	5688-198	1994	LE/370 V1R3 Library, also shipped as part of MVS/ESA SP 5.1 OpenEdition AD/Cycle C/370 Language Support Feature.	end of service
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.	end of service
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.	end of service
C/C++/MVS V3R2	5655-121	1995	C/C++ for MVS/ESA V3R2 Compilers	

Table 2. A History of Compilers and Libraries (continued)

Short Name	Product Number	GA Date	Description	Service Status
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.	
OS/390 R1	5645-001	March 1996	OS/390 R1 includes the C/C++ for MVS/ESA V3R2 Compilers and the OS/390 V1R1 Language Environment.	end of service Jan 31, 2001
OS/390 R2	5645-001	Sept 1996	OS/390 R2 C/C++ is the follow-on to OS/390 R1 C/C++, and includes new optimization options to improve the execution-time performance of C code. OS/390 V1R2 Language Environment comes with OS/390 V1R2.	end of service Jan 31, 2001
OS/390 R3	5645-001	March 1997	OS/390 R3 C/C++ is the follow-on to OS/390 R2 C/C++, and includes new optimization options to improve the execution-time performance of C++ code. OS/390 V1R3 Language Environment comes with OS/390 V1R3.	end of service Mar 31, 2001
OS/390 V2R4	5647-A01	Sept 1997	OS/390 V2R4 C/C++ is the follow-on to OS/390 R3 C/C++, and includes performance improvements for DLLs, conversion of character string literals, and support for the Program Management Binder. OS/390 V2R4 Language Environment comes with OS/390 V2R4.	end of service Mar 31, 2001
OS/390 V2R5	5647-A01	March 1998	OS/390 V2R5 C/C++ is functionally equivalent to OS/390 V2R4 C/C++.	end of service Mar 31, 2001
OS/390 V2R6	5647-A01	Sept 1998	OS/390 V2R6 C/C++ is the follow-on to OS/390 V2R4 C/C++. It 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 comes with OS/390 V2R6.	
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 comes with OS/390 V2R7.	
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 comes with OS/390 V2R8.	

Introduction

Table 2. A History of Compilers and Libraries (continued)

Short Name	Product Number	GA Date	Description	Service Status
OS/390 V2R9	5647-A01	March 2000	<p>OS/390 V2R9 C/C++ is the follow-on to OS/390 V2R6 C/C++. It includes the following new compiler options and suboptions:</p> <ul style="list-style-type: none"> • CHECKOUT(CAST) • COMPRESS • CVFT • DIGRAPH (for C) • IGNERRNO • INITAUTO • IPA(OBJONLY) • PHASEID • ROCONST • ROSTRING • STRICT • TARGET suboptions enhancements <p>It also includes the following #pragma directives:</p> <ul style="list-style-type: none"> • leaves • option_override • reachable <p>OS/390 V2R9 Language Environment comes with OS/390 V2R9.</p>	
OS/390 V2R10	5647-A01	Sept 2000	<p>OS/390 V2R10 C/C++ is the follow-on to OS/390 V2R9 C/C++. It includes the following new compiler options and suboptions:</p> <ul style="list-style-type: none"> • COMPACT • GOFF • IPA(LEVEL(2)) • XPLINK <p>and enhancements to the following compiler options and suboptions:</p> <ul style="list-style-type: none"> • SPILL • TARGET <p>It also includes improvements to:</p> <ul style="list-style-type: none"> • #pragma option_override • Packed decimal optimization in C <p>OS/390 V2R10 Language Environment comes with OS/390 V2R10.</p>	
z/OS V1R1	5694-A01	Mar 2001	<p>z/OS V1R1 C/C++ is functionally equivalent to OS/390 V2R10 C/C++.</p>	

|
|

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 Language Environment V1R1?

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

For example, the *z/OS 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();
}
.
.
.
```

The value of the return code from the `remove()` function changed in LE/370 R3. If an LE/370 R2 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 Language Environment V1R1?

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 only run them with z/OS Language Environment after they are relinked. You cannot continue to run them with the C/370 library.

Introduction

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 V2 link-edit stubs so that they tolerate Language Environment. After your application is relinked using the modified C/370 V2 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 14 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 15 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 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 C/C++ Run-Time Library Reference*, and the *z/OS C/C++ Programming Guide* for more details about the `setenv()` function.

3. Relink your application.

Relinking your application with z/OS Language Environment ensures that you did not link in any non-z/OS Language Environment interfaces. 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 14 for more information.

- Is an SPC application that uses the library
 - Contains calls to `ctest()`
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. Contact your service representative.
If you followed steps 1 to 6, but cannot run your existing load module under z/OS Language Environment, contact your System Programmer to verify 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. If this is not the case, ask your Service Representative to open a Problem Management Record (PMR) against the applicable IBM product. See the APAR member in data set `CBC.SCBCDOC` for information on how to open a PMR.

I Attempt to Recompile My Application and It Fails — Why?

Changes were made between versions and releases of compilers. Several changes were made between C/370 V1 and C/370 V2. In some cases, these changes were made to ensure compliance with C Language standards such as ANSI/ISO. This book describes these changes, and the alterations you may need to make to your code.

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 to z/OS V1R1 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 V1R1 C/C++ product.

Compilers:

- The IBM C/370 V1 compiler, 5688-040
- 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

In this part, z/OS V1R1 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 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 Language Environment V1R1 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 19.

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

Programs that ran with the C/370 V1 or V2R1 library may have to be changed to run with z/OS Language Environment if they have dependencies on any of the input and output behaviors listed in “Chapter 7. Input and Output Operations Compatibility” on page 39.

Differences Between the C/370 V1 and V2 Compilers

If you have programs that were created with C/370 V1, you should be aware of some changes made in C/370 V2 that may affect them. These differences also exist in the z/OS C compiler. See “Chapter 5. C/370 V1 to C/370 V2 Compiler Changes” on page 25 for more information.

Executable Programs That Invoke Debug Tool or dbx

When migrating your application from C/370 to z/OS Language Environment V1R1, 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 16 and in “Considerations for Interlanguage Call (ILC) Applications” on page 17. 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 C/C++ run-time library need to be relinked. You can relink applications 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

From C/370 to z/OS V1R1

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.

If your SPC module has been built with exception handling, 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 to 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.

To 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, obtain and apply 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.

To relink your C/370-COBOL ILC application so that it will only run under z/OS Language Environment, replace the old library objects @ @C2CBL and @ @CBL2C, as described in “Converting Old Executable Programs to New Executable Programs” on page 16 and “Considerations for Interlanguage Call (ILC) Applications” on page 17. 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. To use Fortran and C ILC routines, you must relink all Fortran-C ILC applications containing pre-Language Environment C or Fortran library routines.

The following table outlines when a relink of ILC applications is required, based on languages found in the executable program:

Table 3. Migrations requiring relinking

Language	Relink required
Assembler	No
PL/I	No
Fortran	YES
COBOL	YES *

Notes:

1. * 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 V1R1 C/C++. Otherwise a relink is required.
2. 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 Applications* for more information.

Initialization Compatibility

Both z/OS Language Environment V1R1 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 Language Environment V1R1 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 V1 and V2 libraries.

The following lists describe the initialization schemes:

IBM C/370 Version 1 and Version 2 Initialization

1. Old CEESTART loads IBMBLIIA.
2. IBMBLIIA initializes the Common Library.
3. The Common Library runs C/370-specific initialization.
4. The main program is called.

z/OS Language Environment Initialization

1. The new CEESTART loads CEEBINIT.
2. CEEBINIT initializes z/OS Language Environment.
3. z/OS Language Environment C-specific initialization is run.
4. The main program is called.

z/OS Language Environment Initialization of C/370 Executable Programs

1. Old CEESTART loads CEEBLIIA (as IBMBLIIA).
2. CEEBLIIA (IBMBLIIA) initializes z/OS Language Environment.
3. z/OS Language Environment C-specific initialization is run.
4. The main program is called.

In the third situation listed above, 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, assigned the alias of 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.

From C/370 to z/OS V1R1

To initialize the Common Library environment, ensure that SIBMLINK is concatenated before SCEERUN. To initialize the z/OS Language Environment environment, ensure that SCEERUN is concatenated before SIBMLINK. The version of IBMBLIIA that is found first determines the environment (Language Environment or Common Library) that is initialized.

Converting Old Executable Programs to New Executable Programs

Many sites will 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 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 17 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 those 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 V1R1 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 on page 16 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 C library function ctdli() and if the z/OS 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(CEER00TB)     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(CEER00TB)     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(CEER00TB)     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(CEER00TB)     ALWAYS NEEDED
INCLUDE SYSLIB(@@C2CBL)      REQUIRED FOR C CALLING COBOL
INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS

