z/OS

Language Environment Writing Interlanguage Communication Applications

Version 2 Release 1

SA38-0684-00

Note

Before using this information and the product it supports, read the information in “Notices” on page 267.

This edition applies to Version 2 Release 1 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 6. Communicating between C and Fortran. 85

Chapter 7. Communicating between C++ and Fortran 107

Chapter 8. Communicating between C and PL/I. 127

Chapter 9. Communicating between C++ and PL/I. 149

Chapter 10. Communicating between COBOL and Fortran 169

Chapter 11. Communicating between COBOL and PL/I 185

Chapter 12. Communicating between Fortran and PL/I 207

Chapter 13. Communicating between multiple HLLs 225

Chapter 14. Communicating between assembler and HLLs 233

Chapter 15. ILC under CICS 241

Appendix A. Condition-handling responses. 249

Appendix B. Using nested enclaves 251

Appendix C. Accessibility 263

Notices 267

Index 271

Figures

Tables

About this document

This document supports z/OS (5650-ZOS).

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

Language Environment is the prerequisite runtime environment for applications generated with the following IBM compiler products:

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

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

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

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

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

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

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

This document is written for application programmers and developers to create and run interlanguage communication (ILC) applications under the z/OS IBM Language Environment product.

Language Environment improves ILC between conforming high-level languages (HLLs) because it creates one common runtime environment and it defines data types and constructs that are equivalent across languages.

For application programming, you will need to use this book, *z/OS Language Environment Programming Guide*, and *z/OS Language Environment Programming Reference*.You will also need to use the programming guides of the HLLs you are programming with.

This document is organized into pair-wise chapters that discuss ILC between two languages. There is also a chapter that discusses applications developed in more than two languages (Chapter 13, "Communicating between multiple HLLs," on page 225). ILC with assembler is discussed in Chapter 14, "Communicating between assembler and HLLs," on page 233 and ILC under CICS is discussed in Chapter 15, "ILC under CICS," on page 241.

How to send your comments to IBM

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

Use one of the following methods to send your comments:

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

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:
  z/OS V2R1.0 Language Environment Writing ILC Applications
  SA38-0684-00
- The topic and page number that is related to your comment.
- The text of your comment.

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

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

If you have a technical problem

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

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at z/OS support page (http://www.ibm.com/systems/z/support/).

z/OS Version 2 Release 1 summary of changes

See the following publications for all enhancements to z/OS Version 2 Release 1 (V2R1):

- *z/OS Migration*
- *z/OS Planning for Installation*
- *z/OS Summary of Message and Interface Changes*
- *z/OS Introduction and Release Guide*

Chapter 1. Getting started with Language Environment ILC

Interlanguage communication (ILC) applications are applications built of two or more high-level languages (HLLs) and frequently assembler. ILC applications run outside of the realm of a single language's environment, which creates special conditions, such as how the languages' data maps across load module boundaries, how conditions are handled, or how data can be called and received by each language.

This book helps you create ILC applications using Language Environment-conforming compilers. Most of the book is organized into "pairwise" chapters, which compare how each language handles different aspects of ILC, such as calling, data, reentrance, condition handling, and storage.

If your application contains more than two languages, you should read the section for each pair of languages first. For example, if your application consists of a C main routine that calls a COBOL subroutine, and the C main later calls a PL/I subroutine, read the chapters on C to COBOL and C to PL/I ILC. Then read Chapter 13, “Communicating between multiple HLLs,” on page 225 for additional information about developing multiple-language applications. If you have ILC with assembler or under CICS, see Chapter 14, “Communicating between assembler and HLLs,” on page 233 and Chapter 15, “ILC under CICS,” on page 241.

The benefits of ILC under Language Environment

Performance improves under the single runtime environment. Language Environment ILC applications run in one environment, giving you cooperative ILC support for running mixed-language applications, without the overhead of multiple libraries and library initialization.

The environment is tailored to HLLs at initialization. When you run your ILC applications in Language Environment, the initialization process establishes the Language Environment environment, tailored to the set of HLLs in the main load module. ILC applications follow the Language Environment program model, making program execution consistent and predictable.

Coordinated cleanup is performed at termination. Language Environment terminates in an orderly manner. Resources obtained during the execution of the application are released, regardless of the mix of programming languages in the application.

Cooperative condition handling provides consistency. All languages participating in the ILC application handle conditions cooperatively, making exception and condition handling consistent and predicable.

All ILC applications can reside above the line. Applications can be linked AMODE(31) RMODE(ANY) to reside above the 16M line in storage.

Writing ILC applications

Here are the steps you need to follow to develop an ILC application:

1. Decide which languages to use.

Your application code will need to follow the rules in the compiler programming guides and the *z/OS Language Environment Programming Guide*. Use the pairwise language chapters to identify what levels of HLLs you should be using.

2. Make sure all your ILC applications are Language Environment-conforming.

   Each chapter gives the basics of what you need to do to get your ILC applications to be Language Environment-conforming (adhering to Language Environment's common interface). For detailed information about migration, see the language migration guides.

3. Decide which language will have the main routine.

   Language Environment allows only one routine to be the main routine in an enclave. Each chapter describes how to determine the main routine in an ILC application. If you are using a multiple language application, see Chapter 13, “Communicating between multiple HLLs,” on page 225 to determine how to designate a main routine.

4. Learn how to declare and use data across HLLs.

   Each chapter describes how to use data in an ILC application.

5. Learn how to mix HLL and Language Environment operations. Each HLL has a unique way of using storage, return codes, and performing condition handling. Each chapter describes how to mix these HLL-specific constructs.

Chapter 2. Communicating with XPLINK applications

This chapter describes compatibility between AMODE 31 XPLINK and non-XPLINK programs.

Extra Performance Linkage (XPLINK) offers enhanced linkage between programs, potentially increasing performance significantly when frequent calls are made between small programs. The main focus of XPLINK is to improve speed and efficiency of subroutine calls, through a downward-growing stack, and by passing parameters in registers. Secondary objectives include reducing the function footprint and removing restrictions on function pointers for C and C++ functions compiled with XPLINK. For more information, see *z/OS Language Environment Programming Guide*.

XPLINK applications are supported under the IMS™ environment.

XPLINK compatibility support

XPLINK Compatibility Support is defined as the ability for routines (functions) compiled with NOXPLINK (these may be non-XPLINK C or C++, COBOL, PL/I, or OS Linkage Assembler) to "transparently" call routines that are compiled with XPLINK, and vice versa.

This transparent compatibility is provided at the Program Object (or Load Module, for compatibility with prelinker-built executables) boundary. That is, a Program Object (or Load Module) containing a caller of one linkage type (XPLINK or NOXPLINK) may call a routine compiled with the opposite linkage type as long as the called routine resides in a different Program Object (or Load Module). Program Objects can reside in either a PDSE or the HFS, while Load Modules must reside in a PDS.

The main "call linkage" supporting XPLINK Compatibility is the DLL call mechanism, but C's fetch() and Language Environment's CEEFETCH assembler macro are also supported.

The following are **not** supported for XPLINK in z/OS:

- COBOL dynamic call of an XPLINK function
- PL/I fetch
- Language Environment's CEELOAD assembler macro (traditional LOAD/BALR)

There are other environments that are not supported in an XPLINK environment (the XPLINK(ON) runtime option is in effect), such as AMODE-24 applications. Full details of supported environments are in *z/OS Language Environment Programming Guide*.

ILC calls between XPLINK and non-XPLINK routines

For calls made across a Program Object boundary (such as calls to a DLL or a C fetch()ed function), Language Environment will insert the necessary glue code to perform a transition from XPLINK to non-XPLINK, or vice versa. This glue code must perform the following tasks:

- Switch between the downward-growing stack that XPLINK routines use and the upward-growing stack of non-XPLINK routines.
- Switch between XPLINK and non-XPLINK register conventions.
- Convert between XPLINK and non-XPLINK parameter list and return value formats.

The details of the differences between these different linkage types are documented in *z/OS Language Environment Vendor Interfaces*.

Because of the extra overhead added by the glue code for calls between XPLINK and non-XPLINK routines, an application's overall performance can be affected if the number of calls between XPLINK and non-XPLINK routines is high. Applications that will benefit the most from being recompiled XPLINK are those that make many calls to small functions, all of which have been recompiled XPLINK. C++ applications are typically coded to this model. You should also try to minimize calls made between XPLINK and non-XPLINK routines, even if this means not compiling C or C++ routines that have a high interaction with non-XPLINK routines as XPLINK. For more details on selecting candidate applications for XPLINK, see *z/OS Language Environment Programming Guide*.

ILC between XPLINK and non-XPLINK C

The parameter list format passed by non-XPLINK C is identical to one built in the argument area of an XPLINK caller, except that in the XPLINK case certain parameters may be passed in registers. The argument list may contain addresses of arguments passed indirectly (by reference) or values of arguments passed directly (by value). The end of the parameter list is **not** marked by the high order bit of the last parameter being turned on. Since the end of the argument list is not identified the programmer must ensure that the callee only accesses as many parameters as the caller had arguments.

When an XPLINK function calls a non-XPLINK C function, glue code will use the information encoded at the call site to determine which registers contain parameters. These parameters will be stored in the argument area to construct a complete, contiguous parameter list. When the non-XPLINK C program is given control, register 1 will point to this complete parameter list. Upon return, the returned value is transferred from C to XPLINK conventions by the glue code, again using information encoded at the call site.

When a non-XPLINK C function calls an XPLINK function, glue code will use the interface-mapping flags in the PPA1 of the XPLINK callee to determine which registers should contain parameters. These registers will be loaded from the parameter list, and the rest of the parameter list will be copied into the argument area of the "caller" (in this case, a transitional stack frame). Upon return, the returned value is transferred from XPLINK to C conventions by the glue code, again using information in the interface-mapping flags.

ILC between XPLINK and non-XPLINK C++

The parameter list format passed by FASTLINK C++ is identical to that built in the argument area of an XPLINK caller. In both the FASTLINK and XPLINK cases, some parameters may be passed in registers, although the rules differ about which registers get loaded and when. In the FASTLINK case, any remaining parameters not passed in registers are passed in an argument area at a fixed location in the **callee's** stack frame. In the XPLINK case, remaining parameters are passed in an argument area at a fixed location in the **caller's** stack frame.

ILC between XPLINK and COBOL

The only way a COBOL routine can call an XPLINK-compiled routine is if the caller is compiled with the Enterprise COBOL for z/OS or COBOL for OS/390 & VM compiler with the DLL compiler option, and the target of the call is in a separate DLL, or vice versa.

COBOL Dynamic Call to XPLINK is not supported.

Since XPLINK compatibility is only provided to Language Environment-conforming languages, OS/VS COBOL programs and VS COBOL II programs are not supported in an XPLINK(ON) environment.

The COBOL reusable environment support (the RTEREUS runtime option or the callable interfaces ILBOSTP0 and IGZERRE) cannot be used in an XPLINK(ON) environment.

COBOL ILC with an XPLINK function in a separate DLL can employ either of two parameter passing techniques:

- The "pragmaless" style, where the COBOL programmer explicitly specifies syntax indicating that arguments are to be passed BY VALUE, and uses RETURNING syntax to access C function results.

  RETURNING values and BY VALUE arguments are implemented using pre-XPLINK C linkage conventions, and are designed to enable pragma-less ILC with C or with C++ using EXTERN C. This technique also works with XPLINK. In this case, the glue code will load the necessary XPLINK parameter registers when called from a COBOL function, or store them into the argument area when calling a COBOL function.
- The #pragma linkage(..., COBOL) style, where the COBOL programmer specifies normal BY REFERENCE argument conventions, the C functions specify normal C by-value conventions, and the XL C/C++ compiler introduces code to accommodate both.

  An XPLINK program can identify a called function as using a COBOL-style parameter list (R1 => list of addresses, with the High Order Bit (HOB) of the last parameter turned on):

  ```
  #pragma linkage(called_rtn,COBOL)
  ```

  In this case the XPLINK compiler generates a list of addresses to the actual parameters. The only difference is that XPLINK loads up to the first three of these addresses into registers 1, 2, and 3. When the glue code receives control from the XPLINK caller to swap the stack before giving control to the called COBOL function, it also creates a complete by-reference parameter list and set register 1 with the address of this parameter list.

  An XPLINK function can also specify that it receives COBOL-style parameters as input:

  ```
  #pragma linkage(this_rtn,COBOL)
  ```

  Processing via glue code is similar – when the XPLINK function receives control, the first three parameter addresses have been loaded into the parameter registers.

ILC between XPLINK and PL/I

The only way a PL/I program can call an XPLINK-compiled routine is if the caller is compiled with the EnterprisePL/I for z/OS compiler with the DLL compiler option and the target of the call is in a separate DLL, or vice versa.

PL/I FETCH of an XPLINK program object is not supported.

PL/I Multitasking is not supported in an XPLINK environment.

The processing of ILC calls between PL/I and XPLINK will be very similar to COBOL.

- PL/I parameter list is either pragma-less (by using the BYVALUE PROCEDURE attribute) or specified in the XPLINK program using `#pragma linkage(...,PLI)`.
- Calls from XPLINK to PL/I through glue code will establish an environment conforming to PL/I linkage conventions before giving the PL/I function control.
- Calls from PL/I to XPLINK through glue code will convert from PL/I linkage conventions to XPLINK conventions.

ILC between XPLINK and Assembler

The processing of ILC calls between Language Environment-conforming assembler and XPLINK will be identical to COBOL and PL/I.

Since the assembler programmer has direct control over the format of the parameter list, it can be constructed as either a "C-style" parameter list, or as an OS linkage parameter list. In the latter case, the XPLINK program must specify `#pragma linkage(...,OS)` at its interface with the assembler program.

The format of an OS linkage parameter list, as defined by an XPLINK function, is that the address of the first parameter will be passed in register 1, the address of the second parameter will be passed in register 2, the address of the third parameter will be passed in register 3, and any remaining parameters will be passed by placing their address in the caller's argument area. The high-order bit of the last parameter will be turned on. Note that this is different from the expected "R1 points to a list of addresses", but has better performance characteristics and allows the glue routine to issue the instruction STM R1,R3 to build a complete OS linkage parameter list. It can then set R1 to the address of this list for a call to an OS linkage routine.

There are three flavors of OS linkage that can be used by an XPLINK program:

- **OS_UPSTACK**

  In general, parts compiled XPLINK cannot be combined with parts compiled NOXPLINK in the same program object. One exception to this rule is OS linkage routines that are defined in an XPLINK-compiled caller as OS_UPSTACK. In this case, the XPLINK compiler will generate a call to glue code that performs a transition from the XPLINK caller to the OS linkage callee. The callee will get control with OS linkage conventions (parameter list and registers) and running on a Language Environment-conforming upward-growing stack.

  From XPLINK C code, specified as one of:

  – `#pragma linkage(function_name,OS_UPSTACK)`

  – `#pragma linkage(function_name,OS)` with the OSCALL(UPSTACK) compiler option

  From XPLINK C++ code, specified as one of:

  - `extern "OS_UPSTACK" function_prototype`
  - `extern "OS" function_prototype` with the OSCALL(UPSTACK) compiler option (this is the default)

- **OS_NOSTACK**

  The other exception allowing XPLINK and NOXPLINK parts in the same program object is that XPLINK-compiled routines can call OS linkage routines defined as OS_NOSTACK. In this case, the XPLINK compiler will generate an OS linkage style call (parameter list and registers) directly to the callee. There is no intervening glue code to provide a stack swap. Instead, a 72-byte savearea is provided. This provides much better performance characteristics over OS_UPSTACK calls when the called routine does not require an Language Environment-conforming stack.

  From XPLINK C code, specified as one of:

  - `#pragma linkage(function_name,OS_NOSTACK)`
  - `#pragma linkage(function_name,OS)` with the OSCALL(NOSTACK) compiler option (this is the default)

  From XPLINK C++ code, specified as one of:

  - `extern "OS_NOSTACK" function_prototype`
  - `extern "OS" function_prototype` with the OSCALL(NOSTACK) compiler option

- **OS_DOWNSTACK**

  This defines calls between XPLINK-compiled routines that pass an OS linkage "by reference" parameter list. XPLINK calling conventions are used.

  From XPLINK C code, specified as one of:

  - `#pragma linkage(function_name,OS_DOWNSTACK)`
  - `#pragma linkage(function_name,OS)` with the OSCALL(DOWNSTACK) compiler option

  From XPLINK C++ code, specified as one of:

  - `extern "OS_DOWNSTACK" function_prototype`
  - `extern "OS" function_prototype` with the OSCALL(DOWNSTACK) compiler option

ILC between XPLINK and Fortran

XPLINK compatibility with Fortran is not supported.

PIPI XPLINK considerations

Language Environment Version 1 Release 3 adds support to the preinitialization services (PIPI) to support programs that have been compiled XPLINK. Specifically, it allows programs and subroutines that have been compiled XPLINK to be defined in the PIPI table. For more details, refer to the *z/OS Language Environment Programming Guide*.

Chapter 3. Communicating between C and C++

This topic describes Language Environment's support for C and C++ ILC applications. If you are running a C to C++ ILC application under CICS you should also consult Chapter 15, "ILC under CICS," on page 241.

General facts about C to C++ ILC

C++ is reentrant by default. To create a reentrant C to C++ application, compile the C program with the RENT compiler option. (See "Building a reentrant C to C++ application" on page 11.)

If theC code was not compiled with RENT, the C++ code must contain special directives so it can use global variables defined in the C or C++ program. This information can be found in "Data equivalents" on page 15.

Preparing for ILC

This section describes the topics you should consider before writing a C to C++ ILC application. For help in determining how different versions of C and C++ work together, refer to *z/OS Language Environment Runtime Application Migration Guide*.

Language Environment ILC support

Table 1. Supported languages for Language Environment ILC

HLL pair	C	C++
C–C++	• C/370 Version 2 • AD/Cycle C/370 Version 1 • C/C++ for MVS/ESA • z/OS XL C/C++ compilers	• C/C++ for MVS/ESA • z/OS XL C/C++ compilers

Determining the main routine

In C and C++ the main routine is the function called `main()`. In a C to C++ ILC application only one `main()` function is allowed. Multiple `main()` functions will result in errors. Recursive calls to the `main()` function are not supported in C++.

An entry point is defined for each supported HLL. Table 2 identifies the desired entry point. The table assumes that your code has been compiled using the Language Environment-conforming compilers.

See *z/OS XL C/C++ Runtime Library Reference* for the description of the requirements for fetching C++.

Table 2. Determining the entry point

HLL	Main entry point	Fetched entry point
C	CEESTART	CEESTART or routine name, if `#pragma linkage(,fetchable)` is not used.
C++	CEESTART	CEESTART or routine name, if `#pragma linkage(,fetchable)` is not used.

Declaring C to C++ ILC

If a C function invokes a C++ function or a C++ function invokes a C function, all entry declarations are contained solely within the C++ source. No special declaration is required in the C code.

For C to C++ ILC, the C++ `extern "C"` linkage specification lets the C++ compiler generate parameter lists for C or accept parameter lists from C.

The `extern "C"` linkage specification has the following format:

extern "C" {*declaration*}

declaration is a valid C++ prototype of the C function(s) being called by C++, or the C++ routine being called by C. The braces { } are not required if only one declaration is specified.

If your C or C++ application is compiled with XPLINK or LP64 (which implies AMODE 64 XPLINK), the linkage and parameter passing mechanisms for C and C++ are identical. If you link to a C function from a C++ program, you should still specify `extern "C"` to avoid name mangling.

Declaration for C calling C++

C function

```
#include <stdio.h>

void CPLUSF (int parm);
int  CPLUSF2 (int parm);

int main()  {
  int x,y;

  x=3;
  y=CPLUSF2(x);
  printf("x = %d, y = %d\n",x,y);

  CPLUSF(x);

}
```

C++ function

```
#include <stdio.h>
#include <stdlib.h>
extern "C"  {
  void CPLUSF (int parm);
  int CPLUSF2 (int parm);
}

void CPLUSF (int parm) {

}

int CPLUSF2 (int parm) {
  int myint;
  myint=parm;
  return (myint);
}
```

Declaration for C++ calling C

C++ function

```
#include <stdio.h>
#include <stdlib.h>
extern "C" {
  void CFUNC (int parm);
  int CFUNC2 (int parm);
}

int main() {
  int x,y;

  x=3;
  y=CFUNC2(x);
  printf("x = %d, y = %d\n",x,y);

  CFUNC(x);

}
```

C function

```
#include <stdio.h>

void CFUNC (int parm) {

}

int CFUNC2 (int parm) {
  int myint;
  myint=parm2;
  return (myint);
}
```

Building a reentrant C to C++ application

The XL C++ compiler creates reentrant code by default. However, to create a reentrant C to C++ ILC application, you need to follow the following process:

1. Compile your C++ code.
2. Compile your C code with the RENT, LONGNAME, and DLL parameters. (LONGNAME and DLL are not required but make the prelinking and linking process simpler.)
3. Prelink all C++ and C text decks together using the Language Environment prelinker.

   The Prelink step can be eliminated when the target library for the executable module is either the HFS or a PDSE and the DFSMS Binder is used to link all C++ and C text decks together.
4. Link the text deck created by the prelinker to create your module.

See *z/OS Language Environment Programming Guide* for more information about reentrant applications.

Calling between C and C++

Table 3 describes the types of calls between C and C++ that Language Environment allows:

Table 3. Calls permitted for C to C++ ILC

Direction of call	Static calls	Dynamic calls using DLLs	Fetch/fetchep
C to C++	Yes	Yes	Yes
C++ to C	Yes	Yes	Yes

Table 3. Calls permitted for C to C++ ILC (continued)

Direction of call	Static calls	Dynamic calls using DLLs	Fetch/fetchep

Note:

1. The `fetch()` function can be used to fetch modules, compiled DLL, or modules containing C++ routines, but exported data will not be available.
2. Any of the C or C++ functions can be compiled with the XPLINK option, with the single restriction that you cannot mix XPLINK and non-XPLINK in the same module (that is, XPLINK and non-XPLINK C or C++ cannot be statically bound together).

As of C/C++ for MVS/ESA V3, the compiler provides support for dynamic load libraries (DLLs) which can be used to dynamically access C or C++ functions or data. For more information about DLLs, see *z/OS XL C++ Programming Guide*.

Passing data between C and C++

There are two ways to pass data with C and C++: *by value* and *by reference*. By value means that a temporary copy of the argument is passed to the called function or procedure. By reference means that the address of the argument is passed.

Passing data by value between C and C++

In general, value parameters are passed and received by C++ in the same manner as under C; a non-pointer or reference variable is passed in the parameter list. Any change that happens to a value parameter in the called function does not affect the variable in the caller, as in the following example, where an integer is passed by value from C++ to C:

Sample C++ usage

```
#include <stdio.h>
extern "C" int cfunc(int);

main() {
  int result, y;

  y=5;
  result=cfunc(y);  /* by value */

  if (y==5 && result==6)
    printf("It worked!\n");
}
```

C subroutine

```
#include <stdio.h>

cfunc(int newval)
{
  ++newval;
  return newval;
}
```

Similarly, to pass an int by value from C to C++:

Sample C usage

```
#include <stdio.h>

int cppfunc(int);
main() {
int result, y;

y=5;
result=cppfunc(y);  /* by value */

if (y==5 && result==6)
printf("It worked!\n");
}
```

C++ subroutine

```
#include <stdio.h>

extern "C" {
int cppfunc(int);
}

int cppfunc(int newval)
{
++newval;
return newval;
}
```

Passing data by reference between C and C++

In C, you can pass data by reference by passing a pointer to the item or passing the address of the item. In C++, you can pass a pointer, the address of the item, or a reference variable.

A pointer passed from C to C++ may be received as a pointer or as a reference variable, as in the following example:

Sample C usage

```
#include <stdio.h>

main() {
  int result, y;
  int *x;

  y=5;
  x= &y;

  result=cppfunc(x);
    /* by reference */
  if (y==6)
  printf("It worked!\n");
}
```

C++ subroutine

```
#include <stdio.h>

extern "C" {
  int cppfunc(int *);
}

cppfunc(int *newval)
{     // receive into pointer
  ++(*newval);
  return *newval;
}
```

Sample C usage

```
#include <stdio.h>

main() {
  int result, y;
  int *x;

  y=5;
  x= &y;

  result=cppfunc(x);
    /* by reference */
  if (y==6)
  printf("It worked!\n");
}
```

C++ subroutine

```
#include <stdio.h>

extern "C" {
  int cppfunc(int&);
}

cppfunc(int&; newval)
{  // receive into reference variable
  ++newval;
  return newval;
}
```

A pointer, or the address of a variable, passed from C++ to C must be received as a pointer, as in the following example:

Sample C++ usage

```
#include <stdio.h>
extern "C" {
  int cfunc(int *);
}

main() {
  int result, y;
  int *x;

  y=5;
  x= &y;

  result=cfunc(x);  /* by reference */
  if (y==6)
    printf("It worked!\n");
}
```

C subroutine

```
#include <stdio.h>

cfunc(int *newval)
{   // receive into pointer
  ++(newval);
  return(*newval);
}
```

Similarly, a reference variable passed from C++ to C must be received as a pointer, as in the following example:

Sample C++ usage

```
#include <stdio.h>
extern "C" {
  int cfunc(int *);
}

main() {
  int result, y=5;
  int& x=y;

  result=cfunc(x); /* by reference */
  if (y==6)
  printf("It worked!\n");
}
```

C subroutine

```
#include <stdio.h>

cfunc( int *newval )
{   // receive into pointer
  ++(*newval);
  return *newval;
}
```

Passing C++ objects

Objects can pass freely between C and C++ if the layout of the object in C and C++ is identical. In the following example, `cobj` (C) and `cxxobj` (C++) are identical:

```
struct cobj {
    int    age;
    char*  name;
}

class cxxobj {
public:
    int    age;
    char*  name;
}
```

A C++ structure is just a class declared with the keyword `struct`; its members and base classes are public by default. Therefore, a C++ class is the same as a C++ structure if all data is public. A union is a class declared with the keyword `union`; its members are public by default and holds only one member at a time. In C, a structure is a simple variant of the C++ class.

If a C++ class using features not available to C (for example, virtual functions, virtual base class, private and protected data, or static data members) is passed to C, the results are undefined.

Supported data types passed between C and C++

C data type	Equivalent C++ data type
char	char
signed char	signed char
unsigned char	unsigned char
short, signed short, short int, or signed short int	short, signed short, short int, or signed short int
unsigned short, or unsigned short int	unsigned short, or unsigned short int
int, signed, signed int	int, signed, signed int
unsigned, or unsigned int	unsigned, or unsigned int
long, signed long, long int, or signed long int	long, signed long, long int, or signed long int
unsigned long, or unsigned long int	unsigned long, or unsigned long int
float	float
double	double
long double	long double
struct	struct, some classes
union	union

C data type	Equivalent C++ data type
enum	enum
array	array
pointers to above types, pointers to void	pointers to above types, reference variables of above types, or pointers to void
pointers to functions	pointers to functions
types created by typedef	types created by typedef

Note:

1. C functions invoked from C++ or C++ functions invoked from C must be declared as `extern "C"` in the C++ source.
2. Packed decimal is not supported by C++. If you need to use packed decimal data, declare and modify it in C code using C functions.
3. If C++ classes, using features that are not available in C (see “Passing C++ objects” on page 14 for examples), are passed to C, the results are undefined.

Using aggregates

C structures and unions may map differently than C++ structures, classes, and unions. The C and C++ AGGREGATE compiler options provide a layout of aggregates to help you perform the mapping.

Data equivalents

This section shows how C and C++ data types correspond to each other.

Equivalent data types for C to C++

The following examples illustrate how C and C++ routines within a single ILC application might code the same data types.

Signed one-byte character data

Sample C usage

```
#include <stdio.h>

int cplusf( signed char mc );

int main()
{
  int rc;

  signed char myc='c';

  rc=cplusf(myc); /* by value */
  printf("myc=%c rc=%d\n",myc,rc);

}
```

C++ subroutine

```
#include <stdio.h>
#include <stdlib.h>

extern "C" int cplusf(signed char myc);

int cplusf(signed char myc)
{
  myc='d';
  printf("myc=%c, rc=%d\n",myc,myc);

  return((int)myc);
}
```

Sample C usage

```
#include <stdio.h>

int cplusf( signed char *mc );

int main()
{
  int rc;

  signed char myc='c';

  rc=cplusf(&myc);
  /* by reference */
  printf("myc=%c rc=%d\n",myc,rc);

}
```

C++ subroutine

```
#include <stdio.h>
#include <stdlib.h>

extern "C" int cplusf(signed char&; myc);

int cplusf(signed char&; myc)
{
  myc='d';
  printf("myc=%c, rc=%d\n",myc,myc);

  return((int)myc);
}
```

32-bit unsigned binary integer

Sample C usage

```
#include <stdio.h>

int cplusf( unsigned int mi );

int main()
{
  int rc;

  unsigned int myi=32;

  rc=cplusf(myi);
    /*  by value */
  printf("myi=%u rc=%d\n",myi,rc);

}
```

C++ subroutine

```
#include <stdio.h>
#include <stdlib.h>

extern "C" int cplusf(unsigned int myi);

int cplusf(unsigned int myi)
{
  myi=33;
  printf("myi=%u, rc=%d\n",myi,myi);

  return((int)myi);
}
```

Sample C usage

```
#include <stdio.h>

int cplusf( unsigned int *mi );

int main()
{
  int rc;

  unsigned int myi=32;

  rc=cplusf(&myi);
    /*  by reference */
  printf("myi=%u rc=%d\n",myi,rc);

}
```

C++ subroutine

```
#include <stdio.h>
#include <stdlib.h>

extern "C" int cplusf(unsigned int *myi);

int cplusf(unsigned int *myi)
{
  *myi=33;
  printf("myi=%u, rc=%d\n",*myi,*myi);

  return((int)*myi);
}
```

Structures and typedefs

Sample C usage

```
#include <stdio.h>

typedef struct {
  int   x;
  float y;
} coord;

double cplusf( coord *mycoord);

int main()
{
  double rc;

  coord xyval, *xy;

  xyval.x=2;
  xyval.y=4;

  xy=&xyval;
  rc=cplusf(xy);
    /*  by reference */
  printf("xyval.x=%d xyval.y=%f
    rc=%lf\n",xyval.x,xyval.y,rc);

}
```

C++ subroutine

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int   x;
  float y;
} coord;

extern "C" double cplusf( coord&; mycoord);

double cplusf(coord&; xy)
{
  double rc;

  xy.x=4;
  xy.y=10;
  rc=xy.x * xy.y;
  printf("xy.x=%d, xy.y=%f, rc=%lf\n",
    xy.x,xy.y,rc);

  return(rc);
}
```

Function pointers

Sample C usage

```
#include <stdio.h>

typedef int(FUNC) (int);

void cplusf (FUNC *myfnc);

int myfunc( int value )
{
  int rc;
  rc= printf( "The given value
    was %d\n", value);
  return(rc);
}

int main()
{
  int rc;

  rc = myfunc(3);
  printf("rc=%d\n",rc);

  cplusf(myfunc);

}
```

C++ subroutine

```
#include <stdio.h>
#include <stdlib.h>

extern "C" {
  typedef int(FUNC) (int);
  void cplusf (FUNC *myfnc);
}

void cplusf(FUNC *myfunc)
{
  int rc;

  rc=myfunc(3);
  printf("rc=%d\n",rc);

}
```

Equivalent data types for C++ to C

The following examples illustrate how C++ and C routines within a single ILC application might code the same data types.

16-bit signed binary integer

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>

extern "C" int cfunc(short int msi);

int main()
{
  int rc;

  short mysi = 2;

  rc=cfunc(mysi); /* by value */
  printf("mysi=%hd rc=%d\n",mysi,rc);

}
```

C subroutine

```
#include <stdio.h>
#include <stdlib.h>

int cfunc(short mysi);

int cfunc(short int mysi)
{
  mysi=5;
  printf("mysi=%hd, rc=%d\n",mysi,mysi);

  return((int)mysi);
}
```

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>

extern "C" int cfunc( short int *msi );

int main()
{
  int rc;

  short mysi = 2;
  short *pmysi;

  pmysi=&mysi;
  rc=cfunc(pmysi);
    /* by reference */
  printf("mysi=%hd rc=%d\n",mysi,rc);

}
```

C subroutine

```
#include <stdio.h>
#include <stdlib.h>

int cfunc(short *mysi);

int cfunc(short int *mysi)
{
  *mysi=5;
  printf("mysi=%hd, rc=%d\n",*mysi,*mysi);
  return((int)*mysi);
}
```

Short floating-point number

Sample C++ usage

```
#include <stdio.h>

extern "C" int cfunc( float mf );

int main()
{
  int rc;

  float myf=32;

  rc=cfunc(myf);
    /*  by value */
  printf("myf=%f rc=%d\n",myf,rc);

}
```

C subroutine

```
#include <stdio.h>
#include <stdlib.h>

int cfunc(float myf);

int cfunc(float myf)
{
  myf=33;
  printf("myf=%f, rc=%d\n",myf,33);

  return(33);
}
```

Sample C++ usage

```
#include <stdio.h>

extern "C" int cfunc( float *mf );

int main()
{
  int rc;

  float myf=32;

  rc=cfunc(&myf);
    /*  by reference */
  printf("myf=%f rc=%d\n",myf,rc);

}
```

C subroutine

```
#include <stdio.h>
#include <stdlib.h>

int cfunc(float *myf);

int cfunc(float *myf)
{
  *myf=33;
  printf("myf=%u, rc=%f\n",*myf,*myf);

  return(33);
}
```

Pointer to character

Sample C++ usage

```
#include <stdio.h>

extern "C" void cfunc( char * stuff);

int main()
{
  char *mystuff="fun fun";

  cfunc(mystuff);
    /*  by reference */
  printf("mystuff is %s\n",mystuff);

}
```

C subroutine

```
#include <stdio.h>
#include <stdlib.h>

void cfunc(char *stuff);

void cfunc(char *stuff)
{
  *(stuff+1)='U';
  printf("stuff is %s\n",stuff);
}
```

Function pointers

Sample C++ usage

```
#include <stdio.h>

extern "C" {
  typedef int(FUNC) (int);
  int myfunc( int value );
  void cfunc (FUNC *myfnc);
}

int myfunc( int value )
{
  int rc;
  rc= printf( "The given value
    was %d\n", value);
  return(rc);
}

int main()
{
  int rc;

  rc = myfunc(3);
  printf("rc=%d\n",rc);

  cfunc(myfunc);

}
```

C subroutine

```
#include <stdio.h>
#include <stdlib.h>

typedef int(FUNC) (int);
void cfunc (FUNC *myfnc);

void cfunc(FUNC *myfunc)
{
  int rc;

  rc=myfunc(3);
  printf("rc=%d\n",rc);

}
```

Name scope and name space

In programming languages, the *name scope* is defined as the portion of an application within which a particular declaration applies or is known. *Name space* is defined as the portion of a load module within which a particular declaration applies or is known. These two concepts determine whether a particular declaration in one language will map to a reference in another language.

In C, the name space is determined by compiler options or specific compiler directives. Programs compiled with NORENT place variable declarations in CSECTs. Programs compiled with RENT refer to a writable static area that contains static, external, and string literal variables. For more information about the writable static area, see *z/OS XL C/C++ Programming Guide*.

The XL C++ compiler acts as if everything was compiled with RENT.

The two techniques for creating an executable program are:

- When the executable program is to be stored in a PDSE or HFS, use the binder to combine the output from the XL C/C++ compiler.
- When the executable program is to be stored in a PDS, use the Language Environment Prelinker Utility to combine the output from the XL C/C++ compiler and pass the prelinker output to the binder.

To map data in C to C++ ILC, there are 3 major cases:

1. **C compiled with the RENT option statically bound to C++**

   C++ and C storage areas that result from compiling with RENT are mapped together at prelink time or at bind time when the Prelinker is not used and the output from the binder goes to the HFS or to a member in a PDSE. The name scope is the module boundary and external data will map to each other. The LONGNAME compiler option or the `#pragma map` preprocessor directive may be required to allow mixed-case, or long named external references, to resolve.
2. **C compiled without the RENT option statically bound to C++**

   C++ storage is placed in a different location than C NORENT storage. Therefore, by default, C++ will not look for NORENT C storage in the right place unless you give the XL C++ compiler the `#pragma variable(varname,norent)` directive.

Sample C usage

```
#include <stdio.h>

float mynum = 2;
main() {
  int result, y;
  int *x;

  y=5;
  x=&y;

  result = cppfunc(x);
    /* by reference */
  if (y==6 && mynum==3)
  printf("It worked!\n");
}
```

C++ subroutine

```
#include <stdio.h>

#pragma variable(mynum,norent)
extern float mynum;
extern "C" {
  int extern cppfunc( int * );
}

cppfunc( int *newval )
{      // receive into pointer
  *newval = *newval + 1;
  mynum = mynum + 1;
  return *newval;
}
```

Alternatively, you can specify to the XL C++ compiler that a specific variable is RENT by specifying `#pragma variable(varname,rent)` in C.

Sample C usage

```
#include <stdio.h>
#pragma variable(mynum,rent)

float mynum = 2;
main() {
  int result, y;
  int *x;

  y=5;
  x=&y;

  result = cppfunc(x);
    /* by reference */
  if (y==6 && mynum==3)
  printf("It worked!\n");
}
```

C++ subroutine

```
#include <stdio.h>

extern float mynum;
extern "C" {
  int extern cppfunc( int * );
}

cppfunc( int *newval )
{      // receive into pointer
  *newval = *newval + 1;
  mynum = mynum+ 1;
  return *newval;
}
```

3. **A C to C++ DLL application**

   A DLL application can access any declaration in its own module as well as referencing, implicitly or explicitly, any declaration from another DLL in the same enclave. For further information about what DLLs are and how to use them, see *z/OS XL C/C++ Programming Guide*.

Enhancing performance with packed structures and unions

Data elements of a structure or union are stored in memory on an address boundary specific for that data type. For example, a double value is stored in memory on a doubleword (8-byte) boundary. Gaps can be left in memory between elements of a structure to align elements on their natural boundaries. You can reduce the padding of bytes within a structure by packing that structure with the `_Packed` qualifier in C or by using the `#pragma pack(packed)` directive in C++ prior to the structure declaration.

The memory saved using packed structures might affect runtime performance. Most CPUs access data much more efficiently if it is aligned on appropriate boundaries. With packed structures, members are generally not aligned on appropriate (halfword, fullword, or doubleword) boundaries; the result is that member-accessing operations (`.` and `->`) might be slower. The `_Packed` qualifier in C and `#pragma pack(packed)` in C++ have the same alignment rules for the same structures or unions. `_Packed` affects the definition, and `#pragma pack(packed)` affects the declaration.

Example of packed structures

In the following C example, `fredc` is a packed structure with its members, `a,b,c`, aligned on 1-byte boundaries.