```

From C/370 to z/OS V1R1

```
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(CEER00TB)    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(CEER00TB)    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(CEER00TB)    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(CEER00TB)    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 V1R1 C/C++ product.

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

Compilers:

- The IBM C/370 V1 compiler, 5688-040
- 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

C/370 V1 modules were created with the C/370 V1 library. C/370 V2 modules were created with the C/370 V2 library.

“Chapter 6. Other Migration Considerations” on page 29 has information on run-time options, which may also affect source code compatibility.

Input and Output Operations

You may have to change programs running with the C/370 V1 or V2R1 library if they have dependencies on any of the input and output behaviors listed in “Chapter 11. Input and Output Operations Compatibility” on page 61.

Differences Between the C/370 V1 and V2 Compilers

If you have programs that were created with the C/370 V1 compiler, you should be aware of some changes made in C/370 V2 that may affect your programs. These differences are also in the z/OS C compiler. See “Chapter 5. C/370 V1 to C/370 V2 Compiler Changes” on page 25 for more information.

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 Language Environment V1R1. If you have old load modules (created with the C/370 V1 or V2R1 library) that accidentally generated decimal overflow conditions, they may behave differently with z/OS Language Environment, by raising unexpected SIGFPE exceptions. You cannot migrate such modules to the new library without altering the source, and they are unsupported.

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. 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 release() Function

With the z/OS C compiler and z/OS Language Environment, you can no longer issue a `release()` call against a fetched COBOL, Fortran, or PL/I module. If you do, `release()` returns a nonzero return code. You can still use `release()` with C modules and non-z/OS Language Environment enabled assembler modules.

If your application fetches and releases PL/I, Fortran, or COBOL modules, you must change your source code, then recompile and link it with z/OS Language Environment, if you are dependent on the `release()` return code.

Although `release()` could be issued against any assembler routines with IBM C/370 Version 1 and Version 2, it cannot be issued against z/OS Language Environment-enabled assembler routines. These routines, known as ASM15 routines, are assembled with z/OS Language Environment assembler prologs. ASM15 routines are coded with the CEEENTRY macro. If any assembler routines are rewritten as ASM15 routines, ensure that the calling code does not issue a `release()` call against them.

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 Version 1 and Version 2, 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.

Fetches Main Programs

C/370 V1 and V2 programs that are fetched must now be recompiled without a main entry point. Under z/OS Language Environment, if you attempt to fetch a main program it will fail.

User Exits

If both `CEEBXITA` and `IBMBXITA` are present in a relinked IBM C/370 Version 1 or Version 2 module, `CEEBXITA` will have precedence over `IBMBXITA`.

#line Directive

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 C compiler does not ignore the `#line` directive.

sizeof Operator

The behavior of `sizeof` when applied to a function return type was changed in the C/C++ MVS V3R2 compiler. For example:

```
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 `WSIZE0F` 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 C/C++ Language Reference*. For more information on the `WSIZE0F` compiler option, see *z/OS C/C++ User's Guide*.

System Programming C 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. Under z/OS Language Environment, it has a field containing a number that represents the library product. IBM C/370 Version 1 and Version 2 libraries are product 0, Language Environment is Product 1, and OS/390 Language Environment or z/OS Language Environment is product 2.

In IBM C/370 V1 and 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 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 the *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 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 EDCxxxxa, where xxxx is a number and the a is either I, E, or S. See *z/OS Language Environment Debugging Guide* 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 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 C/C++ User's Guide* for a list of messages.

_Packed Structures and Unions

With the z/OS 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 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. C/370 V1 to C/370 V2 Compiler Changes

This chapter describes some of the changes made between the C/370 V1 and V2 compilers. These changes also appear in the z/OS C compiler. Read this section if you are migrating programs from C/370 V1.

Source Code Incompatibilities

This section describes the changes you may have to make to your source code when moving from C/370 V1.

Characters

You can no longer assign a char the value ' '. A character must be between the single quotation marks. Under C/370 V1, ' ' was the same as '\0'.

A warning is now issued when the CHECKOUT compile-time option is specified, and more than 4 bytes are assigned to a char or more than 2 bytes are assigned to a wchar_t constant. These restrictions did not apply under C/370 V1.

Sign extension now occurs when the #pragma chars(signed) directive is used. Thus the value of '\xff' is -1 when chars are signed. When a signed char literal is converted to int, the sign extension will occur on the most significant specified byte that is not shifted out. These are changes from C/370 V1. See Table 4 for examples.

Table 4. Sign Extensions

Value of signed char literal	Value of int
'\x80\x00'	(int)0xffff8000
'\x80\x00\x00'	(int)0xff800000
'\x80\x00\x00\x00'	(int)0x80000000
'\x80\x00\x00\x00\x80'	0x00000080

Note: A hexadecimal escape sequence represents one char of data, so '\x123456789' is equivalent to '\x89'.

The #pragma comment Directive

If you are using the #pragma comment directive, you must now enclose the characters specified in double quotation marks. In C/370 V1, the double quotation marks were not required.

Structure Declarations

With the z/OS C compiler, you must declare a struct type before any function calls that contain the struct as one of its parameters. Otherwise, the struct in the function call will be incomplete and the parameter passed must be a pointer to void.

For example, the following program will not compile as desired because struct st in func_call is an incomplete struct. The call of func_call with a pointer to a struct will be an incompatible parameter type with the expected pointer to void.

```
int func_call (struct st *s); /* incomplete struct */

struct st { int x, y, z; };
```

From C/370 to z/OS V1R1

```
int main(void)
{
    struct st *t;
    func_call(t);           /* pointer to struct st but func_call */
                           /* can only accept pointer to void */
    printf("%\n", t->y);
}
```

To solve this problem, add a declaration before the function declaration:

```
struct st;

int func_call (struct st *s);

struct st { int x, y, z; };
```

Note: If you have the following declaration,

```
extern struct S my_struct;
```

the type must be completed before the `->` or `.` operators can be performed on `my_struct`. In C/370 V1, the `struct S` had to be a complete type at the time this declaration was reached.

Function Argument Compatibility

If you compile the following example under C/370 V1, the compiler fails to notice that the argument `func1` is incompatible with the prototype for `func2`. The `func2()` function requires a pointer to a void function with an argument of type `void *`, but an argument of type pointer to void function with an argument of type `int *` is passed instead. The z/OS C compiler will generate an error message in this situation.

```
void func2( void (*)(void *) );

void func1(int *);

main() {
    func2( func1 );
}
```

Pointer Considerations

According to the *ANSI 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 C compilers perform some type checking, such as in assignments, argument passing on function calls, and function return codes.

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

With the C/370 V2, the AD/Cycle C/370, and the C/MVS V3R1 compilers, you could not assign `NULL` to an integer value. For example, the following was not allowed:

```
int i = NULL;
```

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

Macro Changes

In `stdio.h`, the `#define` macro `__VSAM_OPEN_AIX_PATH` (a value for the `__amrc` struct `__last_op` field), was replaced in C/370 V2 by `__VSAM_OPEN_ESDS_PATH` and `__VSAM_OPEN_KSDS_PATH`.

Modules compiled with C/370 V1 work with z/OS Language Environment. However, if you plan to compile your source with the z/OS C compiler, you must first change it to use `__VSAM_OPEN_ESDS_PATH` and `__VSAM_OPEN_KSDS_PATH`.

Chapter 6. Other Migration Considerations

This chapter provides additional considerations on migrating applications to z/OS V1R1 C/C++ that were created with one of the following compilers, and with one of the following libraries.

Compilers:

- The IBM C/370 V1 compiler, 5688-040
- 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 22 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 SYSTEM, 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 (hexadecimal 34). For more information about the abend codes and messages see *z/OS Language Environment Debugging Guide*.

SYSMSGGS ddname

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

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).

Compile-Time Options

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

DECK Compile-Time 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 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.

INLINE Compile-Time 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 C compilers, the default is 100 ACUs.

OPTIMIZE Compile-Time Option

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

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

SEARCH and LSEARCH Compile-Time Option

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. Now the z/OS 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 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 Compile-Time Option

Starting with 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) or OPT(2) is in effect.

Starting with C/C++ MVS V3R2, a restriction applies to the TEST compiler option if you are using the z/OS 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.

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

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 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 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 V1 and 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 R1 and R2. These changes were carried into z/OS Language Environment V1R1. In the C/370 library and LE/370 R1 and R2, the defaults for SIGINT, SIGUSR1, and SIGUSR2 were to ignore the signals. As of LE/370 R3, 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:

ASA	terminal id	transaction id	sp	Time Stamp YYYYMMDDHHMMSS	sp	data
1	4	4	1	14	1	108

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 V1 and 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:

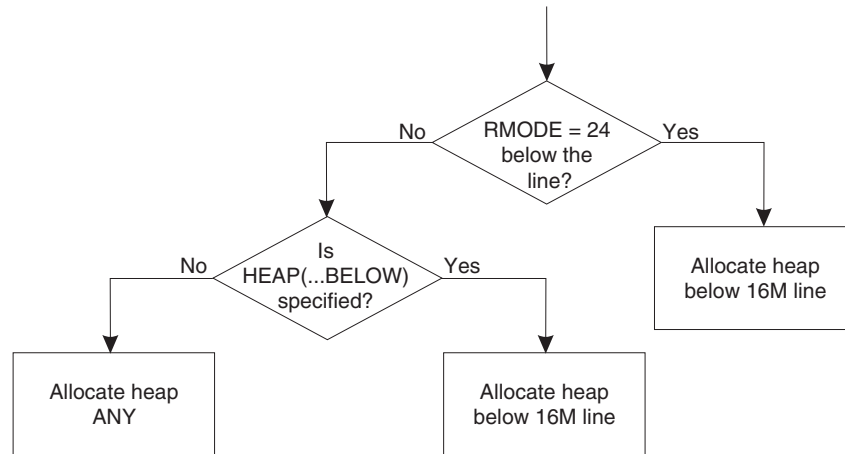


Figure 2. Heap Location Logic

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 Language Environment V1R1 has the potential to invade the user's name space.

For example, z/OS Language Environment-enabled COBOL is available under z/OS Language Environment V1R1, and the following situations are possible:

- If there is a user C function name prefixed with IGZ or ILB that happens to correspond to a COBOL routine, there is the chance of binding in the COBOL routine at link-edit time.
- If there is a fetch() statement for a routine with a name prefixed with IGZ or ILB that happens to correspond to a COBOL routine that is dynamically loaded, it is possible that the COBOL routine will be loaded at run time.

To prevent the first problem, specify the user link libraries ahead of the z/OS Language Environment link libraries.

To prevent the second problem, specify the user execution libraries ahead of the z/OS Language Environment execution libraries.

Passing Control to the Cross System Product

As in IBM C/370 Version 1 and Version 2, control can be passed between Cross System Product (CSP) and z/OS Language Environment in three ways: XFER, DXFER, and CALL.

If you have code that passes control from z/OS Language Environment V1R1 to CSP, which in turn passes control back to z/OS Language Environment V1R1, the behavior is undefined. Code that passes control from CSP to z/OS Language Environment V1R1, which in turn passes control back to CSP, is supported. In summary, z/OS Language Environment V1R1 must appear only once in the chain of passed control.

Syntax for the CC Command

With C/C++ MVS V3R2 and subsequent products, the CC command can be invoked using a new syntax. At customization time, your system programmer can customize the CC EXEC to accept only the old syntax (the one supported by compilers prior to C/MVS 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 only the old syntax, keep in mind that it does not support Hierarchical File System (HFS) files. 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. If you invoke this EXEC with the old syntax, it will not support HFS files.

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

atexit List during abort()

Unlike under IBM C/370 Version 1 and Version 2, the `atexit` list is not driven after a call to `abort()` under z/OS Language Environment.

Time Functions

For z/OS Language Environment, in the absence of customized locale information, the `ctime()`, `localtime()`, and `mktime()` functions return Coordinated Universal Time (UTC).

If you were running with the C/370 V2R2 library, and applied both PTF UN61216 and PTF UN77602, or did not apply either one, the functions will return local time in the absence of customized locale information. Therefore, you will see a change in behavior beginning in OS/390 V2R6 Language Environment.

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 (e.g., 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 C/C++ Programming Guide* for more information about both environment variables and customizing a locale.

Direction of Compiler Messages to stderr

All messages generated by the z/OS C/C++ compiler are sent to `stderr`. In the past, some messages were sent to `stdout`.

Compiler Listings

As of OS/390 C/C++ V2R6, OPT(1) maps to OPT(2). The compiler listing no longer conforms to the format 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 listings for the current release, refer to the *z/OS C/C++ User's Guide*.

Chapter 7. 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 Language Environment V1R1. If your programs performed input and output operations with the following products, you should read the changes listed in this section. These products are:

- The C/370 V1 library
- The C/370 V2R1 library

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 PDSes 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 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.

From C/370 to z/OS V1R1

- 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 <code>fclose()</code> , <code>fopen()</code>
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 <code>fclose()</code> , <code>fopen()</code>
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 5. Appending to ASA Files

abc Written to File, <code>fclose()</code> then append xyz	What You Read from File after <code>fclose()</code> , <code>fopen()</code>	
	Previous release	New release
abc ⇒ xyz	\nabc\nxyz\n	same as previous release

Table 5. 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 ⇒ \xyz	\abc\xyz\n	\abc\n\xyz\n
abc ⇒ \rxyz	\abc\rxyz\n	\abc\n\rxyz\n
abc\n ⇒ xyz	\abc\nxyz\n	same as previous release
abc\n ⇒ \xyz	\abc\nxyz\n	\abc\n\nxyz\n
abc\n ⇒ \rxyz	\abc\nrxyz\n	\abc\n\nrxyz\n
abc\n\n ⇒ xyz	\abc\n\nxyz\n	\abc\n\nxyz\n
abc\n\n ⇒ \xyz	\abc\n\nxyz\n	same as previous release
abc\n\n ⇒ \rxyz	\abc\n\nrxyz\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.

From C/370 to z/OS V1R1

- 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 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 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 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 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 6. Closing and Reopening ASA Files

Written to file	Physical record after close			
		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)	

Table 6. Closing and Reopening ASA Files (continued)

Written to file	Physical record after close					
	Previous behavior			New behavior		
abc\n\n	Char	abc	(1)	Char	abc	(1)
		0	(2)		0	(2)
	Hex	4888	(1)	Hex	4888	(1)
		0123	(2)		0123	(2)
abc\n\n\n	Char	abc	(1)	Char	abc	(1)
		-	(2)		-	(2)
	Hex	4888	(1)	Hex	4888	(1)
		0123	(2)		0123	(2)
abc\r	Char	abc	(1)	same as previous release		
		+	(2)			
	Hex	4888	(1)			
		0123	(2)			
abc\f	Char	abc	(1)	same as previous release		
		1	(2)			
	Hex	4888	(1)			
		0123	(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 C/C++ Programming Guide*.