```
struct ss{
    int a;
    char b;
    double c;
};

_Packed struct ss fredc;
```

In the following C++ example, `fredcplus` is a packed structure with its members, `a,b,c`, aligned on 1-byte boundaries.

```
#pragma pack(packed)

struct ss{
    int a;
    char b;
    double c;
};

struct ss fredcplus;
```

Both fredc and fredcplus have the same storage mapping.

Calling packed structures and unions

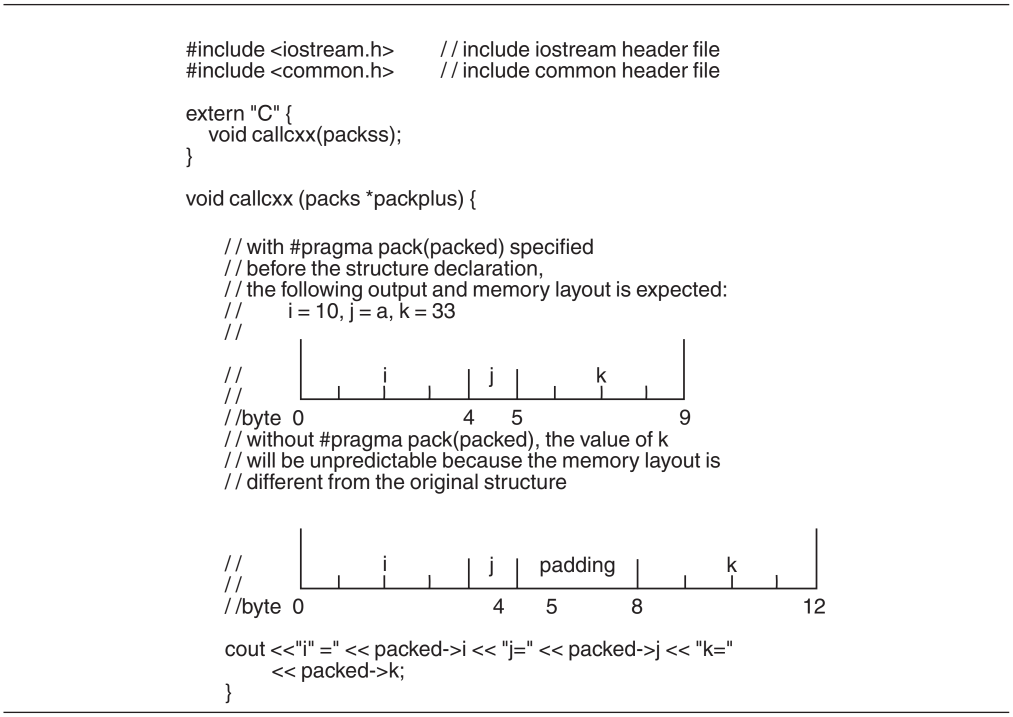
Packed and unpacked objects can have different memory layouts. You can use ILC calls with arguments that are packed structures by using the #pragma pack(packed) support in C++.

```
#ifdef __cplusplus         /* if compiled with C++ compiler */
  #ifndef _Packed         /* define _Packed                 */
    #define _Packed
  #endif

  #pragma pack(packed)    /* 1-byte alignment is used     */
#endif

struct ss {
  int i;
  char j;
  int k;
};

typedef _Packed struct ss   packss;

#ifdef __cplusplus           /* if compiled with C++ compiler */
  #pragma pack(reset)        /* reset alignment rule          */
#endif
```

Figure 1. Common header file (common.h)

```
#include "common.h"       /* include common header file */

void callcxx(packss);

main() {
  packss  packed;

  packed.i = 10;
  packed.j = 'a';
  packed.k = 33;

  callcxx(packed);
}
```

Figure 2. Common header file in C



```
#include <iostream.h>        // include iostream header file
#include <common.h>          // include common header file

extern "C" {
   void callcxx(packss);
}

void callcxx (packs *packplus) {

     // with #pragma pack(packed) specified
     // before the structure declaration,
     // the following output and memory layout is expected:
     //        i = 10, j = a, k = 33
     //

     //                 i          j            k
     //
     //byte  0                     4    5                9
     // without #pragma pack(packed), the value of k
     // will be unpredictable because the memory layout is
     // different from the original structure


     //                 i          j    padding         k
     //
     //byte  0                        4    5      8              12

     cout <<"i" =" << packed->i << "j=" << packed->j << "k="
           << packed->k;
     }
```

Figure 3. Common header file in C++

For more information about `#pragma pack(packed)` or on `_Packed` structures, see *z/OS XL C++ Language Reference*.

Using storage functions in C to C++ ILC

Use the following guidelines if you mix HLL storage constructs:

- If the storage was allocated using Language Environment services, free it using Language Environment services.
- If the storage was allocated using C functions such as `malloc()`, `calloc`, or `realloc()`, free it using `free()`;
- If the storage was allocated using the C++ `new` keyword then it must be deleted with `delete`.
- If your program requires that storage be allocated in one language and deleted in another, use the Language Environment services.

Directing output in ILC applications

When writing output to a standard stream from a C to C++ ILC application, both the C and C++ code should use the C I/O functions such as `printf` and `puts`, which allow the output to be written in the expected order. If your C++ program is directing output to `cout` (the default), you may get output in an unexpected order. See *z/OS XL C/C++ Programming Guide* chapter on IOSTREAMS for further information.

There is no restriction on passing file pointers from C to C++; a file opened using `fopen` in a C program may be closed by using `fclose` in a C++ program, and vice versa.

C to C++ condition handling

C++ exception handling uses `throw()`/`try()`/`catch()`, whereas C uses `signal()`/`raise()` or `sigaction()`/`kill()`. Mixing C and C++ exception handling in a C to C++ ILC module will result in undefined behavior. If you use only the C exception handling model, a C++ routine can register a signal handler via signal to handle exceptions (software/hardware) raised by either a C or C++ routine. However, the behavior of running destructors for static/automatic objects is undefined.

If you use only the C++ exception handling model, only C++ routines will be able to catch()/handle() thrown objects. C routines do not have try()/catch()/throw() abilities nor can they use signal() to register a handler for thrown objects. A C++ routine cannot register a handler via signal() to catch thrown objects; it must use catch() clauses. C routines will ignore thrown objects.

Sample C to C++ applications

```
/* Module/File Name:  EDCCXC  */
 /**********************************************************************/
 /* CPROGRAM, compile with rent and longname                           */
 /**********************************************************************/
 #include <stdio.h>
 #include <stdlib.h>

 typedef struct {
   int length;
   int width;
 } rectngl;

 extern void printrec( rectngl r );
 extern void change_width( rectngl* r , int w);

 int CFUNC( rectngl  *mine )
 {
   int rc;

   printrec( *mine);
   change_width( mine, 5);

 #ifdef debug
   printrec( *mine);
 #endif

   printrec( *mine);

   if (mine->width !=5)
     return(1);
   else
     return(0);

}
```

Figure 4. C++main routine

```
/* Module/File Name:  EDCCXCX   */
/*********************************************************************/
/* CXXPROG - prelink with CPROG                                      */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>

#include "verify.h"

class rectangle {
  public:
    int length;
    int width;

    rectangle(int l, int w ) { length = l; width = w; }
    void show()
    { printf("Length: %d  Width: %d\n",length,width); }
    void set_width( int size ) { width=size; }
    void set_length( int size) { length=size; }
};

extern "C" {
  int CFUNC( rectangle  *small );
  void printrec( rectangle r ) { r.show();};
  void change_width( rectangle* r, int nw) { r->set_width(nw);};
};

int main( )
{
  int rc;

  rectangle myrec(10,2);

  rc=CFUNC( &myrec);

  if (rc == 1)
    fail(__FILE__,__LINE__);
  else
    check(__FILE__);
}
```

Figure 5. Csubroutine

Chapter 4. Communicating between C and COBOL

This topic describes Language Environment's support for C and COBOL ILC applications. If you are running a C to COBOL ILC application under CICS, you should also consult Chapter 15, "ILC under CICS," on page 241.

General facts about C to COBOL ILC

- With Enterprise COBOL for z/OS, COBOL for OS/390 & VM, or COBOL for MVS & VM, the `#pragma linkage` directive is not required for most calls between C and COBOL routines (although it can still be used). See "Calling between C and COBOL" on page 31 for the cases that require it. With COBOL/370 and VS COBOL II, `#pragma linkage` is required in the C routines.
- A C to COBOL application can be constructed to be reentrant.
- Language Environment does not support the automatic passing of return codes between C and COBOL routines in an ILC application.
- There is no ILC support between AMODE 31 and AMODE 64 applications. COBOL does not support AMODE 64.
- ILC is not supported in the C multitasking facility environment.

Preparing for ILC

This section describes topics you might want to consider before writing an application that uses ILC. For help in determining how different versions of HLLs work together, refer to the migration guides for the HLLs you plan to use.

Language Environment ILC support

Language Environment provides ILC support between the following combinations of C and COBOL:

Table 4. Supported languages for Language Environment ILC

HLL pair	C (31–bit mode only)	COBOL
C to COBOL	• C/370 Version 1 Release 2 • C/370 Version 2 Release 1 • IBM C/C++ for MVS/ESA • z/OS XL C/C++ Compilers	• VS COBOL II Version 1 Release 3 and later • COBOL/370 Release 1 • COBOL for MVS & VM Release 2 • COBOL for OS/390 & VM • Enterprise COBOL for z/OS

Note: C refers to both the pre-Language Environment-conforming and Language Environment-conforming versions of C. COBOL refers to VS COBOL II, COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL for z/OS.

Language Environmentdoes not support ILC between OS/VS COBOL and C. See Chapter 15, "ILC under CICS," on page 241 for the allowable ILC on CICS.

Migrating ILC applications

Relinking

ILC applications that contain load modules with pre-Language Environment-conforming C routines or VS COBOL II programs must be relinked, either under Language Environment or with the migration tool provided in C/370

Version 2 Release 2. You should relink using the migration tool if your application will be used under both C/370 Version 2 Release 2 and Language Environment.

When you relink under Language Environment, explicitly include the following:

- The @@C2CBL and @@CBL2C routines. @@C2CBL is called when a C routine calls a COBOL program. @@CBL2C is called (in C) when a C routine is designated as linkage COBOL.
- IGZEBST or IGZENRI, depending on whether the COBOL program was compiled with RES or NORES, respectively.

Recompiling

You do not need to recompile an existing ILC application with a Language Environment-conforming compiler, but you can do so in order to take advantage of Language Environment's condition handling behavior.

Compiling and linking considerations

Compiling

If the C library is installed above the 16M line, compile your COBOL program using the RENT compiler option.

Linking

When link-editing ILC application load modules, there are different considerations for the main load module (the load module that contains the main routine) and for a load module that is fetched or dynamically called.

For the main load module, you should present your main routine to the linkage editor first in order to avoid an incorrectly chosen entry point. See "Determining the main routine" for information about how to identify the main routine.

For load modules that will be fetched or dynamically called, the entry point of the load module must be as follows:

- When C is the called routine and it does not specify any `#pragma linkage`, then the routine name must be the entry point.
- When C is the called routine and it specifies `#pragma linkage(...,COBOL)`, then the routine name must be the entry point.
- When C is the called routine and it specifies `#pragma linkage(,fetchable)`, then CEESTART or the routine name must be the entry point.
- When COBOL is the called routine, the program name must be the entry point.

To specify the entry point of a load module, use the binder's ENTRY control statement.

Determining the main routine

In Language Environment, only one routine can be the main routine; no other routine in the enclave can use syntax that indicates it is main. If you write the main routine in C, you must use language syntax to identify the routine as the main routine. If you use COBOL as the first program in the enclave that is to gain control, the program is effectively designated main by being the first to run.

In C, the same routine can serve as both the main routine and subroutines if recursively called. In such a case, the new invocation of the routine is not considered a second main routine within the enclave, but a subroutine. With a VS

COBOL II or COBOL/370 single enclave, a recursively called main program is not permitted; Enterprise COBOL for z/OS, COBOL for OS/390 & VM and COBOL for MVS & VM support recursion.

Table 5 describes how C and COBOL identify the main routine.

Table 5. How C and COBOL main routines are determined

Language	When determined	Explanation
C	Compilation	Determined in the C source file by declaring a C function named main. The same routine can be used both as a main and subroutine if it is recursively called.
COBOL	Run time	Determined dynamically. If it is the first program to run, it is a main program. COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL for z/OS support recursion, but the main program cannot be called recursively within a single enclave in VS COBOL II or COBOL/370.

An entry point is defined for each supported HLL. Table 6 identifies the desired entry point. The table assumes that your code was compiled using the Language Environment-conforming compilers.

Table 6. Determining the entry point

HLL	Main entry point	Fetched or dynamically called entry point
C	CEESTART	CEESTART or routine name if `#pragma linkage(,fetchable)` is used. In all other cases, the routine name.
COBOL	Name of the first object program to get control in the object module	Program name

Note: Specify ENTRY statement on link for the function name when not going through prelinker. Use `#pragma map` to remap a name and use the remapped name on ENTRY statement when prelinking.

C and COBOL routines that make up an ILC application are executed together in a single run unit (the equivalent of a Language Environment enclave). However, unlike in earlier versions of COBOL (VS COBOL II and OS/VS COBOL), the first COBOL program in a run unit is no longer necessarily considered the main program. If the first COBOL program is not the first program in the enclave to run, it is considered a subroutine in the Language Environment enclave.

Declaring C to COBOL ILC

No special linkage declaration is required for ILC between C and Enterprise COBOL for z/OS, COBOL for OS/390 & VM, or COBOL for MVS & VM. However, for COBOL/370 and VS COBOL II, a C `#pragma linkage` directive is required for both static and dynamic calls. The C `#pragma linkage` directive can still be used with Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM, but it is no longer required, as these levels of the COBOL products support direct use of C-style linkage conventions via language constructs such as BY VALUE arguments and function RETURNING values. Further, with the support of DLL linkage conventions in the Enterprise COBOL for z/OS and COBOL for OS/390 & VM products, COBOL applications may directly interoperate with reentrant C modules that are linked as DLLs.

When a #pragma linkage directive is required, all entry declarations are made in the C code, both in the case where C calls COBOL and vice versa. The C #pragma linkage directive lets the C compiler generate parameter lists for COBOL or accept them from COBOL. It also ensures that writable static pointers are passed correctly for reentrant C modules and, for calls from pre-Language Environment-conforming COBOL, verifies that Language Environment has been properly established.

The #pragma linkage directive has the following format:

```
#pragma linkage(function_name, COBOL)
```

function_name can be up to eight characters (Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM allow 160 characters). *function_name* is either the COBOL program being called by C, or the C routine being called by COBOL.

Note: When declaring without #pragma linkage the reference to COBOL is for Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM. When declaring with #pragma linkage the reference to COBOL is for Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II.

Declaring C to COBOL ILC without #pragma linkage

Declaration for C calling COBOL:

C function	COBOL program

`void COBRTN(int);`
`COBRTN(x);  /* x by value */`
	`01 X PIC S9(9) BINARY.`
	`PROCEDURE DIVISION USING BY VALUE X.`

Declaration for COBOL calling C:

COBOL program	C function

`01 X PIC S9(9) BINARY.`
`CALL "CENTRY" USING BY VALUE X.`
	`void CENTRY(int x) {`
	`}`

Declaring C to COBOL ILC with #pragma linkage

Declaration for C calling COBOL:

C function	COBOL program

`#pragma linkage(CBLRTN,COBOL)`
`void CBLRTN(int p1);`
`CBLRTN(p1);`
	`01 P1 PIC S9(9) USAGE IS BINARY`
	`PROCEDURE DIVISION USING P1.`

Declaration for COBOL calling C:

COBOL program	C function

`01 P1 PIC S9(9) USAGE IS BINARY`
`CALL 'CFUNC' USING BY CONTENT P1.`
	`#pragma linkage(CFUNC,COBOL)`
	`void CFUNC(int p1) {`
	`}`

Calling between C and COBOL

This section describes the types of calls permitted between C and COBOL, and considerations when using dynamic calls and fetch.

Types of calls permitted

The following tables describe the types of calls that are supported between C and COBOL when running with Language Environment.

- Table 7 shows which calls are supported from COBOL programs to C routines that use *#pragma linkage(...,COBOL)*.
- Table 8 on page 32 shows which calls are supported from COBOL programs to C routines that do not use *#pragma linkage(...,COBOL)*.
- Table 9 on page 33 shows which calls are supported from C routines to COBOL programs.

Table 7. Support for calls from COBOL to C with #pragma linkage(...,COBOL)

		Target		
Caller	**Call type**	**Non-reentrant C or naturally reentrant C**	**Reentrant C that does not export functions or variables**	**Reentrant C that exports functions or variables**
VS COBOL II (1)	static	Yes	Yes with restrictions (2)	Yes with restrictions (2)
COBOL/370 (1)	static	Yes	Yes	Yes
COBOL (5) compiled with NODLL	static	Yes	Yes	Yes
VS COBOL II (1)	dynamic	Yes (3)	Yes (3)	No
COBOL/370 (1)	dynamic	Yes (3)	Yes (3)	No
COBOL (5) compiled with NODLL	dynamic	Yes (3)	Yes (3)	No
COBOL (6) compiled with DLL	CALL "literal" to a function within the module	Yes	Yes	Yes
COBOL (6) compiled with DLL	CALL "literal" to a function exported from a DLL	No	No	Yes (4)
COBOL (6) compiled with DLL	CALL identifier to a function exported from a DLL	No	No	Yes (4)

Table 7. Support for calls from COBOL to C with #pragma linkage(...,COBOL) (continued)

		Target		
Caller	**Call type**	**Non-reentrant C or naturally reentrant C**	**Reentrant C that does not export functions or variables**	**Reentrant C that exports functions or variables**

Note:

1. When the caller is VS COBOL II or COBOL/370, all calls must be to *void* functions.
2. Static calls are supported from VS COBOL II to reentrant C in the following cases:
   - The call is done in the main load module.
   - The call is done in a load module whose entry point is a Language Environment-conforming program or routine that was called using COBOL dynamic call.
   - The call is done in a load module whose entry point is not a Language Environment-conforming program or routine that was called using COBOL dynamic call, and there are no C routines in the main load module, and no reentrant C routines have been previously called in any other load module.
   - The call is done in a module that was called using C fetch.
3. Dynamically called load modules cannot contain any DLL routines that export functions or variables.
4. In this case, the C code can also be compiled with the XPLINK option. The XPLINK option implies the DLL option — all XPLINK compiled code is automatically DLL-enabled. The XPLINK C code must reside in a separate module from the COBOL caller.
5. COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL for z/OS.
6. COBOL for OS/390 & VM or Enterprise COBOL for z/OS

Table 8. Support for calls from COBOL to C without #pragma linkage(...,COBOL)

		Target		
Caller	**Call type**	**Non-reentrant C or naturally reentrant C**	**Reentrant C that does not export functions or variables**	**Reentrant C that exports functions or variables**
VS COBOL II (1)	static	No	No	No
COBOL/370 (1)	static	No	No	No
COBOL (5) compiled with NODLL	static	Yes	Yes with restrictions (2)	Yes with restrictions (2)
VS COBOL II (1)	dynamic	No	No	No
COBOL/370 (1)	dynamic	No	No	No
COBOL (5) compiled with NODLL	dynamic	Yes (3)	Yes (3)	No
COBOL (6) compiled with DLL	CALL "literal" to a function within the module	Yes	Yes	Yes
COBOL (6) compiled with DLL	CALL "literal" to a function exported from a DLL	No	No	Yes (4)
COBOL (6) compiled with DLL	CALL identifier to a function exported from a DLL	No	No	Yes (4)

Table 8. Support for calls from COBOL to C without #pragma linkage(...,COBOL) (continued)

		Target		
Caller	**Call type**	**Non-reentrant C or naturally reentrant C**	**Reentrant C that does not export functions or variables**	**Reentrant C that exports functions or variables**

Note:

1. For VS COBOL II or COBOL/370, #pragma linkage(...,COBOL) is required.
2. Static calls are supported for calls from COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL for z/OS compiled with NODLL to reentrant C that exports functions or variables in the following cases:
   - The call is done in the main load module.
   - The call is done in a DLL that was called using DLL linkage.
3. Dynamically called load modules cannot contain any DLL routines that export functions or variables.
4. In this case, the C code can also be compiled with the XPLINK option. The XPLINK option implies the DLL option — all XPLINK compiled code is automatically DLL-enabled. The XPLINK C code must reside in a separate module from the COBOL caller.
5. COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL for z/OS.
6. COBOL for OS/390 & VM or Enterprise COBOL for z/OS

Table 9. Support for calls from C to COBOL

		Target		
Caller	**Call type**	**VS COBOL II or COBOL/370**	**COBOL (3) compiled with NODLL**	**COBOL (4) compiled with DLL**
C with #pragma linkage(...,COBOL) and compiled with NODLL	static	Yes (1)	Yes	Yes
C without #pragma linkage(...,COBOL) and compiled with NODLL	static	No	Yes	Yes
C with #pragma linkage(...,COBOL) and compiled with NODLL	fetch	Yes (1, 2)	Yes (2)	No
C without #pragma linkage(...,COBOL) and compiled with NODLL	fetch	No	Yes (2)	No
C with #pragma linkage(...,COBOL) and compiled with DLL	static	Yes (1)	Yes	Yes
C without #pragma linkage(...,COBOL) and compiled with DLL	static	No	Yes	Yes
C with #pragma linkage(...,COBOL) and compiled DLL	fetch	Yes (1,2)	Yes (2)	No
C without #pragma linkage(...,COBOL) and compiled DLL	fetch	No	Yes (2)	No
C with #pragma linkage(...,COBOL) and compiled DLL	dynamic (called function is exported from a DLL)	No	No	Yes
C without #pragma linkage(...,COBOL) and compiled with DLL	dynamic (called function is exported from a DLL)	No	No	Yes

Table 9. Support for calls from C to COBOL (continued)

		Target		
Caller	**Call type**	**VS COBOL II or COBOL/370**	**COBOL (3) compiled with NODLL**	**COBOL (4) compiled with DLL**
C with #pragma linkage(...,COBOL) and compiled with XPLINK	static	No	No	No
C without #pragma linkage(...,COBOL) and compiled with XPLINK	static	No	No	No
C with #pragma linkage(...,COBOL) and compiled with XPLINK	fetch	No	Yes (2)	No
C without #pragma linkage(...,COBOL) and compiled with XPLINK	fetch	No	Yes (2)	No
C with #pragma linkage(...,COBOL) and compiled with XPLINK	dynamic (called function is exported from a DLL)	No	No	Yes
C without #pragma linkage(...,COBOL) and compiled with XPLINK	dynamic (called function is exported from a DLL)	No	No	Yes

Note:

1. When the target of the call is VS COBOL II or COBOL/370, the called COBOL program must be declared as a *void* function.
2. Fetched COBOL load modules cannot contain any DLL routines that export functions or variables.
3. COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL for z/OS.
4. COBOL for OS/390 & VM or Enterprise COBOL for z/OS.

Performance considerations

The performance of COBOL calling C is best when `#pragma linkage(...,COBOL)` is not used.

Dynamic call/fetch considerations - non-DLL modules

Both C and COBOL provide language constructs that support the dynamic loading, execution, and deletion of user-written routines. The C `fetch()` library function dynamically loads a load module that you specify into main storage. The module can be invoked later from a C application (see *z/OS XL C/C++ Programming Guide* for more information about `fetch()`). In COBOL, you can use the dynamic CALL statement to dynamically load a load module into main storage. For more information about the CALL statement, see the appropriate version of the programming guide in the Enterprise COBOL for z/OS library (http://www-01.ibm.com/support/docview.wss?uid=swg27036733).

Both C and COBOL support multiple-level fetches or dynamic calls (for example, Routine 1 fetches Routine 2, which in turn fetches Routine 3, and so on).

User-written condition handlers registered using CEEHDLR can be fetched, but must be written in the same language as the fetching language.

C fetching C with COBOL statically linked

ILC between C and COBOL is supported within both a fetching C load module and a fetched C load module.

C fetching COBOL

You can use the C `fetch()` function to fetch a COBOL program and invoke it later using a function pointer. The declaration of a COBOL fetched program within a C routine is shown in Figure 6. The figure indicates C fetching either an Enterprise COBOL for z/OS or COBOL for OS/390 & VM program or a COBOL for MVS & VM program. If COBOL/370 or VS COBOL II were used, the C `#pragma linkage` directive would be used, as `#pragma linkage(CBL_FUNC, COBOL)`.

```
typedef void CBL_FUNC();
      .
      .
      .

CBL_FUNC *fetch_ptr;
fetch_ptr = (CBL_FUNC*) fetch("COBEP");   /* fetch the routine */
fetch_ptr(args);                          /* call COBEP        */
```

Figure 6. C fetching a COBOL program

You can use the C `release()` function to release a COBOL program that was explicitly loaded by `fetch()`. A COBOL CANCEL cannot be issued against any routine dynamically loaded using the C `fetch()` function.

COBOL dynamically calling COBOL with C statically linked

ILC between COBOL and C is supported within both a dynamically calling COBOL load module and a dynamically called COBOL load module.

Restriction: COBOL cannot dynamically call a C function that has been compiled XPLINK.

COBOL dynamically calling non-Language Environment conforming assembler with C statically linked

COBOL can dynamically call non-Language Environment conforming assembler which then calls a statically linked C routine provided the C routine is non-reentrant or naturally reentrant (no writable static). The C routine must have the #pragma linkage(...,COBOL) directive coded.

Note: If C has not yet been initialized, calling a reentrant C routine will work. The C routine must have the #pragma linkage(...,COBOL) directive coded. Due to the difficulty of the user ensuring that C has not yet been initialized, IBM recommends the use of one of the methods described in this section.

A preferable method would be to have COBOL dynamically call the C routine directly. You can also have COBOL dynamically call Language Environment conforming assembler, which then calls a statically linked C routine. In these cases, there would be no reentrancy restrictions.

Cancel considerations

A COBOL program can use the CANCEL statement to cancel a load module that contains C.

COBOL dynamically calling C

A COBOL program can dynamically CALL a C routine.

Dynamic call/fetch considerations - DLL modules

A DLL module differs from a regular load module in that the original source code was compiled using the DLL option of the XL C or COBOL compiler, and then uses the prelinker or binder facilities to import or export symbols.

Restriction: The DLL approach versus the COBOL dynamic call/C fetch approach are two different mechanisms for achieving the goal of structuring the application as multiple, separate modules. The two approaches do not mix, however. One or the other should be chosen. See the appropriate version of the programming guide in the Enterprise COBOL for z/OS library (http://www-01.ibm.com/support/docview.wss?uid=swg27036733), which explains this in some detail and gives some trade-offs between the DLL approach and the dynamic call approach. See also *z/OS Language Environment Programming Guide* or *z/OS XL C/C++ Programming Guide*.

Passing data between C and COBOL

In VS COBOL II and COBOL/370, you can pass parameters two ways:

By reference (indirect)
COBOL BY REFERENCE

By value (indirect)
COBOL BY CONTENT

In Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM, you can pass parameters three ways:

By reference (indirect)
COBOL BY REFERENCE

By value (indirect)
COBOL BY CONTENT

By value (direct)
COBOL BY VALUE

Under Language Environment, the term *by value* means that a temporary copy of the argument is passed to the called function or procedure. Any changes to the parameter made by the called routine will not alter the original parameter passed by the calling routine. Under Language Environment, the term *by reference* means that the actual address of the argument is passed. Any changes to the parameter made by the called routine can alter the original parameter passed by the calling routine.

Further, the term *direct* means that the argument is passed in the parameter list. The term *indirect* means that a pointer to the argument is passed in the parameter list.

There are two ways to pass data between C and COBOL: one way uses `#pragma linkage` in the C routine; the other does not. Both methods are discussed separately.

Passing data between C and COBOL without #pragma

Note: The reference to COBOL, in the sections explaining the use of COBOL without #pragma, applies only to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM.

When data is passed between C and COBOL without `#pragma linkage (COBOL)`, the standard language linkages are used with no extra level of indirection introduced.

Passing data by value between C and COBOL

Copies of variables can be passed between C and COBOL. On return, the actual value of the variable remains unchanged, regardless of how it may have been modified in the called routine.

Passing by value (direct): To pass data by value (direct) from C to COBOL, the variables are passed by C as arguments on a function call and received by COBOL as BY VALUE parameters. Conversely, to pass data by value (direct) from COBOL to C, the variables are passed by COBOL as BY VALUE arguments and received by C as function parameters. In all cases, the variable must be declared in C and COBOL with compatible base data types. For example, if a C function called FROMCOB is to receive a parameter passed by value (direct) of type int, the function prototype declaration would look like this:

```
void FROMCOB(int)
```

Passing by value (indirect): Data cannot be passed from C to COBOL; however, data can be passed by value (indirect) from COBOL to C. In this case, the variable is passed as a BY CONTENT argument and received by C as a pointer to the given type. For example, if a C function called FROMCOB is to receive a parameter passed by value (indirect) of type int, the function prototype declaration would look like this:

```
void FROMCOB(int *)
```

The C function must dereference the pointer to access the actual value. If the value of the pointer is modified by the C function, as opposed to modifying the value that the pointer points to, the results on return to COBOL are unpredictable. Thus, passing values by value (indirect) from COBOL to C should be used with caution, and only in cases where the exact behavior of the C function is known.

Table 10 on page 38 shows the supported data types for passing by value (direct) and Table 11 on page 38 shows the supported data types for passing by value (indirect).

Passing data by reference (indirect) between C and COBOL

A parameter can be passed by reference (indirect) between C and COBOL. By reference (indirect) means that the actual address of the argument is passed to the called function or procedure; any changes to the parameter made by the called routine can alter the original parameter passed by the calling routine.

To pass data by reference (indirect) from C to COBOL, the variables are passed by C as function arguments, which are pointers to a given type or the address of a given variable, and received by COBOL as BY REFERENCE parameters. Conversely, to pass data by reference (indirect) from COBOL to C, the variables are passed by COBOL as BY REFERENCE arguments and received by the C function as pointers to a given type.

The C function must dereference the pointer to access the actual value. If the value of the pointer is modified by the C function, as opposed to modifying the value that the pointer points to, the results on return to COBOL are unpredictable. Thus, passing values by reference (indirect) from COBOL to C should be used with caution, and only in cases where the exact behavior of the C function is known.

Table 11 shows the supported data types for passing by reference (indirect).

Supported data types passed by value (direct) between C and COBOL

Table 10 identifies the data types that can be passed by value (direct) as parameters between C and COBOL applications.

Table 10. Supported data types passed by value (direct) without #pragma

C	COBOL (by value)
char	PIC X, PIC A
signed short int	PIC S9(4) USAGE IS BINARY
unsigned short int	PIC 9(4) USAGE IS BINARY
signed int, signed long int	PIC S9(9) USAGE IS BINARY
unsigned int	PIC 9(9) USAGE IS BINARY, LENGTH OF
unsigned long int	PIC 9(9) USAGE IS BINARY
float	COMP-1
double	COMP-2
pointer to...	POINTER, ADDRESS OF

If the COBOL program receives `int` parameters from the C calling function that might have a value that is larger than that declared as the maximum size by the COBOL picture clause, the COBOL program must either be compiled with the TRUNC(BIN) compiler option or each binary data item that receives `int` parameters from C must be declared as USAGE IS COMP-5. Taking these actions will guarantee that truncation of high-order digits does not occur. For more information about the TRUNC(BIN) compiler option or about using COMP-5 data items, see the appropriate version of the programming guide in the Enterprise COBOL for z/OS library (http://www-01.ibm.com/support/docview.wss?uid=swg27036733).

Supported data types passed between C and COBOL by reference (indirect) and from COBOL to C either by value (indirect) or by reference (indirect)

Table 11 identifies the data types that can be passed between C and COBOL by Reference (Indirect) and from COBOL to C either by Value (Indirect) or by Reference (Indirect).

Table 11. Supported data types passed between C and COBOL by reference (Indirect) and from COBOL to C either by value (indirect) or by reference (indirect) without #pragma

C (Pointer to...)	COBOL (by content/by reference)
char	PIC X, PIC A
signed short int	PIC S9(4) USAGE IS BINARY
unsigned short int	PIC 9(4) USAGE IS BINARY
signed int, signed long int	PIC S9(9) USAGE IS BINARY
unsigned int	PIC 9(9) USAGE IS BINARY, LENGTH OF
unsigned long int	PIC 9(9) USAGE IS BINARY

Table 11. Supported data types passed between C and COBOL by reference (Indirect) and from COBOL to C either by value (indirect) or by reference (indirect) without #pragma (continued)

C (Pointer to...)	COBOL (by content/by reference)
float	COMP-1
double	COMP-2
pointer to...	POINTER, ADDRESS OF
decimal	USAGE IS PACKED-DECIMAL
struct	Groups
type array[n]	Tables (OCCURS n TIMES)

Note:

1. You must specify a size for type array.
2. If the COBOL program receives **int** parameters from the C calling function that might have a value that is larger than that declared as the maximum size by the COBOL picture clause, the COBOL program must either be compiled with the TRUNC(BIN) compiler option or each binary data item that receives **int** parameters from C must be declared as USAGE IS COMP-5. Taking these actions will guarantee that truncation of high-order digits does not occur. For more information about the TRUNC(BIN) compiler option or about using COMP-5 data items, see the appropriate version of the programming guide in the Enterprise COBOL for z/OS library (http://www-01.ibm.com/support/docview.wss?uid=swg27036733).
3. COBOL turns on the high-order bit of the address of the last parameter when it is passed by reference. This can cause problems in the C program if it is using the address (since it will be treated as a negative number). If a C program does need to use the address of the last parameter, one of the following techniques can be used to bypass this problem:
   - If the COBOL program is an Enterprise COBOL program, instead of passing the parameter by reference, pass the address of the item by value. For example, use a call statement that looks like the following code:

     ```
     CALL "C" using by value address of C-PARM1
                    by value address of C-PARM2
     ```

   - If the COBOL program is not Enterprise COBOL, code needs to be added to mask out the high-order bit in the C routine. The sample code shows how to do this:

     ```
         #include <stdio.h>
         #include <string.h>
         void A1CC01BA(char* myString)
         {
           myString = (char*)((int)myString & 0x7fffffff);
           printf("My String: %s \n", myString);
           return;
         }
     ```

Handling function returns between C and COBOL

In COBOL, values can be returned to COBOL programs as COBOL returning variables from C functions using standard C function returns. This is the recommended approach for passing modified values back from a C function to a COBOL program.

Note: When a COBOL program calls a void C function, the RETURN-CODE special register contents will be unpredictable upon return from the call.

The following examples illustrate how to declare data types for using function returns in C to COBOL applications.

Declaration for C calling COBOL

Sample C usage

```
#include <stdio.h>
void cobrtn (int);

int main()
{
  int x,y;
  x=1;
  y=cobrtn(x);  /* x by value */
  printf("y=%i.",y);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 A PIC S9(9) USAGE IS BINARY.
01 B PIC S9(9) USAGE IS BINARY.