Error Handling

The general return code for errors is now `E0F`. 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. `E0F` 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. Starting with 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* manual.

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.

From C/370 to z/OS V1R1

- 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.

From C/370 to z/OS V1R1

Part 3. From Pre-OS/390 Releases of C/C++ to z/OS V1R1 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 V1R1 C/C++ product.

Compilers:

- The AD/Cycle C/370 V1R1 compiler, 5688-216
- The AD/Cycle C/370 V1R2 compiler, 5688-216
- The IBM C/C++ for MVS/ESA V3R1 compiler, 5655-121
- The IBM C/C++ for MVS/ESA V3R2 compiler, 5655-121, also known as the IBM OS/390 C/C++ V1R1 compiler, 5645-001
- 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
- Language Environment V1R4, 5688-198
- Language Environment V1R5, 5688-198
- The OpenEdition AD/Cycle C/370 Language Support Feature of MVS/ESA SP V5R1, 5655-068 and 5655-069
- The C/C++ Language Feature of MVS/ESA SP V5R2, 5655-068 and 5655-069
- The C/C++ Language Feature of MVS/ESA SP V5R2, 5655-068 and 5655-069
- OS/390 V1R1 Language Environment, 5645-001

Note: The OS/390 V1R1 compiler and library were equivalent to the final MVS/ESA compiler and library.

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 V1R1 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.

References To These Products	Also Apply To These Products
LE/370 R3	MVS/ESA SP V5R1 OpenEdition AD/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 V5R2 (Modification 2)
OS/390 R1	IBM C/C++ for MVS V3R2 compiler and Language Environment R5

Chapter 8. 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 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 Language Environment V1R1 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 9. Source Program Compatibility” on page 51.

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

Programs running with LE/370 V1R1 or V1R2 may not work without changes if they have dependencies on any of the input and output behavior listed in “Chapter 11. Input and Output Operations Compatibility” on page 61.

System Programming C Facility (SPC) Executable Programs

If you have an LE/370 V1R1 or V1R2 SPC application that was built with exception handling (that is linked with EDCXERR, EDCXABRT and EDCXHDLR), you must relink it with z/OS Language Environment V1R1 using the SCEESPC data set.

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

Using the LINK Macro to Initiate a main()

When the LINK macro was used to initiate one C `main()` from another in LE/370 V1R0, 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. Starting with LE/370 V1R1, and continuing through to z/OS Language Environment V1R1 run-time options are no longer propagated.

With LE/370 V1R0, using LINK to initiate a child `main()` restricted you from using standard streams in the child and from using memory files in the child. Starting with LE/370 V1R1 and continuing through to z/OS Language Environment V1R1, these restrictions no longer apply. Therefore, the parent's standard streams and memory files are shared by the child.

Inheritance of Run-Time Options with EXEC CICS LINK

When an EXEC CICS LINK command was used with LE/370 V1R1, run-time options were inherited from an ancestor. Users who used STACK and HEAP to tune C-CICS applications had to take particular note of this. Because of this inheritance, 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. For programs running under later releases of Language Environment, including z/OS Language Environment V1R1, run-time options are no longer inherited.

STAE/NOSPIE and SPIE/NOSTAE Mapping

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. If the STAE option and SPIE option are used, they map to TRAP(ON,SPIE). If NOSTAE and NOSPIE are used, they map to TRAP(OFF). When the values are mixed, for example, STAE/NOSPIE, they map to TRAP(ON,SPIE). In LE/370 V1R1, SPIE/NOSTAE and STAE/NOSPIE are mapped to TRAP(OFF).

Class Library Execution Incompatibilities

There are execution incompatibilities between the class libraries provided with the C++/MVS™ V3R1M0, V3R1M1, V3R2M0 compilers and the libraries provided with the z/OS C++ compiler. You must recompile and relink applications that are dynamically bound to those class libraries for the following migration paths:

- Collection Class
 - From C++/MVS V3R1M1 or V3R2M0 (C++ version) to z/OS V1R1 C/C++
 - From C++/MVS V3R1M0 to z/OS V1R1 C/C++
- Application Support Class
 - From C++/MVS V3R1M1 or V3R2M0 (C++ version) to z/OS V1R1 C/C++
 - From C++/MVS V3R1.0 to z/OS V1R1 C/C++

Refer to “Appendix. Class Library Migration Considerations” on page 79 for some background information about class libraries and compatibility considerations.

Chapter 9. Source Program Compatibility

In general, you can use source programs with the z/OS V1R1 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/MVS V3R1 or V3R2 compiler running with Language Environment V1R4 or later
- C++/MVS V3R1, and C++/MVS V3R2 programs running with Language Environment V1R4 or later

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

“Chapter 10. Other Migration Considerations” on page 55 has information on run-time options, which may also affect source code compatibility.

Input and Output Operations

You may have to change programs that ran with the LE/370 R1 or R2 library so that they work with z/OS Language Environment, if they have dependencies on any of the input and output behaviors listed in “Chapter 11. Input and Output Operations Compatibility” on page 61.

SIGFPE Exceptions

Decimal overflow conditions were masked in R1 and R2 of LE/370. These conditions were enabled when the packed decimal data type was introduced in the AD/Cycle C/370 R2 compiler, and continue to be enabled with z/OS Language Environment V1R1.

If you have old load modules that accidentally generated decimal overflow conditions, they may behave differently with z/OS Language Environment V1R1 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 R1 or R2 that explicitly manipulated the program mask may require source alteration to execute correctly under z/OS Language Environment V1R1. 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:
 - Modify the assembler code to save the existing mask, set the new value, and when finished, restore the saved mask.

From Pre-OS/390 Releases to z/OS V1R1

- Change the C code so that the produced SIGFPE signal is ignored in the called code. 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 Directive

The AD/Cycle C/370 and C/MVS V3R1 compilers ignored the #line directive when either the EVENTS or the TEST compiler option was in effect. As of C/MVS V3R2, the compiler does not ignore the #line directive.

sizeof Operator

The behavior of sizeof when applied to a function return type was changed in the C/C++ MVS V3R2 compiler. For example:

```
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 OS/390 V2R4 C/C++ and subsequent compilers, you can use #pragma wsizeof or the WSIZE0F 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 the *z/OS C/C++ Language Reference*. For more information on the WSIZE0F compiler option, see the *z/OS C/C++ User's Guide*.

_Packed Structures and Unions

If you are migrating from an AD/Cycle C/370 compiler to the z/OS 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 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).

Supporting the ANSI standard

The C/C++ MVS V3R2 and z/OS C/C++ compilers support the 1992 draft of the ANSI standard, and no longer interpret wide-character constants that contain escape sequences.

LANGLVL(ANSI)

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

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.** See *z/OS C/C++ User's Guide* for a list of compiler messages.

Collection Class Library Source Code Incompatibilities

There are source code incompatibilities between the native Collection Class Libraries available with the C++/MVS V3R1 and z/OS C++ compilers. You must change your source code if you are migrating to z/OS C++ from C++/MVS V3R1 and your application makes use of either of the following:

newCursor method

The return type of the `newCursor` method is now a pointer to the abstract cursor class `ICursor` (`*ICursor`).

Deriving from Reference Classes

Deriving from Reference Classes without overriding existing Collection Class member functions is still possible. However, you can no longer

From Pre-OS/390 Releases to z/OS V1R1

override existing Collection Class functions and use your derived Collection Class in a polymorphic way without additional effort. Refer to the chapter about "Polymorphism and the Collections", in the *OS/390 C/C++ IBM Open Class Library User's Guide* for more information.

These changes were made in the Collection Class Library that was available with the C++/MVS V3R1M1 compiler, and do not affect you if you are migrating from the C++/MVS V3R1M1, C++/MVS V3R2, or any OS/390 C++ compiler.

Also, the structure of the Collection Classes changed in MVS V3R1M1. All classes, including the concrete classes, are now related in an abstract hierarchy. The abstract hierarchy makes use of virtual inheritance. When you subclass from a Collection Class and implement your own copy constructor, you must initialize the virtual base class `IACollection<Element>` in your derived classes. Therefore, if you subclassed from a concrete Collection Class that was shipped with C++/MVS V3R1, and are migrating to the Collection Classes that are shipped with z/OS V1R1 C/C++, you will have to change the implementation of your copy constructor by adding the virtual base class initialization.

Refer to "Appendix. Class Library Migration Considerations" on page 79 for some background information about class libraries and compatibility considerations.

DSECT Utility

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 10. 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 V1R1 C/C++” on page 47 to the z/OS V1R1 C/C++ product.

Class Library Object Module Incompatibilities

There are object incompatibilities between the class libraries provided with the IBM C++/MVS V3R1M0, V3R1M1, V3R2M0 compilers and the libraries provided with the z/OS C++ compilers. You must recompile and relink applications that are dynamically bound to those class libraries, for the following migration paths:

- Collection Class
 - From C++/MVS V3R1M1 or V3R2M0 (C++ version) to z/OS V1R1 C/C++
 - From C++/MVS V3R1M0 to z/OS V1R1 C/C++
- Application Support Class
 - From C++/MVS V3R1M1 or V3R2M0 (C++ version) to z/OS V1R1 C/C++
 - From C++/MVS V3R1M0 to z/OS V1R1 C/C++

Refer to “Appendix. Class Library Migration Considerations” on page 79 for some background information about class libraries and compatibility considerations.

Removal of Database Access Class Library Utility

Starting with 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++/MVS 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 compile-time options
```

SYSMMSGs and SYSXMSGs ddnames

With the C/C++ MVS V3R2 and z/OS 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 SYSMMSGs and SYSXMSGs ddnames, you must now use the NATLANG run-time option. If you specify data sets for these ddnames, they are ignored.

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 warnings, and in very rare

From Pre-OS/390 Releases to z/OS V1R1

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.** See *z/OS C/C++ User's Guide* for a description of compiler messages and return codes.

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.

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 Language Environment V1R1 product.

The differences or incompatibilities are:

- The defaults for the SIGINT, SIGTERM, SIGUSR1, and SIGUSR2 signals changed in LE/370 R3, 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.

Compile-Time Options

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

OPTIMIZE Compile-Time Option

In the AD/Cycle C/370 compilers:

- `OPT(0)` mapped to `N0OPT`
- `OPT` and `OPT(1)` mapped to `OPT(1)`
- `OPT(2)` mapped to `OPT(2)`

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

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

Starting with the OS/390 V2R6 C/C++ compiler:

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

IDL Compile-Time Option

As of OS/390 V2R4 C/C++, 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.

INLINE Compile-Time Option

The defaults for the C compile INLINE compiler option were changed in the C/MVS V3R1 compiler. In the past, the default for the threshold suboption was 250 ACUs (Abstract Code Units). With the C/MVS V3 and the z/OS C compilers, the default is 100 ACUs.

SEARCH and LSEARCH Compile-Time Option

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 searched only the libraries specified for the last LSEARCH option. Now the z/OS C compilers search 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 C/C++ SEARCH option more than once, the z/OS C++ compilers search 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 Compile-Time Option

Starting with 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) or OPT(2) is in effect.

Starting with 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.

HALT Compile-Time Option

The C++/MVS V3R2 and z/OS C++ compilers do not accept 33 as a valid parameter for the HALT compile-time option.

Syntax for the CC Command

With the C/C++ MVS V3R2 and z/OS C/C++ compilers, the CC command can be invoked using a new syntax. 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++ MVS 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 only the old syntax, keep in mind that it does not support Hierarchical File System (HFS) files. 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. If you invoke this EXEC with the old syntax, it does not support HFS files.

Refer to the z/OS Program Directory for more information about installation and customization, and to the *z/OS 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 (e.g., 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 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 an LE formatted dump was generated (which continues to be produced under z/OS Language Environment V1R1).

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 may also modify CEEBDATX and CEECDATX to suppress the dumps. The exits are available in the z/OS Language Environment V1R1 sample library.

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:

ASA	terminal id	transaction id	sp	Time Stamp YYYYMMDDHHMMSS	sp	data
1	4	4	1	14	1	108

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 Language Environment V1R1 from LE/370 V1R1 or V1R2.