PROCEDURE DIVISION USING BY VALUE A
                   RETURNING B.
       COMPUTE B = A + 1
       GOBACK.
END PROGRAM COBRTN.
```

Declaration for COBOL calling C

COBOL program

```
LINKAGE SECTION.
01 P1 PIC S9(9) USAGE IS BINARY
01 P2 PIC S9(9) USAGE IS BINARY
CALL 'CFUNC' USING BY VALUE P1
     RETURNING P2.
```

C function

```
int CFUNC(int p1){
  int p2;
  p2=p1;
  return p2;
}
```

Passing data between C and COBOL with #pragma

Note:

The reference to COBOL with #pragma applies to VS COBOL II, COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL for z/OS.

When data is passed between C and COBOL with `#pragma linkage (COBOL)`, the C compiler generates the appropriate addressing code which introduces an extra level of indirection on the C side for non-pointer types. Pointers, however, are passed directly, meaning that for COBOL to receive a pointer to a C data type, C must pass a pointer to a pointer to the C data type. Conversely, if COBOL returns a pointer to a data type, C receives a pointer to a pointer to the data type.

Passing data by value (indirect) between C and COBOL

Copies of variables can be passed between C and COBOL routines. On return, the actual value of the variables remains unchanged regardless of how it may have been modified in the called routine.

Value arguments can be passed BY CONTENT from COBOL programs and received as C function parameters when declared with the appropriate base type. Conversely, C function arguments can be passed by value from C functions and received as COBOL parameters. The C compiler generates the appropriate addressing code required to access the parameter values; you can write your C function, which interoperates with COBOL, as if it were in a C-only environment. It can be moved to a C-only environment simply by removing the `#pragma linkage` directive. For example, if a C function called FROMCOB is to receive a parameter passed BY CONTENT of type int, the function prototype declaration would look like this:

```
void FROMCOB(int)
```

Table 12 shows the supported data types for passing by value (indirect).

Supported data types passed by value (indirect) between C and COBOL

Table 12 identifies the data types that can be passed by value (indirect) as parameters between C and COBOL.

Table 12. Supported data types passed by value (Indirect) with #pragma

C	COBOL
signed int, signed long int	PIC S9(9) USAGE IS BINARY
double	COMP-2
pointer to...	POINTER, ADDRESS OF
struct	Groups
type array[*n*]	Tables (OCCURS *n* TIMES)

Passing data by reference (indirect) between C and COBOL

A parameter can be passed by reference (indirect) between C and COBOL, which means the actual address of the argument is passed to the called function or procedure; any changes to the parameter made by the called routine can alter the original parameter passed by the calling routine.

To pass data by reference (indirect) from C to COBOL, the variables are passed by C as function arguments, which are pointers to a given type or the address of a given variable, and received as COBOL parameters. Conversely, to pass data by reference (indirect) from COBOL to C, the variables are passed from COBOL as BY REFERENCE arguments and received by a C function as pointers to a given type. For example, if a C function called FROMCOB is to receive a parameter passed by reference (indirect) of type int, the function prototype declaration would look like this:

```
void FROMCOB(int *)
```

The C function must dereference the pointer to access the actual value. If the value of the pointer is modified by the C function, as opposed to modifying the value that the pointer points to, the results on return to COBOL are unpredictable. Therefore, passing values by reference (indirect) from COBOL to C should be used with caution, and only in cases where the exact behavior of the C function is known.

Table 13 shows the supported data types for passing by reference (indirect).

Supported data types passed by reference (indirect) between C and COBOL

Table 13 identifies the data types that can be passed by reference (indirect) between C and COBOL.

Table 13. Supported data types passed by reference (indirect) with #pragma

C	COBOL
signed short int	PIC S9(4) USAGE IS BINARY
signed int, signed long int	PIC S9(9) USAGE IS BINARY

Table 13. Supported data types passed by reference (indirect) with #pragma (continued)

C	COBOL
float	COMP-1
double	COMP-2
pointer to...	POINTER, ADDRESS OF
decimal	USAGE IS PACKED-DECIMAL
struct	Groups
type array[n]	Tables (OCCURS n TIMES)

Passing strings between C and COBOL

C and COBOL have different string data types:

C strings
Logically unbounded length and are terminated by a NULL (the last byte of the string contains X'00')

COBOL PIC X(*n*)
Fixed-length string of characters of length *n*

You can pass strings between COBOL and C routines, but you must match what the routine interface demands with what is physically passed. Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM both have strings like previous COBOLs, as well as null-terminated literal strings like C.

Refer to “Sample ILC applications” on page 56 to see how string data is passed between C and COBOL.

Using aggregates

Aggregates (arrays, strings, or structures) are mapped differently by C and COBOL and are not automatically mapped. You must completely declare every byte in the structure to ensure that the layouts of structures passed between the two languages map to one another correctly. The XL C compile-time option AGGREGATE and the COBOL compiler option MAP provide a layout of structures to help you perform the mapping.

Data equivalents

This section describes how C and COBOL data types correspond to each other.

Equivalent data types for C to COBOL

The following examples illustrate how C and COBOL routines within a single ILC application might code the same data types.

Note: In the declarations that follow, examples showing the use of COBOL without #pragma apply only to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM. The examples showing COBOL with #pragma apply to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II.

One-byte character data without #pragma

Sample C usage

```
#include <stdio.h>
void cobrtn (char);

int main()
{
  char x;
  x='a';
  cobrtn(x);  /* x by value */
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC X.
PROCEDURE DIVISION USING BY VALUE X.
       DISPLAY X.
       GOBACK.
END PROGRAM COBRTN.
```

One-byte character data with #pragma

Sample C usage

```
#pragma linkage (cobrtn,COBOL)
#include <stdio.h>
void cobrtn (char*);

int main()
{
  char x;
  x='a';
  cobrtn(&x);    /* x by reference */
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC X.
PROCEDURE DIVISION USING X.
       DISPLAY X
       GOBACK.
END PROGRAM COBRTN.
```

16-bit signed binary Integer without #pragma

Sample C usage

```
#include <stdio.h>
void cobrtn (short int);

int main()
{
  short int x;
  x=5;
  cobrtn(x);  /* x by value */
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC S9(4) BINARY.
PROCEDURE DIVISION USING BY VALUE X.
       DISPLAY X.
       GOBACK.
END PROGRAM COBRTN.
```

16-bit signed binary integer with #pragma

Sample C usage

```
#pragma linkage(cobrtn,COBOL)
#include <stdio.h>
void cobrtn (short int*);

int main()
{
  short int x;
  x=5;
  cobrtn(&x);  /* x by reference */
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC S9(4) BINARY.
PROCEDURE DIVISON USING X.
      DISPLAY X
      GOBACK.
END PROGRAM COBRTN.
```

32-bit signed binary integer without #pragma

Sample C usage

```
#include <stdio.h>
void cobrtn (int);

int main()
{
  int x;
  x=5;
  cobrtn(x);  /* x by value */
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC S9(9) BINARY.
PROCEDURE DIVISION USING BY VALUE X.
      DISPLAY X.
      GOBACK.
END PROGRAM COBRTN.
```

32-bit signed binary integer with #pragma

Sample C usage

```
#pragma linkage(cobrtn,COBOL)
#include <stdio.h>
void cobrtn (int, int*);

int main()
{
  int x,y;
  x=5;
  y=6;
  cobrtn(x,&y);   /* x by value */
}         /* y by reference */
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC S9(9) BINARY.
01 Y PIC S9(9) BINARY.
PROCEDURE DIVISION USING X Y.
      DISPLAY X Y
      GOBACK.
END PROGRAM COBRTN.
```

Long floating-point number without #pragma

Sample C usage

```
#include <stdio.h>
void cobrtn (double);

int main()
{
  double x;
  x=3.14159265;
  cobrtn(x);    /* x by value */
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X COMP-2.
PROCEDURE DIVISION USING BY VALUE X
      DISPLAY Y.
      GOBACK.
END PROGRAM COBRTN.
```

Long floating-point number with #pragma

Sample C usage

```
#pragma linkage(cobrtn,COBOL)
#include <stdio.h>
void cobrtn (double, double*);

int main()
{
  double x,y;
  x=3.14159265;
  y=4.14159265;
  cobrtn(x,&y);    /* x by value */
}         /* y by reference */
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X COMP-2.
01 Y COMP-2.
PROCEDURE DIVISION USING X Y.
      DISPLAY X Y
      GOBACK.
END PROGRAM COBRTN.
```

Structure with #pragma

Sample C usage

```
#pragma linkage (cobrtn,COBOL)
#include <stdio.h>
struct stype {
  int s1;
  int s2;};
void cobrtn (struct stype,
             struct stype*);

int main()
{
  struct stype struc1, struc2;
  struc1.s1=1;
  struc1.s2=2;
  struc2.s1=3;
  struc2.s2=4;
  cobrtn(struc1,&struc2);
    /* struc1 by value */
}    /* struc2 by reference */
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 STRUC1.
    05 S11 PIC S9(9) BINARY.
    05 S12 PIC S9(9) BINARY.
01 STRUC2.
    05 S21 PIC S9(9) BINARY.
    05 S22 PIC S9(9) BINARY.
PROCEDURE DIVISION USING STRUC1 STRUC2.
    DISPLAY S11 S12 S21 S22
    GOBACK.
END PROGRAM COBRTN.
```

Array with #pragma

Sample C usage

```
#pragma linkage(cobrtn,COBOL)
#include <stdio.h>
void cobrtn (int array[2]);

int main()
{
  int array[2];
  array[0]=1;
  array[1]=2;
  cobrtn(array);
    /* array by reference */
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 ARRAY.
    05 ELE PIC S9(9) BINARY OCCURS 2.
PROCEDURE DIVISION USING ARRAY.
    DISPLAY ELE(1) ELE(2)
    GOBACK.
END PROGRAM COBRTN.
```

Fixed-length decimal data with #pragma

Sample C usage

```
#pragma linkage(cobrtn,COBOL)
#include <stdio.h>
#include <decimal.h>
void cobrtn (decimal(5,2)*);

int main()
{
  decimal(5,2) x;
  x=123.45d;
  cobrtn(&x);   /* x by reference */
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC 999V99 COMP-3.
PROCEDURE DIVISION USING X.
      DISPLAY X
      GOBACK.
END PROGRAM COBRTN.
```

Equivalent data types for COBOL to C

The following examples illustrate how COBOL to C routines within a single ILC application might code the same data types.

Note: In the declarations that follow, examples showing the use of COBOL without #pragma apply only to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM. The examples showing COBOL with #pragma apply to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II.

32-bit signed binary integer without #pragma

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X PIC S9(9) BINARY.
01 Y PIC S9(9) BINARY.
PROCEDURE DIVISION.
       MOVE 1 TO X.
* X BY VALUE ***
       CALL "CENTRY" USING BY VALUE X
                 RETURNING Y.
       GOBACK.
END PROGRAM COBRTN.
```

C function

```
#include <stdio.h>

int centry(int x)
{
  int y=2;
  printf("%d %d \n",x,y);
  return y;
}
```

32-bit signed binary integer with #pragma

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X PIC S9(9) BINARY.
01 Y PIC S9(9) BINARY.
PROCEDURE DIVISION.
       MOVE 1 TO X.
       MOVE 2 TO Y.
* X BY VALUE, Y BY REFERENCE ***
       CALL "CENTRY" USING BY CONTENT X
                 BY REFERENCE Y.
       GOBACK.
END PROGRAM COBRTN.
```

C function

```
#pragma linkage (centry,COBOL)
#include <stdio.h>

void centry (int x, int *y)
{
  printf("%d %d \n",x,*y);
  return;
}
```

Long floating-point number without #pragma

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X COMP-2.
01 Y COMP-2.
PROCEDURE DIVISION.
       MOVE 3.14159265 TO X.
* X BY VALUE ***
       CALL "CENTRY" USING BY VALUE X
                 RETURNING Y.
       GOBACK.
END PROGRAM COBRTN.
```

C function

```
#include <stdio.h>

double centry (double x)
{
  double y=4.14159265;
  printf("%f %f \n",x,y);
  return y;
}
```

Long floating-point number with #pragma

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X COMP-2.
01 Y COMP-2.
PROCEDURE DIVISION.
       MOVE 3.14159265 TO X.
       MOVE 4.14159265 TO Y.
* X BY VALUE, Y BY REFERENCE ***
       CALL "CENTRY" USING BY CONTENT X
                 BY REFERENCE Y.
       GOBACK.
END PROGRAM COBRTN.
```

C function

```
#pragma linkage (centry,COBOL)
#include <stdio.h>

void centry (double x, double *y)
{
printf("%f %f \n",x,*y);
return;
}
```

Structure without #pragma

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 STRUC1.
    05 S11 PIC S9(9) BINARY.
    05 S12 PIC S9(9) BINARY.
PROCEDURE DIVISION.
    CALL "CENTRY" RETURNING STRUC1.
    GOBACK.
END PROGRAM COBRTN.
```

C function

```
#include <stdio.h>
struct stype {
  int s1;
  int s2;  };
struct stype centry()
{
  struct stype struc2;
  struc2.s1=3;
  struc2.s2=4;
  return struc2;
}
```

Structure with #pragma

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 STRUC1.
    05 S11 PIC S9(9) BINARY VALUE 1.
    05 S12 PIC S9(9) BINARY VALUE 2.
01 STRUC2.
    05 S21 PIC S9(9) BINARY VALUE 3.
    05 S22 PIC S9(9) BINARY VALUE 4.
PROCEDURE DIVISION.
* STRUC1 BY VALUE**
* STRUC2 BY REFERENCE ***
    CALL "CENTRY" USING BY CONTENT STRUC1
                    BY REFERENCE STRUC2.
    GOBACK.
END PROGRAM COBRTN.
```

C function

```
#pragma linkage(centry,COBOL)
#include <stdio.h>
struct stype {
  int s1;
  int s2;  };
void centry (struct stype struc1,
    struct stype *struc2)
{
  printf("%d %d %d %d \n",struc1.s1,
    struc1.s2,struc2->s1,struc2->s2);
  return;
}
```

Fixed-length decimal data without #pragma

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X PIC 999V99 COMP-3.
PROCEDURE DIVISION.
    CALL "CENTRY" RETURNING X.
    GOBACK.
END PROGRAM COBRTN.
```

C function

```
#include <stdio.h>
#include <decimal.h>

decimal(5,2) centry()
{
  decimal(5,2) x=123.45;
  printf("%D(5,2)\n",x);
  return x;
}
```

Fixed-length decimal data with #pragma

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X PIC 999V99 COMP-3.
PROCEDURE DIVISION.
    MOVE 123.45 TO X.
* X BY REFERENCE ***
    CALL "CENTRY" USING X.
    GOBACK.
END PROGRAM COBRTN.
```

C function

```
#pragma linkage (centry,COBOL)
#include <stdio.h>
#include <decimal.h>

void centry (decimal(5,2) x)
{
  printf("%D(5,2)\n",x);
  return;
}
```

Name scope of external data

In programming languages, the *name scope* is defined as the portion of an application within which a particular declaration applies or is known. The name scope of external data differs between C and COBOL. The scope of external data under C is the load module; under COBOL, it is the enclave (or run unit). Figure 7 on page 49 and Figure 8 on page 49 illustrate these differences.

Because the name scope for C and COBOL is different, external variables do not map between C and COBOL; external variables with the same name are considered separate between C and COBOL.

If your application relies on the separation of external data, do **not** give the data the same name in both languages within a single application. If you give the data in each load module a different name, you can change the language mix in the application later, and your application still behaves as you expect it to.

DLL considerations

In DLL code, external variables are mapped across the load module boundary. DLLs are shared at the enclave level. Therefore, a single copy of a DLL applies to all modules in an enclave, regardless of whether the DLL is loaded implicitly (through a reference to a function or variable) or explicitly (through `dllload()`). See *z/OS Language Environment Programming Guide* for information about building and managing DLL code in your applications.

COBOL data declared with the EXTERNAL attribute are independent of DLL support. These data items are managed by the COBOL runtime, and are accessible by name from any COBOL program in the run-unit that declares them, regardless of whether the programs are in DLLs or not.

In particular, the facilities for exporting and importing external variables from DLLs implemented in z/OS C/C++ do not apply to COBOL external data. Hence C/C++ external data and COBOL external data are always in separate name spaces, regardless of DLL considerations.

For C/C++, non-DLL applications have external data which is only shared within the load module.

However, for DLL applications, C/C++ external data is now (optionally) accessible to all C/C++ routines in the enclave.

Name scope of external data in a C application

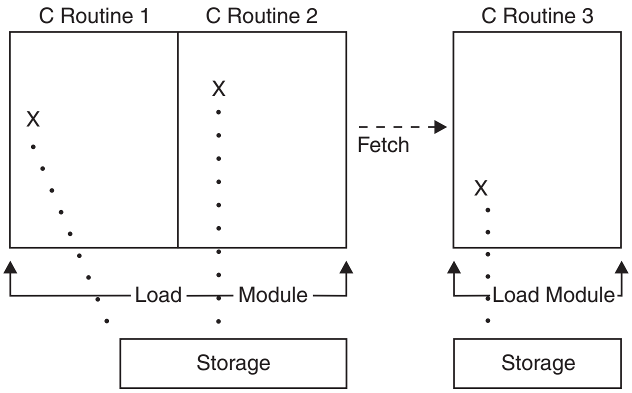


Figure 7. Name scope of external variables for C fetch

In Figure 7, external data declared in C Routine 1 maps to that declared in C Routine 2 in the same load module. When a fetch to C Routine 3 in another load module is made, the external data does not map, because the name scope of external data in C is the load module.

Name scope of external data in a COBOL run unit

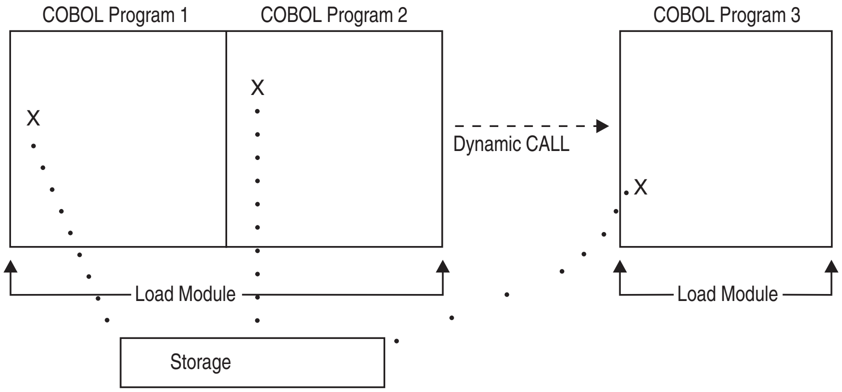


Figure 8. Name scope of external variables for COBOL dynamic call

In Figure 8, Routines 1, 2, and 3 comprise a COBOL run unit. External data declared in COBOL Program 1 maps to that declared in COBOL Program 2 in the same load module. When a dynamic CALL to COBOL Program 3 in another load module is made, the external data still maps, because the name scope of external data in COBOL is the enclave.

Name space of external data

In programming languages, the *name space* is defined as the portion of a load module within which a particular declaration applies or is known. Like the name scope, the name space of external data differs between C and COBOL.

Figure 9 and Figure 10 illustrate that within the same load module, the name space of COBOL programs is the same. However, the name spaces of a COBOL program and a C routine within the same load module are not the same. If you give external data the same name in both languages, an incompatibility in external data mapping can occur.

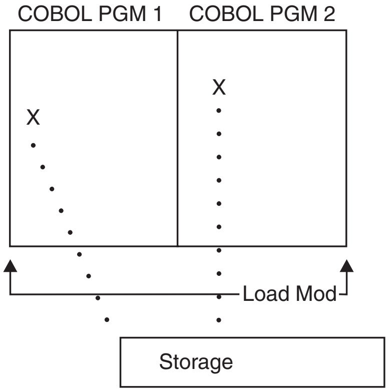


Figure 9. Name space of external data for COBOL static call to COBOL

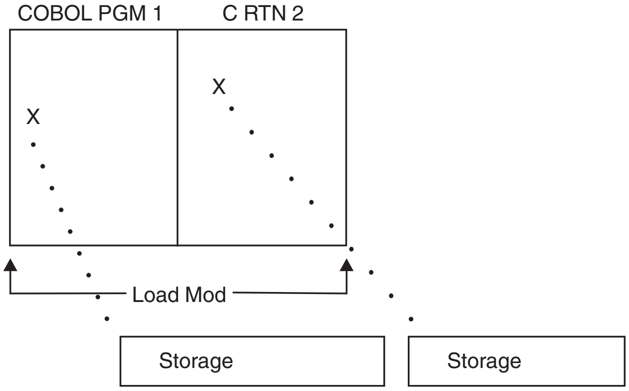


Figure 10. Name space of external data in COBOL static call to COBOL

Directing output in ILC applications

C and COBOL do not share files, except the Language Environment message file (the ddname specified in the Language Environment MSGFILE runtime option). You must manage all other files to ensure that no conflicts arise. Performing I/O operations on the same ddname might cause abnormal termination.

Under C, runtime messages and other related output are directed to the default MSGFILE ddname. `stderr` output is also by default directed to the MSGFILE ddname. `stdout` is not by default directed to the MSGFILE ddname, but can be redirected to do so. Also, output from `printf` can be interspersed with output from the COBOL DISPLAY statement and output from Language Environment by redirecting `stdout` to `stderr` (for example, passing `1>&2` as a command-line parameter).

For more information about how to redirect C output, see *z/OS XL C/C++ Programming Guide*.

Under COBOL, runtime messages and other related output are directed to the MSGFILE ddname. Output from COBOL DISPLAY UPON SYSOUT is directed to the default MSGFILE ddname only when the OUTDD compiler option ddname matches the MSGFILE ddname; this applies to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II.

Interspersing output when C is the main routine

To intersperse output from C and COBOL when the main routine is in C, compile your COBOL program with the default OUTDD (OUTDD=SYSOUT) if you are using the default MSGFILE ddname. If you have overridden the default MSGFILE ddname, you must compile your COBOL program using an OUTDD that specifies the same name as the MSGFILE. You must redirect `stdout` to `stderr` for the main C routine.

Interspersing output when COBOL is the main routine

To intersperse output from C and COBOL when the main program is in COBOL, compile your COBOL program with the default OUTDD (OUTDD=SYSOUT). In the C routine, redirect `stdout` to `stderr` by placing the line `stdout=stderr` in your program. If you have overridden the default MSGFILE ddname, you must compile your COBOL program using an OUTDD ddname that specifies the same name as the MSGFILE.

C POSIX multithreading

COBOL programs can run in more than one thread as long as all of the COBOL programs used in the enclave are enabled for multithreading.

A COBOL program is enabled for multithreading when it is compiled with the Enterprise COBOL for z/OS compiler using the THREAD compiler option.

A COBOL program is not enabled for multithreading when it is compiled with any of the following compilers:

- Enterprise COBOL for z/OS (with the NOTHREAD compiler option)
- COBOL for OS/390 & VM
- COBOL for MVS & VM
- COBOL/370
- VS COBOL II

POSIX-conforming C/C++ applications can communicate with COBOL programs enabled for multithreading on one or more threads.

POSIX-conforming C/C++ applications can communicate with COBOL programs *not* enabled for multithreading; however, there are limitations in the support:

- COBOL can only be used in one thread.
- When COBOL is run in a thread that is not the initial process thread (IPT), there is no support to dynamically call a program that contains C/C++ writeable static.

- When COBOL is run in a thread that is not the initial thread and the thread terminates, COBOL frees the resources it acquired except for any DLLs it loaded. The freeing of resources includes deleting any load modules it loaded for dynamic call processing.

POSIX-conforming C/370 applications can communicate with assembler routines on any thread when the assembler routines use the CEEENTRY/CEETERM macros or the EDCPRLG/EDCEPIL macros provided by C/C++.

C to COBOL condition handling

This section provides two scenarios of condition handling behavior in a C to COBOL ILC application. If an exception occurs in a C routine, the set of possible actions is as described in "Exception occurs in C." If an exception occurs in a COBOL program, the set of possible actions is as described in "Exception occurs in COBOL" on page 54.

Keep in mind that some conditions can be handled only by the HLL of the routine in which the exception occurred. For example, in a COBOL program, a statement can have a clause that adds condition handling to a verb, such as the ON SIZE ERROR clause of a COBOL DIVIDE verb (which includes the logical equivalent of a divide-by-zero condition). This type of condition is handled completely within COBOL.

For a detailed description of Language Environment condition handling, see *z/OS Language Environment Programming Guide*.

Enclave-terminating language constructs

Enclaves can be terminated for reasons other than an unhandled condition of severity 2 or greater. In Language Environment ILC, you can issue an HLL language construct to terminate a C to COBOL enclave from either a COBOL or C routine.

C

Examples of C language constructs that terminate the enclave are: `kill()`, `abort()`, `raise(SIGTERM)`, `raise(SIGABND)`, and `exit()`. When you use a C language construct to terminate an enclave, the T_I_S (Termination Imminent Due to STOP) condition is raised. After T_I_S has been processed and all user code has been removed from the stack, the C atexit list is honored.

COBOL

Examples of C language constructs that terminate the enclave are: `kill()`, `abort()`, `raise(SIGTERM)`, `raise(SIGABND)`, and `exit()`. When you use a C language construct to terminate an enclave, the T_I_S (Termination Imminent Due to STOP) condition is raised. After T_I_S has been processed and all user code has been removed from the stack, the C atexit list is honored.

Exception occurs in C

In this scenario, a COBOL main program invokes a C subroutine and an exception occurs in the C subroutine. Refer to Figure 11 on page 53 throughout the following discussion.

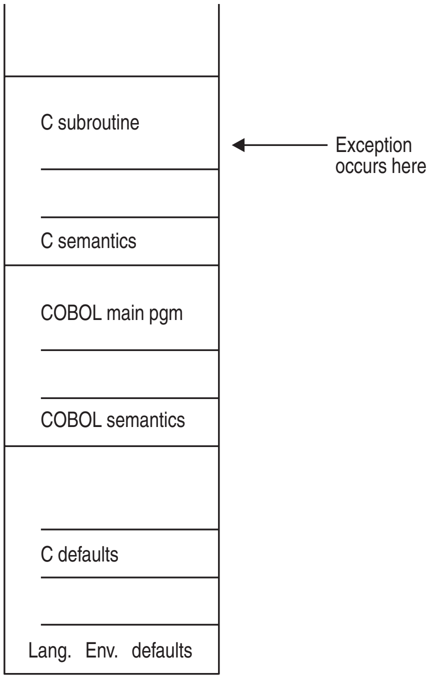


Figure 11. Stack contents when the exception occurs in C

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, C determines whether the exception in the C routine should be enabled and treated as a condition. If any of the following are true, the exception is ignored, and processing continues at the next sequential instruction after which the exception occurred:
   - You specified `SIG_IGN` for the exception in a call to `signal()`.

     Note: The system or user abend corresponding to the `signal(SIGABND)` or the Language Environment message 3250 is not ignored. The enclave is terminated.
   - The exception is one of those listed as masked in Table 63 on page 249, and you have not enabled it using the CEE3SPM callable service.
   - You did not specify any action, but the default action for the condition is `SIG_IGN` (see Table 63 on page 249).
   - You are running under CICS and a CICS handler is pending.

   If you did none of these things, the condition is enabled and processed as a condition.
2. If a user-written condition handler has been registered using CEEHDLR on the C stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, there is not a user-written condition handler registered for the condition, so the condition is percolated.
3. The global error table is now examined for signal handlers that have been registered for the condition.

   If there is a signal handler registered for the condition, it is given control. If it issues a resume or a call to `longjmp()`, the condition handling step ends.

Processing resumes in the routine to which the resume cursor points. You must be careful when issuing a `longjmp()` in an application that contains a COBOL program; see "CEEMRCR and COBOL" on page 56 for details.

In this example no C signal handler is registered for the condition, so the condition is percolated.

4. The condition is still unhandled. If C does not recognize the condition, or if the C default action (listed in Table 63 on page 249) is to terminate, the condition is percolated.
5. If a user-written condition handler has been registered using CEEHDLR on the COBOL stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points. You must be careful when moving the resume cursor in an application that contains a COBOL program; see "CEEMRCR and COBOL" on page 56 for details.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
6. If the condition is of severity 0 or 1, the Language Environment default actions take place, as described in Table 62 on page 249.
7. If the condition is of severity 2 or above, Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition) and redrive the stack. Condition handling now enters the termination imminent step.
8. If, on the second pass of the stack, no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Exception occurs in COBOL

In this scenario, a C main routine invokes a COBOL subroutine and an exception occurs in the COBOL subroutine. Refer to Figure 12 on page 55 throughout the following discussion.

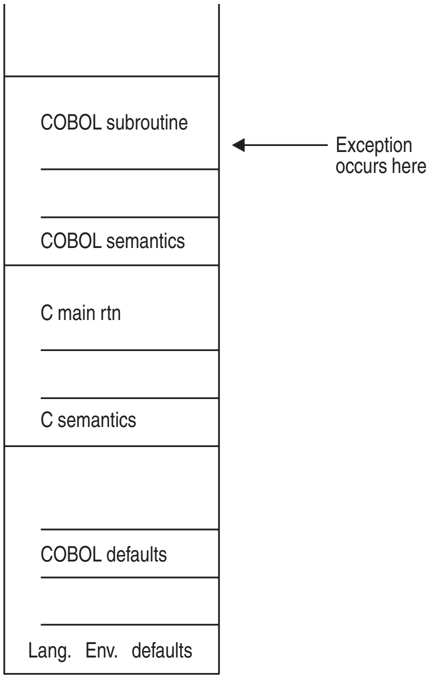


Figure 12. Stack contents when the COBOL exception occurs

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, COBOL determines if the exception should be ignored or handled as a condition.
   - If the exception is to be ignored, control is returned to the next sequential instruction after where the exception occurred.
   - If the exception is to be enabled and processed as a condition, the condition handling step takes place.
2. If a user-written condition handler has been registered using CEEHDLR on the COBOL stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated. You must be careful when moving the resume cursor in an application that contains a COBOL program; see CEEMRCR and COBOL for details.
3. If a user-written condition handler has been registered for the condition (as specified in the global error table) using CEEHDLR on the C stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
4. If a C signal handler has been registered for the condition, it is given control. If it moves the resume cursor or issues a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points. You must be careful when moving the resume cursor in an application that contains a COBOL program; see “CEEMRCR and COBOL” on page 56 for details.

In this example, no C signal handler is registered for the condition, so the condition is percolated.

5. If the condition has a Facility_ID of IGZ, the condition is COBOL-specific. The COBOL default actions for the condition take place. If COBOL does not recognize the condition, condition handling continues.
6. If the condition is of severity 0 or 1, Language Environment default actions take place, as described in Table 62 on page 249.
7. If the condition is of severity 2 or above, Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition) and redrive the stack. Condition handling now enters the termination imminent step.
8. If, on the second pass of the stack, no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

CEEMRCR and COBOL

When you make a call to CEEMRCR to move the resume cursor, or issue a call to `longjmp()`, and a COBOL routine is removed from the stack, the COBOL routine can be re-entered via another call path.

If the terminated routine is one of the following, the routine remains active. If the COBOL program does not specify RECURSIVE in the PROGRAM-ID, a recursion error is raised if you attempt to enter the routine again.

- A VS COBOL II, COBOL/370, COBOL for MVS & VM, or COBOL for OS/390 & VM program compiled with the CMPR2 option
- A VS COBOL II program that does not contain nested programs and is compiled with the NOCMPR2 compiler option
- A COBOL/370, COBOL for MVS & VM, or COBOL for OS/390 & VM program compiled with the NOCMPR2 option that does not use the combination of the INITIAL attribute, nested routines, and file processing in the same compilation unit.
- An Enterprise COBOL for z/OS program that does not use the combination of the INITIAL attribute, nested routines, and file processing in the same compilation unit.

In addition, if the routine is a COBOL routine with the INITIAL attribute and containing files, the files are closed. (COBOL supports VSAM and QSAM files and these files are closed.)

Sample ILC applications

Figure 13 on page 57, Figure 14 on page 58, and Figure 15 on page 58 contain an example of an ILC application. The C C1 routine dynamically calls the COBOL CBL1 program. CBL1 statically calls C routine C2.

```
 /*Module/File Name:  EDCCCB  */
 /*******************************************************************/
 /*    Illustration of Interlanguage Communication between C/MVS    */
 /*    and COBOL.  All parameters passed by reference.              */
 /*                                                                 */
 /*             C1 ==========> CBL1 ---------> C2                   */
 /*                  dynamic         static                         */
 /*                   call            call                          */
 /*******************************************************************/
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>

 typedef void CBLrtn();
 #pragma linkage(CBLrtn,COBOL)
 CBLrtn *rtn_ptr;

 /********************* C1 routine  example *************************/

 main()
 {
   signed short int short_int  = 2;
   signed long int  long_int   = 4;
   double           floatpt    = 8.0;
   char             string[80];

   fprintf(stderr,"main  STARTED\n");
   rtn_ptr = (CBLrtn *) fetch("CBL1");  /* get the address of CBL1 */

   if ( rtn_ptr == 0 )                  /* check result of fetch   */
      printf("fetch failed\n");
   else                                 /* call to CBL1            */

     rtn_ptr ( short_int, long_int, floatpt, string );

  fprintf(stderr,"main  ENDED\n");
} /* end of main */
```

Figure 13. Dynamic call from C to COBOL program

```
CBL LIB,QUOTE
      *Module/File Name:  IGZTILCC

       IDENTIFICATION DIVISION.
       PROGRAM-ID.  CBL1.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77  var1               PIC S9(9) BINARY VALUE 5.
       01  msg-string       PIC X(80).

       LINKAGE SECTION.
       77  int2                PIC S9(4) BINARY.
       77  int4                PIC S9(9) BINARY.
       77  float              COMP-2.
       77  char-string      PIC X(80).

       PROCEDURE DIVISION USING int2 int4
                            float char-string.

          DISPLAY "CBL1 STARTED".

             IF (int2 NOT = 2) THEN
               DISPLAY "INT2 NOT = 2".

             IF (int4 NOT = 4) THEN
               DISPLAY "INT4 NOT = 4".

             IF (float NOT = 8.0) THEN
               DISPLAY "FLOAT NOT = 8".
       ********************************************************
       * Place null-character-terminated string in parameter  *
       ********************************************************

             STRING "PASSED CHARACTER STRING",
                         LOW-VALUE
                         DELIMITED BY SIZE INTO msg-string

       ********************************************************
       * Make a static CALL to a C routine                    *
       ********************************************************
             CALL "C2" USING var1, msg-string.

             DISPLAY "CBL1 ENDED".
             GOBACK.
```

Figure 14. Static call from COBOL to C routine

```
 /*Module/File Name:  EDCCCB2 */

 /*    C2 routine example     */

 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>

 #pragma linkage(C2,cobol)
 void C2( int*, char*);

 void C2( int* num, char* strng)
 {
   printf(stderr,"num is %d, string
              is %s\n",*num, strng);
 }  /* end of main */
```

Figure 15. Statically called C routine

Chapter 5. Communicating between C++ and COBOL

This topic describes Language Environment's support for C++ to COBOL ILC applications. If you are running a C++ to COBOL ILC application under CICS, you should also consult Chapter 15, "ILC under CICS," on page 241.

General facts about C++ to COBOL ILC

- Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM can use the C-style linkage and non-void function return values to pass and receive parameters by value.
- Language Environment does not support the passing of return codes between C++ and COBOL routines in an ILC application.
- There is no ILC support between AMODE 31 and AMODE 64 applications. COBOL does not support AMODE 64.

Preparing for ILC

This section describes topics you need to consider before writing a C++ to COBOL ILC application. For help in determining how different versions of HLLs work together, refer to the migration guides for the HLLs you plan to use.

Language Environment ILC support

Language Environment provides ILC support between the following combinations of C++ and COBOL:

Table 14. Supported languages for Language Environment ILC

HLL pair	C++	COBOL
C++ to COBOL	• IBM C/C++ for MVS/ESA • z/OS XL C/C++ Compilers	• VS COBOL II Version 1 Release 3 and later • COBOL/370 Release 1 • COBOL for MVS & VM Release 2 • COBOL for OS/390 & VM • Enterprise COBOL for z/OS

Note: The considerations for ILC support described in the table apply to all versions of C++ and COBOL listed, except where otherwise noted. COBOL refers to VS COBOL II, COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL for z/OS.

Compiling considerations

If the C++ library is installed above the 16M line, compile your COBOL program using the RENT compiler option.

Determining the main routine

In Language Environment, only one routine can be the main routine; no other routine in the enclave can use syntax that indicates it is main. If you write the main routine in C++, you must use language syntax to identify the routine as the main routine. If you use COBOL as the first program in the enclave that is to gain control, the program is effectively designated main by being the first to run.

Table 15 on page 60 describes how C++ and COBOL identify the main routine.

Table 15. How C++ and COBOL main routines are determined

Language	When determined	Explanation
C++	Compile time	Determined in the C++ source file by declaring a C++ function named main.
COBOL	Run time	Determined dynamically. If it is the first program to run, it is a main program.

An entry point is defined for each supported HLL. Table 16 identifies the desired entry point. The table assumes that your code has been compiled using the Language Environment-conforming compilers.

Table 16. Determining the entry point

HLL	Main entry point	Fetched entry point
C++	CEESTART	Entry point specified as #pragma fetchable, or explicitly named during link-edit
COBOL	Name of the first object program to get control in the object module	Program name

COBOL and C++ routines that make up an ILC application are executed together in a single run unit (the equivalent of a Language Environment enclave). However, unlike in earlier versions of COBOL (VS COBOL II and OS/VS COBOL), the first COBOL program in a run unit is no longer necessarily considered the main routine. If the first COBOL program is not the first routine in the enclave to run, it is considered a subroutine in the Language Environment enclave.

Declaring C++ to COBOL ILC

If a C++ function invokes a COBOL program or a COBOL program invokes a C++ function, all entry declarations are contained solely within the C++ source. No special declaration is required within the COBOL program.

Declarations for extern "C" linkage

With either Enterprise COBOL for z/OS, COBOL for OS/390 & VM, or COBOL for MVS & VM, you can pass and receive parameters using “C-style” linkage and non-void function return values. To use this new function, you must include in your C++ code an `extern "C"` linkage specification instead of the `extern "COBOL"` linkage specification.

The `extern "C"` linkage specification has the following format:

```
extern "C" {declaration}
```

where *declaration* is a valid C++ prototype of the COBOL program being called by C++, or the C++ routine being called by COBOL

Note: The reference to COBOL in the following declarations applies only to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM.

Declaration for C++ calling COBOL (extern "C"):

C++ function

```
extern "C" {
void CBLRTN( double p2 );
}

main() {
  double myval = 5;

  CBLRTN(myval);
  printf("myval=%f\n",myval);

}
```

COBOL program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBLRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X COMP-2.
PROCEDURE DIVISION USING BY VALUE X.
       DISPLAY X
       GOBACK.
END PROGRAM CBLRTN.
```

Declaration for COBOL calling C++ (extern "C"):

COBOL program

```
ID DIVISION.
PROGRAM-ID. COBPROG.
WORKING-STORAGE SECTION.
01 P1 PIC S9(9) USAGE IS BINARY.
PROCEDURE DIVISION.
CALL 'CPLUSF' USING BY VALUE P1.
GOBACK.
```

C++ function

```
extern "C" {
  void CPLUSF( int p1 );
}
void CPLUSF( int parm ) {
  printf("parm=%d\n");
}
```

Declarations for extern "COBOL" linkage

For C++ to COBOL ILC, the C++ `extern "COBOL"` linkage specification lets the XL C++ compiler generate parameter lists for or accept them from COBOL.