Direction of Compiler Messages to `stderr`

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

Array `new`

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

If you are migrating from the base C/C++ MVS V3R1 compiler to z/OS V1R1 C/C++, and you have written your own global `new` operator, it is no longer called when you create an array object. For example:

```
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 listings for the current release, refer to the *z/OS C/C++ User's Guide*.

Chapter 11. Input and Output Operations Compatibility

Changes were made to input and output support in the C/370 V2R2 and LE/370 Release 3 libraries. These changes also apply to z/OS Language Environment V1R1. If your programs performed input and output operations with the following products, you should read the changes listed in this section. These products are:

- 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 PDSes 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 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.

From Pre-OS/390 Releases to z/OS V1R1

- 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 <code>fclose()</code> , <code>fopen()</code>
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 <code>fclose()</code> , <code>fopen()</code>
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 7. Appending to ASA Files

abc Written to File, <code>fclose()</code> then append xyz	What You Read from File after <code>fclose()</code> , <code>fopen()</code>	
	Previous release	New release
abc ⇒ xyz	\nabc\nxyz\n	same as previous release

Table 7. 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 ⇒ \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
abc\n ⇒ \rxyz	\nabc\rxyz\n	\nabc\n\rxyz\n
abc\n\n ⇒ xyz	\nabc\n\nxyz\n	\nabc\n\nxyz\n
abc\n\n ⇒ \nxyz	\nabc\n\nxyz\n	same as previous release
abc\n\n ⇒ \rxyz	\nabc\n\nrxyz\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

From Pre-OS/390 Releases to z/OS V1R1

- 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 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 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 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 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 8. Closing and Reopening ASA Files

Written to file	Physical record after close			
		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)	

Table 8. Closing and Reopening ASA Files (continued)

Written to file	Physical record after close					
	Previous behavior			New behavior		
abc\n\n	Char	abc	(1)	Char	abc	(1)
		0	(2)		0	(2)
	Hex	4888	(1)	Hex	4888	(1)
		0123	(2)		0123	(2)
abc\n\n\n	Char	abc	(1)	Char	abc	(1)
		-	(2)		-	(2)
	Hex	4888	(1)	Hex	4888	(1)
		0123	(2)		0123	(2)
abc\r	Char	abc	(1)	same as previous release		
		+	(2)			
	Hex	4888	(1)			
		0123	(2)			
abc\f	Char	abc	(1)	same as previous release		
		1	(2)			
	Hex	4888	(1)			
		0123	(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 C/C++ Programming Guide*.

Error Handling

The general return code for errors is now `E0F`. 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. `E0F` 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. Starting with 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* manual.

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.

From Pre-OS/390 Releases to z/OS V1R1

- 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.

From Pre-OS/390 Releases to z/OS V1R1

Part 4. From OS/390 C/C++ to z/OS V1R1 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 V1R1 C/C++ product.

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 compiler, 5647-A01
- IBM OS/390 C/C++ V2R6 compiler, 5647-A01
- IBM OS/390 C/C++ V2R9 compiler, 5647-A01

Libraries:

- OS/390 V1R2 Language Environment, 5645-001
- OS/390 V1R3 Language Environment, 5645-001
- OS/390 V2R4 Language Environment, 5647-A01
- OS/390 V2R5 Language Environment, 5647-A01
- OS/390 V2R6 Language Environment, 5647-A01
- OS/390 V2R7 Language Environment, 5647-A01
- OS/390 V2R8 Language Environment, 5647-A01
- OS/390 V2R9 Language Environment, 5647-A01

Notes:

1. The z/OS V1R1 compiler and library are equivalent to the OS/390 V2R10 compiler and library.
2. The OS/390 V1R1 compiler and library were equivalent to the final MVS/ESA compiler and library.

Chapter 12. Changes Between Releases of OS/390 C/C++

This chapter describes the changes you may need to make if you are migrating from a previous release of OS/390 C/C++ to z/OS V1R1 C/C++.

Compiler

Memory Consideration

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.

Pragma reachable and leaves

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 now 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 the related `sigsetjmp`, `siglongjmp`, and so on) 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`, refer to *z/OS C/C++ Language Reference*.

Reentrant Variables

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 using the following JCL procedures with the output directed to a non-PDSE: `EDCCCL`, `EDCCCLG`, `EDCL`, and `EDCLG`.

If the output data set is non-PDSE, you need the prelinker. The following JCL procedures use the prelinker: `EDCCPL`, `EDCCPLG`, `EDCPL`, and `EDCPLG`.

From Previous Releases of OS/390 C/C++ to z/OS V1R1 C/C++

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.

Compiler Options

ARCHITECTURE Option

As of OS/390 V2R10, the default value of the ARCHITECTURE compiler option is 2. In previous releases, it was 0.

Removal of SOM-Related Options

The following SOM-related compiler options are no longer supported:

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

Removal of IDL Compile-Time Option

As of OS/390 V2R4 C/C++, 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.

OPTIMIZE Compile-Time Option

In the OS/390 C/C++ V1R2, V1R3, and V2R4 compilers:

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

Starting with the OS/390 V2R6 C/C++ compiler:

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

TARGET (OS) Option

The TARGET option now supports the following operating system levels: OSV1R2, OSV1R3, OSV2R4, OSV2R5, OSV2R6, OSV2R7, OSV2R8, OSV2R9, and OSV2R10. This allows you to compile an application using the current compiler, and then link and run the application on a lower level system. Refer to *z/OS C/C++ User's Guide* for details.

Interprocedural Analysis

IPA Object Module Binary Compatibility

Release-to-release binary compatibility is maintained by the z/OS 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.

From Previous Releases of OS/390 C/C++ to z/OS V1R1 C/C++

- 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. If this is attempted, an error diagnostic message will be issued by the IPA Link.
- Note that if the IPA object is recompiled by a later IPA Compile, additional optimizations may be performed and the resulting application program may perform better.

An exception to this is the IPA object files produced by the OS/390 Release 2 C IPA Compile. These must be recompiled from the program source using an OS/390 V1R3 or later compiler before attempting to process them with the z/OS V1R1 C/C++ IPA Link.

IPA Link Step Defaults

Starting with OS/390 C/C++ 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. 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.

Starting with OS/390 C/C++ V2R6, the default optimization level for the IPA Link step is OPT(2).

Data Types

Floating Point Type to Integer Conversion

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 */
}
```

Long Long Data Type

As of OS/390 V2R9, the C/C++ compiler and Language Environment support long long data types. The `_LONG_LONG` macro is predefined for all language levels other than ANSI.

In previous releases of the compiler, some users defined their own `_LONG_LONG` macro. If you want your code to be portable, you should remove this user-defined macro before compiling a program under V1R1.

Language Environment

Name Conflicts with Run-Time Library Functions

When taking code previously compiled and link-edited on a system below 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. Starting with OS/390 V2R4 many `pragma map`s 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 9 lists the functions that had `pragma map`s deleted in OS/390 V2R4.

Table 9. Functions That Had Pragma Maps Deleted

<code>__loc1()</code>	<code>__atoe()</code>	<code>__atoe_l()</code>	<code>__cnvblk()</code>	<code>__dlight()</code>
<code>__etoa()</code>	<code>__etoa_l()</code>	<code>__gderr()</code>	<code>__getipc()</code>	<code>__ipdbcs()</code>
<code>__ipdspx()</code>	<code>__iphost()</code>	<code>__ipmsgc()</code>	<code>__ipnode()</code>	<code>__iptcpn()</code>
<code>__opargf()</code>	<code>__operrf()</code>	<code>__opindf()</code>	<code>__opoptf()</code>	<code>__sigerr()</code>
<code>__sigign()</code>	<code>__sigpro()</code>	<code>__tzone()</code>	<code>__wsinit()</code>	<code>__longjmp()</code>
<code>_setjmp()</code>	<code>_tolower()</code>	<code>_toupper()</code>	<code>accept()</code>	<code>access()</code>
<code>alarm()</code>	<code>a64l()</code>	<code>basename()</code>	<code>bcmp()</code>	<code>bcopy()</code>
<code>bind()</code>	<code>brk()</code>	<code>bzero()</code>	<code>catclose()</code>	<code>catgets()</code>
<code>catopen()</code>	<code>cclass()</code>	<code>chaudit()</code>	<code>chdir()</code>	<code>chmod()</code>
<code>chown()</code>	<code>chroot()</code>	<code>clearenv()</code>	<code>clearenv()</code>	<code>close()</code>
<code>closedir()</code>	<code>closelog()</code>	<code>clrmemf()</code>	<code>confstr()</code>	<code>connect()</code>
<code>creat()</code>	<code>crypt()</code>	<code>ctdli()</code>	<code>ctdli()</code>	<code>ctermid()</code>
<code>ctermid()</code>	<code>cuserid()</code>	<code>cuserid()</code>	<code>dirname()</code>	<code>drand48()</code>

From Previous Releases of OS/390 C/C++ to z/OS V1R1 C/C++

Table 9. Functions That Had Pragma Maps Deleted (continued)

dup()	dup2()	dynalloc()	dynfree()	ecvt()
encrypt()	endgrent()	endpwent()	erand48()	execl()
execle()	execlp()	execv()	execve()	execvp()
fattach()	fchmod()	fchdir()	fchmod()	fcntl()
fcvt()	fdelrec()	fdetach()	fetch()	fetchep()
ffs()	fileno()	fldata()	flocate()	fntmsg()
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()	getsyntax()	getuid()	getutxid()	getw()
getwd()	glob()	globfree()	grantpt()	hcreate()
hdestroy()	hsearch()	iconv()	index()	insque()
ioctl()	ioctl()	isatty()	isnan()	jrnd48()
kill()	killpg()	lchown()	long48()	lfind()
link()	listen()	lockf()	lrnd48()	lsearch()
lseek()	lstat()	l64a()	maxcoll()	maxdesc()
memccpy()	mkdir()	mkfifo()	mkstemp()	mktemp()
mmap()	mount()	mprotect()	mrnd48()	msgctl()
msgget()	msgrcv()	msgsnd()	msgxrcv()	msync()
munmap()	nftw()	nice()	nlist()	nrnd48()
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()	rexc()	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()
srnd48()	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()

From Previous Releases of OS/390 C/C++ to z/OS V1R1 C/C++

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 (e.g., 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 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_UCS environment variable to "0". Refer to *z/OS C/C++ Programming Guide* for more information about the _ICONV_UCS 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 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.

Class Library

Removal of SOM[®] Support

As of OS/390 V2R10, the IBM System Object Model[™] (SOM) is no longer supported in the C++ compiler and the IBM Open Class[™] Library. The SOM-enabled class library DLLs have been stabilized at the OS/390 V2R9 level and continue to be shipped as a run-time environment only.