The `extern "COBOL"` linkage specification has the following format:

```
extern "COBOL" { declaration }
```

declaration is a valid C++ prototype of the COBOL program being called by C++, or the C++ routine being called by COBOL.

Note: The reference to COBOL in the following declarations applies to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II.

Declaration for C++ calling COBOL (extern "COBOL"):

C++ function

```
extern "COBOL" {
  void CBLRTN( double p2 );
}

main() {
  double myval = 5;

  CBLRTN(myval);
  printf("myval=%f\n",myval);

}
```

COBOL program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X COMP-2.
PROCEDURE DIVISION USING BY REFERENCE X.
     DISPLAY X
     GOBACK.
END PROGRAM COBRTN.
```

Declaration for COBOL calling C++ (extern "COBOL"):

COBOL program

```
ID DIVISION.
PROGRAM-ID. COBPROG.
WORKING-STORAGE SECTION.
01 P1 PIC S9(9) USAGE IS BINARY.
PROCEDURE DIVISION.
CALL 'CPLUSF' USING BY REFERENCE P1.
GOBACK.
```

C++ function

```
extern "COBOL" {
  void CPLUSF( int p1 );
}
void CPLUSF( int parm ) {
  printf("parm=%d\n");
}
```

Building a reentrant C++ to COBOL application

By default, the XL C++ compiler generates object modules with constructed reentrancy. Therefore, linking a C++ to COBOL application is a two-step process:

1. Prelink all C++ and COBOL text decks together using the prelinker.
2. Link the text deck generated by the prelinker to create your module.

Calling between C++ and COBOL

The following tables describe the types of calls that are supported between C++ and COBOL when running with Language Environment.

- Table 17 shows which calls are supported from COBOL programs to C++ routines.
- Table 18 on page 63 shows which calls are supported from C++ routines to COBOL programs.

Within a given enclave containing ILC between COBOL and C++, the application may be structured in one of the following ways:

- As a single statically-bound load module.
- As a DLL application, with multiple DLL load modules, each containing DLL-enabled COBOL and C++ routines.
- As an application using multiple load modules and using COBOL dynamic CALLs. In this case, all instances of C++ ILC must be statically-bound within the main load module. The only exception would be if `extern "COBOL"` or `extern "C"` is specified for all C++ functions called by COBOL that are in dynamically called load modules that do not contain any DLL routines that export functions or variables.
- As an application using multiple load modules and using fetch.

Table 17. Support for calls from COBOL to C++

		Target		
Caller	**Call type**	**C++ with extern "COBOL"**	**C++ with extern "C"**	**C++ without extern**
VS COBOL II (1)	static	Yes with restrictions (2)	No	No
COBOL/370 (1)	static	Yes	No	No
COBOL (5) compiled with NODLL	static	Yes	Yes	No
VS COBOL II (1)	dynamic	No	No	No
COBOL/370 (1)	dynamic	No	No	No
COBOL (5) compiled with NODLL	dynamic	Yes (3)	Yes (3)	No

Table 17. Support for calls from COBOL to C++ (continued)

		Target		
Caller	**Call type**	**C++ with extern "COBOL"**	**C++ with extern "C"**	**C++ without extern**
COBOL (6) compiled with DLL	CALL "literal" to a function within the module	Yes	Yes	No
COBOL (6) compiled with DLL	CALL "literal" to a function exported from a DLL	Yes (4)	Yes (4)	No
COBOL (6) compiled with DLL	CALL identifier to a function exported from a DLL	Yes (4)	Yes (4)	No

Note:

1. When the caller is VS COBOL II or COBOL/370, all calls must be to *void* functions.
2. Static calls are supported from VS COBOL II to C++ with extern "COBOL" in the following cases:
   - The call is done in the main load module.
   - The call is done in a load module whose entry point is a Language Environment-conforming program or routine that was called using COBOL dynamic call.
   - The call is done in a module that was called using fetch.
   - The call is done in a DLL that was called using DLL linkage.
3. Dynamically called load modules cannot contain any DLL routines that export functions or variables. Use the binder DYNAM(NO) control statement to prevent marking the load module as a DLL.

   The XL C/C++ compiler may mark certain internal symbols as exported, resulting in the C++ executable being a DLL. To prevent the XL C++ compiler from doing this, use the NOEXPORT(NOSYS) compiler option.
4. In this case, the C++ code can also be compiled with the XPLINK option. The XPLINK C++ code must reside in a separate module from the COBOL caller.
5. COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL for z/OS
6. COBOL for OS/390 & VM or Enterprise COBOL for z/OS

Table 18. Support for calls from C++ to COBOL

		Target		
Caller	**Call type**	**VS COBOL II or COBOL/370**	**COBOL (4) compiled with NODLL**	**COBOL (5) compiled with DLL**
C++ with extern "COBOL"	function is within the module	Yes (1)	Yes	Yes
C++ with extern "C"	function is within the module	No	Yes	Yes
C++ without extern	function is within the module	No	Yes	No
C++ with extern "COBOL"	fetch	Yes (1, 2)	Yes (2, 3)	No
C++ with extern "C"	fetch	No	Yes (2, 3)	No
C++ without extern	fetch	No	No	No
C++ with extern "COBOL"	function is exported from a DLL	No	No	Yes (3)
C++ with extern "C"	function is exported from a DLL	No	No	Yes (3)

Table 18. Support for calls from C++ to COBOL (continued)

		Target		
Caller	**Call type**	**VS COBOL II or COBOL/370**	**COBOL (4) compiled with NODLL**	**COBOL (5) compiled with DLL**
C++ without extern	function is exported from a DLL	No	No	No

Note:

1. When the target of the call is VS COBOL II or COBOL/370, the called COBOL program must be declared as a *void* function.
2. Fetched load modules cannot contain any DLL routines that export functions or variables.
3. In this case, the C++ code can also be compiled with the XPLINK option. The XPLINK C++ code must reside in a separate module from the COBOL caller.
4. COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL for z/OS
5. COBOL for OS/390 & VM or Enterprise COBOL for z/OS

Passing data between C++ and COBOL

In VS COBOL II and COBOL/370, you can pass parameters two ways:

By reference (indirect)
COBOL BY REFERENCE

By value (indirect)
COBOL BY CONTENT

In Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM, you can pass parameters three ways:

By reference (indirect)
COBOL BY REFERENCE

By value (indirect)
COBOL BY CONTENT

By value (direct)
COBOL BY VALUE

Under Language Environment, the term *by value* means that a temporary copy of the argument is passed to the called function or procedure. Any changes to the parameter made by the called routine will not alter the original parameter passed by the calling routine. Under Language Environment, the term *by reference* means that the actual address of the argument is passed. Any changes to the parameter made by the called routine can alter the original parameter passed by the calling routine.

Further, the term *direct* means that the argument is passed in the parameter list. The term *indirect* means that a pointer to the argument is passed in the parameter list.

There are two ways to pass data between C++ and COBOL; one way uses `extern "C"` in the C linkage, and the other uses `extern "COBOL"`. Both methods are discussed separately in the following sections.

Passing data between C++ and COBOL with extern "C"

Note: The reference to COBOL with `extern "C"` applies only to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM.

Passing data by value between C++ and COBOL

Copies of variables can be passed between C++ and COBOL. On return, the actual value of the variable remains unchanged, regardless of how it may have been modified in the called routine.

To pass data by value (direct) from C++ to COBOL, the variables are passed by C++ as arguments on a function call and received by COBOL as BY VALUE parameters. Conversely, to pass data by value (direct) from COBOL to C++, the variables are passed by COBOL as BY VALUE arguments and received by C++ as function parameters. In all cases, the variable must be declared in C++ and COBOL with compatible base data types. For example, if a C++ function called FROMCOB is to receive a parameter passed by value (direct) of type int, the function prototype declaration would look like this:

```
void FROMCOB(int)
```

Data can also be passed by value (indirect) from COBOL to C++. In this case, the variable is passed as a BY CONTENT argument and received by C++ as a pointer to the given type. For example, if a C++ function called FROMCOB is to receive a parameter passed by value (indirect) of type int, the function prototype declaration would look like this:

```
void FROMCOB(int *)
```

The C++ function must dereference the pointer to access the actual value. If the value of the pointer is modified by the C++ function, as opposed to modifying the value that the pointer points to, the results on return to COBOL are unpredictable. Thus, passing values by value (indirect) from COBOL to C++ should be used with caution, and only in cases where the exact behavior of the C++ function is known.

Table 19 on page 66 shows the supported data types for passing by value (direct) and Table 20 on page 67 shows the supported data types for passing by value (indirect).

Passing data by reference (indirect) between C++ and COBOL

A parameter can be passed by reference (indirect) between C++ and COBOL. By reference (indirect) means that the actual address of the argument is passed to the called function or procedure; any changes to the parameter made by the called routine can alter the original parameter passed by the calling routine.

To pass data by reference (indirect) from C++ to COBOL, the variables are passed by C++ as function arguments, which are pointers to a given type or the address of a given variable, and received by COBOL as BY REFERENCE parameters. Conversely, to pass data by reference (indirect) from COBOL to C++, the variables are passed by COBOL as BY REFERENCE arguments and received by C++ function as pointers to a given type.

The C++ function must dereference the pointer to access the actual value. If the value of the pointer is modified by the C++ function, as opposed to modifying the value that the pointer points to, the results on return to COBOL are unpredictable. Thus, passing values by reference (indirect) from COBOL to C++ should be used with caution, and only in cases where the exact behavior of the C++ function is known.

Table 20 on page 67 shows the supported data types for passing by reference (indirect).

Handling function returns between C++ and COBOL

In COBOL, values can be returned to COBOL programs as COBOL returning variables from C++ functions using standard C++ function returns. This is the recommended approach for passing modified values back from a C++ function to a COBOL procedure.

Note: In the declarations that follow, COBOL refers to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM.

The following examples illustrate how to declare data types for using function returns in C++ to COBOL ILC applications.

Declaration for COBOL calling C++:

COBOL program

```
LINKAGE SECTION.
01 P1 PIC S9(9) USAGE IS BINARY.
01 P2 PIC S9(9) USAGE IS BINARY.
CALL 'CFUNC' USING BY VALUE P1
     RETURNING P2.
```

C++ function

```
int CFUNC(int p1){
  int p2;
  p2=p1;
  return p2;
}
```

Declaration for C++ calling COBOL:

Sample C usage

```
#include <stdio.h>
extern "C" {void cobrtn (int);};

int main()
{
  int x,y;
  x=1;
  y=cobrtn(x);  /* x by value */
  printf("y=%i.",y);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 A PIC S9(9) USAGE IS BINARY.
01 B PIC S9(9) USAGE IS BINARY.

PROCEDURE DIVISION USING BY VALUE A
                   RETURNING B.
       COMPUTE B = A + 1
       GOBACK.
END PROGRAM COBRTN.
```

Supported data types passed by value (direct) between C++ and COBOL

Table 19 identifies the data types that can be passed by value (direct) as parameters between C++ and COBOL applications.

Table 19. Supported data types passed by value (direct) with extern "C"

C++	COBOL (by value)
char	PIC X, PIC A
signed short int	PIC S9(4) USAGE IS BINARY
unsigned short int	PIC 9(4) USAGE IS BINARY
signed int, signed long int	PIC S9(9) USAGE IS BINARY
unsigned int	PIC 9(9) USAGE IS BINARY, LENGTH OF
unsigned long int	PIC 9(9) USAGE IS BINARY
float	COMP-1
double	COMP-2
pointer to...	POINTER, ADDRESS OF

Note:

1. If the COBOL program receives **int** parameters from the C calling function that might have a value that is larger than that declared as the maximum size by the COBOL picture clause, the COBOL program must either be compiled with the TRUNC(BIN) compiler option or each binary data item that receives **int** parameters from C must be declared as USAGE IS COMP-5. Taking these actions will guarantee that truncation of high-order digits does not occur. For more information about the TRUNC(BIN) compiler option or about using COMP-5 data items, see the appropriate version of the programming guide in the Enterprise COBOL for z/OS library (http://www-01.ibm.com/support/docview.wss?uid=swg27036733).

Supported data types passed by value (indirect) or by reference (indirect) between C++ and COBOL

Table 20 identifies the data types that can be passed by value (indirect) or by reference (indirect) between C++ and COBOL applications.

Table 20. Supported data types passed by value (indirect) or by reference (indirect) with extern "C"

C++ (Pointer to...)	COBOL (by content/by reference)
char	PIC X, PIC A
signed short int	PIC S9(4) USAGE IS BINARY
unsigned short int	PIC 9(4) USAGE IS BINARY
signed int, signed log int	PIC S9(9) USAGE IS BINARY
unsigned int	PIC 9(9) USAGE IS BINARY, LENGTH OF
unsigned long int	PIC 9(9) USAGE IS BINARY
float	COMP-1
double	COMP-2
pointer to...	POINTER, ADDRESS OF
struct	Groups
type array[*n*]	Tables (OCCURS *n* TIMES)

Note:

1. You must specify a size for type array.
2. If the COBOL program receives **int** parameters from the C calling function that might have a value that is larger than that declared as the maximum size by the COBOL picture clause, the COBOL program must either be compiled with the TRUNC(BIN) compiler option or each binary data item that receives **int** parameters from C must be declared as USAGE IS COMP-5. Taking these actions will guarantee that truncation of high-order digits does not occur. For more information about the TRUNC(BIN) compiler option or about using COMP-5 data items, see the appropriate version of the programming guide in the Enterprise COBOL for z/OS library (http://www-01.ibm.com/support/docview.wss?uid=swg27036733).
3. COBOL always turns on the high-order bit of the address of the last parameter. This can cause problems in the C++ program if it is using the address (since it will be treated as a negative number). If a C++ program does need to use the address of the last parameter, one of the following techniques can be used to bypass this problem:
   - If the COBOL program is an Enterprise COBOL program, instead of passing the parameter by reference, pass the address of the item by value. For example, use a call statement that looks like this:

```
CALL "C" using by value address of C-PARM1
              by value address of C-PARM2
```

- If the COBOL program is not Enterprise COBOL, code must be added to mask out the high-order bit in the C++ routine. The sample code shows how to do this:

```
#include <stdio.h>
#include <string.h>
void A1CC01B1(char* myString)
{
  myString = (char*)((int)myString & 0x7fffffff);
  printf("My String: %s \n", myString);
  return;
}
```

Passing data between C++ and COBOL with extern "COBOL"

Note: The reference to COBOL with `extern "COBOL"`, applies to VS COBOL II, COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL for z/OS.

When data is passed between XL C++ and COBOL with `extern "COBOL"`, the C++ compiler generates the appropriate addressing code which introduces an extra level of indirection on the C++ side for non-pointer types. Pointers, however, are passed directly; meaning that for COBOL to receive a pointer to a C++ data type, C++ must pass a pointer to a pointer to the C++ data type. Conversely, if COBOL returns a pointer to a data type, C++ receives a pointer to a pointer to the data type.

Passing data by value (indirect) between C++ and COBOL

Copies of variable can be passed between C++ and COBOL routines. On return, the actual value of the variable remains unchanged regardless of how it may have been modified in the called routine.

Value arguments can be passed BY CONTENT from COBOL programs and received as C++ function parameters when declared with the appropriate base type. Conversely, C++ function arguments can be passed by value (indirect) from C++ functions and received as COBOL parameters. The C++ compiler generates the appropriate addressing code required to access the parameter values; you can write your C++ function, which is interoperable with COBOL, as if it were in a C++-only environment. It can be moved to a C++-only environment simply by removing the `extern "COBOL"`. For example, if a C++ function called FROMCOB is to receive a parameter passed BY CONTENT of type int, the function prototype declaration would look like this:

```
void FROMCOB(int)
```

Table 21 shows the supported data types for passing by value (indirect).

Supported data types passed by value (indirect) between C++ and COBOL

Table 21 identifies the data types that can be passed by value (indirect) as parameters between C++ and COBOL.

Table 21. Supported data types passed by value (indirect) with extern "COBOL"

C++	COBOL
Signed int, signed long int	PIC S9(9) USAGE IS BINARY
Double	COMP-2

Table 21. Supported data types passed by value (indirect) with extern "COBOL" (continued)

C++	COBOL
Pointer to...	POINTER, ADDRESS OF
Sruct	Groups
Type array[*n*]	Tables (OCCURS *n* TIMES)

Passing data by reference (indirect) between C++ and COBOL

A parameter can be passed by reference (indirect) between C++ and COBOL, which means the actual address of the argument is passed to the called function or procedure; any changes to the parameter made by the called routine can alter the original parameter passed by the calling routine.

To pass data by reference (indirect) from C++ to COBOL, the variables are passed by C++ as function arguments, which are pointers to a given type or the address of a given variable, and received as COBOL parameters. Conversely, to pass data by reference (indirect) from COBOL to C++, the variables are passed from COBOL as BY REFERENCE arguments and received by a C++ function as pointers to a given type. For example, if a C++ function called FROMCOB is to receive a parameter passed by reference (indirect) of type int, the function prototype declaration would look like this:

```
void FROMCOB(int *)
```

The C++ function must dereference the pointer to access the actual value. If the value of the pointer is modified by the C++ function, as opposed to modifying the value that the pointer points to, the results on return to COBOL are unpredictable. Therefore, passing values by reference (indirect) from COBOL to C++ should be used with caution, and only in cases where the exact behavior of the C++ function is known.

Table 22 shows the supported data types for passing by reference (indirect).

Supported data types passed by reference (indirect) between C++and COBOL

Table 22 identifies the data types that can be passed by reference (indirect) between C++ and COBOL.

Table 22. Supported data types passed by reference (indirect) with extern "COBOL"

C++	COBOL
Signed short int	PIC S9(4) USAGE IS BINARY
Signed int, signed long int	PIC S9(9) USAGE IS BINARY
Float	COMP-1
Double	COMP-2
Pointer to...	POINTER, ADDRESS OF
Struct	Groups
Type array[*n*]	Tables (OCCURS *n* TIMES)

Passing strings between C++ and COBOL

C++ and COBOL have different string data types:

C++ strings
Logically unbounded length and are terminated by a NULL (the last byte of the string contains X'00')

COBOL PIC X(n)
Fixed-length string of characters of length *n*

You can pass strings between COBOL and C++ routines, but you must match what the routine interface demands with what is physically passed.

Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM have strings like previous COBOLs, as well as null-terminated literal strings like C++.

Refer to “Sample ILC applications” on page 82 to see how string data is passed between C++ and COBOL.

Using aggregates

Aggregates (arrays, strings, or structures) are mapped differently by C++ and COBOL and are not automatically mapped. You must completely declare every byte in the structure to ensure that the layouts of structures passed between the two languages map to one another correctly. The XL C++ compile-time option AGGREGATE and the COBOL compiler option MAP provide a layout of structures to help you perform the mapping.

In C++, a structure is simply a class declared with the `struct` keyword; its members and base classes are public by default. A C++ class is the same as a C++ structure if the only data is public. If a C++ class that uses features unavailable to COBOL (such as virtual functions, virtual base classes, private data, protected data, static data members, or inheritance) is passed to a COBOL program, the results are undefined.

Data equivalents

This section shows how C++ and COBOL data types correspond to each other.

Equivalent data types for C++ to COBOL

This section uses language samples to describe data type equivalencies in C++ to COBOL applications.

Note: In the declarations that follow, examples showing the use of extern "C" apply only to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM. The examples showing extern "COBOL" apply to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II.

Signed one-byte character data with extern "C"

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>
extern "C" {void COBRTN (char);}

int main()
{
  char x;
  x='a';
  COBRTN(x);   /* x by value */
  exit(0);
}
```

COBOL subroutune

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC X.
PROCEDURE DIVISION USING BY VALUE X.
     DISPLAY X.
     GOBACK.
END PROGRAM COBRTN.
```

Signed one-byte character data with extern "COBOL"

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>
extern "COBOL" {void COBRTN (char*);}

  int main()
  {
  char x;
  x='a';
COBRTN(&x);   /* x by reference */
exit(0);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC X.
PROCEDURE DIVISION USING X.
     DISPLAY X
     GOBACK.
END PROGRAM COBRTN.
```

16-bit signed binary integer with extern "C"

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>
extern "C" {void COBRTN
  (short int);}

int main()
{
  short int x;
  x=5;
  COBRTN(x);    /* x by value */
  exit(0);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC S9(4) BINARY.
PROCEDURE DIVISION USING BY VALUE X.
     DISPLAY X.
     GOBACK.
END PROGRAM COBRTN.
```

16-bit signed binary integer with extern "COBOL"

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>
extern "COBOL" {void COBRTN
(short int*);}

int main()
{
  short int x;
  x=5;
  COBRTN(&x);   /* x by reference */
  exit(0);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC S9(4) BINARY.
PROCEDURE DIVISION USING X.
     DISPLAY X
     GOBACK.
END PROGRAM COBRTN.
```

32-bit signed binary integer with extern "C"

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>
extern "C" {void COBRTN (int);}

int main()
{
  int x;
  x=5;
  COBRTN(x);  /* x by value */
  exit(0);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC S9(9) BINARY.
PROCEDURE DIVISION USING BY VALUE X.
     DISPLAY X.
     GOBACK.
END PROGRAM COBRTN.
```

32-bit signed binary integer with extern "COBOL"

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>
extern "COBOL" {void COBRTN (int, int*);}

int main()
{
  int x,y;
  x=5;
  y=6;
  COBRTN(x,&y);  /* x by value */
    /* y by reference */
  exit(0);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC S9(9) BINARY.
01 Y PIC S9(9) BINARY.
PROCEDURE DIVISION USING X Y.
     DISPLAY X Y
     GOBACK.
END PROGRAM COBRTN.
```

Long floating-point number with extern "C"

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>
extern "C" {void COBRTN
    (double);}

int main()
{
  double x;
  x=3.14159265;
  COBRTN(x);   /* x by value */
  exit(0);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X COMP-2.
PROCEDURE DIVISION USING BY VALUE X.
     DISPLAY X.
     GOBACK.
END PROGRAM COBRTN.
```

Long floating-point with extern "COBOL"

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>
extern "COBOL" {void COBRTN
    (double, double*);}

int main()
{
  double x,y;
  x=3.14159265;
  y=4.14159265;
  COBRTN(x,&y);   /* x by value */
    /* y by reference */
  exit(0);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 X COMP-2.
01 Y COMP-2.
PROCEDURE DIVISION USING X Y.
     DISPLAY X Y
     GOBACK.
END PROGRAM COBRTN.
```

Structure with extern "COBOL"

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>
struct stype {
  int s1;
  int s2;};
extern "COBOL" {void COBRTN
  (struct stype,struct stype*);}

int main()
{
  struct stype struc1,struc2;
  struc1.s1=1;
  struc1.s2=2;
  struc2.s1=3;
  struc2.s2=4;
  COBRTN(struc1,&struc2);
    /* struc1 by value */
    /* struc2 by reference */
  exit(0);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 STRUC1.
     05 S11 PIC S9(9) BINARY.
     05 S12 PIC S9(9) BINARY.
01 STRUC2.
     05 S21 PIC S9(9) BINARY.
     05 S22 PIC S9(9) BINARY.
PROCEDURE DIVISION USING STRUC1 STRUC2.
     DISPLAY S11 S12 S21 S22
     GOBACK.
END PROGRAM COBRTN.
```

Array with extern "COBOL"

Sample C++ usage

```
#include <stdio.h>
#include <stdlib.h>
extern "COBOL" {void COBRTN
     (int array[2]);}

int main()
{
  int array=[2];
  array[0]=1;
  array[1]=2;
  COBRTN(array);
    /* array by reference */
  exit(0);
}
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 ARRAY.
     05 ELE PIC S9(9) BINARY OCCURS 2.
PROCEDURE DIVISION USING ARRAY.
     DISPLAY ELE(1) ELE(2)
     GOBACK.
END PROGRAM COBRTN.
```

Equivalent data types for COBOL to C++

This section uses language samples to describe data type equivalencies in COBOL to C++ applications.

Note: In the declarations that follow, examples showing the use of extern "C" apply only to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM. The examples showing extern "COBOL" apply to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II.

32-bit signed binary integer with extern "C"

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X PIC S9(9) BINARY.
01 Y PIC S9(9) BINARY.
PROCEDURE DIVISION.
     MOVE 1 TO X.
* X BY VALUE ***
     CALL "CENTRY" USING BY VALUE X
       RETURNING Y.
     GOBACK.
END PROGRAM COBRTN.
```

C++ function

```
#include <stdio.h>
extern "C" {int CENTRY (int x);}

int CENTRY(int x)
{
  int y=2;
  printf("%d %d \n",x,y);
  return y;
}
```

32-bit signed binary integer with extern "COBOL"

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X PIC S9(9) BINARY.
01 Y PIC S9(9) BINARY.
PROCEDURE DIVISION.
     MOVE 1 TO X.
     MOVE 2 TO Y.
* X BY VALUE, Y BY REFERENCE ***
     CALL "CENTRY" USING BY CONTENT X
       BY REFERENCE Y.
     GOBACK.
END PROGRAM COBRTN.
```

C++ function

```
#include <stdio.h>
extern "COBOL" {void CENTRY (int x, int *y);}

void CENTRY(int x, int *y)
{
  printf("%d %d \n",x,*y);
  return;
}
```

Long floating-point number with extern "C"

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X COMP-2.
01 Y COMP-2.
PROCEDURE DIVISION.
     MOVE 3.14159265 TO X.
* X BY VALUE ***
     CALL "CENTRY" USING BY VALUE X
       RETURNING Y.
     GOBACK.
END PROGRAM COBRTN.
```

C++ function

```
#include <stdio.h>
extern "C" {double CENTRY
     (double x);}

double CENTRY(double x)
{
  double y=4.14159265;
  printf("%f %f \n",x,y);
  return y;
}
```

Long floating-point number with extern "COBOL"

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X COMP-2.
01 Y COMP-2.
PROCEDURE DIVISION.
     MOVE 3.14159265 TO X.
     MOVE 4.14159265 TO Y.
* X BY VALUE, Y BY REFERENCE ***
     CALL "CENTRY" USING BY CONTENT X
       BY REFERENCE Y.
     GOBACK.
END PROGRAM COBRTN.
```

C++ function

```
#include <stdio.h>
extern "COBOL" {void CENTRY
     (double x, double *y);}

void CENTRY(double x, double *y)
{
  printf("%f %f \n",x,*y);
  return;
}
```

Structure with extern "C"

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 STRUC1.
     05 S11 PIC S9(9) BINARY.
     05 S12 PIC S9(9) BINARY.
PROCEDURE DIVISION.
     CALL "CENTRY" RETURNING STRUC1.
     GOBACK.
END PROGRAM COBRTN.
```

C++ function

```
#include <stdio.h>
struct stype {
  int S1;
  int S2;  } struc1;
extern "C" {struct stype CENTRY()

struct stype CENTRY()
}
  struc1.s1=1;
  struc1.s2=2;
  printf("%d %d \n",struc1.s1,
    struc1.s2);
  return struc1;
}
```

Structure with extern "COBOL"

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRTN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 STRUC1.
     05 S11 PIC S9(9) BINARY VALUE 1.
     05 S12 PIC S9(9) BINARY VALUE 2.
01 STRUC2.
     05 S21 PIC S9(9) BINARY VALUE 3.
     05 S22 PIC S9(9) BINARY VALUE 4.
PROCEDURE DIVISION.
* STRUC1 BY VALUE STRUC2 BY REFERENCE ***
     CALL "CENTRY"
       USING BY CONTENT STRUC1
             BY REFERENCE STRUC2.
     GOBACK.
END PROGRAM COBRTN.
```

C++ function

```
#include <stdio.h>
struct stype {
  int S1;
  int S2;  };
extern "COBOL" {void CENTRY
     (struct stype struc1,
      struct stype *struc2);}

void CENTRY(struct stype struc1,
      struct stype *struc2)
{
printf("%d %d %d %d \n",struc1.s1,
      struc1.s2,struc2->s1,struc2->s2);
return;
}
```

Name scope of external data

In programming languages, the *name* scope is defined as the portion of an application within which a particular declaration applies or is known. The name scope of external data differs between C++ and COBOL. The scope of external data under C++ is the load module; under COBOL, it is the enclave (or run unit). Figure 7 on page 49 and Figure 8 on page 49 illustrate these differences.

Because the name scope for C++ and COBOL is different, external variable mapping between C++ and COBOL routines is not supported. External variables with the same name are considered separate between C++ and COBOL.

If your application relies on the separation of external data, do not give the data the same name in both languages within a single application. If you give the data in each load module a different name you can change the language mix in the application later, and your application still behaves as you expect it to.

DLL considerations

In DLL code, external variables are mapped across the load module boundary. DLLs are shared at the enclave level. Therefore, a single copy of a DLL applies to all modules in an enclave, regardless of whether the DLL is loaded implicitly (through a reference to a function or variable) or explicitly (through `dllload()`). See *z/OS Language Environment Programming Guide* for information about building and managing DLL code in your applications.

COBOL data declared with the EXTERNAL attribute are independent of DLL support. These data items are managed by the COBOL runtime, and are accessible by name from any COBOL program in the run-unit that declares them, regardless of whether the programs are in DLLs or not.

In particular, the facilities for exporting and importing external variables from DLLs implemented in OS/390 C/C++, do not apply to COBOL external data. Hence C/C++ external data and COBOL external data are always in separate name spaces, regardless of DLL considerations.

For C/C++, non-DLL applications have external data which is only shared within the load module.

However, for DLL applications, C/C++ external data is now (optionally) accessible to all C/C++ routines in the enclave.

Name space of external data

In programming languages, the *name space* is defined as the portion of a load module within which a particular declaration applies or is known. Like the name scope, the name space of external data differs between C++ and COBOL.

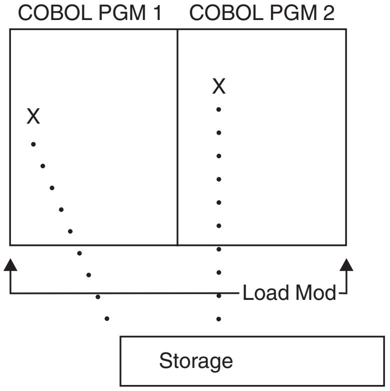


Figure 16. Name space of external data for COBOL static call to COBOL

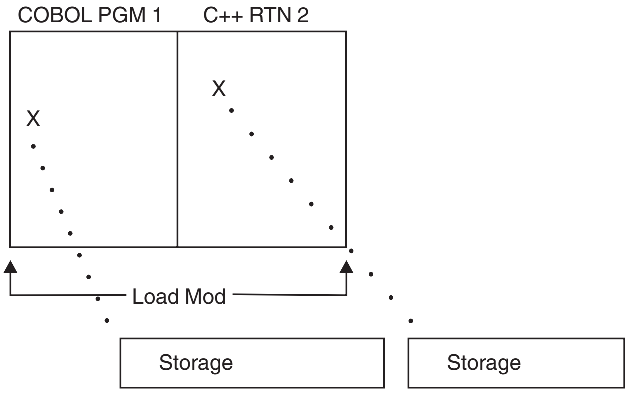


Figure 17. Name space of external data in COBOL static call to C++

Figure 16 on page 76 and Figure 17 illustrate that within the same load module, the name space of COBOL programs is the same. However, the name spaces of a COBOL program and a C++ routine within the same load module are not the same. If you give external data the same name in both languages, an incompatibility in external data mapping can occur.

Directing output in ILC applications

C++ and COBOL do not share files, except the Language Environment message file (the ddname specified in the Language Environment MSGFILE runtime option). You must manage all other files to ensure that no conflicts arise. Performing I/O operations on the same ddname might cause abnormal termination.

Under C++, runtime messages and other related output are directed to the default MSGFILE ddname. `stderr` output is also by default directed to the MSGFILE ddname. `stdout` is not by default directed to the MSGFILE ddname, but can be redirected to do so. Also, output from `printf` can be interspersed with output from the COBOL DISPLAY statement and output from Language Environment by redirecting `stdout` to `stderr` (for example, passing `1>&2` as a command-line parameter).

For information about redirecting C++ output, see *z/OS XL C/C++ Programming Guide*.

If you are using the C++ iostreams class, `cout` is directed to the same place as `stdout`, and `cerr`/`clog` are directed to the same place as `stderr`.

Under COBOL, runtime messages and other related output are directed to the MSGFILE ddname. Output from COBOL DISPLAY UPON SYSOUT is directed to the default MSGFILE ddname only when the OUTDD compiler option ddname matches the MSGFILE ddname; this applies to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II.

Interspersing output when C++ Is the main routine

To intersperse output from C++ and COBOL when the main routine is coded in C++, compile your COBOL program using the default OUTDD (OUTDD=SYSOUT) if you are using the default MSGFILE ddname. If you have overridden the default

MSGFILE ddname, you must compile your COBOL program using an OUTDD that specifies the same name as the MSGFILE. You must redirect `stdout` to `stderr` when C++ is the main routine.

Interspersing output when COBOL Is the main program

To intersperse output from C++ and COBOL when the main routine is coded in COBOL, compile your COBOL program using the default OUTDD (OUTDD=SYSOUT). In your C++ routine, add a line `stdout = stderr`. If you have overridden the default MSGFILE ddname, you must compile your COBOL program using an OUTDD ddname that specifies the same name as the MSGFILE.

C++ to COBOL condition handling

This section provides two scenarios of condition handling behavior in a C++ to COBOL ILC application. If an exception occurs in a C++ routine, the set of possible actions is as described in “Exception occurs in C++” on page 79. If an exception occurs in a COBOL program, the set of possible actions is as described in “Exception occurs in COBOL” on page 80.

Keep in mind that some conditions can be handled only by the HLL of the routine in which the exception occurred. For example, in a COBOL program, a statement can have a clause that adds condition handling to a verb, such as the ON SIZE clause of a COBOL DIVIDE verb (which includes the logical equivalent of a divide-by-zero condition). This condition is handled completely within COBOL.

C++ exception handling constructs `try()/throw()/catch()` cannot be used with Language Environment and COBOL condition handling. If you use C exception handling constructs (`signal()/raise()`) in your C++ routine, condition handling will proceed as described in this section. Otherwise, you will get undefined behavior in your programs if you mix the C++ constructs with the C constructs. For a detailed description of Language Environment condition handling, see *z/OS Language Environment Programming Guide*.

Enclave-terminating language constructs

Enclaves can be terminated for reasons other than an unhandled condition of severity 2 or greater. In Language Environment ILC, you can issue an HLL language construct to terminate a C++ to COBOL enclave from either a C++ or COBOL routine.

C language constructs available under C++

Among the C language constructs that terminate an enclave are `abort()`, `exit()`, `raise(SIGABRT)`, and `raise(SIGTERM)`. Destructors are run at library termination.

If you call `abort()`, `raise(SIGABRT)`, or `exit()`, the T_I_S (Termination Imminent Due to STOP) condition is raised. After T_I_S has been processed and all user code has been removed from the stack, the C atexit list is honored.

COBOL language constructs

The COBOL language constructs that cause the enclave to terminate are:

- STOP RUN

  COBOL's STOP RUN is equivalent to the C++ `exit()` function. If you code a COBOL STOP RUN statement, the T_I_S (Termination Imminent Due to STOP) condition is raised. After T_I_S has been processed and all user code has been removed from the stack, the C++ atexit list is honored.

- Call to ILBOABN0 or CEE3ABD

  Calling ILBOABN0 or CEE3ABD causes T_I_U to be signaled. Condition handlers are given a chance to handle the abend. If the abend remains unhandled, normal Language Environment termination activities occur. For example, the C++ atexit list is honored and the Language Environment assembler user exit gains control.

  User-written condition handlers written in COBOL must be compiled with Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM or COBOL/370.

Exception occurs in C++

In this scenario, a COBOL main program invokes a C++ subroutine and an exception occurs in the C++ subroutine. Refer to Figure 18 throughout the following discussion.

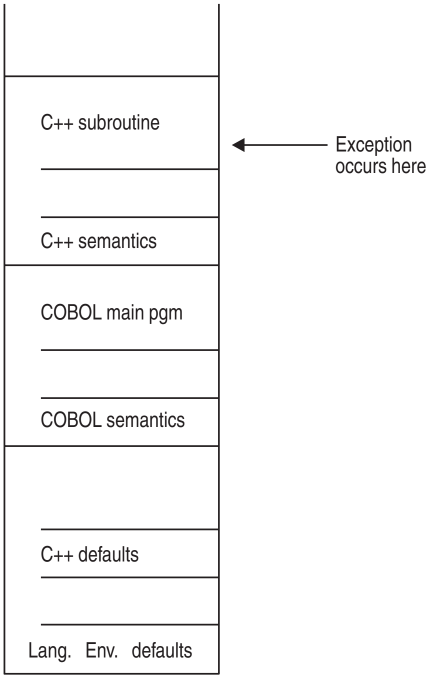


Figure 18. Stack contents when the C/C++ exception occurs

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, C++ determines whether the exception in the C++ routine should be enabled and treated as a condition. If any of the following are true, the exception is ignored, and processing continues at the next sequential instruction after where the exception occurred:
   - You specified `SIG_IGN` for the exception in a call to `signal()`.

     Note: The system or user abend corresponding to the `signal(SIGABND)` or the Language Environment message 3250 is not ignored. The enclave is terminated.
   - The exception is one of those listed as masked in Table 63 on page 249, and you have not enabled it using the CEE3SPM callable service.
   - You did not specify any action, but the default action for the condition is `SIG_IGN` (see Table 63 on page 249).

- You are running under CICS and a CICS handler is pending.

If you did none of these things, the condition is enabled and processed as a condition.

2. If a user-written condition handler was registered using CEEHDLR on the C++ stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points. You must be careful when moving the resume cursor in an application that contains a COBOL program; see “CEEMRCR and COBOL” on page 82 for details.

   In this example, there is not a user-written condition handler registered for the condition, so the condition is percolated.
3. The global error table is now examined for signal handlers that was registered for the condition.

   If there is a signal handler registered for the condition, it is given control. If it issues a resume or a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points. You must be careful when issuing a `longjmp()` in an application that contains a COBOL program; see “CEEMRCR and COBOL” on page 82 for details.

   In this example, no C signal handler is registered for the condition, so the condition is percolated.
4. The condition is still unhandled. If C++ does not recognize the condition, or if the C++ default action (listed in Table 63 on page 249) is to terminate, the condition is percolated.
5. If a user-written condition handler was registered using CEEHDLR on the COBOL stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
6. If the condition is of severity 0 or 1, the Language Environment default actions take place, as described in Table 62 on page 249.
7. If the condition is of severity 2 or above, Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition) and redrive the stack. Condition handling now enters the termination imminent step.
8. If, on the next pass of the stack, no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Exception occurs in COBOL

In this scenario, a C++ main routine invokes a COBOL subroutine and an exception occurs in the COBOL subroutine. Refer to Figure 19 on page 81 throughout the following discussion.

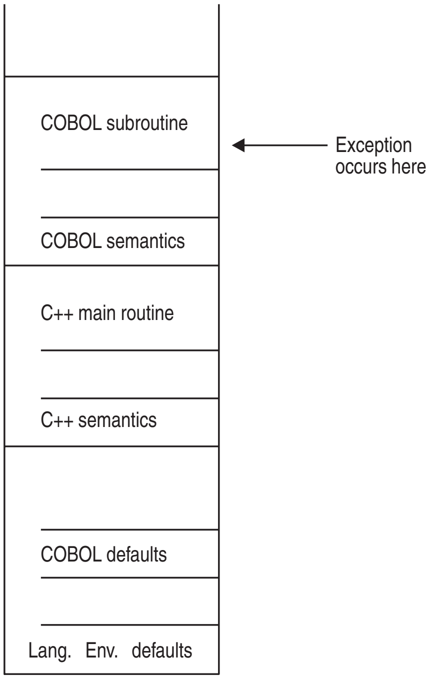


Figure 19. Stack contents when the exception occurs in COBOL

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, COBOL determines if the exception should be ignored or handled as a condition.
   - If the exception is to be ignored, control is returned to the next sequential instruction after where the exception occurred.
   - If the exception is to be enabled and processed as a condition, the condition handling step takes place.
2. If a user-written condition handler was registered using CEEHDLR on the COBOL stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points. Note that you must be careful when moving the resume cursor in an application that contains a COBOL program. See “CEEMRCR and COBOL” on page 82 for details.

   In this example no user-written condition handler is registered for the condition, so the condition is percolated.
3. If a user-written condition handler was registered for the condition using CEEHDLR on the C++ stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example no user-written condition handler is registered for the condition, so the condition is percolated.
4. If a C signal handler was registered for the condition, it is given control. If it moves the resume cursor or issues a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points.

   In this example, no C signal handler is registered for the condition, so the condition is percolated.

5. If the condition has a Facility_ID of IGZ, the condition is COBOL-specific. The COBOL default actions for the condition take place. If COBOL does not recognize the condition, condition handling continues.
6. If the condition is of severity 0 or 1, Language Environment default actions take place, as described in Table 62 on page 249.
7. If the condition is of severity 2 or higher, Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition) and redrive the stack. Condition handling now enters the termination imminent step.
8. If on the second pass of the stack no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

CEEMRCR and COBOL

When you make a call to CEEMRCR to move the resume cursor, or issue a call to `longjmp()`, and a COBOL program is removed from the stack, the COBOL program can be re-entered via another call path. (This will not work for VS COBOL II programs.)

If the COBOL program does not specify RECURSIVE in the PROGRAM-ID, a recursion error is raised if you attempt to enter the routine again.

- A VS COBOL II, COBOL/370, COBOL for MVS & VM, or COBOL for OS/390 & VM program compiled with the CMPR2 option
- A VS COBOL II program compiled with the NOCMPR2 option (which does not use nested routines)
- A COBOL/370, COBOL for MVS & VM, or COBOL for OS/390 & VM program compiled with the NOCMPR2 option that does not use the combination of the INITIAL attribute, nested routines, and file processing in the same compilation unit.
- An Enterprise COBOL for z/OS program that does not use the combination of the INITIAL attribute, nested routines, and file processing in the same compilation unit.

In addition, if the routine is a COBOL routine with the INITIAL attribute and containing files, the files are closed. (COBOL supports VSAM and QSAM files and these files are closed.)

Sample ILC applications

Figure 20 on page 83 and Figure 21 on page 84 contain an example of an ILC application. The C++ routine CPP1 statically calls the COBOL CBL1 program. CBL1 statically CALLs C++ routine CPP2.

```
/* Module/File Name: EDCXCB  */
/******************************************************************/
/*   Illustration of Interlanguage Communication between C++      */
/*   and COBOL.  All parameters passed by reference.              */
/*                                                                */
/*           CPP1 ---------> CBL1 ---------> CPP2                 */
/*                 static          static                         */
/*                  call            call                          */
/******************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern "COBOL" {
   void CBL1(short *, long, float, char *);
   void CPP2( int& num, char * newstring);
}

/******************** CPP1 routine example **********************/

int main()
{
  signed short int short_int  = 2;
  signed long int  long_int   = 4;
  double           floatpt    = 8.0;
  char             string[80];


  CBL1( &short_int, long_int, floatpt, string );

  fprintf(stderr,"main  ENDED\n");
}

void CPP2( int& num, char* newstring)
{
   fprintf(stderr,"num is %d, newstring is %s\n",num,newstring);
   fprintf(stderr,"CPP2 ended\n");
}
```

Figure 20. Static call from C++ to COBOL program

```
*   CBL LIB,QUOTE
*Module/File Name:  IGZTXCB
****************** CBL1 routine example ********************

 IDENTIFICATION DIVISION.
 PROGRAM-ID.  CBL1.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77  var1                 PIC S9(9) BINARY VALUE 5.
 01  msg-string           PIC X(80).

 LINKAGE SECTION.
 77  int2                 PIC S9(4) BINARY.
 77  int4                 PIC S9(9) BINARY.
 77  float                COMP-2.
 77  char-string          PIC X(80).

     PROCEDURE DIVISION USING int2 int4 float char-string.

          DISPLAY "CBL1 STARTED".

          IF (int2 NOT = 2) THEN
            DISPLAY "INT2 NOT = 2".

          IF (int4 NOT = 4) THEN
            DISPLAY "INT4 NOT = 4".

          IF (float NOT = 8.0) THEN
            DISPLAY "FLOAT NOT = 8".


   * Place null-character-terminated string in parameter
         STRING "PASSED CHARACTER STRING", X'00' LOW-VALUE
                DELIMITED BY SIZE INTO msg-string

   * MAKE A STATIC CALL TO C FUNCTION
         CALL "CPP2" USING var1, msg-string.

         DISPLAY "CBL1 ENDED".
         GOBACK.
```

Figure 21. Static call from COBOL to C++ routine

Chapter 6. Communicating between C and Fortran

This topic describes Language Environment's support for C to Fortran ILC applications.

General facts about C to Fortran ILC

- A load module consisting of object code compiled with any Fortran compiler link-edited with object code compiled in another language is not reentrant, regardless of whether the Fortran routine was compiled with the RENT compiler option.
- Return codes cannot be passed from a C routine to the Fortran routine that invoked it.
- Fortran routines cannot operate under CICS.
- Support for Fortran on VM is not provided by Language Environment.
- Several C and Fortran library routines have identical names. You must resolve name conflicts before link-editing your C to Fortran ILC applications. See *z/OS Language Environment Programming Guide* for link-editing information.
- Fortran routines cannot operate in an XPLINK environment, so ILC with C routines compiled XPLINK is not supported.
- There is no ILC support between AMODE 31 and AMODE 64 applications. Fortran does not support AMODE 64.

Preparing for ILC

This section describe topics you might want to consider before writing an ILC application. To determine how different versions of HLLs work together, refer to the migration guides for the HLLs you plan to use.

Language Environment ILC support

Language Environment provides ILC support between the following combinations of C and Fortran:

Table 23. Supported languages for Language Environment ILC

HLL pair	C	Fortran
C to Fortran	• C/370 Version 1 Release 2 • C/370 Version 2 Release 1 • AD/Cycle C/370 Version 1 • IBM C/C++ for MVS/ESA • z/OS XL C/C++ compilers	• FORTRAN IV G1 • FORTRAN IV H Extended • VS FORTRAN Version 1, except modules compiled with Release 2.0 or earlier and that either pass character arguments to, or receive character arguments from, subprograms. • VS FORTRAN Version 2, except modules compiled with Releases 5 or 6 and whose source contained any parallel language constructs or parallel callable services, or were compiled with either of the compiler options PARALLEL or EC.

Note: Dynamic calls from Fortran are available in VS FORTRAN Version 2 Release 6 only.

Migrating ILC applications

All C to Fortran ILC applications need to be relinked before running. See *z/OS Language Environment Programming Guide* for information about how to relink applications to run with Language Environment. You can find other helpful migration information, especially concerning runtime option compatibility, in *z/OS Language Environment Runtime Application Migration Guide*.

Determining the main routine

In Language Environment, only one routine can be the main routine; no other routine in the enclave can use syntax that indicates it is main.

A C function is designated as a main routine because its function definition gives its name as `main`. The entry point into the load module is CEESTART. In C, the same routine can serve as both the main routine and subroutine if it is recursively called. In such a case, the new invocation of the routine is not considered a second main routine within the enclave, but a subroutine.

A Fortran routine is designated as a main routine with a PROGRAM statement, which indicates the name of the main routine. A main routine can also be designated by the absence of PROGRAM, SUBROUTINE, or FUNCTION statements, in which case the name of the main routine is the default value `MAIN` (or `MAIN#` for VS FORTRAN Version 2 Releases 5 and 6). The name of the main routine is the entry point into the load module.

An entry point is defined for each supported HLL. Table 24 identifies the main and fetched entry points for each language. The table assumes that your code was compiled using the Language Environment-conforming compilers.

Table 24. Determining the entry point

HLL	Main entry point	Fetched entry point
C	CEESTART	CEESTART or routine name, if `#pragma linkage(,fetchable)` is not used.
Fortran	Name on the PROGRAM statement. In the absence of PROGRAM, SUBROUTINE, or FUNCTION statements, the default value is `MAIN` (or `MAIN#` in VS FORTRAN Version 2 Releases 5 and 6).	Subprogram name

Declaring C to Fortran ILC

A `#pragma linkage` directive is required for C to call Fortran and for Fortran to call C. All entry declarations are made in the C code; no special declaration is required in the Fortran routine. The directive allows the C routine to pass arguments and return values to and from the Fortran routine using an internal argument list compatible with Fortran.

The `#pragma linkage` directive has the following format:

```
#pragma linkage(identifier, FORTRAN[, RETURNCODE])
```

identifier is either the name of the Fortran routine (a function or subroutine), a C function, or, for dynamic calls, the name of a typedef that refers to a Fortran routine.

RETURNCODE is optional and applies only to a called Fortran routine: it specifies that *identifier* is a Fortran routine that returns an alternate return code to the C routine.

Example of declaration for C calling Fortran

The following example shows a partial C routine that calls a Fortran function. The calling C routine contains the `#pragma linkage` directive for the Fortran function FORTFCN, the function prototype for the Fortran function, and a static call to the Fortran function.

C function

```
#pragma linkage (fortfcn, FORTRAN)
⋮
double fortfcn(int, double [100]);
⋮
int index;
double list[100];
double value;
⋮
value=fortfcn(index, list);
```

Fortran function

```
FUNCTION FORTFCN (INDEX, LIST) RESULT (VALUE)
INTEGER*4 INDEX
REAL*8 LIST(0:99)
REAL*8 VALUE
⋮
VALUE=LIST(INDEX)
END
```

Example of declaration for Fortran calling C

The following example shows a partial Fortran routine that calls a C function. The called C routine contains the `#pragma linkage` directive for the C function CFCN and the function definition for the C function.

Fortran Routine

```
INTEGER*4 INDEX
REAL*8 LIST(0:99)
REAL*8 VALUE
REAL*8 CFCN
⋮
VALUE=CFCN(INDEX, LIST)
```

C Routine

```
#pragma linkage (cfcn, FORTRAN)
⋮
double cfcn(int *index, double list [])
{
⋮
}
```

Calling between C and Fortran

This section describes the types of calls permitted between C and Fortran, and considerations when using dynamic calls and fetch.

Types of calls permitted

Table 25 describes the types of calls between C and Fortran that Language Environment allows:

Table 25. Calls permitted for C and Fortran ILC

ILC direction	Static calls	Dynamic calls
C to Fortran	Yes	Yes
Fortran to C	Yes	Yes

Note: Dynamic call refers to C fetching a Fortran routine or Fortran dynamically calling a C routine.

Dynamic call/fetch considerations

This section describes the dynamic calling/fetching mechanisms supported between C and Fortran.

C fetching Fortran

In C, dynamic calls are made by invoking the `fetch()` function and then later invoking the fetched routine with the returned fetch pointer. The fetched routine can either be a C or Fortran routine: the C routine can then fetch or statically call a

Fortran routine; the Fortran routine can then statically call a C routine. (If the statically linked C routine is within a dynamically loaded module with a Fortran entry point, the C routine must be either nonreentrant or naturally reentrant.) In the fetched load module, a routine can dynamically call other C or Fortran routines, regardless of whether the routines are reentrant.

A C routine that fetches a Fortran routine cannot contain a `fork()` function. Although you cannot run an application with `fork()`, you can run with POSIX(ON). For a full description of running under POSIX, see *z/OS Language Environment Programming Guide*.

Restriction: When a C routine fetches a Fortran routine, the dynamically loaded module can contain only routines written in those languages that already exist in a previous load module. (The routine in the previous load module need not be called; it only needs to be present.) For a Fortran routine to be recognized, ensure that at least one of the following is present in a previous load module:

- A Fortran main program
- A Fortran routine that causes one or more Fortran runtime library routines to be link-edited into the load module. If the Fortran routine contains either an I/O statement, a mathematical function reference, or a call to any Fortran callable service (such as CPUTIME), then a library routine is included, and this requirement is satisfied.
- The Fortran signature CSECT, CEESG007. Use the following linkage editor statement to include CEESG007 if neither of the two previous conditions is true:

  ```
  INCLUDE SYSLIB(CEESG007)
  ```

Fortran dynamically calling C

Dynamic calls are made in Fortran by specifying the name of the routine to be loaded with the DYNAMIC compiler option, and then using the same name in a CALL statement. The dynamically called routine can be either C or Fortran, and it can in turn statically call either a C or Fortran routine. (If the statically linked C routine is within a dynamically loaded module with a Fortran entry point, the C routine must be either nonreentrant or naturally reentrant.) In the dynamically loaded module, a routine can dynamically call other C or Fortran routines, regardless of whether the routines are reentrant.

Neither a C nor a Fortran routine can delete a dynamically loaded routine that was dynamically loaded in a Fortran routine.

Invoking functions with returned values

Both C and Fortran can invoke the other language as a function that returns a value: in C, this would be a function that returns something other than void; in Fortran, a function is a routine that begins with a FUNCTION statement. Only certain data types, however, can be used as function return values. See Table 28 on page 90 for a list of the supported data types that can be used as function return values.

Calling Fortran library routines

You can statically call a Fortran library routine, such as CPUTIME, from C. However, you cannot dynamically call a Fortran library routine from C.

Passing data between C and Fortran

This section describes the data types that can be passed between C and Fortran. In the C-to-Fortran passing direction, most of the data types can be passed by reference only; several can also be passed by value. In the Fortran-to-C passing direction, however, the only passing method allowed is by reference.

Under Language Environment, the term *by value* means that a temporary copy of the argument is passed to the called function or procedure. Any changes to the parameter made by the called routine will not alter the original parameter passed by the calling routine. Under Language Environment, the term *by reference* means that the actual address of the argument is passed. Any changes to the parameter made by the called routine can alter the original parameter passed by the calling routine.

This section also includes information about passing an alternate return code from Fortran to C.

Supported data types between C and Fortran

Table 26 lists the data types that can be passed between C and Fortran by reference.

Table 26. Supported data types passed by reference

C	Fortran
signed short int	INTEGER*2
signed int, signed long int	INTEGER*4
float	REAL*4
double	REAL*8
long double	REAL*16
signed char	INTEGER*1
unsigned char	UNSIGNED*1 or CHARACTER*1
char[*n*]	CHARACTER**n*
Address of supported data types and aggregates Examples: Address of an integer (...int**...) Address of an array of integers (...int(**)[8]...)	 POINTER (X,Y) INTEGER*4 Y POINTER (X,Y) INTEGER*4 Y(8)

Supported data types for passing by value

Table 27 lists the data types that can be passed from C to Fortran by value.

Table 27. Supported data types for passing by value from C to Fortran

C	Fortran
signed int, signed long int	INTEGER*4
double	REAL*8
long double	REAL*16

Supported data types for passing function return values

Table 28 lists the data types that can be passed as function return values from a C function to a Fortran routine.

Table 28. Supported data types for passing as function return values from C to Fortran

C	Fortran
signed short int	INTEGER*2
signed int, signed long int	INTEGER*4
float	REAL*4
double	REAL*8
long double	REAL*16
signed char	INTEGER*1
unsigned char	UNSIGNED*1

Passing an alternate return code from Fortran to C

You can pass an alternate return code to a C routine from a Fortran subroutine by specifying the called Fortran subroutine in the `#pragma linkage` directive. The Fortran subroutine will produce an alternate return code when alternate returns are specified as dummy arguments on the SUBROUTINE statement.

In an all-Fortran application, the alternate returns provide a way to return to a point in the calling program other than to the point immediately following the CALL statement. The following example illustrates how a Fortran routine would call a Fortran subroutine to use an alternate return:

```
CALL FSUB (ARG1, *22, ARG2, *66)
```

In this example, `*22` and `*66` specify two labels (22 and 66) to which control can be passed rather than to the point following the CALL statement. The corresponding subroutine would be coded as follows:

```
SUBROUTINE FSUB (DARG1, DARG2, *, *)
```

When the FSUB subroutine executes the following RETURN statement, control would pass to the calling program at the second alternate return, at label 66.

```
RETURN 2
```

There is no alternate return point feature in C. However, if you specify the RETURNCODE suboption on the `#pragma linkage` directive in your C routine, you can use the `fortrc()` function to get the alternate return code from the RETURN statement in the Fortran subroutine. The `fortrc()` function reference applies to the call to Fortran immediately preceding it; you must not have any C code between the Fortran subroutine and the `fortrc()` function reference.

In the following example, the C routine calls the subroutine FSUB, whose SUBROUTINE and RETURN statements are shown above. The `fortrc()` function returns an alternate return code of 2.

```
#includes <stdlib.h>
#pragma linkage (fsub, FORTRAN, RETURNCODE)
void fsub (float, float);
int rc:
⋮
fsub(1.0,2.0);
```

```
rc=fortrc();
⋮
```

The RETURNCODE suboption is optional. It indicates to the C compiler that `fsub` is a Fortran routine returning an alternate return code. You cannot pass return code values from a called C function to a calling Fortran routine.

Passing character data

Character data can be received by a Fortran routine only when the routine that receives the data declares the data to be of fixed length. Therefore, the Fortran form CHARACTER*(*) cannot be used by a Fortran routine to receive character data. An array of characters can be processed in a C routine only when the Fortran routine or the C routine produces the terminating null character.

Mapping arrays between C and Fortran

Arrays can be passed between C and Fortran routines when the array passed has its elements in contiguous storage locations. In addition, in a called Fortran routine, the declaration of the array must indicate a constant number of elements along each dimension.

In C, arrays of more than one dimension are arranged in storage in row major order, while in Fortran they are arranged in column major order. You need to reference the corresponding element of the other language by reversing the order of the subscripts. For example, in an array of floating point integers, the C declaration would be `float [10][20]` while the Fortran declaration would be `REAL*4(20,10)`.

Another difference in using arrays is that unless specified otherwise, the lower bound (the lowest subscript value) of a Fortran array is 1. In C, the lowest subscript value is always 0, so you must adjust either the declared lower bound in the Fortran routine or the subscript you are using when you reference the value in C or Fortran.

For example, the following two arrays have the same storage mapping:

C `float da[10][20];`

Fortran
`REAL*4 DA(20,10)`

The following two elements also represent the same storage:

C `da[4][8]`

Fortran
`DA(9,5)`

Data equivalents

This section describes how C and Fortran data types correspond to each other.

Equivalent data types for C to Fortran

The following examples illustrate how C and Fortran routines within a single ILC application might code the same data types. The examples might be clearer to you if you first read “Passing data between C and Fortran” on page 89, which describes how a C routine can receive parameters that are passed by value and by reference.

16-bit signed binary integer

Sample C usage (by Reference)

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
void cfort( short int * );
main()
{
  short int x;
  x=5;
  cfort(&x);
  printf
    ("Updated value in C: %d\n", x);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG )
INTEGER*2 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
ARG = ARG + 1
END
```

Note: Because `short int` is an example of a parameter which must be passed using an C explicit pointer, you cannot code `cfort(x)`, passing `x` by value.

32-bit signed binary integer

Sample C usage (by Value)

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
int cfort( int );
main()
{
  int x, y;
  x=5;
  y = cfort(x);
  printf
    ("Value returned to C: %d\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
INTEGER*4 CFORT
INTEGER*4 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Sample C usage (by Reference)

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
int cfort( int * );
main()
{
  int x, y;
  x=5;
  y = cfort(&x);
  printf
    ("Value returned to C: %d\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
INTEGER*4 CFORT
INTEGER*4 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Short floating-point number

Sample C usage (by Reference)

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
float cfort( float * );
main()
{
  float x, y;
  x=5.0F;
  y = cfort(&x);
  printf
    ("Value returned to C: %f\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
REAL*4 CFORT
REAL*4 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Long floating-point number

Sample C usage (by Value)

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
double cfort(double);
main()
{
  double x, y;
  x=12.5;
  y=cfort(x);
  printf
    ("Value returned to C: %f\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
REAL*8 CFORT
REAL*8 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Sample C usage (by Reference)

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
double cfort(double *);
main()
{
  double x, y;
  x=12.5;
  y=cfort(&x);
  printf
    ("Value returned to C: %f\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
REAL*8 CFORT
REAL*8 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Extended floating-point number

Sample C usage (by Value)

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
long double cfort(long double);
main()
{
  long double x, y;
  x=12.1L;
  y=cfort(x);
  printf
    ("Value returned to C: %Lf\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
REAL*16 CFORT
REAL*16 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Sample C usage (by Reference)

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
long double cfort(long double *);
main()
{
  long double x, y;
  x=12.1L;
  y=cfort(&x);
  printf
    ("Value returned to C: %Lf\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
REAL*16 CFORT
REAL*16 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Signed one-byte character data

Sample C usage (by Reference)

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
void cfort(signed char *);
main()
{
signed char x, y;
  x=-5;
  cfort(&x);
  printf
    ("Updated value in C: %d\n", x);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG )
INTEGER*1 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
ARG = ARG - 1
END
```

Unsigned one-byte character data

Sample C usage (by Reference)

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
void cfort(unsigned char *,
    unsigned char *);
main()
{
  unsigned char x, y;
  x='a';
  cfort(&x,&y);
  printf
    ("Value returned to C: %c\n", y);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG1, ARG2 )
CHARACTER*1 ARG1, ARG2
PRINT *, 'FORTRAN ARG1 VALUE: ', ARG1
ARG2 = ARG1
END
```

Fixed-length character data

Sample C usage

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
void cfort(char [10], char [10]);
main()
{
  char x[10] = "1234567890";
  char y[10];
  cfort(x,y);
  printf
    ("Value returned to C: %10.10s\n", y);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG1, ARG2 )
CHARACTER*10 ARG1, ARG2
PRINT *, 'FORTRAN ARG1 VALUE: ', ARG1
ARG2 = ARG1
END
```

Array

Sample C usage

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
void cfort(float[]);
main()
{
  float matrix[3] = {0.0F,1.0F,2.0F};
  cfort(matrix);
  printf
    ("Updated values in C: %f %f %f\n",
    matrix[0], matrix[1], matrix[2]);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG )
REAL*4 ARG(3)
PRINT *, 'FORTRAN ARG VALUES:', ARG
DO J = 1, 3
  ARG(J) = ARG(J) + 1.0
ENDDO
END
```

Address of an integer

Sample C usage

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
void cfort (int **);
main()
{
  int i, *temp;
  i=5;
  temp=&i;
  cfort(&temp);
  printf
    ("Updated integer value in C: %d\n", i);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG )
POINTER*4 (ARG, Y)
INTEGER*4 Y
PRINT *,
1  'FORTRAN INTEGER ARG VALUE:', Y
Y = Y + 1
END
```

Address of an array

Sample C usage

```
#pragma linkage (cfort,FORTRAN)
#include <stdio.h>
void cfort(int(**)[]);
main()
{
  int matrix[3] = {0,1,2};
  int (*temp)[] = &matrix;
  cfort(&temp);
  printf
    ("Updated values in C: %d %d %d\n",
    matrix[0], matrix[1],
    matrix[2]);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG )
POINTER*4 (ARG, Y)
INTEGER*4 Y(3)
PRINT *,
1  'FORTRAN ARRAY ARG VALUES:', Y
DO J = 1, 3
  Y(J) = Y(J) + 1
ENDDO
END
```

Equivalent data types for Fortran to C

The following examples illustrate how C and Fortran routines within a single ILC application might code the same data types. The examples might be clearer to you if you first read “Passing data between C and Fortran” on page 89, which describes how a C routine can receive parameters that are passed by value and by reference.

16-bit signed binary integer

Sample Fortran usage

```
INTEGER*2 X
X = 5
CALL CENTRY(X)
PRINT *,
1  'UPDATED VALUE IN FORTRAN:', X
END
```

C function (by Reference)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
void centry(short int *x)
{
  printf("C int arg value: %d\n",*x);
  *x += 1;
}
```

32-bit signed binary integer

Sample Fortran usage (by Value)

```
INTEGER*4 X, Y, CENTRY
X = 5
Y = CENTRY((X))
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C function (by Value)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
int centry(int x)
{
  printf("C arg value: %d\n",x);
  return(x);
}
```

Sample Fortran usage

```
INTEGER*4 X, Y, CENTRY
X = 5
Y = CENTRY(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C function (by Reference)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
int centry(int *x)
{
  printf("C int arg value: %d\n",*x);
  return(*x);
}
```

Short floating-point number

Sample Fortran usage

```
REAL*4 X, Y, CENTRY
X = 5.0
Y = CENTRY(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C function (by Reference)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
float centry(float *x)
{
  printf("C float arg value: %f\n",*x);
  return(*x);
}
```

Long floating-point number

Sample Fortran usage (by Value)

```
REAL*8 X, Y, CENTRY
X = 12.5D0
Y = CENTRY((X))
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C function (by Value)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
double centry(double x)
{
  printf("C arg value: %f\n",x);
  return(x);
}
```

Sample Fortran usage

```
REAL*8 X, Y, CENTRY
X = 5.0D0
Y = CENTRY(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C function (by Reference)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
double centry(double *x)
{
  printf
    ("C double arg value: %f\n",*x);
  return(*x);
}
```

Extended floating-point number

Sample Fortran usage (by Value)

```
REAL*16 X, Y, CENTRY
X = 12.1Q0
Y = CENTRY((X))
PRINT *,
1 'VALUE RETURNED TO FORTRAN:', Y
END
```

C function (by Value)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
long double centry(long double x)
{
  printf("C arg value: %Lf\n",x);
  return(x);
}
```

Sample Fortran usage

```
REAL*16 X, Y, CENTRY
X = 5.0Q0
Y = CENTRY(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C function (by Reference)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
long double centry(long double *x)
{
  printf
    ("C long double arg value:
    %Lf\n", *x);
  return(*x);
}
```

Signed one-byte character data

Sample Fortran usage

```
INTEGER*1 X
X = -5
CALL CENTRY(X)
PRINT *,
1  'UPDATED VALUE IN FORTRAN:', X
END
```

C function (by Reference)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
void centry(signed char *x)
{
  printf("C char arg value: %d\n",*x);
  *x -= 1;
}
```

Unsigned one-byte character data

Sample Fortran usage

```
CHARACTER*1 X, Y
X = 'A'
CALL CENTRY(X, Y)
PRINT *,
1  'VALUE RETURNED TO FORTRAN: ', Y
END
```

C function (by Reference)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
void centry
  (unsigned char *x, unsigned char *y)
{
  printf("C char arg value: %c\n",*x);
  *y = *x;
}
```

Fixed-length character data

Sample Fortran usage

```
CHARACTER*10 X, Y
X = '1234567890'
CALL CENTRY(X, Y)
PRINT *,
1  'VALUE RETURNED TO FORTRAN: ', Y
END
```

C function (by Reference)

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
#include <string.h>
void centry(char x[10], char y[10])
{
  printf
  ("C char array arg: %10.10s\n",x);
  memcpy(y, x, 10);
}
```

Array

Sample Fortran usage

```
REAL*4 MATRIX(3) / 1.0, 2.0, 3.0 /
CALL CENTRY(MATRIX)
PRINT *,
1  'UPDATED VALUES IN FORTRAN:', MATRIX
END
```

C function

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
void centry(float x[3])
{
  int index;
  printf
  ("C arg values: %f %f %f\n",
  x[0], x[1], x[2]);
  for (index = 0; index <= 2; index++)
  x[index] -= 1.0F;
}
```

Address of an integer

Sample Fortran usage

```
POINTER*4 (P, I)
INTEGER*4 I, J
P = LOC(J)
I = 5
CALL CENTRY(P)
PRINT *,
1  'UPDATED VALUE IN FORTRAN:', I
END
```

C function

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
void centry(int **x)
{
  printf("C int arg value: %d\n",**x);
  **x += 1;
}
```

Address of an array

Sample Fortran usage

```
POINTER*4 (P, I)
INTEGER*4 I(3)
INTEGER*4 J(3) / 1, 2, 3 /
P = LOC(J)
CALL CENTRY (P)
PRINT *,
1  'UPDATED VALUES IN FORTRAN:', I
END
```

C function

```
#pragma linkage (centry,FORTRAN)
#include <stdio.h>
void centry(int (**x)[3])
{
  int index;
  printf
    ("C int array arg values: %d %d %d\n",
    (**x)[0], (**x)[1], (**x)[2]);
  for (index = 0; index <= 2; index++)
    (**x)[index] -= 1;
}
```

External data

External data in a C routine can be shared with a common block of the same name in a Fortran routine in the same load module under the following conditions:

- The C static data is declared outside of any functions
- The C external data is NORENT, either from compiling the source file as NORENT, or by marking the variable as NORENT with `#pragma variable(xxxx, NORENT)`.
- The Fortran static common blocks are either used only with one load module in an application or they are declared as private common blocks. A private common block is not shared across load modules and must by created in any of the following ways:
  - Specified or implied by the PC compiler option
  - Referenced by Fortran object code produced by the VS FORTRAN Version 2 Release 4 compiler or earlier
  - In an application that executes with the PC runtime option.

Directing output in ILC applications

Language Environment does not provide support to coordinate the use of C and Fortran files. Routines written in Fortran and C do share the Language Environment message file, however (the ddname specified in the Language Environment MSGFILE runtime option). You must manage all other files to ensure that no conflicts arise. For example, performing output operations on the same ddname is likely to cause unpredictable results.

Under C, runtime messages and other related output are directed to the default MSGFILE ddname. `stderr` output is also by default directed to the MSGFILE ddname. `stdout` is not by default directed to the MSGFILE ddname, but can be redirected to do so. Also, output from `printf` can be interspersed with output from the Fortran PRINT statement and output from Language Environment by redirecting `stdout` to `stderr` (for example, passing `1>&2` as a command-line parameter).

For more information about how to redirect C output, see *z/OS XL C/C++ Programming Guide*.

Fortran runtime messages, output written to the print unit, and other output (such as output from the SDUMP callable service) are directed to the file specified by the MSGFILE runtime option. To direct this output to the file with the ddname FT*nn*F001 (where *nn* is the two-digit error message unit number), specify the runtime option MSGFILE(FT*nn*F001). If the print unit is different than the error message unit (if the PRTUNIT and the ERRUNIT runtime options have different values), output from a PRINT statement won't be directed to the Language Environment message file.

C to Fortran condition handling

This section provides two scenarios of condition handling behavior in a C to Fortran ILC application. If an exception occurs in a C routine, the set of possible actions is as described in “Exception occurs in C” on page 100. If an exception occurs in a Fortran program, the set of possible actions is as described in “Exception occurs in Fortran” on page 102.

Some conditions can be handled only by the HLL of the routine in which the exception occurred. For example, when an ERR or IOSTAT specifier is present on a

Fortran I/O statement and an error is detected while executing that statement, the Fortran semantics take precedence over Language Environment condition handling. In this case, control returns immediately to the Fortran program and no condition is signaled to Language Environment.

See *z/OS Language Environment Programming Guide* for a detailed description of Language Environment condition handling. For information about Fortran condition handling semantics, see *VS FORTRAN Version 2 Language and Library Reference*.

Enclave-terminating language constructs

Enclaves can be terminated for reasons other than an unhandled condition of severity 2 or greater. HLL constructs that cause the termination of a single language enclave also cause the termination of a C to Fortran enclave. In Language Environment ILC, you can issue the language construct to terminate the enclave from a C or Fortran routine.

C

Examples of C language constructs that terminate the enclave are: `kill()`, `abort()`, `raise(SIGTERM)`, `raise(SIGABND)`, and `exit()`. When you use a C language construct to terminate an enclave, the T_I_S (Termination Imminent Due to STOP) condition is raised. After T_I_S has been processed and all user code has been removed from the stack, the C atexit list is honored.

Fortran

The Fortran language constructs that cause the enclave to terminate are:
- A STOP statement
- An END statement in the main routine
- A call to EXIT or SYSRCX
- A call to DUMP or CDUMP

All of the constructs listed above except the END statement cause the T_I_S (Termination Imminent due to STOP) condition to be signaled.

Exception occurs in C

This scenario describes the behavior of an application that contains a C and Fortran routine. In this scenario, a Fortran main routine invokes a C subroutine. An exception occurs in the C subroutine.

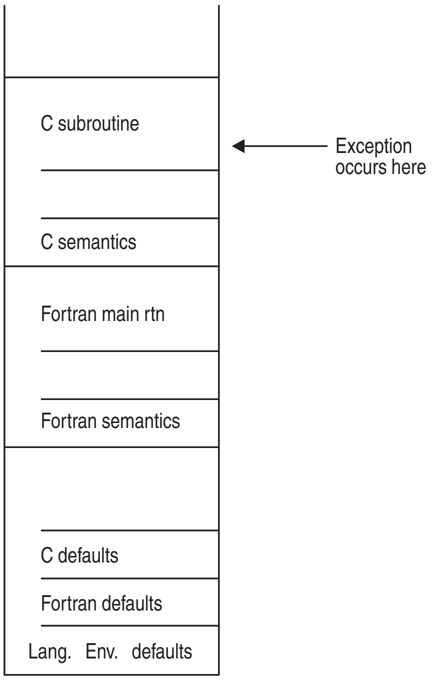


Figure 22. Stack contents when the exception occurs in C

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, C determines whether the exception in the C routine should be enabled and treated as a condition. If any of the following are true, the exception is ignored, and processing continues at the next sequential instruction after where the exception occurred:
   - You specified `SIG_IGN` for the exception in a call to `signal()`.

     Note: The system or user abend corresponding to the `signal(SIGABND)` or the Language Environment message 3250 is not ignored. The enclave is terminated.
   - The exception is one of those listed as masked in Table 63 on page 249, and you have not enabled it using the CEE3SPM callable service.
   - You did not specify any action, but the default action for the condition is `SIG_IGN` (see Table 63 on page 249).

   If you did none of these things, the condition is enabled and processed as a condition.
2. If a user-written condition handler has been registered using CEEHDLR on the C stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, there is not a user-written condition handler registered for the condition, so the condition is percolated.
3. The C global error table is now examined for signal handlers that have been registered for the condition.

   If there is a signal handler registered for the condition, it is given control. If it issues a resume or a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points.

In this example no C signal handler is registered for the condition, so the condition is percolated.

4. The condition is still unhandled. If C does not recognize the condition, or if the C default action (listed in Table 63 on page 249) is to terminate, the condition is percolated.
5. There is no user-written condition handler on the Fortran stack frame (because CEEHDLR cannot be called from a Fortran routine), and the condition is percolated.
6. If the condition is of severity 0 or 1, the Language Environment default actions take place, as described in Table 62 on page 249.
7. If the condition is of severity 2 or above, Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition) and redrive the stack. Condition handling now enters the termination imminent step.
8. If, on the second pass of the stack, no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Exception occurs in Fortran

This scenario describes the behavior of an application that contains a Fortran and a C routine. In this scenario, a C main routine invokes a Fortran subroutine. An exception occurs in the Fortran subroutine. Refer to Figure 23 throughout the following discussion.