Removal of Database Access Class Library Utility

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

Part 5. Appendixes

Appendix. Class Library Migration Considerations

This appendix provides some background information on the types of class libraries that are available with the C++/MVS V3 and z/OS C/C++ compilers.

The following four class libraries are available for use with the z/OS C++ compilers, beginning with the C++/MVS V3R1M0 compiler:

- I/O Stream Class Library
- Complex Mathematics Class Library
- Application Support Class Library
- Collection Class Library

The C++/MVS V3R2 compiler introduced the Database Access Class Library Utility, which was removed in OS/390 V2R4.

The I/O Stream, Complex Mathematics Class, Application Support Class, and Collection Class libraries are offered in native C++ versions only. All libraries are available in both statically bindable and dynamically bindable forms.

In previous releases, the Application Support Class library and the Collection Class library were also offered in SOM versions. Starting with OS/390 V2R10, these SOM-enabled class libraries have been removed. The SOM-enabled class library DLLs have been stabilized at the V2R9 level and continue to be shipped as a run-time environment only.

In native C++ class libraries, references to methods are dependent upon the order of the method entries in a virtual function table. When new methods are added to a library, the order of the methods can change, and therefore existing applications using those methods may no longer work. Between releases or modification levels, migration impacts may also occur if there are changes to the interfaces or semantics of existing functions within a class library.

Whether an application is statically or dynamically bound to a class library will also determine whether or not there are executable incompatibilities. Statically-bound applications do not usually encounter release-to-release executable incompatibilities unless they are recompiled/relinked from source or relinked from objects with the new release. Dynamically-bound applications, however, may encounter release-to-release executable incompatibilities. Source and object incompatibilities may occur regardless of whether an application is statically or dynamically bound.

For more information on the topics mentioned above, refer to the following:

- *OS/390 C/C++ IBM Open Class Library User's Guide*
- *OS/390 C/C++ IBM Open Class Library Reference*
- The "Building and Using Dynamic Link Libraries" chapter in the *z/OS C/C++ Programming Guide* (for information on dynamic linking)
- *z/OS C/C++ User's Guide* (for information on static linking)

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, 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 OR CONDITIONS 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.
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
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 the z/OS 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. 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 IBM's application programming interfaces.

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 C/C++ programs.

Trademarks

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

AD/Cycle	C/370	C/MVS
C++/MVS	CICS	CICS/ESA
IBM	IMS	IMS/ESA
Language Environment	MVS	MVS/ESA
Open Class	OpenEdition	OS/390

S/370
System/370

S/390
System Object Model

SOM
z/OS

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 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 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 *IBM Online Library Omnibus Edition z/OS Collection*, SK2T-6700, or on a tape available with z/OS.

z/OS

- *z/OS Introduction and Release Guide*, GA22-7502
- *z/OS Planning for Installation*, GA22-7504
- *z/OS Summary of Message Changes*, SA22-7505
- *z/OS Information Roadmap*, SA22-7500

z/OS C/C++

- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS C/C++ Language Reference*, SC09-4764
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C Curses*, SA22-7820
- *z/OS C/C++ Compiler and Run-Time Migration Guide*, SC09-4763
- *z/OS C/C++ Reference Summary*, SX09-1319
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364
- *Debug Tool User's Guide and Reference*, SC09-2137

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 Migration Guide*, GA22-7565
- *z/OS Language Environment Writing Interlanguage Applications*, SA22-7563

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
- *Programming Guide*, SC26-9049
- *Language Reference*, SC26-9046
- *Diagnosis Guide*, GC26-9047
- *Licensed Program Specifications*, GC26-9044
- *Installation and Customization under z/OS*, GC26-9045
- *Millenium Language Extensions*, GC26-9266

PL/I

- *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

- *CICS Application Programming Guide*, SC34-5702
- *CICS Application Programming Reference*, SC34-5703
- *CICS Distributed Transaction Programming Guide*, SC34-5708
- *CICS Front End Programming Interface User's Guide*, SC34-5710
- *CICS Messages and Codes*, GC33-5716
- *CICS Resource Definition Guide*, SC34-5722
- *CICS System Definition Guide*, SC34-5725
- *CICS System Programming Reference*, SC34-5726
- *CICS User's Handbook*, SX33-6116
- *CICS Family: Client/Server Programming*, SC34-1435
- *CICS Transaction Server for OS/390 Migration Guide*, GC34-5699
- *CICS Transaction Server for OS/390 Release Guide*, GC34-5701
- *CICS Transaction Server for OS/390: Planning for Installation*, GC34-5700

DB2

- *DB2 Administration Guide*, SC26-8957
- *DB2 Application Programming and SQL Guide*, SC26-8958
- *DB2 Call Level Interface Guide and Reference*, SC26-8959
- *DB2 Command Reference*, SC26-8960
- *DB2 Data Sharing: Planning and Administration*, SC26-8961
- *DB2 Installation Guide*, GC26-8970
- *DB2 Messages and Codes*, GC26-8979
- *DB2 Reference for Remote DRDA Requesters and Servers*, SC26-8964
- *DB2 SQL Reference*, SC26-8966
- *DB2 Utility Guide and Reference*, SC26-8967

IMS/ESA

- *IMS/ESA Application Programming: Design Guide*, SC26-8728
- *IMS/ESA Application Programming: Transaction Manager*, SC26-8729
- *IMS/ESA Application Programming: Database Manager*, SC26-8727
- *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS*, SC26-8726

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*, SC26-7394
- *z/OS DFSMS Program Management*, SC27-1130

INDEX

Special Characters

__librel() 22
#line directive 21, 52
_packed 54
_Packed structures 23, 53
_Packed unions 23, 53
_VSAM_OPEN_AIX_PATH macro 27
_VSAM_OPEN_ESDS_PATH macro 27
_VSAM_OPEN_KSDS_PATH macro 27

A

abnormal termination 33, 56
abort() function 36
ABTERMENC default option 76
ANSI
 LANGLVL(ANSI) 53
 standards 53
ARCHITECTURE compiler option 72
array new 59
ASA files
 closing 40, 62
 closing and reopening 42, 64
 writing to 40, 62
ASM15 routines 20
Assembler interlanguage calls 14
atexit list during abort() 36

C

C/370 V1 to C/370 V2 compiler changes 25
CC command 36, 58
CEEBDATX 58
CEEBLIIA 15
CEEBXITA 21
CEECDATX 58
CEEEV003 22
CEESTART 14, 15
char data type 25
CHECKOUT compile-time option 25
CICS
 abend codes and messages 33
 and versions of C/370 libraries 33
 Application Programmer Interface 34
 reason codes 33
 standard stream support 34, 59
 stderr 34
 transient data queue names 34
 using HEAP option 34
class library incompatibilities
 Application Class
 load module 50
 object module 55
 Collection Class
 load module 50
 object module 55
 source code 53

class library incompatibilities (*continued*)
 IO Stream Class
 load module 50
CLISTs, changes affecting 29, 55
COBOL
 interlanguage calls 14
 library routines 35
code points 23, 53
command-line parameters
 passing to a program 30
 z/OS Language Environment error handling 30
compatibility
 exception handling
 from C/370 V1 or V2 33, 56
 function argument 26
 input/output
 from C/370 V1 or V2 39
 from pre-OS/390 releases 61
 load module
 from C/370 V1 or V2 7, 13
 from pre-OS/390 releases 49
 general information 7
 other considerations
 AD/Cycle C/370 to z/OS V1R1 C 55, 56
 C/370 V1 or V2 compiler to z/OS V1R1 C
 compiler 29, 31
 C/MVS V3R1 to z/OS V1R1 C 56
 from C/370 V1 or V2 29
 from pre-OS/390 releases 55
 NOOPTIMIZE 32, 56, 72
 OPTIMIZE 32, 56, 72
PSW mask
 from C/370 V2R1 33
 from pre-OS/390 releases 56
source program
 C/370 V1 or V2 compiler to z/OS V1R1 C
 compiler 19
 C/370 V1 to C/370 V2 25
 from C/370 V1 19
 from C/370 V2 19
 general information 9
 with AD/Cycle C/370 compiler 51
 with C++/MVS compiler 51
 with C/MVS compiler 51
System Programming C Facility
 C/370 V1 or V2 compiler to z/OS V1R1 C
 compiler 19
 C/370 V1 or V2 to z/OS Language
 Environment 32
compile-time options
 CHECKOUT 25
 DECK 31
 HALT 57
 IDL 57, 72
 INLINE 31, 57
 IPA 72
 LANGLVL(ANSI) 53
 LSEARCH 32, 57

compile-time options (*continued*)