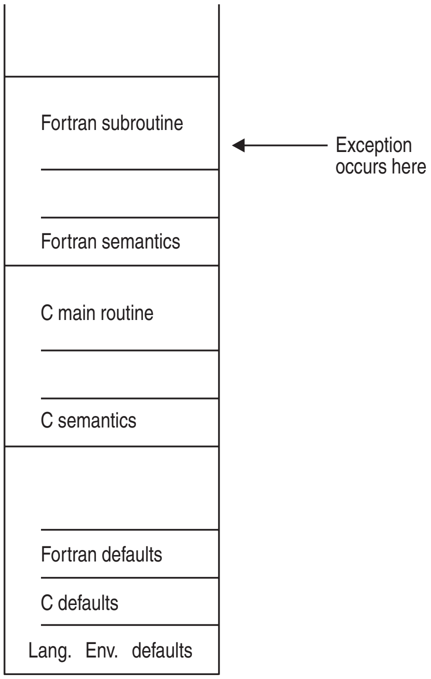


Figure 23. Stack contents when the exception occurs

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. If an I/O error is detected on a Fortran I/O statement that contains an ERR or IOSTAT specifier, Fortran semantics take precedence. The exception is not signaled to the Language Environment condition handler.
2. In the enablement step, Fortran treats all exceptions as conditions. Processing continues with the condition handling step.

3. There is no user-written condition handler on the Fortran stack frame (because CEEHDLR cannot be called from a Fortran routine), and the condition is percolated.
4. If a user-written condition handler has been registered for the condition (as specified in the global error table) using CEEHDLR on the C stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
5. If a C signal handler has been registered for the condition, it is given control. If it moves the resume cursor or issues a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points.

   In this example, no C signal handler is registered for the condition, so the condition is percolated.
6. If the condition is of severity 0 or 1, Language Environment default actions take place, as described in Table 62 on page 249.
7. If the condition is of severity 2 or above, Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition) and redrive the stack. Condition handling now enters the termination imminent step.
8. If on the second pass of the stack no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Sample ILC applications

Fortran main program that calls a C++ function:

```
@PROCESS LIST
      PROGRAM CEFOR2C
*   Module/File Name: AFHCFOR
***************************************************************
*    FUNCTION   :  Interlanguage communications call to a     *
*                  a C program.                               *
*                                                             *
*    This example illustrates an interlanguage call from      *
*    a Fortran main program to a C function.                  *
*    The parameters passed across the call from Fortran       *
*    to C have the following declarations:                    *
*                                                             *
*    Fortran INTEGER*2    to C short as pointer               *
*    Fortran INTEGER*4    to C int                            *
*    Fortran REAL*4       to C float                          *
*    Fortran REAL*8       to C double                         *
*    Fortran CHARACTER*23 to C as pointer to pointer to CHAR  *
***************************************************************
***************************************************************
*    DECLARATIONS OF VARIABLES FOR THE CALL TO C              *
***************************************************************
      INTEGER*4    J
      EXTERNAL     CECFFOR
      INTEGER*4    CECFFOR
      INTEGER*2    FOR_SHORT       / 15 /
      INTEGER*4    FOR_INT         / 31 /
      REAL*4       FOR_FLOAT       / 53.99999 /
      REAL*8       FOR_DOUBLE      / 3.14159265358979312D0 /
      POINTER*4   (FOR_POINTER, CHAR_POINTEE)
      CHARACTER*23 CHARSTRING      /'PASSED CHARACTER STRING'/
      CHARACTER*23 CHAR_POINTEE
**************************************************************
```

```
*     PROCESS STARTS HERE                                     *
**************************************************************
      PRINT *, '*********************************'
      PRINT *, 'FORTRAN CALLING C EXAMPLE STARTED'
      PRINT *, '*********************************'
      FOR_POINTER = LOC(CHARSTRING)
      PRINT *, 'CALLING C FUNCTION'
      J = CECFFOR( LOC(FOR_SHORT), FOR_INT, FOR_FLOAT,
     1   FOR_DOUBLE, LOC(FOR_POINTER))
      PRINT *,  'RETURNED FROM C FUNCTION'
      IF (J /= 999) THEN
         PRINT *, 'ERROR IN RETURN CODE FROM C'
      ENDIF
      PRINT *, '*******************************'
      PRINT *, 'FORTRAN CALLING C EXAMPLE ENDED'
      PRINT *, '*******************************'
      END /*Module/File Name:  EDCCFOR  */
```

Cfunction invoked by a Fortran program:

```
#pragma linkage (CECFFOR,FORTRAN)
 #include <stdio.h>
 #include <string.h>
 #include <math.h>

  /************************************************************
   * This is an example of a C function invoked by a Fortran  *
   * program.                                                 *
   * CECFFOR is called from Fortran program CEFOR2C with the  *
   * following list of arguments:                             *
   *  Fortran INTEGER*2    to C short as pointer              *
   *  Fortran INTEGER*4    to C int                           *
   *  Fortran REAL*4       to C float                         *
   *  Fortran REAL*8       to C double                        *
   *  Fortran CHARACTER*23 to C as pointer to pointer to char *
   ************************************************************/
  int CECFFOR (short **c_short,
                int *c_int,
                float *c_float,
                double *c_double,
                char *** c_character_string
                )
  {
     int ret=999;    /* Fortran program expects 999 returned */
     fprintf(stderr,"CECFFOR STARTED\n");
  /***********************************************************
   * Compare each passed argument against the C value.       *
   * Issue an error message for any incorrectly passed       *
   * parameter.                                              *
   ***********************************************************/
     if (**c_short != 15)
     {
       fprintf(stderr,"**c_short not = 15\n");
       --ret;
     }
     if (*c_int != 31)
     {
       fprintf(stderr,"*c_int not = 31\n");
       --ret;
     }
     if (fabs(53.99999 - *c_float) > 1.0E-5F)
     {
       fprintf(stderr,
            "fabs(53.99999 - *c_float) > 1.0E-5F, %f\n", *c_float);
       --ret;
     }
      if (fabs(3.14159265358979312 - *c_double) > 1.0E-13)
   {
     fprintf(stderr,
          "fabs(3.14159265358979312 - *c_double) > 1.0E-13\n");
     --ret;
   }
    if (memcmp(**c_character_string,"PASSED CHARACTER STRING",23)
              != 0)
   {
     fprintf(stderr,"**c_character_string not %s\n",
     "\"PASSED CHARACTER STRING\"");
     --ret;
   } /***********************************************************
 * Fortran program will check for a correct return code.   *
***********************************************************/
   fprintf(stderr,"CECFFOR ENDED\n");
   return(ret);
}
```

Chapter 7. Communicating between C++ and Fortran

This topic describes Language Environment's support for C++ and Fortran ILC applications.

General facts about C++ to Fortran ILC

- A load module consisting of object code compiled with any Fortran compiler link-edited with object code compiled in another language is not reentrant, regardless of whether the Fortran routine was compiled with the RENT compiler option.
- Return codes cannot be passed from a C++ routine to the Fortran routine that invoked it.
- Fortran routines cannot operate under CICS.
- Support for Fortran on VM is not provided by Language Environment.
- Several C++ and Fortran library routines have identical names; you will need to resolve name conflicts before link-editing your C++ to Fortran applications. See *z/OS Language Environment Programming Guide* for link-editing information.
- C++ is supported on MVS only.
- Fortran routines cannot operate in an XPLINK environment, so ILC with C routines compiled XPLINK is not supported.
- There is no ILC support between AMODE 31 and AMODE 64 applications. Fortran does not support AMODE 64.

Preparing for ILC

This section describe topics you might want to consider before writing an ILC application. To determine how different versions of HLLs work together, refer to the migration guides for the HLLs you plan to use.

Language Environment ILC support

Language Environment provides ILC support between the following combinations of C++ and Fortran:

Table 29. Supported languages for Language Environment ILC

HLL pair	C++	Fortran
C++ to Fortran	• IBM C/C++ for MVS/ESA • z/OS XL C/C++ compilers	• FORTRAN IV G1 • FORTRAN IV H Extended • VS FORTRAN Version 1, except modules compiled with Release 2.0 or earlier and that either pass character arguments to, or receive character arguments from, subprograms. • VS FORTRAN Version 2, except modules compiled with Releases 5 or 6 and whose source contained any parallel language constructs or parallel callable services, or were compiled with either of the compiler options PARALLEL or EC.

Determining the main routine

In Language Environment, only one routine can be the main routine; no other routine in the enclave can use syntax that indicates it is main.

A C++ function is designated as a main routine because its function definition gives its name as `main`. The entry point into the load module is CEESTART. In C++, the same routine can serve as both the main routine and subroutine if it is recursively called. In such a case, the new invocation of the routine is not considered a second main routine within the enclave, but a subroutine.

A Fortran routine is designated as a main routine with a PROGRAM statement, which indicates the name of the main routine. A main routine can also be designated by the absence of PROGRAM, SUBROUTINE, or FUNCTION statements, in which case the name of the main routine is the default value `MAIN` (or `MAIN#` for VS FORTRAN Version 2 Releases 5 and 6).

An entry point is defined for each supported HLL. Table 30 identifies the main and fetched entry point for each language. The table assumes that your code was compiled using the Language Environment-conforming compilers.

Table 30. Determining the entry point

HLL	Main entry point	Fetched entry point
C++	CEESTART	Not supported
Fortran	Name on the PROGRAM statement. In the absence of PROGRAM, SUBROUTINE, or FUNCTION statements, the default value `MAIN` (or `MAIN#` in VS FORTRAN Version 2 Releases 5 and 6) is used.	Subprogram name

Declaring C++ to Fortran ILC

An `extern` linkage specification is required for C++ to call Fortran and for Fortran to call C++. All entry declarations are made in the C++ code; no special declaration is required in the Fortran routine. The specification allows the C++ routine to pass arguments and return values to and from the Fortran routine using an internal argument list compatible with Fortran.

The `extern` linkage specification has the following format:

```
extern "FORTRAN" {declaration}
```

declaration is a valid C++ prototype of the Fortran program being called by C++, or the C++ routine being called by Fortran.

Example of declaration for C++ calling Fortran

The following example shows a partial C++ routine that calls a Fortran function. The calling C++ routine contains the `extern "FORTRAN"` linkage specification for the Fortran function FORTFCN, the function prototype for the Fortran function, and a static call to the Fortran function.

C++ routine

```
extern "FORTRAN"
  {double fortfcn(int, double *);}
⋮
double fortfcn(int, double [100]);
⋮
int index;
double list[100];
double value;
⋮
value=fortfcn(index, list);
```

Fortran function

```
FUNCTION FORTFCN
   (INDEX, LIST) RESULT (VALUE)
INTEGER*4 INDEX
REAL*8 LIST(0:99)
REAL*8 VALUE
⋮
VALUE=LIST(INDEX)
END
```

Example of declaration for Fortran calling C++

The following example shows a partial Fortran routine that calls a C++ function. The called C++ function contains the `extern "FORTRAN"` linkage specification for the C++ function CFCN and the function definition for the C++ function.

Fortran routine

```
INTEGER*4 INDEX
REAL*8 LIST(0:99)
REAL*8 VALUE
REAL*8 CFCN
⋮
VALUE=CFCN(INDEX, LIST)
```

C++ function

```
extern "FORTRAN"
   {double cfcn(int *, double *);}
⋮
double cfcn(int *index, double list [])
{
⋮
return list[*index];
}
```

Calling between C++ and Fortran

This section describes the types of calls permitted between C++ and Fortran, and considerations when using dynamic calls and fetch.

Types of calls permitted

Table 31 describes the types of calls between C++ and Fortran that Language Environment allows:

Table 31. Calls permitted for C++ and Fortran ILC

ILC direction	Static calls	Dynamic calls	Fetch
C++ to Fortran	Yes	N/A	C++ does not support `fetch()`
Fortran to C++	Yes	Yes	N/A

Dynamic call/fetch considerations

All of the rules described here for dynamic call and fetch assume that compiled code conforms to the list of supported products in Chapter 7, “Communicating between C++ and Fortran,” on page 107.

Fortran dynamically calling C++

Dynamic calls are made in Fortran by specifying the name of the routine to be loaded with the DYNAMIC compiler option, and then using the same name in a CALL statement. The dynamically called routine can be either C++ or Fortran, and it can in turn statically call either a C++ or Fortran routine. (If the statically linked C++ routine is within a dynamically loaded module with a Fortran entry point, the C++ routine must be either nonreentrant or naturally reentrant.) In the dynamically loaded module, a routine can dynamically call other C++ or Fortran routines, regardless of whether the routines are reentrant.

Neither a C++ nor a Fortran routine can delete a dynamically loaded routine that was dynamically loaded in a Fortran routine.

Invoking functions with returned values

Both C++ and Fortran can invoke the other language as a function that returns a value: in C++, this would be a function that returns something other than void; in Fortran, the equivalent of a function is a routine that begins with a FUNCTION statement. Only certain data types, however, can be used as function return values. See Table 34 on page 111 for a list of the supported data types that can be used as function return values.

Calling Fortran library routines

You can statically call a Fortran library routine, such as CPUTIME, from C++. However, you cannot dynamically call a Fortran library routine from C++.

Passing data between C++ and Fortran

This section describes the data types that can be passed between C++ and Fortran. In the C++-to-Fortran passing direction, most of the data types can be passed by reference only; several can be passed by value. In the Fortran-to-C++ passing direction, however, the only passing method allowed is by reference.

Under Language Environment, the term *by value* means that a temporary copy of the argument is passed to the called function or procedure. Any changes to the parameter made by the called routine will not alter the original parameter passed by the calling routine. Under Language Environment, the term *by reference* means that the actual address of the argument is passed. Any changes to the parameter made by the called routine can alter the original parameter passed by the calling routine.

Supported data types between C++ and Fortran

Table 32 lists the data types that can be passed between C++ and Fortran by reference.

Table 32. Supported data types passed by reference

C++	Fortran
signed short int	INTEGER*2
signed int, signed long int	INTEGER*4
float	REAL*4
double	REAL*8
long double	REAL*16
signed char	INTEGER*1
unsigned char	UNSIGNED*1 or CHARACTER*1
char[*n*]	CHARACTER**n*
Address of supported data types and aggregates	
Examples: Address of an integer (...int**...)	POINTER (X,Y) INTEGER*4 Y
Address of an array of integers (...int(**)[8]...)	POINTER (X,Y) INTEGER*4 Y(8)

Supported data types for passing by value

Table 33 lists the data types that can be passed from C++ to Fortran by value.

Table 33. Supported data types for passing by value from C++ to Fortran

C++	Fortran
signed int, signed long int	INTEGER*4
double	REAL*8

Table 33. Supported data types for passing by value from C++ to Fortran (continued)

C++	Fortran
long double	REAL*16

Supported data types for passing function return values

Table 34 lists the data types that can be passed as function return values from a C++ function to a Fortran routine.

Table 34. Supported data types for passing as function return values from C++ to Fortran

C++	Fortran
signed short int	INTEGER*2
signed int, signed long int	INTEGER*4
float	REAL*4
double	REAL*8
long double	REAL*16
signed char	INTEGER*1
unsigned char	UNSIGNED*1

Passing an alternate return code from Fortran to C++

You can pass an alternate return code to a C++ routine from a Fortran subroutine by specifying the called Fortran subroutine in the `extern "FORTRAN"` linkage specification. The Fortran subroutine produces an alternate return code when alternate returns are specified as dummy arguments on the SUBROUTINE statement.

In an all-Fortran application, the alternate returns provide a way to return to a point in the calling program other than to the point immediately following the CALL statement. The following example illustrates how a Fortran routine would call a Fortran subroutine to use an alternate return:

```
CALL FSUB (ARG1, *22, ARG2, *66)
```

In this example, `*22` and `*66` specify two labels (22 and 66) to which control can be passed rather than to the point following the CALL statement. The corresponding subroutine would be coded as follows:

```
SUBROUTINE FSUB (DARG1, DARG2, *, *)
```

When the FSUB subroutine executes the following RETURN statement, control would pass to the calling program at the second alternate return, at label 66.

```
RETURN 2
```

In C++, you can use the `fortrc()` function and `extern "FORTRAN"` linkage specification to get the alternate return code from the Fortran RETURN statement of the Fortran call immediately preceding it. You must not have any other C++ code between the Fortran routine call and `fortrc()`, otherwise the result is undefined.

In the following example, the C++ routine calls the subroutine FSUB, whose SUBROUTINE and RETURN statements are shown above. The `fortrc()` function returns an alternate return code of 2.

```
extern "FORTRAN" {void fsub (float, float);}
#includes <stdlib.h>
int rc:
⋮
fsub(1.0,2.0);
rc=fortrc();
⋮
```

You cannot pass return code values from a called C++ function to a calling Fortran routine.

Passing character data

Character data can be received by a Fortran routine only when the routine that receives the data declares the data to be of fixed length. Therefore, the Fortran form CHARACTER*(*) cannot be used by a Fortran routine to receive character data. An array of characters can be processed in a C++ routine only when the Fortran routine or the C++ routine produces the terminating null character.

Mapping arrays between C++ and Fortran

Arrays can be passed between C++ and Fortran routines when the array passed has its elements in contiguous storage locations. In addition, in a called Fortran routine, the declaration of the array must indicate a constant number of elements along each dimension.

In C++, arrays of more than one dimension are arranged in storage in row major order, while in Fortran they are arranged in column major order. You need to reference the corresponding element of the other language by reversing the order of the subscripts. For example in an array of floating point integers, the C++ declaration would be `float [10]` while the Fortran declaration would be `REAL*4(20,10)`.

Another difference in using arrays is that unless specified otherwise, the lower bound (the lowest subscript value) of a Fortran array is 1. In C++, the lowest subscript value is always 0, so you must adjust either the declared lower bound in the Fortran routine or the subscript you are using when you reference the value in C or Fortran.

For example, the following two arrays have the same storage mapping:

C++ `float da[10][20];`

Fortran
`REAL*4 DA(20,10)`

The following two elements also represent the same storage:

C++ `da[4][8]`

Fortran
`DA(9,5)`

Data equivalents

This section describes how C++ and Fortran data types correspond to each other.

Equivalent data types for C++ to Fortran

The following examples illustrate how C++ and Fortran routines within a single ILC application might code the same data types. The examples might be clearer to you if you first read “Passing data between C++ and Fortran” on page 110, which describes how a C++ routine can receive parameters that are passed by value and by reference.

16-bit signed binary integer

Sample C++ usage (by Reference)

```
extern "FORTRAN"
{ void cfort( short int &);; }
#include <stdio.h>
main()
{
  short int x;
  x=5;
  cfort(x);
  printf ("Updated value in C: %d\n", x);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG )
INTEGER*2 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
ARG = ARG + 1
END
```

Note: Because `short int` is an example of a parameter which must be passed using a C++ explicit pointer, you cannot code `cfort(x)`, passing `x` by value.

32-bit signed binary integer

Sample C++ usage (by Value)

```
extern "FORTRAN"
{ int cfort( int ); }
#include <stdio.h>
main()
{
  int x, y;
  x=5;
  y = cfort(x);
  printf ("Value returned to C: %d\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
INTEGER*4 CFORT
INTEGER*4 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Sample C++ usage (by Reference)

```
extern "FORTRAN"
{ int cfort( int & ); }
#include <stdio.h>
main()
{
  int x, y;
  x=5;
  y = cfort(x);
  printf ("Value returned to C: %d\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
INTEGER*4 CFORT
INTEGER*4 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Short floating-point number

Sample C++ usage (by Reference)

```
extern "FORTRAN"
{ float cfort( float & ); }
#include <stdio.h>
main()
{
  float x, y;
  x=5.0F;
  y = cfort(x);
  printf
    ("Value returned to C: %f\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
REAL*4 CFORT
REAL*4 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Long floating-point number

Sample C++ usage (by Value)

```
extern "FORTRAN"
{ double cfort(double); }
#include <stdio.h>
main()
{
  double x, y;
  x=12.5;
  y=cfort(x);
  printf
    ("Value returned to C: %f\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
REAL*8 CFORT
REAL*8 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Sample C++ usage (by Reference)

```
extern "FORTRAN"
{ double cfort(double &);; }
#include <stdio.h>
main()
{
  double x, y;
  x=12.5;
  y=cfort(x);
  printf
    ("Value returned to C: %f\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
REAL*8 CFORT
REAL*8 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Extended floating-point number

Sample C++ usage (by Value)

```
extern "FORTRAN"
{ long double cfort(long double); }
#include <stdio.h>
main()
{
  long double x, y;
  x=12.1L;
  y=cfort(x);
  printf
    ("Value returned to C: %Lf\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
REAL*16 CFORT
REAL*16 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Sample C++ usage (by Reference)

```
extern "FORTRAN"
{ long double cfort(long double &);; }
#include <stdio.h>
main()
{
  long double x, y;
  x=12.1L;
  y=cfort(x);
  printf
    ("Value returned to C: %Lf\n", y);
}
```

Fortran function

```
FUNCTION CFORT ( ARG )
REAL*16 CFORT
REAL*16 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
CFORT = ARG
END
```

Signed one-byte character data

Sample. C++ usage (by Reference)

```
extern "FORTRAN"
{ void cfort(signed char &);; }
#include <stdio.h>
main()
{
  signed char x, y;
  x=-5;
  cfort(x);
  printf
    ("Updated value in C: %d\n", x);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG )
INTEGER*1 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
ARG = ARG - 1
END
```

Unsigned one-byte character data

Sample C++ usage (by Reference)

```
extern "FORTRAN"
{ void cfort(unsigned char &,
    unsigned char &); }
#include <stdio.h>
main()
{
  unsigned char x, y;
  x='a';
  cfort(x,y);
  printf
    ("Value returned to C: %c\n", y);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG1, ARG2 )
CHARACTER*1 ARG1, ARG2
PRINT *, 'FORTRAN ARG1 VALUE: ', ARG1
ARG2 = ARG1
END
```

Fixed-length character data

Sample C++ usage

```
extern "FORTRAN"
{ void cfort(char [10], char [10]); }
#include <stdio.h>
main()
{
  char x[10] = "1234567890";
  char y[10];
  cfort(x,y);
  printf
    ("Value returned to C: %10.10s\n", y);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG1, ARG2 )
CHARACTER*10 ARG1, ARG2
PRINT *, 'FORTRAN ARG1 VALUE: ', ARG1
ARG2 = ARG1
END
```

Array

Sample C++ usage

```
extern "FORTRAN"
{ void cfort(float[]); }
#include <stdio.h>
main()
{
  float matrix[3] = {0.0F,1.0F,2.0F};
  cfort(matrix);
  printf
    ("Updated values in C: %f %f %f\n",
    matrix[0], matrix[1], matrix[2]);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG )
REAL*4 ARG(3)
PRINT *, 'FORTRAN ARG VALUES:', ARG
DO J = 1, 3
  ARG(J) = ARG(J) + 1.0
ENDDO
END
```

Address of an integer

Sample C++ usage

```
extern "FORTRAN"
{ void cfort (int *&);; }
#include <stdio.h>
main()
{
  int i, *temp;
  i=5;
  temp=&i;
  cfort(temp);
  printf
    ("Updated integer value in C: %d\n", i);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG )
POINTER*4 (ARG, Y)
INTEGER*4 Y
PRINT *,
1  'FORTRAN INTEGER ARG VALUE:', Y
Y = Y + 1
END
```

Address of an array

Sample C++ usage

```
extern "FORTRAN"
{ void cfort(int(**)[]); }
#include <stdio.h>
main()
{
  int matrix[3] = {0,1,2};
  int (*temp)[] = &matrix;
  cfort(&temp);
  printf
    ("Updated values in C: %d %d %d\n",
    matrix[0], matrix[1], matrix[2]);
}
```

Fortran subroutine

```
SUBROUTINE CFORT ( ARG )
POINTER*4 (ARG, Y)
INTEGER*4 Y(3)
PRINT *,
1  'FORTRAN ARRAY ARG VALUES:', Y
DO J = 1, 3
  Y(J) = Y(J) + 1
ENDDO
END
```

Equivalent data types for Fortran to C++

The following examples illustrate how C++ and Fortran routines within a single ILC application might code the same data types. The examples might be clearer to you if you first read "Passing data between C++ and Fortran" on page 110, which describes how a C++ routine can receive parameters that are passed by value and by reference.

16-bit signed binary integer

Sample Fortran usage

```
INTEGER*2 X
X = 5
CALL CENTRY(X)
PRINT *,
1  'UPDATED VALUE IN FORTRAN:', X
END
```

C++ function (by Reference)

```
extern "FORTRAN"
{ void centry(short int &); }
#include <stdio.h>
void centry(short int &x)
{
  printf("C int arg value: %d\n",x);
  x += 1;
}
```

32-bit signed binary integer

Sample Fortran usage (by Value)

```
INTEGER*4 X, Y, CENTRY
X = 5
Y = CENTRY((X))
PRINT *,
1 'VALUE RETURNED TO FORTRAN:', Y
END
```

C++ function (by Value)

```
extern "FORTRAN"
{ int centry(int); }
#include <stdio.h>
int centry(int x)
{
  printf("C arg value: %d\n",x);
  return(x);
}
```

Sample Fortran usage

```
INTEGER*4 X, Y, CENTRY
X = 5
Y = CENTRY(X)
PRINT *,
1 'VALUE RETURNED TO FORTRAN:', Y
END
```

C++ function (by Reference)

```
extern "FORTRAN"
{ int centry(int &); }
#include <stdio.h>
int centry(int &x)
{
  printf("C int arg value: %d\n",x);
  return(x);
}
```

Short floating-point number

Sample Fortran usage

```
REAL*4 X, Y, CENTRY
X = 5.0
Y = CENTRY(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C++ function (by Reference)

```
extern "FORTRAN"
{ float centry(float &); }
#include <stdio.h>
float centry(float &x)
{
  printf("C float arg value: %f\n",x);
  return(x);
}
```

Long floating-point number

Sample Fortran usage (by Value)

```
REAL*8 X, Y, CENTRY
X = 12.5D0
Y = CENTRY((X))
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C++ function (by Value)

```
extern "FORTRAN"
{ double centry(double); }
#include <stdio.h>
double centry(double x)
{
  printf("C arg value: %f\n",x);
  return(x);
}
```

Sample Fortran usage

```
REAL*8 X, Y, CENTRY
X = 5.0D0
Y = CENTRY(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C++ function (by Reference)

```
extern "FORTRAN"
{ double centry(double &); }
#include <stdio.h>
double centry(double &x)
{
  printf("C double arg value: %f\n",x);
  return(*x);
}
```

Extended floating-point number

Sample Fortran usage (by Value)

```
REAL*16 X, Y, CENTRY
X = 12.1Q0
Y = CENTRY((X))
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C++ function (by Value)

```
extern "FORTRAN"
{ long double centry(long double); }
#include <stdio.h>
long double centry(long double x)
{
  printf("C arg value: %Lf\n",x);
  return(x);
}
```

Sample Fortran usage

```
REAL*16 X, Y, CENTRY
X = 5.0Q0
Y = CENTRY(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

C++ function (by Reference)

```
extern "FORTRAN"
{ long double centry(long double &); }
#include
long double centry(long double &x)
{
printf
  ("C long double arg value:
    %Lf\n", x);
return(x);
}
```

Signed one-byte character data

Sample Fortran usage

```
INTEGER*1 X
X = -5
CALL CENTRY(X)
PRINT *,
1  'UPDATED VALUE IN FORTRAN:', X
END
```

C++ function (by Reference)

```
extern "FORTRAN"
{ void centry(signed char &); }
#include <stdio.h>
void centry(signed char &x)
{
  printf("C char arg value: %d\n",x);
  x -= 1;
}
```

Unsigned one-byte character data

Sample Fortran usage

```
CHARACTER*1 X, Y
X = 'A'
CALL CENTRY(X, Y)
PRINT *,
1  'VALUE RETURNED TO FORTRAN: ', Y
END
```

C++ function (by Reference)

```
extern "FORTRAN"
{ void centry
(unsigned char *, unsigned char &); }
#include <stdio.h>
void centry
  (unsigned char &x; unsigned char &y)
{
  printf("C char arg value: %c\n",x);
  y = x;
}
```

Fixed-length character data

Sample Fortran usage

```
CHARACTER*10 X, Y
X = '1234567890'
CALL CENTRY(X, Y)
PRINT *,
1  'VALUE RETURNED TO FORTRAN: ', Y
END
```

C++ function (by Reference)

```
extern "FORTRAN"
{ void centry(char [10],
char [10]); }
#include <stdio.h>
#include <string.h>
void centry(char x[10],
  char y[10])
{
  printf
    ("C char array arg: %10.10s\n",x);
  memcpy(y, x, 10);
}
```

Array

Sample Fortran usage

```
REAL*4 MATRIX(3) / 1.0, 2.0, 3.0 /
CALL CENTRY(MATRIX)
PRINT *,
1  'UPDATED VALUES IN FORTRAN:', MATRIX
END
```

C++ function

```
extern "FORTRAN"
{ void centry(float [3]); }
#include <stdio.h>
void centry(float x[3])
{
  int index;
  printf
    ("C arg values: %f %f %f\n",
    x[0], x[1], x[2]);
  for (index = 0; index <= 2; index++)
  x[index] -= 1.0F;
}
```

Address of an integer

Sample Fortran usage

```
POINTER*4 (P, I)
INTEGER*4 I, J
P = LOC(J)
I = 5
CALL CENTRY(P)
PRINT *,
1  'UPDATED VALUE IN FORTRAN:', I
END
```

C++ function

```
extern "FORTRAN"
{ void centry(int **); }
#include <stdio.h>
void centry(int **x)
{
  printf("C int arg value: %d\n",**x);
  **x += 1;
}
```

Address of an array

Sample Fortran usage

```
POINTER*4 (P, I)
INTEGER*4 I(3)
INTEGER*4 J(3) / 1, 2, 3 /
P = LOC(J)
CALL CENTRY (P)
PRINT *,
1  'UPDATED VALUES IN FORTRAN:', I
END
```

C++ function

```
extern "FORTRAN"
{ void centry(int (**)[3]); }
#include <stdio.h>
void centry(int (**x)[3])
{
  int index;
  printf
    ("C int array arg values:
    %d %d %d\n",
    (**x)[0], (**x)[1], (**x)[2]);
for (index = 0; index <= 2; index++)
  (**x)[index] -= 1;
}
```

External data

External data in a C++ routine can be shared with a common block of the same name in a Fortranroutine in the same load module under the following conditions:

- The C++ static data is declared outside of any functions
- The C++ external data is declared NORENT by using `#pragma variable(var, NORENT)`. Otherwise, C++ variables are always RENT.
- The Fortran static common blocks are either used only with one load module in an application or they are declared as private common blocks. A private common block is not shared across load modules and must by created in any of the following ways:
  - Specified or implied by the PC compiler option
  - Referenced by Fortran object code produced by the VS FORTRAN Version 2 Release 4 compiler or earlier
  - In an application that executes with the PC runtime option

Directing output in ILC applications

Language Environment does not provide support to coordinate the use of most C++ and Fortran files; however they can share the Language Environment message file which is the *ddname* specified in the Language Environment MSGFILE runtime option. You must manage all other files to ensure that no conflicts arise. For example, performing output operations on the same ddname is likely to cause unpredictable results.

Under C++, runtime messages and other related output are directed to the default MSGFILE ddname. `stderr` output is also by default directed to the MSGFILE ddname. `stdout` is not, by default, directed to the MSGFILE ddname, but can be redirected to do so. Also, output from `printf` can be interspersed with output from the Fortran PRINT statement and output from Language Environment by redirecting `stdout` to `stderr` (for example, passing `1>&2` as a command-line parameter).

For more information about how to redirect C++ output, see *z/OS XL C/C++ Programming Guide*.

Fortran runtime messages, output written to the print unit, and other output (such as output from the SDUMP callable service) are directed to the file specified by the MSGFILE runtime option. To direct this output to the file with the ddname FT*nn*F001, (where *nn* is the two-digit error message unit number), specify the runtime option MSGFILE(FT*nn*F001). If the print unit is different than the error message unit (if the PRTUNIT and the ERRUNIT runtime options have different values), output from a PRINT statement won't be directed to the Language Environment message file.

C++ to Fortran condition handling

This section provides two scenarios of condition handling behavior in a C to Fortran ILC application. If an exception occurs in a C routine, the set of possible actions is as described in “Exception occurs in C” on page 100. If an exception occurs in a Fortran program, the set of possible actions is as described in “Exception occurs in Fortran” on page 102.

Some conditions can be handled only by the HLL of the routine in which the exception occurred. For example, when an ERR or IOSTAT specifier is present on a Fortran I/O statement and an error is detected while executing that statement, the

Fortran semantics take precedence over Language Environment condition handling. In this case, control returns immediately to the Fortran program and no condition is signaled to Language Environment.

C++ exception handling constructs `try()/throw()/catch()` cannot be used with Language Environment and Fortran condition handling. If you use C exception handling constructs (`signal()/raise()`) in your C++ routine, condition handling will proceed as described in this section. Otherwise, you will get undefined behavior in your programs if you mix the C++ constructs with the C constructs.

See *z/OS Language Environment Programming Guide* for a detailed description of Language Environment condition handling. For information about Fortran condition handling semantics, see *VS FORTRAN Version 2 Language and Library Reference*.

Enclave-terminating language constructs

Enclaves can be terminated for reasons other than an unhandled condition of severity 2 or greater. HLL constructs that cause the termination of a single language enclave also cause the termination of a C to Fortran enclave. In Language Environment ILC, you can issue the language construct to terminate the enclave from a C++ or Fortran routine.

C language constructs available under C++

Among the C language constructs that terminate an enclave are `abort()`, `exit()`, `raise(SIGABND)`, and `raise(SIGTERM)`. When you use a C language construct to terminate an enclave, the T_I_S (Termination Imminent Due to STOP) condition is raised. After T_I_S has been processed and all user code has been removed from the stack, the C atexit list is honored.

Fortran

The Fortran language constructs that cause the enclave to terminate are:
- A STOP statement
- An END statement in the main routine
- A call to EXIT or SYSRCX
- A call to DUMP or CDUMP

All of the constructs listed above except the END statement cause the T_I_S (Termination Imminent due to STOP) condition to be signaled.

Exception occurs in C++

This scenario describes the behavior of an application that contains a C++ and Fortran routine. In this scenario, a Fortran main routine invokes a C++ subroutine. An exception occurs in the C++ subroutine. Refer to Figure 24 on page 122 throughout the following discussion.

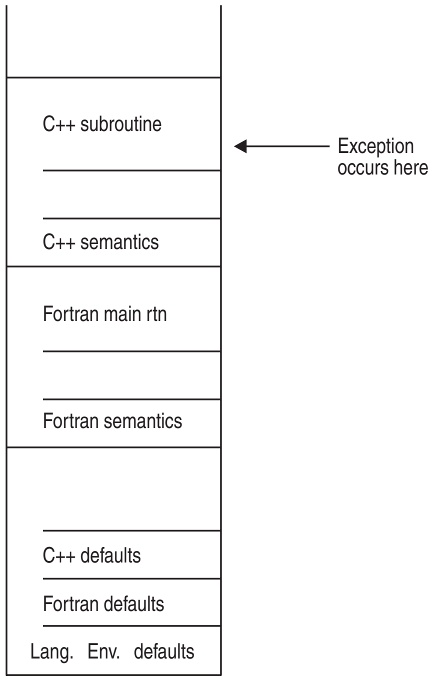


Figure 24. Stack contents when the exception occurs in C++

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, C++ determines whether the exception in the C++ routine should be enabled and treated as a condition. If any of the following are true, the exception is ignored, and processing continues at the next sequential instruction after where the exception occurred:
   - You specified `SIG_IGN` for the exception in a call to `signal()`.

     Note: The system or user abend corresponding to the `signal(SIGABND)` or the Language Environment message 3250 is not ignored. The enclave is terminated.
   - The exception is one of those listed as masked in Table 63 on page 249, and you have not enabled it using the CEE3SPM callable service.
   - You did not specify any action, but the default action for the condition is `SIG_IGN` (see Table 63 on page 249).

   If you did none of these things, the condition is enabled and processed as a condition.
2. If a user-written condition handler has been registered using CEEHDLR on the C++ stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, there is not a user-written condition handler registered for the condition, so the condition is percolated.
3. The C global error table is now examined for signal handlers that have been registered for the condition.

   If there is a signal handler registered for the condition, it is given control. If it issues a resume or a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points.

In this example no C++ signal handler is registered for the condition, so the condition is percolated.

4. The condition is still unhandled. If C++ does not recognize the condition, or if the C++ default action (listed in Table 63 on page 249) is to terminate, the condition is percolated.
5. There is no user-written condition handler on the Fortran stack frame (because CEEHDLR cannot be called from a Fortran routine), and the condition is percolated.
6. If the condition is of severity 0 or 1, the Language Environment default actions take place, as described in Table 62 on page 249.
7. If the condition is of severity 2 or above, Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition) and redrive the stack. Condition handling now enters the termination imminent step.
8. If, on the second pass of the stack, no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Exception occurs in Fortran

This scenario describes the behavior of an application that contains a Fortran and a C++ routine. In this scenario, a C++ main routine invokes a Fortran subroutine. An exception occurs in the Fortran subroutine. Refer to Figure 25 throughout the following discussion.

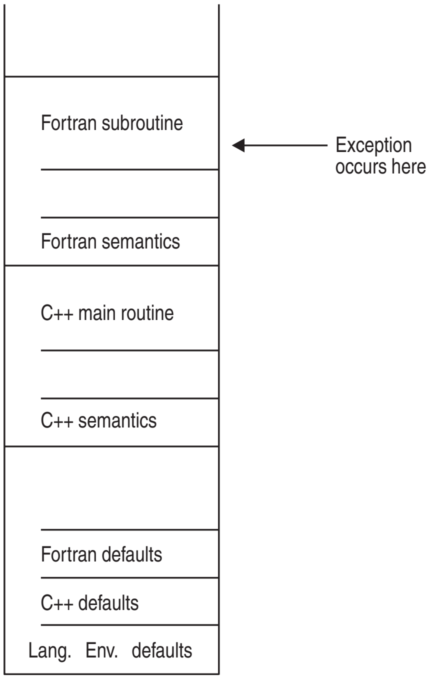


Figure 25. Stack contents when the exception occurs in Fortran

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. If an I/O error is detected on a Fortran I/O statement that contains an ERR or IOSTAT specifier, Fortran semantics take precedence. The exception is not signaled to the Language Environment condition handler.
2. In the enablement step, Fortran treats all exceptions as conditions. Processing continues with the condition handling step.

3. There is no user-written condition handler on the Fortran stack frame (because CEEHDLR cannot be called from a Fortran routine), and the condition is percolated.
4. If a user-written condition handler has been registered for the condition (as specified in the C global error table) using CEEHDLR on the C++ stack frame, it is given control. If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
5. If a C signal handler has been registered for the condition, it is given control. If it moves the resume cursor or issues a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points.

   In this example, no C signal handler is registered for the condition, so the condition is percolated.
6. If the condition is of severity 0 or 1, Language Environment default actions take place, as described in Table 62 on page 249.
7. If the condition is of severity 2 or above, Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition) and redrive the stack. Condition handling now enters the termination imminent step.
8. If on the second pass of the stack no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Sample ILC applications

```
@PROCESS LIST
      PROGRAM CEFO2CP
*   Module/File Name: AFHCPFOR
*************************************************************
*    FUNCTION    : Interlanguage communications call to     *
*                  a C++ program.                           *
*                                                           *
*    This example illustrates an interlanguage call from    *
*    a Fortran main program to a C++ function.              *
*    The parameters passed across the call from Fortran     *
*    to C++ have the following declarations:                *
*                                                           *
*    Fortran INTEGER*2    to C++ short as pointer           *
*    Fortran INTEGER*4    to C++ int                        *
*    Fortran REAL*4       to C++ float                      *
*    Fortran REAL*8       to C++ double                     *
*    Fortran CHARACTER*23 to C++ as char                    *
*************************************************************
*************************************************************
*    DECLARATIONS OF VARIABLES FOR THE CALL TO C++          *
*************************************************************
      INTEGER*4        J
      EXTERNAL         CECFFOR
      INTEGER*4        CECFFOR
      INTEGER*2        FOR_SHORT        / 15 /
      INTEGER*4        FOR_INT          / 31 /
      REAL*4           FOR_FLOAT        / 53.99999 /
      REAL*8           FOR_DOUBLE       / 3.14159265358979312D0 /
      CHARACTER*23     CHARSTRING       /'PASSED CHARACTER STRING'/
*************************************************************
*     PROCESS STARTS HERE                                   *
*************************************************************
      PRINT *, '*****************************************'
      PRINT *, 'FORTRAN CALLING C++ EXAMPLE STARTED'    *
```

```
     PRINT *, '*****************************************'
     FOR_POINTER = LOC(CHARSTRING)
     PRINT *, 'CALLING C++ DUNCTION'
     J = CECFFOR( FOR_SHORT, FOR_INT, FOR_FLOAT,
    1   FOR_DOUBLE, CHARSTRING)
     PRINT *,  'RETURNED FROM C++ FUNCTION'
     IF (J /= 999) THEN
        PRINT *, 'ERROR IN RETURN CODE FROM C++'
     ENDIF
     PRINT *, '***************************************'
     PRINT *, 'FORTRAN CALLING C++ EXAMPLE ENDED'    *
     PRINT *, '***************************************'
     END
/*Module/File Name:  EDCCPFOR  */

extern "FORTRAN"
  { int CECFFOR (short &, int &, float &, double &, char * ) }
#include <stdio.h>
#include <string.h>
#include <math.h>
 /**************************************************************
  * This is an example of a C++ function invoked by a         *
  * Fortran program.                                          *
  * CECFFOR is called from Fortran program CEFOR2CP with the  *
  * following list of arguments:                              *
  *  Fortran INTEGER*2      to C short                        *
  *  Fortran INTEGER*4      to C int                          *
  *  Fortran REAL*4         to C float                        *
  *  Fortran REAL*8         to C double                       *
  *  Fortran CHARACTER* 23  to C char                         *
  **************************************************************/
 int CECFFOR (short & c_short,
              int  & c_int,
              float & c_float,
              double & c_double,
              char * c_character_string
              )
{
   int ret=999;    /* Fortran program expects 999 returned */
   fprintf(stderr,"CECFFOR STARTED\n");
/************************************************************
 * Compare each passed argument against the C value.        *
 * Issue an error message for any incorrectly passed        *
 * parameter.                                               *
 ***********************************************************/
   if (c_short != 15)
   {
     fprintf(stderr,"c_short not = 15\n");
     --ret;
   }

   if (c_int != 31)
   {
     fprintf(stderr,"c_int not = 31\n");
     --ret;
   }

   if (fabs(53.99999 - c_float) > 1.0E-5F)
   {
     fprintf(stderr,
          "fabs(53.99999 - c_float) > 1.0E-5F, %f\n", c_float);
     --ret;
   }

   if (fabs(3.14159265358979312 - c_double) > 1.0E-13)
   {
     fprintf(stderr,
         "fabs(3.14159265358979312 - c_double) > 1.0E-13,
          %f.14\n",c_double);
    --ret;
  }  if (memcmp(c_character_string,"PASSED CHARACTER STRING",23)
             != 0)

  {
    fprintf(stderr,"c_character_string not %s\n",
    "\"PASSED CHARACTER STRING\"");
    --ret;
  }
 /****************************************************************
 * Fortran program will check for a correct return code.         *
 ****************************************************************/
    fprintf(stderr,"CECFFOR ENDED\n");
    return(ret);
}
```

Chapter 8. Communicating between C and PL/I

This topic describes Language Environment's support for C and PL/I ILC applications. If you are running a C to PL/I ILC application under CICS, you should also consult Chapter 15, "ILC under CICS," on page 241.

General facts about C to PL/I ILC

- ILC between C and Enterprise PL/I for z/OS is discussed in the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).
- PL/I routines can run in non-Initial Process Threads (non-IPTs) created by the C routines in z/OS UNIX-conforming C to PL/I applications.
- PL/I Multitasking Facility (MTF) does not support C.
- With Enterprise PL/I for z/OS, a C program that is fetched from a PL/I main must be compiled with the most recent C compiler. If another version of C is desired, specify OPTIONS(ASM).
- Language Environment does not support passing return codes between PL/I and C routines in an ILC application.
- A C NULL is X'00000000'; a PL/I NULL is X'FF000000'; a PL/I SYSNULL is X'00000000'. Comparisons against a NULL value and other uses of the NULL value must therefore be done with care.
- There is no ILC support between AMODE 31 and AMODE 64 applications. PL/I does not support AMODE 64.

Preparing for C to PL/I ILC

This section describes topics you might want to consider before writing an application that uses ILC. For help in determining how different versions of HLLs work together, refer to the migration guides for the HLLs you plan to use.

Language Environment ILC support

Table 35. Supported languages for Language Environment ILC

HLL pair	C	PL/I
C–PL/I	• C/370 Version 1 • C/370 Version 2 • AD/Cycle C/370 Version 1 • IBM C/C++ for MVS/ESA • z/OS XL C/C++ compilers	• OS PL/I Version 2 Release 2 or later • Enterprise PL/I for z/OS • PL/I for MVS & VM

Note: In this chapter, C refers to both the pre-Language Environment- and Language Environment-conforming versions of C. PL/I refers to OS PL/I, Enterprise PL/I for z/OS, and PL/I for MVS & VM.

Migrating C to PL/I ILC applications

Language Environment allows you to run ILC applications that were compiled under previous versions of C and PL/I. In general, you do not need to relink or recompile an existing C to PL/I ILC application in Language Environment.

Determining the main routine

In Language Environment, only one routine can be the main routine. If a PL/I routine is identified as a main routine in an ILC application using PROC OPTIONS(MAIN) and a C main function does not exist, the PL/I main routine is the first to gain control. If a C main function exists, but no PL/I main routine is identified in the ILC application, the C main function gains control first.

If both a PL/I main routine identified by PROC OPTIONS(MAIN) and a C main function exist in the same ILC application, this is a user error. However, the error is not detected by Language Environment.

An entry point is defined for each supported HLL. Table 36 identifies the desired entry point. The table assumes that your code was compiled using the Language Environment-conforming compilers.

Table 36. Determining the entry point

HLL	Main entry point	Fetched entry point
C	CEESTART	CEESTART or routine name if `#pragma linkage(,fetchable)` is not used.
PL/I	CEESTART	CEESTART if OPTIONS(FETCHABLE) is used, or routine name.

When link-editing a PL/I module that is fetched, the name of the routine that is being fetched must be the entry point of the load module, unless the FETCHABLE option is specified on the PROCEDURE statement in the PL/I routine. When link-editing a C module that is fetched, the name of the routine being fetched must be the entry point of the load module, unless `#pragma linkage (,fetchable)` is specified in the C routine. You cannot have more than one entry point in an ILC application with `#pragma linkage (,fetchable)` or PL/I FETCHABLE option on the PROCEDURE statement. This error is not detected by Language Environment, but can cause unpredictable results.

Declaring C to PL/I ILC

Declaring a C entry point in a PL/I routine has the same syntax as declaring another PL/I entry point. A C routine can be replaced by a PL/I routine without altering the PL/I code that calls the routine. Likewise, if a C routine calls a PL/I routine, the PL/I procedure contains no explicit declaration indicating control is being passed from the C routine. The declaration is contained within the C routine.

In C, you must declare that the C entry point receives control from a PL/I routine. This declaration is in the form of a `pragma`. The body of the C function is the same as if the routine were called from another C function. Calling a PL/I routine and being called by a PL/I routine are handled by the same `#pragma` preprocessor directive. No special linkage declaration is required for ILC between C and Enterprise PL/I for z/OS.

Declaration for C calling PL/I

C function

```
#pragma linkage(PLIFUNC, PLI)
double PLIFUNC( double );     / C prototype /

int main()
{
  double val,result;

  val=6.2
  result=PLIFUNC(val);
  printf("val=%f\n",val result);
}
```

PL/I routine

```
PLIFUNC: Proc(arg) options(reentrant)
    returns(float binary(53));
  Dcl arg float binary(53);

  Return (34.0);
  End;
```

Declaration for PL/I calling C

PL/I Routine

```
PLIPROG:  Proc options(main, reentrant);
  Dcl cfunc external entry
    returns(fixed bin(31));
  Dcl arg fixed bin(31);
  Dcl a fixed bin(31);
  Arg = 10;
  A = cfunc(arg);
  End;
```

C function

```
#pragma linkage(CFUNC, PLI)
int CFUNC( int parm ) {

return (5);
}
```

Building a reentrant C to PL/I application

A PL/I to C application can be constructed to be reentrant. Compile all PL/I routines in an ILC application by using the REENTRANT option of the OPTIONS attribute of the PROCEDURE statement. Compile all your C routines with RENT.

Calling between C and PL/I

This section describes the types of calls permitted between C and PL/I as well as dynamic call/fetch considerations.

Types of calls permitted

Table 37 describes the types of calls between C and PL/I that Language Environment allows:

Table 37. Calls permitted for C and PL/I

ILC direction	Static calls	Dynamic calls using DLLs	Fetch/Call	Comments
C to PL/I	Yes	Yes (1)	Yes	
PL/I to C	Yes	Yes (1)	Yes	C must be non-reentrant or naturally reentrant.

Note: Enterprise PL/I for z/OS supports calls to a C DLL, and also allows calls from C to a PL/I DLL. In this case, the C code could also optionally be compiled with the XPLINK option.

Dynamic call/fetch considerations

Both PL/I and C can specify only one fetchable entry point for an entire load module. In general, target routines need to be recompiled with a Language

Environment-conforming compiler. C fetching PL/I is the only exception. If a PL/I routine is resident in the main load module, the target routine does not need to be recompiled.

If a load module is introduced as a result of the PL/I FETCH statement or the C `fetch()` function and the load module contains any ILC or the fetching and fetched routines are written in different languages, then the load module cannot be deleted using a corresponding PL/I RELEASE statement or the C `release()` function.

PL/I fetch cannot be used to load a C function that was compiled with the XPLINK option.

User-written condition handlers registered using CEEHDLR can be fetched, but must be written in the same language as the fetching language.

C fetching PL/I

The C `fetch()` function supports fetching a PL/I routine and subsequent invocation using a function pointer. The fetched PL/I routine can make additional calls (either static or dynamic) to other C routines. When a C routine issues the fetch to a load module having a PL/I entry point with statically linked C routines, the C routines might have constructed reentrancy.

When a PL/I routine is dynamically introduced into the enclave as a result of a fetch, the fetch restrictions described in the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735) apply. Enterprise PL/I for z/OS has lifted some of the fetch restrictions. For more information, see the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).

If a PL/I procedure is to be dynamically loaded, you must specify one of the following:

- The routine name as the entry point when you link-edit it as described in the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).
- OPTIONS(FETCHABLE) on the PROCEDURE statement and recompile.

The declaration of a PL/I fetched routine within a C routine is shown in Figure 26.

```
typedef int PLIFUNC();
#pragma linkage (PLIFUNC, PLI)
    .
    .
    .

PLIFUNC *fetch_ptr;
fetch_ptr = (PLIFUNC*) fetch("PLIENT");  /* fetch the routine */
fetch_ptr(args);                         /* call PLIENT       */
```

Figure 26. C fetching a PL/I routine

PL/I fetching C

A PL/I routine can fetch a C function or another PL/I routine that is statically linked to a C function. Any C routine that is either directly or indirectly fetched by PL/I must be either naturally reentrant or be non-reentrant (that is, it cannot have constructed reentrancy via the RENT option and the prelinker).

The declaration of a C fetched routine within a PL/I routine is shown in Figure 27.

```
DCL CENTRY EXTERNAL ENTRY;  /* declare C entry point */
    .
    .
    .

FETCH CENTRY;         /* fetch the routine     */
CALL CENTRY(args);    /* call routine          */
```

Figure 27. PL/I fetching a C routine

Passing data between C and PL/I

There are two sets of data types that you can pass between C and PL/I routines: data types passed *by reference* using C explicit pointers in the routine, and data types passed *by value* without using C explicit pointers.

When a parameter is passed by reference, the parameter itself is passed. A copy of the parameter is not made. Any changes to the parameter made by the called routine can alter the original parameter passed by the calling routine. When a parameter is passed by value, a copy of the parameter is passed. Any changes to the parameter made by the called routine cannot alter the original parameter passed by the calling routine.

Passing pointers from C to PL/I

Pointers can be passed and returned between C and PL/I routines. Because the C `#pragma linkage(PLI)` specifies that pointers, unlike other parameters, are passed directly, there is one level of indirection less on the PL/I side.

In order for PL/I to receive a pointer to a PL/I data type, C must pass a pointer to a pointer to the C data type. Conversely, if PL/I returns a pointer to a data type, C receives a pointer to a pointer to the data type.

Structures, arrays, and strings should be passed between C and PL/I only by using pointers.

The non-address bits in all fullword pointers declared in PL/I source code must always be zero. If they are not, results are unpredictable.

Passing pointers from PL/I to C

Pointers to various data objects can be passed from PL/I and accepted by a function written in C.

Because the C `#pragma linkage(PLI)` specifies that pointers, unlike other parameters, are passed directly, an extra level of indirection is added when passing a pointer value from PL/I to C. If PL/I passes or returns a pointer to a type, C receives a pointer to a pointer to the type.

PL/I parameters that require a locator or descriptor should not be passed directly. This includes parameters that are structures, arrays, or strings. These parameters can be passed indirectly from PL/I by using a pointer to the associated data. For more information about data descriptors, see the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).

The non-address bits in all fullword pointers declared in PL/I source code must always be zero. If they are not, results are unpredictable.

Receiving value parameters in C

If you enclose in parentheses the argument you pass from a PL/I routine to a C routine, the argument is passed by value. C should receive the parameter as the equivalent C type. The XL C compiler generates the appropriate addressing code required to access the parameter values.

You can write your PL/I-callable function as if it were in a C-only environment; you can move it to a C-only environment by removing the `#pragma` directive.

Receiving reference parameters in C

If you do not enclose in parentheses the argument you pass from a PL/Iroutine to a C routine, the argument is passed by reference. C should receive the parameter as a pointer to the equivalent C type.

For example, if a C function named `FROMPLI` is called from PL/Iwith an integer argument, the C prototype declaration should be:

```
int FROMPLI(int *);
```

A parameter passed from PL/I by reference is received and used by C as a value parameter provided that its value is not altered. If the value of such a parameter is altered, the effect on the original PL/I variable is undefined.

Data types passed using C pointers (by reference)

Table 38 identifies the data types that can be passed as parameters between C and PL/I applications with the use of explicit pointers, or by reference, under C. Conversely, reference parameters passed by PL/I to C are received as pointers to the equivalent data type.

Table 38. Supported data types between C and PL/I using C pointers (by reference)

C	PL/I
signed short int	REAL FIXED BINARY(15,0)
signed int	REAL FIXED BINARY(31,0)
signed long int	REAL FIXED BINARY(31,0)
float	FLOAT BINARY(21) FLOAT DECIMAL(06) FLOAT BINARY (21) is the preferred equivalent for float.
double	FLOAT BINARY(53) FLOAT DECIMAL(16) FLOAT BINARY (53) is the preferred equivalent for double.
long double	FLOAT BINARY(109) FLOAT DECIMAL(33) FLOAT BINARY (109) is the preferred equivalent for long double.
pointer to. . .	POINTER
decimal(n,p)	FIXED DECIMAL(n,p)
Note: Data storage alignment must match.	

Data types passed by value

Table 39 on page 133 identifies the data types that can be passed as parameters between C and PL/I applications without the use of explicit C pointers. Parameters that are not pointers are passed by value.

In order for a C routine to pass a parameter without using a pointer, the argument should be passed, and the PL/I routine should receive the parameter as the equivalent PL/I data type.

Table 39. Supported data types between C and PL/I without using C pointers (by value)

C	PL/I
signed int	REAL FIXED BINARY(31,0)
signed long int	REAL FIXED BINARY(31,0)
double	FLOAT DECIMAL(16)
double	FLOAT BINARY(53)
long double	FLOAT DECIMAL(33)
long double	FLOAT BINARY(109)
decimal(n,p)	FIXED DECIMAL(n,p)

Note: The preferred PL/I data declarations for the C double and long double data types are FLOAT BINARY(53) and FLOAT BINARY(109), respectively. Data storage alignment must match.

Passing strings between C and PL/I

C and PL/I have different string data types:

C strings
Logically unbounded length and are terminated by a NULL (the last byte of the string contains X'00').

PL/I CHAR(n) VARYING
A halfword-prefixed string of characters with a maximum length *n* characters. The current length is held in the halfword prefix.

PL/I CHAR(n) VARYINGZ
A null-terminated string of characters with a maximum length of *n* characters.

PL/I CHAR(n)
A fixed-length string of characters of length *n*. There is no halfword prefix indicating the length.

You can pass strings between C and PL/I routines, but you must match what the routine interface demands with what is physically passed.

Using aggregates

Aggregates (arrays, strings, or structures) are mapped differently by C and PL/I and are not automatically mapped. Be sure to completely declare every byte in the aggregate so there are no open fields. Doing so helps ensure that the layouts of aggregates passed between the two languages map to one another correctly. The C and PL/I AGGREGATE compile-time options provide a layout of aggregates to help you perform the mapping.

For more information about PL/I structure mapping, see the appropriate language reference and programming guide.

Data equivalents

This section describes how C and PL/I data types correspond to each other.

Equivalent data types for C to PL/I

The following examples illustrate how C and PL/I routines within a single ILC application might code the same data types. The examples might be clearer to you if you first read "Passing data between C and PL/I" on page 131, which describes how a C routine can receive parameters that are passed by value and by reference.

16-bit signed binary integer

Sample C usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
short int cpli( short int*);
main() {
  short int x, y;
  x=5;
  y = cpli(&x);  /* by reference */
}
```

PL/I subroutine

```
CPLI: PROC(ARG) RETURNS (FIXED BIN(15));
    DCL ARG FIXED BIN (15);
⋮
    RETURN (ARG);
    END;
```

Note: Because `short int` is an example of a parameter which must be passed using a C explicit pointer, you cannot code `y = cpli(x)`, passing `x` by value.

32-bit signed binary integer

Sample C usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
int extern cpli( int );
main() {
  int x, y;
  x=5;
  y = cpli(x);  /* by value */
}
```

PL/I subroutine

```
CPLI: PROC(ARG) RETURNS (FIXED BIN(31));
    DCL ARG FIXED BIN (31);
⋮
    RETURN (ARG);
    END;
```

Sample C usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
int extern cpli( int *);
main() {
  int x, y;
  x=5;
  y = cpli(&x);  /* by reference */
}
```

PL/I subroutine

```
CPLI: PROC(ARG) RETURNS (FIXED BIN(31));
    DCL ARG FIXED BIN (31);
⋮
    RETURN (ARG);
    END;
```

Long floating-point number

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
main()
{
  void cpli(double);
  double x, y;
  x=12.5;
  cpli(x);  /* by value */
}
```

PL/I subroutine

```
CPLI: PROC(ARG)
       RETURNS (FLOAT BINARY(53));
    DCL ARG FLOAT BINARY(53);
⋮
    RETURN (34.0);
    END;
```

Sample C usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
main()
{
  void double cpli(double*);
  double x;
  x=12.5;
  cpli(&x);  /* by reference */

}
```

PL/I subroutine

```
CPLI: PROC(ARG);
    DCL ARG FLOAT BINARY(53);
⋮
    END;
```

Extended floating-point number

Sample C usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
main()
{
  long double cpli(long double);
  long double x, y;
  x=12.1;
  y=cpli(x); /* by value    */
}
```

PL/I subroutine

```
CPLI: PROC(ARG) RETURNS (FLOAT BIN(109));
    DCL ARG FLOAT BIN(109);
⋮
    RETURN (ARG);
    END;
```

Sample C usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
main()
{
  long double cpli(long double*);
  long double x, y;
  x=12.01.../* many digits */;
  y=cpli(&x);     /* by reference */
}
```

PL/I subroutine

```
CPLI: PROC(ARG)
    RETURNS (FLOAT BIN(109));
    DCL ARG FLOAT BIN(109);
⋮
    RETURN (ARG);
    END;
```

Pointer to an Integer

Sample C usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
main()
{
  int i, *temp;
  void cpli (int **);
  i = 5;
  temp=&i;
  cpli(&temp);
}
```

PL/I subroutine

```
CPLI: PROC (ARG);
    DCL ARG POINTER;
    DCL ART FIXED BIN(31,0) BASED (ARG);
⋮
    END;
```

Pointer to an array

Sample C usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
main()
{
  int matrix[5];
  int *temp([] = &matrix);
  int i;
  void cpli(int(**)[]);
  for(i=0;i<5;i++);
    matrix[i] = i;
  cpli(&temp);
}
```

PL/I subroutine

```
CPLI: PROC (ARG);
    DCL ARG POINTER;
    DCL I FIXED BIN(31)
    DCL ART(5) FIXED BIN(31,0) BASED(ARG);
⋮
    END;
```

Pointer to a structure

Sample C usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
main()
{
  struct date   {
    int day;
    int month;
    int year   } today;
  struct date *temp = &today;
  void cpli(struct date **);
  int i;
cpli (&temp);
}
```

PL/I subroutine

```
CPLI: PROC (ARG);
    DCL ARG POINTER;
    DCL 1 TODAY BASED (ARG),
        2 DAY FIXED BIN(31),
        2 MONTH FIXED BIN(31),
        2 YEAR FIXED BIN(31);
⋮
END;
```

Fixed-length decimal data

Sample C usage

```
#pragma linkage (pdec, PLI)
#include <stdio.h>
#include <decimal.h>
decimal(5,2) gpd;
main()   {
  decimal(5,2) pd1;
  printf("Packed decimal text\n");
  pd1 = 52d;
  pdec(pd1);
  if (gpd ! = 57d)
{
    printf("Fixed decimal error\n");
    printf("Expect: 57\n");
    printf("Result: %D(5,2)\n",gpd);
}
  printf("Value: %D(5,2)\n:, gpd);
  printf("Finished test\n");
}
```

PL/I subroutine

```
PDEC: PROC (X);
    DCL X FIXED DEC(5,2);
    DCL GPD FIXED DEC(5,2) EXTERNAL;
    X = X+5;
    GPD = X;
    END;
```

Equivalent data types for PL/I to C

The following examples illustrate how C and PL/I routines within a single ILC application might code the same data types. The examples might be clearer to you if you first read "Passing data between C and PL/I" on page 131, which describes how a C routine can receive parameters that are passed by value and by reference.

32-bit signed binary integer

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
    DCL CENTRY EXTERNAL ENTRY
      RETURNS (FIXED BIN(31));
    DCL X FIXED BIN(31);
    DCL Y FIXED BIN(31);
    X = 5;
    /* BY VALUE */
    Y=CENTRY((X));
END MY_PROG;
```

C function

```
#pragma linkage (centry,PLI)
#include <stdio.h>
int centry(int x)
{
  printf("x is %d\n",x);
  return(x);
}
```

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
    DCL CENTRY EXTERNAL ENTRY
      RETURNS (FIXED BIN(31));
    DCL X FIXED BIN(31);
    DCL Y FIXED BIN(31);
    X = 5;
    /* BY REFERENCE */
    Y=CENTRY(X);
END MY_PROG;
```

C function

```
#pragma linkage (centry,PLI)
#include <stdio.h>
int centry(int *x)
{
  printf("*x is %d\n",x);
  return(*x);
}
```

Long floating-point number

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
    DCL CENTRY EXTERNAL ENTRY
      RETURNS (FLOAT DEC(16));
    DCL X FLOAT DEC(16);
    DCL Y FLOAT DEC(16);
    X = 3.14159265;
    /* BY VALUE */
    Y=CENTRY((X));
END MY_PROG;
```

C function

```
#pragma linkage (centry,PLI)
#include <stdio.h>
double centry(double x)
{
  printf("x is %f\n",x);
  return(x);
}
```

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
    DCL CENTRY EXTERNAL ENTRY
      RETURNS (FLOAT DEC(16));
    DCL X FLOAT DEC(16);
    DCL Y FLOAT DEC(16);
    X = 3.14159265;
    /* BY REFERENCE */
   Y=CENTRY(X);
END MY_PROG;
```

C function

```
#pragma linkage (centry,PLI)
#include <stdio.h>
double centry(double *x)
{
  printf("*x is %f\n",x);
  return(*x);
}
```

Extended floating-point number

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
    DCL CENTRY EXTERNAL ENTRY
      RETURNS (FLOAT DEC(33));
    DCL X FLOAT DEC(33);
    DCL Y FLOAT DEC(33);
    X = 12.5;
    /* BY VALUE */
    Y=CENTRY((X));
END MY_PROG;
```

C function

```
#pragma linkage (centry,PLI)
#include <stdio.h>
long double centry(long double x)
{
  printf("x is %Lf\n",x);
  return(x);
}
```

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
    DCL CENTRY EXTERNAL ENTRY
      RETURNS (FLOAT DEC(33));
    DCL X FLOAT DEC(33);
    DCL Y FLOAT DEC(33);
    X = 12.5;
    /* BY REFERENCE */
    Y=CENTRY(X);
END MY_PROG;
```

C function

```
#pragma linkage (centry,PLI)
#include <stdio.h>
long double centry(long double *x)
{
  printf("*x is %Lf\n",x);
  return(*x);
}
```

Pointer to an integer

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
    DCL CENTRY EXTERNAL ENTRY;
    DCL I FIXED BIN(31);
    DCL P POINTER;
    P = ADDR(I);
    I = 5;
    CALL CENTRY (P);
END MY_PROG;
```

C function

```
#pragma linkage (centry,PLI)
#include <stdio.h>
void centry(int **x)
{
  printf("Value is %d\n",**x);
}
```

Pointer to an array

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
    DCL CENTRY EXTERNAL ENTRY;
    DCL I(5) FIXED BIN(31,0);
    DCL J FIXED BIN(31);
    DCL P POINTER;
    P = ADDR(I);
    DO J = 1 TO 5;
      I(J) = J;
    END;
    CALL CENTRY (P);
END MY_PROG;
```

C function

```
#pragma linkage (centry,PLI)
#include <stdio.h>
void centry(int **x)
{
  printf("Value is %d\n",**x);
  }
```

Pointer to a structure

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
    DCL CENTRY EXTERNAL ENTRY;
    DCL 1 TODAY,
        2 DAY FIXED BIN(31),
        2 MONTH FIXED BIN(31),
        2 YEAR FIXED BIN(31);
    DCL P POINTER;
    P = ADDR(TODAY);
    CALL CENTRY (P);
END MY_PROG;
```

C function

```
#pragma linkage (centry,PLI)
#include <stdio.h>
struct date   {
  int day;
  int month;
  int year;  };
void centry(struct date **x)
{
  printf("Day is %d\n",(*x)->day);
}
```

Fixed-length decimal data

Sample PL/I usage

```
PLIPROG: PROC OPTIONS(MAIN, REENTRANT);
    DCL CFUNC EXTERNAL ENTRY
      (FIXED DEC(5,0));
    DCL ARG FIXED DEC(5,0);
    DCL A FIXED DEC(5);
    ARG = 10;
    A = CFUNC(ARG);
END;
```

C function

```
#include <decimal.h>
#pragma linkage (CFUNC,PLI)

void CFUNC(decimal(5,0));

void CFUNC( decimal(5,0) parm ) {

  if (parm==10d)
    printf("Value is good\n");
  prinf("The parm is %D(5,0)\n",parm);
}
```

Name scope of external data

In programming languages, the *name scope* is defined as the portion of an application within which a particular declaration applies or is known. The name scope of static external data for PL/I and static variables defined outside of any function for C is the load module. If your application contains PL/I procedures and non-reentrant C routines, PL/I's external data maps to C's external data only within a load module. After you cross a load module boundary, external data does not map. In addition, the external data does not map if any C function in the application is compiled with the XL C RENT compile-time option.

Figure 28 on page 140 illustrates the name scope of external variables in a PL/I to C enclave, if the C function is non-reentrant. The `routine` can be a PL/I procedure or C routine. If Routine 3 is a PL/I procedure, however, it cannot have any variables with the EXTERNAL attribute; therefore, the name scope of Routine 3 in the figure refers only to C routines.

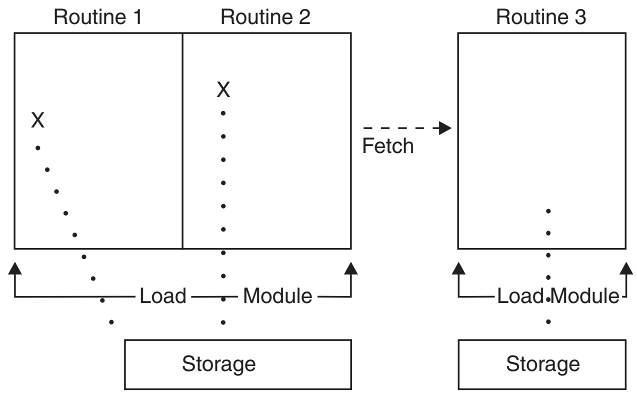


Figure 28. Name scope of external variables for PL/I or C fetch

In Figure 28, external data declared in Routine 1 maps to that declared in Routine 2 in the same load module. If the fetch is made to a C Routine 3 in another load module is made, the external data does not map, because the name scope of external data in C is the load module. If the fetch is made to a PL/I Routine 3, the routine is not allowed to have any variables declared with the EXTERNAL attribute.

When the name scopes of PL/I and C are the same, do not give external data the same name in a PL/I to C application if you cross a load module boundary.

DLL considerations

In DLL code, external variables are mapped across the load module boundary. DLLs are shared at the enclave level. Therefore, a single copy of a DLL applies to all modules in an enclave, regardless of whether the DLL is loaded implicitly (through a reference to a function or variable) or explicitly (through `dllload()`). See *z/OS Language Environment Programming Guide* for information about building and managing DLL code in your applications.

Name space of external data

In programming languages, the *name space* is defined as the portion of a load module within which a particular declaration applies or is known. Within the same load module, the name space of external data under both PL/I and C is the same. Therefore, PL/I's and C's external data map to each other, provided that the C routine is non-reentrant or naturally reentrant.

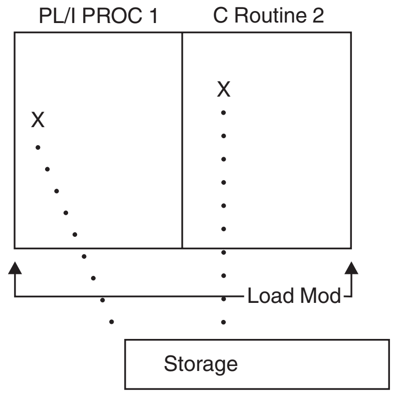


Figure 29. Name space of external data in PL/I static call to C

Figure 29 on page 140 illustrates that within the same load module, the name spaces of PL/I and C routines are the same. Therefore you can give external data the same name in a PL/I to C application, if no load module boundary is crossed.

Using storage functions in C to PL/I ILC

Use the following guidelines when you mix HLL storage constructs and Language Environment storage services:

Storage allocated using the PL/I ALLOCATE statement that:
- Is within a PL/I AREA, or
- Is of the storage class CONTROLLED, or
- Has the REFER option

must be released by the PL/I FREE statement. Storage with these characteristics cannot be released by the Language Environment callable service CEEFRST or by an HLL construct such as the C `free()` function.

Storage allocated as a result of the PL/I ALLOCATE statement that is of the storage class BASED can be released by CEEFRST or an HLL construct such as the C `free()` function if the structure:
- Is completely declared,
- Requires no pad bytes to be added automatically by the compiler, and
- Does not contain the REFER option

Directing output in ILC applications

Under Language Environment, PL/I runtime output such as runtime messages and ON condition SNAP output is directed to the destination specified in the Language Environment runtime option MSGFILE. The PL/I user-specified output, such as the output of the PUT SKIP LIST statement, remains directed to the PL/I STREAM PRINT file SYSPRINT. You can still have runtime and user-specified output directed to the same destination, as under OS PL/I, by specifying MSGFILE(SYSPRINT). Seethe IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735). for details about using the MSGFILE(SYSPRINT) option.

Under Language Environment, C runtime output such as runtime messages is directed to the destination specified in the Language Environment runtime option MSGFILE. `stderr` output is also directed to the destination of the MSGFILE option. Normally, `stdout` output is not directed to the destination of the MSGFILE option. You can redirect `stdout` output to the destination of the MSGFILE option by passing arguments `1>&2` to a C main routine, where `1>&2` associates `stdout` with `stderr`, or by placing `stdout=stderr` in your program. For information about redirecting C output, see *z/OS XL C/C++ Programming Guide*.

Using SYSPRINT as the default stream output file

SYSPRINT serves as the default stream output file for PL/I and C. In the absence of one language, the other uses SYSPRINT without any problem. If both languages are used in an application, C yields to PL/I's use of SYSPRINT by redirecting the C stream output to other destinations. However, in the following two cases, the redirection of C's stream output is not possible and the results can be unpredictable:

1. The main load module does not have a PL/I PUT statement and C uses SYSPRINT for its stream output in the main load module. Later, a fetched subroutine load module contains a PL/I PUT statement that PL/I starts to use SYSPRINT for its stream output.
2. When Language Environment preinitialization services are used for a subroutine environment, C uses SYSPRINT for its stream output in the first CEEPIPI(*call_sub,...*) then PL/I users SYSPRINT for its PUT statement in the second CEEPIPI(*call_sub*).

Directing user-specified output to destination of MSGFILE

You can direct PL/I and C user-specified output to the same destination as the runtime output by specifying MSGFILE(SYSPRINT). In this case, PL/I and C manage their own I/O buffers, line counters, and so on, for their own user-specified output. Therefore, MSGFILE(SYSPRINT) must be used carefully because PL/I output from the PUT statement, C output from `printf`, and runtime output can be interspersed with one another.

C POSIX multithreading

POSIX-conforming C applications can communicate with PL/I routines in any thread created by C routines. POSIX-conforming C applications can communicate with assembler routines on any thread when the assembler routines use the CEEENTRY/CEETERM macros provided by Language Environment or the EDCPRLG/EDCEPIL macros provided by C/370. If a `fork()` command is issued, the target of the `fork()` must be another C routine.

C to PL/I condition handling

This section offers two scenarios of condition handling behavior in a C to PL/I ILC application. If an exception occurs in a C routine, the set of possible actions is as described in “Exception occurs in C” on page 143. If an exception occurs in a PL/I routine, the set of possible actions is as described in “Exception occurs in PL/I” on page 145.

Keep in mind that if there is a PL/I routine currently active on the stack, PL/I language semantics can be applied to handle conditions that occur in non-PL/I routines within an ILC application. For example, PL/I semantics apply to Language Environment hardware conditions that map directly to PL/I conditions such as ZERODIVIDE, even if they occur in a non-PL/I routine. Also, PL/I treats any unknown condition of severity 2 or greater as the ERROR condition. In a case in which a C-specific condition of severity 2 or greater is passed to a PL/I stack frame, an ERROR ON-unit can handle it on the first pass of the stack.

See *z/OS Language Environment Programming Guide* for a detailed description of Language Environment condition handling.

Enclave-terminating language constructs

Enclaves might be terminated due to reasons other than an unhandled condition of severity 2 or greater. The language constructs that cause a single language application to be terminated also cause a C to PL/I application to be terminated.

C

Typical C language constructs that cause the application to terminate are:

- The abort(), raise(SIGTERM), raise(SIGABRT), kill(), pthread_kill(), and exit() function calls.

  If you call abort(), raise(SIGABRT), or exit(), the T_I_S (Termination Imminent Due to STOP) condition is raised. After T_I_S has been processed and all user code has been removed from the stack, the C atexit list is honored.

PL/I

The PL/I language constructs that cause the application to terminate are:

- A STOP statement, or an EXIT statement

  If you code a STOP or EXIT statement, the T_I_S (Termination Imminent Due to STOP) condition is raised. After T_I_S has been processed and after all user code has been removed from the stack, the C atexit list is honored.
- A call to PLIDUMP with the S or E option

  If you call PLIDUMP with the S or E option, neither termination imminent condition is raised, and the C atexit list is not honored before the enclave is terminated. See *z/OS Language Environment Debugging Guide* for syntax of the PLIDUMP service.