SEARCH 32, 57

TEST 32, 57

conversion overflow 73

CSP (Cross System Product)

CALL 35

DXFR 35

XFER 35

ctest() 13

ctime() 36, 58, 76

D

data types

long long 74

dbx 13

ddnames

SYSERR 29

SYSPRINT 29

SYSTEM 29

Debug Tool 13

decimal overflow exceptions 32, 56

DECK compile-time option 31

DSECT utility 54

dumps 13

E

EDC_COMPAT 8

EDCSTART 14

EDCXV 22

environment variables

_EDC_COMPAT 42, 64

EXECs

CC 36, 58

changes affecting 29, 55

F

fetch() function 20

fetchd main programs 21

fetchep() function 20

fflush() 41, 63

fgetpos() 41, 63

fopen() 61

Fortran interlanguage calls 14

freopen() 61

fseek() 41, 63

function return type 21, 52

H

HALT compile-time option 57

HEAP run-time option

default size 31

parameters 31

with CICS 34

hexadecimal numbers 25

I

IBMBLIIA 15

IBMBXITA 21

IDL compile-time option 57, 72

initialization compatibility 15

INLINE compile-time option 31, 57

input/output

ASA files

closing and reopening 42, 64

closing files 40, 62

writing to files 40, 62

closing and reopening files

ASA files 42, 64

closing files

ASA files 40, 62

compatibility 39, 61

error handling 43, 65

file I/O changes 39, 61

FILENAME_MAX 44, 66

fldata() 43, 65

ftell() encoding 42, 64

L_tmpnam 44, 66

opening files 39, 61

repositioning within files 41, 63

standard streams 44, 66

terminal I/O 44, 66

VSAM I/O 44, 66

writing to files

ASA files 40, 62

other considerations 39, 61

interlanguage calls

Assembler 14

COBOL 14

Fortran 14

PL/I 14

ISAINC run-time option 30

isainc with #pragma runopts 32

ISASIZE run-time option 30

isasize with #pragma runopts 32

J

JCL

changes affecting 29, 55

CXX parameter 55

L

LANGLVL(ANSI) compile-time option 53

LANGUAGE run-time option 30

language with #pragma runopts 32

library functions

__librel() 22

abort() 36

ctest() 13

ctime() 36, 58, 76

fetch() 20

fetchep() 20

fflush() 41, 63

fgetpos() 41, 63

library functions *(continued)*

- fseek() 41, 63
- librel 22
- localtime() 36, 58, 76
- mktime() 36, 58, 76
- realloc() 21
- release() 20
- tmpnam() 44, 66
- ungetc() 41, 63
- librel function 22
- line directive 21, 52
- LINK macro 49
- listings 37, 60
- load modules
 - compatibility
 - from C/370 V1 or V2 13
 - from pre-OS/390 releases 7, 49
 - initialization 15
 - converting old executable programs 16
 - System Programming C Facility 13, 49
- localtime() 36, 58, 76
- long long data type 74
- LSEARCH compile-time option 32, 57

M

macros

- _LONG_LONG 74
- _VSAM_OPEN_AIX_PATH macro 27
- _VSAM_OPEN_ESDS_PATH macro 27
- _VSAM_OPEN_KSDS_PATH macro 27
- LINK 49

memory requirement 71

messages

- contents 29
- differences between C/370 and AD/Cycle C/370 V1R2 29
- differences between C/370 and Language Environment 29
- differences between C/370 and z/OS Language Environment 22
- differences between C/370 and z/OS V1R1 C 22
- differences between compilers 53, 55
- direction of messages to stderr 36, 59
- perror() 22
- prefixes 29
- strerror() 22

mktime() 36, 58, 76

Model Tool 71

N

- new, array version 59
- NOOPTIMIZE compile-time option 32, 56, 72
- NOSPIE run-time option 50
- NOSTAE run-time option 50
- Notices 81
- NULL 26

O

- opening files 61
- OPTIMIZE compile-time option 32, 56, 72
- overflow processing 73

P

- packed 54
- Packed structures 23, 53
- Packed unions 23, 53
- PDS 39, 61
- PDSE 39, 61
- perror() 22
- PL/I interlanguage calls 14
- pointers 26
- pragma
 - chars(signed) 25
 - comment 25
 - leaves 71
 - pack 54
 - reachable 71
 - runopts 32
 - variable 71
 - wsizeof 21, 52
- program mask 20, 51
- PSW mask 20, 52

R

- realloc() function 21
- reentrant variables 71
- region size 71
- release() function 20
- relink requirements
 - ctest() 13
 - interlanguage calls with COBOL 14, 17
 - SPC exception handling 13, 49
- REPORT run-time option 30
- report with #pragma runopts 32
- return codes differences
 - between C/370 and Language Environment 29
 - between C/370 and z/OS V1R1 C 22
 - between compilers 53, 55
- Run-time options
 - ending options list 30
 - HEAP 31
 - ISAINC 30
 - ISASIZE 30
 - LANGUAGE 30
 - NOSPIE 50
 - NOSTAE 50
 - passing to program 30
 - REPORT 30
 - slash (/) 30
 - SPIE 30, 50
 - STACK 30
 - STAE 30, 50
 - THREADSTACK 76
 - using with CICS 50

S

- SCEERUN 15, 16
- SEARCH compile-time option 32, 57
- SIBMLINK 15, 16
- SIGFPE 51
- SIGFPE exceptions 19
- SIGINT 33, 56
- sign extension 25
- SIGTERM 33, 56
- SIGUSR1 33, 56
- SIGUSR2 33, 56
- sizeof() 21, 52
- SOM 72, 76
- source program
 - compatibility 9
 - with AD/Cycle C/370 compiler 51
 - with C++/MVS compiler 51
 - with C/MVS compiler 51
- SPIE run-time option 30, 50
- spie with #pragma runopts 32
- STACK run-time option
 - default size 30
 - parameters 31
- STAE run-time option 30, 50
- stae with #pragma runopts 32
- Standards, ANSI 53
- stderr 29, 34, 59
- strerror() 22
- structure declarations used as function parameters 25
- SYSERR ddname 29
- SYSPRINT ddname 29
- System Object Model 72, 76
- System Programming C Facility
 - applications built with EDCXSTRX 22
 - CEEEV003 22
 - EDCXV 22
 - relinking modules 13, 49
 - source changes 22
 - with #pragma runopts 32
- SYSTEM ddname 29

T

- Target 72
- TEST compile-time option 57
 - PATH suboption 32
- THREADSTACK run-time option 76

U

- UCS-2 converters 76
- ungetc()
 - effect upon behavior of fflush() 41, 63
 - effect upon behavior of fgetpos() 41, 63
 - effect upon behavior of fseek() 41, 63
- unhandled conditions 33, 56
- user exits
 - CEEBDATX 58
 - CEEBXITA 21
 - CEECDATX 58

- user exits (*continued*)
 - IBMBXITA 21
- UTF-8 converters 76

V

- variables
 - reentrant 71

W

- wchar_t data type 25
- WSIZEOF compiler option 21, 52



Printed in the United States of America

SC09-4763-00