Exception occurs in C

This scenario describes the behavior of an application that contains a C and a PL/I routine. Refer to Figure 30 throughout the following discussion. In this scenario, a PL/I main routine invokes a C subroutine. An exception occurs in the C subroutine.

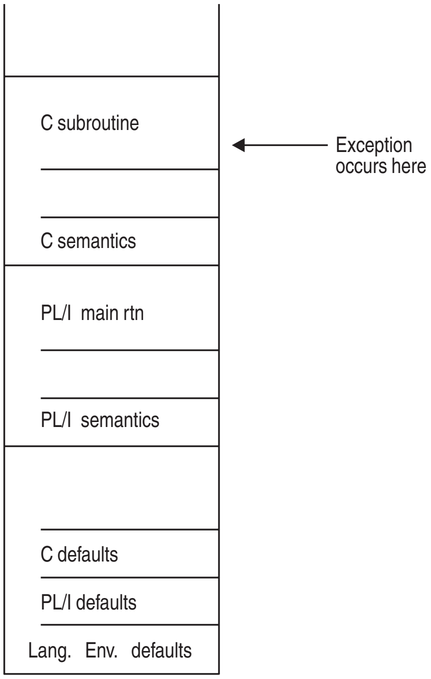


Figure 30. Stack contents when the exception occurs in C

The actions taken are the following:

1. In the enablement step, it is determined whether the exception in the C routine should be enabled and treated as a condition. If any of the following are true, the exception is ignored, and processing continues at the next sequential instruction after where the exception occurred:
   - You specified SIG_IGN for the exception in a call to signal().

Note: The system or user abend corresponding to the `signal(SIGABND)` or the Language Environment message 3250 is not ignored. The enclave is terminated.

- The exception is one of those listed as masked in Table 63 on page 249, and you have not enabled it using the CEE3SPM callable service.
- You did not specify any action, but the default action for the condition is `SIG_IGN` (see Table 63 on page 249).
- You are running under CICS and a CICS handler is pending.

If you did none of these things, the condition is enabled and processed as a condition.

2. If a user-written condition handler has been registered on the stack frame using CEEHDLR, it is given control.

   If it issues a resume, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
3. If a C signal handler has been registered for the condition on the C stack frame, it is given control. If it successfully issues a resume or a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points.

   In this case, there is not a C signal handler registered for the condition.
4. The condition is still unhandled. If C does not recognize the condition, or if the C default action (listed in Table 63 on page 249) is to terminate, the condition is percolated.
5. If a user-written condition handler has been registered on the PL/I stack frame using CEEHDLR, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine at the point where the resume cursor points. In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
6. If an ON-unit has been established for the condition being processed on the PL/I stack frame, it is given control. If it issues a GOTO out-of-block, the condition handling step ends. Execution resumes at the label of the GOTO. In this example, no ON-unit is established for the condition, so the condition is percolated.
7. What happens next depends on whether the condition is promotable to the PL/I ERROR condition. The following can happen:
   - If the condition is not promotable to the PL/I ERROR condition, then the Language Environment default actions take place, as described in Table 62 on page 249. Condition handling ends.
   - If the PL/I default action for the condition is to promote it to the PL/I ERROR condition, the condition is promoted, and another pass is made of the stack to look for ERROR ON-units or user-written condition handlers. If an ERROR ON-unit or user-written condition handler is found, it is invoked.
   - If either of the following occurs:
     - An ERROR ON-unit or user-written condition handler is found, but it does not issue a GOTO out of block or similar construct
     - No ERROR ON-unit or user-written condition handler is found

     then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled condition). Condition handling now enters the termination imminent step. Because T_I_U maps to the PL/I FINISH condition, a FINISH ON-unit or user-written condition handler is run if the stack frame in which it is established is reached.

- If no condition handler moves the resume cursor and issued a resume, Language Environment terminates the thread.

Exception occurs in PL/I

This scenario describes the behavior of an application that contains a PL/I and a C routine. Refer to Figure 31 throughout the following discussion. In this example, a C main routine invokes a PL/I subroutine. An exception occurs in the PL/I subroutine.

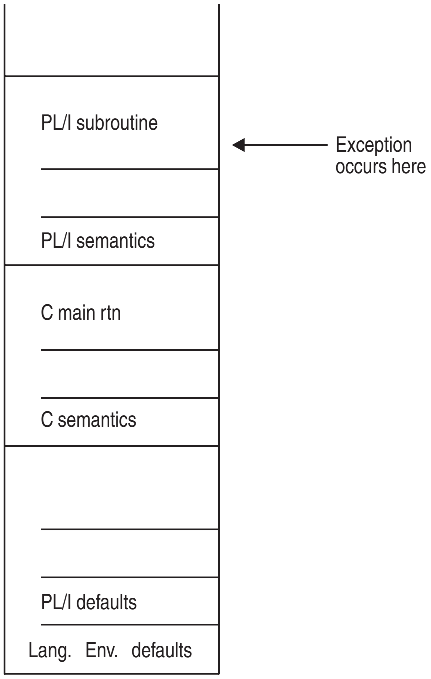


Figure 31. Stack contents when the PL/I exception occurs

The actions taken are the following:

1. In the enablement step, PL/I determines if the exception that occurred should be handled as a condition according to the PL/I rules of enablement.
   - If the exception is to be ignored, control is returned to the next sequential instruction after where the exception occurred.
   - If the exception is to be enabled and processed as a condition, the condition handling step takes place.
2. If a user-written condition handler has been registered on the PL/I stack frame using CEEHDLR, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine at the point where the resume cursor points. In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
3. If an ON-unit has been established for the condition being processed on the PL/I stack frame, it is given control. If it issues a GOTO out-of-block, the condition handling step ends. Execution resumes at the label of the GOTO. In this example, no ON-unit is established for the condition, so the condition is percolated.

4. If a user-written condition handler has been registered using CEEHDLR on the C stack frame, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   Note: There are special considerations for resuming from some IBM conditions of severity 2 or greater. See the chapter on coding a user-written condition handler in *z/OS Language Environment Programming Guide* for more information.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
5. If a C signal handler has been registered for the condition, it is given control. If it successfully issues a resume or a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points.

   In this example no C signal handler is registered for the condition, so the condition is percolated.
6. What happens next depends on whether the condition is promotable to the PL/I ERROR condition. The following can happen:
   - If the condition is not promotable to the PL/I ERROR condition, then the Language Environment default actions take place, as described in Table 62 on page 249. Condition handling ends.
   - If the PL/I default action for the condition is to promote it to the PL/I ERROR condition, the condition is promoted, and another pass of the stack is made to look for ERROR ON-units or user-written condition handlers. If an ERROR ON-unit or user-written condition handler is found, it is invoked.
   - If either of the following occurs:
     – An ERROR ON-unit or user-written condition handler is found, but it does not issue a GOTO out of block or similar construct
     – No ERROR ON-unit or user-written condition handler is found

     then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled condition). Condition handling now enters the termination imminent step. Because T_I_U maps to the PL/I FINISH condition, a FINISH ON-unit is run if the stack frame in which it is established is reached.
   - If no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Fixed-point overflow

The XL C compiler assumes that fixed-point overflow exceptions will be handled by S/390® hardware, while the PL/I compiler assumes the runtime library will process these exceptions. Therefore, if a C to PL/I ILC application incurs a fixed-point overflow, Language Environment will try to handle this exception in the runtime library. This will result in increased CPU utilization compared to an application using only C.

Sample C to PL/I ILC applications

PL/I main routine calling a C subroutine:

```
 *PROCESS LC(101),OPT(0),S,MAP,LIST,STMT,A(F),AG;
   CEPLI2C: PROC OPTIONS(MAIN);
   /*Module/File Name: IBMCPL        */
   /************************************************************/
   /* FUNCTION   :  Interlanguage communications call to       *
   /*               a C program.                               *
   /* This example illustrates an interlanguage call from      *
```

```
/* a PL/I main program to a C subroutine.                  *
/* The parameters passed across the call from PL/I to      *
/* C have the following declarations:                      *
/*                                                         *
/* PL/I fixed bin(15,0) to C short as pointer to BIN       *
/* PL/I fixed bin(31,0) to C int                           *
/* PL/I float bin(53) to C double                          *
/* PL/I float bin(109) to C long double                    *
/* PL/I characters to C as pointer to pointer to CHAR      *
/***********************************************************/
/***********************************************************/
/* DECLARES FOR THE CALL TO C                              *
/***********************************************************/
   DCL ADDR               BUILTIN;
   DCL J                  FIXED BIN(31,0);
   DCL CECFPLI            EXTERNAL ENTRY RETURNS(FIXED BIN(31,0));
   DCL PL1_SHORT          FIXED BIN(15,0) INIT(15);
   DCL PL1_INT            FIXED BIN(31,0) INIT(31);
   DCL PL1_DOUBLE         FLOAT BIN(53) INIT (53.99999);
   DCL PL1_LONG_DOUBLE    FLOAT BIN(109) INIT(3.14151617);
   DCL PL1_POINTER        PTR;
   DCL CHARSTRING         CHAR(23) INIT('PASSED CHARACTER STRING');

/************************************************************/
/*  PROCESS STARTS HERE                                      *
/************************************************************/
   PUT SKIP LIST ('**********************************');
   PUT SKIP LIST ('PL/I CALLING C/370 EXAMPLE STARTED');
   PUT SKIP LIST ('**********************************');
   PL1_POINTER = ADDR(CHARSTRING);
   PUT SKIP LIST ('Calling C/370 subroutine');
   J = CECFPLI( ADDR(PL1_SHORT), PL1_INT, PL1_DOUBLE,
      PL1_LONG_DOUBLE, ADDR(PL1_POINTER));
   PUT SKIP LIST ('Returned from C/370 subroutine');
   IF (J ¬= 999) THEN
      PUT SKIP LIST ('Error in return code from C/370');
   PUT SKIP LIST ('**********************************');
   PUT SKIP LIST ('PL/I CALLING C/370 EXAMPLE ENDED  ');
   PUT SKIP LIST ('**********************************');
END CEPLI2C;
```

C routine called by PL/I main routine

```
/*Module/File Name:  EDCCPL   */
#pragma linkage (CECFPLI,PLI)
#include <stdio.h>
#include <string.h>
 /******************************************************************
  *This is an example of a C program invoked by a PL/I program.    *
  *CECFPLI is called from PL/I program CEPLI2C with the following  *
  *list of arguments:                                              *
  * PL/I fixed bin(15,0) to C short as pointer to BIN              *
  * PL/I fixed bin(31,0) to C int                                  *
  * PL/I float bin(53) to C double                                 *
  * PL/I float bin(109) to C long double                           *
  * PL/I characters to C as pointer to pointer to CHAR             *
  ******************************************************************/
 int CECFPLI (short **c_short,
              int *c_int,
              double *c_double,
              long double *c_long_double,
              char *** c_character_string
               )
 {
    int ret=999; /* pli is expecting 999 returned */
    fprintf(stderr,"CECFPLI STARTED\n");
 /******************************************************************
 * Compare each passed argument against the C value.              *
 * Issue an error message for any incorrectly passed parameter.   *
 ******************************************************************/
   if (**c_short != 15)
   {
     fprintf(stderr,"**c_short not = 15\n");
     --ret;
   }
   if (*c_int != 31)
   {
     fprintf(stderr,"*c_int not = 31\n");
     --ret;
   }
   if ((53.99999 - *c_double) >1.0E-14)
   {
     fprintf(stderr,
          "53.99999 - *c_double not >1.0E-14\n");
     --ret;
   }
   if ((3.14151617 - *c_long_double) >1.0E-16)
   {
     fprintf(stderr,
          "3.14151617 - *c_long_double not >1.0E-16\n");
     --ret;
   }
   if (memcmp(**c_character_string,"PASSED CHARACTER STRING",23)
             != 0)
   {
     fprintf(stderr,"**c_character_string not %s\n",
     "\"PASSED CHARACTER STRING\"");
     --ret;
   }
/******************************************************************
 * PL/I will check for a correct return code.                     *
 ******************************************************************/
   fprintf(stderr,"CECFPLI ENDED\n");
   return(ret);
}
```

Chapter 9. Communicating between C++ and PL/I

This topic describes Language Environment's support for C++ and PL/I ILC applications. If you are running a C++ to PL/I ILC application under CICS, you should also consult Chapter 15, “ILC under CICS,” on page 241.

General facts about C++ to PL/I ILC

- ILC between C++ and Enterprise PL/I for z/OS is discussed in the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).
- Language Environment does not support passing return codes between PL/I and C++ routines in an ILC application.
- PL/I Multitasking Facility (MTF) does not support C++.
- A C++ NULL is X'00000000'; a PL/I NULL is X'FF000000'; a PL/I SYSNULL is X'00000000'. Comparisons against a NULL value and other uses of the NULL value must therefore be done with care.
- There is no ILC support between AMODE 31 and AMODE 64 applications. PL/I does not support AMODE 64.

Preparing for ILC

This section describes topics you might want to consider before writing an application that uses ILC. For help in determining how different versions of HLLs work together, refer to the migration guides for the HLLs you plan to use.

Language Environment ILC support

Language Environment supports ILC between the following combinations of C++ and PL/I:

Table 40. Supported languages for Language Environment ILC

HLL pair	C++	PL/I
C++–PL/I	• IBM C++ for MVS/ESA • z/OS XL C/C++ compilers	• PL/I for MVS & VM • Enterprise PL/I for z/OS

Determining the main routine

In Language Environment, only one routine can be the main routine. If a PL/I routine is identified as a main routine in an ILC application by PROC OPTIONS(MAIN), and a C++ main function does not exist, the PL/I main routine is the first to gain control. If a C++ main function exists, but no PL/I main routine is identified in the ILC application, the C++ main function gains control first.

If both a PL/I main routine identified by PROC OPTIONS(MAIN) and a C++ main function exist in the same ILC application, this is a user error. However, the error is not detected by Language Environment.

An entry point is defined for each supported HLL. Table 41 on page 150 identifies the desired entry point. The table assumes that your code has been compiled using the Language Environment-conforming compilers.

Table 41. Determining the entry point

HLL	Main entry point	Fetched entry point
C++	CEESTART	Not supported
PL/I	CEESTART	CEESTART or routine name, if OPTIONS(FETCHABLE) is used.

Declaring C++ to PL/I ILC

Declaring a C++ entry point in a PL/I routine has the same syntax as declaring another PL/I entry point. A C++ routine can be replaced by a PL/I routine without altering the PL/I code that calls the routine. Likewise, if a C++ routine calls a PL/I routine, the PL/I procedure contains no explicit declaration indicating control is being passed from the C++ routine. The declaration is contained within the C++ routine.

In C++, you must declare that the C++ entry point receives control from a PL/I routine. This declaration is in the form of an `extern "PLI"` linkage specification. The body of the C++ function is the same as if the routine were called from another C++ function. Calling a PL/I routine and being called by a PL/I routine are handled by the same `extern "PLI"` linkage specification.

Declaration for C++ calling PL/I

C++ function

```
extern "PLI" {
double PLIFUNC( double );
    / C++ prototype /
}

int main() {
  double val,result;

  val=7.1;
  result=PLIFUNC(val);
  printf("val=%f, result=%f\n",
    val,result);
}
```

PL/I Routine

```
PLIFUNC: Proc(arg) options(reentrant)
    returns(float binary(53));
  Dcl arg float binary(53);
  Return (34.0);
  End;
```

Declaration for PL/I calling C++

PL/I function

```
PLIPROG: Proc options(main, reentrant);
  Dcl CXXFUNC external entry
    returns(fixed bin(31));
  Dcl arg fixed bin(31);
  Dcl a fixed bin(31);
  Arg = 10;
  A = CXXFUNC(arg);
  End;
```

C++ Routine

```
extern "PLI" {
  int CXXFNC( int parm ) {
}

int CXXFNC( int parm ) {

  return(5);
}
```

Building a reentrant C++ to PL/I application

The XL C++ compiler creates reentrant code by default. To create a reentrant C++ to PL/I application, follow this process:

1. Compile the C++ routines.
2. (Optional) Compile all PL/I routines in the ILC application using the REENTRANT option of the OPTIONS attribute of the PROCEDURE statement.

3. Run the generated C++ and PL/I object code through the Language Environment prelinker to generate a single text deck.
4. Run the text deck provided by the prelinker through the linkage editor to produce a load module.

Note: A reentrant C++ to PL/I application has different semantics than a non-reentrant one.

Calling between C++ and PL/I

Table 42 describes the types of calls between C++ and PL/I that Language Environment allows:

Table 42. Calls permitted for C++ and PL/I ILC

ILC direction	Static calls	Dynamic calls using DLLs	Fetch/Calls
C++ to PL/I	Yes	Yes (1)	C++ does not support `fetch()` for PL/I
PL/I to C++	Yes	Yes (1)	No

Note: Enterprise PL/I for z/OS supports calls to a C++ DLL, and also allows calls from C++ to a PL/I DLL. In this case, the C++ code could also optionally be compiled with the XPLINK option.

Passing data between C++ and PL/I

There are two sets of data types that you can pass between C++ and PL/I routines: data types passed *by reference* using C++ explicit pointers explicitly in the routine, and data types passed *by value* without using C++ explicit pointers.

By reference means the parameter itself is passed. A copy of the parameter is not made. Any changes to the parameter made by the called routine can alter the original parameter passed by the calling routine. By value means a copy of the parameter is passed. Any changes to the parameter made by the called routine cannot alter the original parameter passed by the calling routine.

Passing pointers from C++ to PL/I

Pointers can be passed and returned between C++ and PL/I routines. Because the C++ `extern "PLI"` linkage specifies that pointers, unlike other parameters, are passed directly, there is one level of indirection less on the PL/I side.

In order for PL/I to receive a pointer to a PL/I data type, C++ must pass a pointer to a pointer to the C++ data type. Conversely, if PL/I returns a pointer to a data type, C++ receives a pointer to a pointer to the data type.

Structures, arrays, and strings should be passed between C++ and PL/I only by using pointers.

The non-address bits in all fullword pointers declared in PL/I source code should always be zero. If they are not, results are unpredictable.

Passing pointers from PL/I to C++

Pointers to various data objects can be passed from PL/I and accepted by a function written in C++.

An extra level of indirection is added when passing a pointer value from PL/I to C++, because the C++ extern "PLI" linkage specification passes pointers directly. If PL/I passes or returns a pointer to a type, C++ receives a pointer to a pointer to the type.

PL/I parameters that require a locator or descriptor should not be passed directly. This includes parameters that are structures, arrays, or strings. These parameters can be passed indirectly from PL/I by using a pointer to the associated data. For more information about data descriptors, see the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).

The non-address bits in all fullword pointers declared in PL/I source code should always be zero. If they are not, results are unpredictable.

Receiving value parameters in C++

If you enclose in parentheses the argument you pass from a PL/I routine to a C++ routine, the argument is passed by value. C++ should receive the parameter as the equivalent C++ type. The XL C++ compiler generates the appropriate addressing code required to access the parameter values.

You can write your PL/I-callable function as if it were in a C++-only environment; you can move it to a C++-only environment simply by removing the extern "PLI" linkage specification.

Receiving reference parameters in C++

If a parameter is not enclosed in parentheses a PL/I routine will pass it by reference to a C++ routine. C++ should then receive the parameter as a pointer to the equivalent C++ type.

For example, if a C++ function named FROMPLI is to receive a parameter having the type int, the function prototype declaration looks like this:

```
int FROMPLI(int *i);
or
int FROMPLI(int &i);
```

A parameter passed from PL/I by reference may be received and used by C++ as a value parameter provided that its value is not altered. If the value of such a parameter is altered, the effect on the original PL/I variable is undefined.

Supported data types passed using C++ pointers (by reference)

Table 43 identifies the data types that can be passed as parameters between C++ and PL/I applications with the use of explicit pointers under C++. Parameters that are pointers are passed by reference to PL/I. Conversely, reference parameters passed by PL/I to C++ are received as pointers to the equivalent data type or as a C++ reference variable such as int& i.

Anything that is passed with a pointer can be passed with a reference variable; the effect is the same. Reference variables are generally easier to use.

Table 43. Supported data types between C++ and PL/I using C++ pointers (by reference)

C++	PL/I
signed short int	REAL FIXED BINARY(15,0)
signed int	REAL FIXED BINARY(31,0)

Table 43. Supported data types between C++ and PL/I using C++ pointers (by reference) (continued)

C++	PL/I
signed long int	REAL FIXED BINARY(31,0)
float	FLOAT DECIMAL(06) FLOAT BINARY (21) is the preferred equivalent for float.
double	FLOAT DECIMAL(16) FLOAT BINARY (53) is the preferred equivalent for double.
long double	FLOAT DECIMAL(33) FLOAT BINARY (109) is the preferred equivalent for long double.
pointer to . . .	POINTER
Note: Data storage alignment must match.	

Supported data types passed by value

Table 44 identifies the data types that can be passed by value (without using a C++ pointer) between C++ and PL/I applications.

In order for a C++ routine to pass a parameter without using a pointer, the argument should be passed, and the PL/I routine should receive the parameter as the equivalent PL/I data type.

Table 44. Supported data types between C++ and PL/I by value

C++	PL/I
signed int	REAL FIXED BINARY(31,0)
signed long int	REAL FIXED BINARY(31,0)
double	FLOAT DECIMAL(16)
double	FLOAT BINARY(53)
long double	FLOAT DECIMAL(33)
long double	FLOAT BINARY(109)
Note: The preferred PL/I data declarations for the C double and long double data types are FLOAT BINARY(53) and FLOAT BINARY(109), respectively. Data storage alignment must match.	

Passing strings between C++ and PL/I

C++ and PL/I have different string data types:

C++ strings
: Logically unbounded length and are terminated by a NULL (the last byte of the string contains X'00').

PL/I CHAR(n) VARYING
: A halfword-prefixed string of characters with a maximum length *n* characters. The current length is held in the halfword prefix.

PL/I CHAR(n)
: A fixed-length string of characters of length *n*. There is no halfword prefix indicating the length.

You can pass strings between C++ and PL/I routines, but you must match what the routine interface demands with what is physically passed.

Using aggregates

Aggregates (arrays, strings, or structures) are mapped differently by C++ and PL/I and are not automatically mapped. Be sure to completely declare every byte in the aggregate so there are no open fields. Doing so helps ensure that the layouts of aggregates passed between the two languages map to one another correctly. The C++ and PL/I AGGREGATE compiler time options provide a layout of aggregates to help you perform the mapping.

In C++, a structure is a class declared with the `struct` keyword; its members and base classes are public by default. A C++ class is the same as a C++ structure if the only data is public. If a C++ class that uses features unavailable to PL/I (such as virtual functions, virtual base classes, private data, protected data, static data members, or inheritance) is passed to PL/I, the results are undefined.

For more information about PL/I structure mapping, see the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).

Data equivalents

This section describes how C++ and PL/I data types correspond to each other.

Equivalent data types for C++ to PL/I

The following examples illustrate how C++ and PL/I routines within a single ILC application might code the same data types.

16-bit signed binary integer

Sample C++ usage

```
#include <stdio.h>

extern "PLI" {
  short int cpli( short int * );
}

main() {
  short int x, y;
  x=5;
  y = cpli(&x);  /* by reference */
}
```

PL/I subroutine

```
CPLI: PROC(ARG) RETURNS (FIXED BIN(15));
    DCL ARG FIXED BIN (15);
⋮
    RETURN (ARG);
    END;
```

Note: Because `short int` is an example of a parameter which must be passed using an C++ explicit pointer, you cannot code `y = cpli(x)`, passing `x` by value.

32-bit signed binary integer

Sample C++ usage

```
#include <stdio.h>

extern "PLI" {
  int cpli( int );
}

main() {
  int x, y;
  x=5;
  y = cpli(x);  /* by value */
}
```

PL/I subroutine

```
CPLI: PROC(ARG) RETURNS (FIXED BIN(31));
    DCL ARG FIXED BIN (31);
⋮
    RETURN (ARG);
    END;
```

Sample C++ usage

```
#include <stdio.h>

extern "PLI" {
  int cpli( int *);
}

main() {
  int x, y;
  x=5;
  y = cpli(&x);  /* by reference */
}
```

PL/I subroutine

```
CPLI: PROC(ARG) RETURNS (FIXED BIN(31));
    DCL ARG FIXED BIN (31);
⋮
    RETURN (ARG);
    END;
```

Long floating-point number

Sample C++ usage

```
#include <stdio.h>

extern "PLI" {
  double cpli(double);
}

main()
{
  double x, y;
  x=12.5;
  cpli(x);  /* by value */
}
```

PL/I subroutine

```
CPLI: PROC(ARG)
       RETURNS (FLOAT BINARY(53));
    DCL ARG FLOAT BINARY(53);
⋮
    RETURN (ARG);
    END;
```

Sample C++ usage

```
#include <stdio.h>

extern "PLI" {
  void double cpli(double *);
}

main()
{
  double x;
  x=12.5;
  cpli(&x);  /* by reference */

}
```

PL/I subroutine

```
CPLI: PROC(ARG);
    DCL ARG FLOAT BINARY(53);
⋮
    END;
```

Extended floating-point number

Sample C++ usage

```
#include <stdio.h>

extern "PLI" {
  long double cpli(long double);
}

main()
{
  long double x, y;
  x=12.1
  y=cpli(x); /* by value    */
}
```

PL/I subroutine

```
CPLI: PROC(ARG) RETURNS (FLOAT BIN(109));
    DCL ARG FLOAT BIN(109);
⋮
    RETURN (ARG);
    END;
```

Sample C++ usage

```
#include <stdio.h>

extern "PLI" {
  long double cpli(long double*);
}

main()
{
  long double x, y;
  x=12.1
  y=cpli(&x);     /* by reference */
}
```

PL/I subroutine

```
CPLI: PROC(ARG)
    RETURNS (FLOAT BIN(109));
    DCL ARG FLOAT BIN(109);
⋮
    RETURN (ARG);
    END;
```

Pointer to an integer

Sample C++ usage

```
#include <stdio.h>

extern "PLI" {
  void cpli (int **);
}

main()
{
  int i, *temp;
  i = 5;
  temp=&i;
  cpli(&temp);
}
```

PL/I subroutine

```
CPLI: PROC (ARG);
    DCL ARG POINTER;
    DCL ART FIXED BIN(31,0)
      BASED (ARG);
⋮
    END;
```

Pointer to an array

Sample C++ usage

```
#include <stdio.h>

extern "PLI" {
  int (*temp[] = &matrix);
}

main()
{
  int matrix[5]
  int i;
  void cpli(int(**)[]);
  for(i=0;i<5;i++);
    matrix[i] = i;
  cpli(&temp);
}
```

PL/I subroutine

```
CPLI: PROC (ARG);
    DCL ARG POINTER;
    DCL I FIXED BIN(31)
    DCL ART(5) FIXED BIN(31,0)
    BASED (ARG);
⋮
    END;
```

Pointer to a structure

Sample C++ usage

```
#include <stdio.h>

struct date   {
  int day;
  int month;
  int year   } today;

extern "PLI" {
  void cpli(struct date **);
}

main()
{
  struct date *temp = &today;
  int i;
  cpli (&temp);
}
```

PL/I subroutine

```
CPLI: PROC (ARG);
    DCL ARG POINTER;
    DCL 1 TODAY BASED (ARG),
        2 DAY FIXED BIN(31),
        2 MONTH FIXED BIN(31),
        2 YEAR FIXED BIN(31);
⋮
    END;
```

Equivalent data types for PL/I to C++

This section contains examples that illustrate how C++ and PL/I routines within a single ILC application might code the same data types.

32-bit signed binary integer

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
    DCL CENTRY EXTERNAL ENTRY
      RETURNS (FIXED BIN(31));
    DCL X FIXED BIN(31);
    DCL Y FIXED BIN(31);
    X = 5;
    /* BY VALUE */
    Y=CENTRY((X));
END MY_PROG;
```

C++ function

```
#include <stdio.h>

extern "PLI" {
  int centry(int x)
}

int centry(int x)
{
  printf("x is %d/n",x);
  return(x);
}
```

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
     DCL CENTRY EXTERNAL ENTRY
       RETURNS (FIXED BIN(31));
     DCL X FIXED BIN(31);
     DCL Y FIXED BIN(31);
     X = 5;
     /* BY REFERENCE */
     Y=CENTRY(X);
END MY_PROG;
```

C++ function

```
#include <stdio.h>

extern "PLI" {
  int centry(int *x)
}

int centry(int *x)
{
  printf("x is %d\n",x);
  return(*x);
}
```

Long floating-point number

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
     DCL CENTRY EXTERNAL ENTRY
       RETURNS (FLOAT DEC(16));
     DCL X FLOAT DEC(16);
     DCL Y FLOAT DEC(16);
     X = 3.14159265;
     /* BY VALUE */
     Y=CENTRY((X));
END MY_PROG;
```

C++ function

```
#include <stdio.h>

extern "PLI" {
  double centry(double x)
}

double centry(double x)
{
  printf("x is %f\n",x);
  return(x);
}
```

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
     DCL CENTRY EXTERNAL ENTRY
       RETURNS (FLOAT DEC(16));
     DCL X FLOAT DEC(16);
     DCL Y FLOAT DEC(16);
     X = 3.14159265;
     /* BY REFERENCE */
     Y=CENTRY(X);
END MY_PROG;
```

C++ function

```
#include <stdio.h>

extern "PLI" {
double centry(double *x)
}

double centry(double *x)
{
printf("x is %f\n",x);
return(*x);
}
```

Extended floating-point number

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
     DCL CENTRY EXTERNAL ENTRY
       RETURNS (FLOAT DEC(33));
     DCL X FLOAT DEC(33);
     DCL Y FLOAT DEC(33);
     X = 12.5;
     /* BY VALUE */
     Y=CENTRY((X));
END MY_PROG;
```

C++ function

```
#include <stdio.h>

extern "PLI" {
  long double centry(long double x);
}

long double centry(long double x)
{
  printf("x is %Lf\n",x);
  return(x);
}
```

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
     DCL CENTRY EXTERNAL ENTRY
       RETURNS (FLOAT DEC(33));
     DCL X FLOAT DEC(33);
     DCL Y FLOAT DEC(33);
     X = 12.5;
     /* BY REFERENCE */
     Y=CENTRY(X);
END MY_PROG;
```

C++ function

```
#include <stdio.h>

extern "PLI" {
  long double centry(long double *x)
}

  printf("x is %Lf\n",x);
  return(*x);
}
```

Pointer to an integer

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
     DCL CENTRY EXTERNAL ENTRY;
     DCL I FIXED BIN(31);
     DCL P POINTER;
     P = ADDR(I);
     I = 5;
     CALL CENTRY (P);
END MY_PROG;
```

C++ function

```
#include <stdio.h>

extern "PLI" {
  void centry(int **x)
}

void centry(int **x)
{
  printf("Value is %d\n",x);
}
```

Pointer to an array

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
     DCL CENTRY EXTERNAL ENTRY;
     DCL I(5) FIXED BIN(31,0);
     DCL J FIXED BIN(31);
     DCL P POINTER;
     P = ADDR(I);
     DO J = 1 TO 5;
       I(J) = J;
     END;
     CALL CENTRY (P);
END MY_PROG;
```

C++ function

```
#include <stdio.h>

extern "PLI" {
  void centry(**x);
}

void centry(**x)
{
  /*   ...   */
}
```

Pointer to a structure

Sample PL/I usage

```
MY_PROG: PROC OPTIONS(MAIN);
     DCL CENTRY EXTERNAL ENTRY;
     DCL 1 TODAY,
         2 DAY FIXED BIN(31),
         2 MONTH FIXED BIN(31),
         2 YEAR FIXED BIN(31);
     DCL P POINTER;
     P = ADDR(TODAY);
     CALL CENTRY (P);
END MY_PROG;
```

C++ function

```
struct date   {
  int day;
  int month;
  int year;  };
extern "PLI" {
  void centry (struct date **x);
}

void centry(struct date **x)
{
  printf("Day is %d\n",(x)->day);
}
```

Name scope of external data

In programming languages, the *name scope* is defined as the portion of an application within which a particular declaration applies or is known. The name scope of static external data for PL/I and static variables defined outside of any function for C++ is the load module.

Because C++ is reentrant, PL/I and C++ external data do not map by default. However, you can map specific variable by using the `pragma variable` directive to specify that a C++ variable is NORENT.

DLL considerations

In DLL code, external variables are mapped across the load module boundary. DLLs are shared at the enclave level. Therefore, a single copy of a DLL applies to all modules in an enclave, regardless of whether the DLL is loaded implicitly (through a reference to a function or variable) or explicitly (through `dllload()`). See *z/OS XL C++ Programming Guide* for information about building and managing DLL code in your applications.

Name space of external data

In programming languages, the *name space* is defined as the portion of a load module within which a particular declaration applies or is known. With statically linked C++ to PL/I ILC, PL/I external data is not accessible from C++ and C++ external data is not accessible from PL/I unless the `#pragma variable` directive is used to specify that the specific variable is NORENT.

Using storage functions in C++ to PL/I ILC

Use the following guidelines when mixing HLL storage constructs and Language Environment storage services.

The C++ `new` and `delete` statements should not be mixed with the C `malloc()` and `free()` statements. Storage allocated with `new` can be deallocated only with `delete` and vice versa.

Storage allocated using the PL/I ALLOCATE statement that:
- Is within a PL/I AREA, or
- Is of the storage class CONTROLLED, or
- Has the REFER option

must be released by the PL/I FREE statement. Storage with these characteristics cannot be released by the Language Environment callable service CEEFRST or by an HLL construct such as the C++ `free()` function.

Storage allocated as a result of the PL/I ALLOCATE statement that is of the storage class BASED can be released by CEEFRST or an HLL construct such as the C++ `free()` function if the structure:
- Is completely declared,
- Requires no pad bytes to be added automatically by the compiler, and
- Does not contain the REFER option

Directing output in ILC applications

Under Language Environment, PL/I runtime output such as runtime messages and ON condition SNAP output is directed to the destination specified in the Language Environment runtime option MSGFILE. The PL/I user-specified output, such as the output of the PUT SKIP LIST statement remains directed to the PL/I STREAM PRINT file, SYSPRINT. You can still have the runtime output and the user-specified output directed to the same destination, like under OS PL/I, by specifying MSGFILE(SYSPRINT). See the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735) for details of using the MSGFILE(SYSPRINT) option.

Under Language Environment, C++ runtime output such as runtime messages is directed to the destination specified in the Language Environment runtime option MSGFILE. `stderr` output is also directed to the destination of the MSGFILE option. Normally, `stdout` output is not directed to the destination of the MSGFILE option. You can redirect `stdout` output to the destination of the MSGFILE option by passing arguments `1>&2` to a C++ main routine, where `1>&2` associates `stdout` with `stderr`, or by placing `stdout=stderr` in your program. For information about redirecting C++ output, see *z/OS XL C/C++ Programming Guide*.

Using SYSPRINT as the default stream output file

SYSPRINT serves as the default stream output file for PL/I and C++. In the absence of one language, the other uses SYSPRINT without any problem. If both languages are used in an application, C++ yields to PL/I's use of SYSPRINT by redirecting the C++ stream output to other destinations. However, in the following two cases, the redirection of C++'s stream output is not possible and the results can be unpredictable:

1. The main load module does not have a PL/I PUT statement and C++ uses SYSPRINT for its stream output in the main load module. Later, a fetched subroutine load module contains a PL/I PUT statement that PL/I starts to use SYSPRINT for its stream output.
2. When Language Environment preinitialization services are used for a subroutine environment, C++ uses SYSPRINT for its stream output in the first CEEPIPI(*call_sub,...*) then PL/I users SYSPRINT for its PUT statement in the second CEEPIPI(*call_sub*).

Directing user-specified output to destination of MSGFILE

You can direct PL/I and C++ user-specified output to the same destination as the runtime output by specifying MSGFILE(SYSPRINT). In this case, PL/I and C++ manage their own I/O buffers, line counters, etc., for their own user-specified output. Therefore, MSGFILE(SYSPRINT) must be used carefully because PL/I output from the PUT statement, C++ output from `printf`, and runtime output can be interspersed with one another.

C++ to PL/I condition handling

This section offers two scenarios of condition handling behavior in a C++ to PL/I ILC application. If an exception occurs in a C++ routine, the set of possible actions is as described in “Exception occurs in C++” on page 162. If an exception occurs in a PL/I routine, the set of possible actions is as described in “Exception occurs in PL/I” on page 164.

Keep in mind that if there is a PL/I routine currently active on the stack, PL/I language semantics can be applied to handle conditions that occur in non-PL/I routines within an ILC application. For example, PL/I semantics apply to Language Environment hardware conditions that map directly to PL/I conditions such as ZERODIVIDE, even if they occur in a non-PL/I routine. Also, PL/I treats any unknown condition of severity 2 or greater as the ERROR condition. In a case in which a C-specific condition of severity 2 or greater is passed to a PL/I stack frame, an ERROR ON-unit can handle it on the first pass of the stack.

C++ exception handling constructs `try/throw/catch` cannot be used with Language Environment and PL/I condition handling. If you use C exception handling constructs (`signal/raise`) in your C++ routine, condition handling will proceed as described in this section. Otherwise, you will get undefined behavior in your programs if you mix the C constructs with the C++ constructs.

For a detailed description of Language Environment condition handling, see *z/OS Language Environment Programming Guide*.

Enclave-terminating language constructs

Enclaves might be terminated due to reasons other than an unhandled condition of severity 2 or greater. The language constructs that cause a single language application to be terminated also cause a C++ to PL/I application to be terminated. Those language constructs of interest are listed in the following sections.

C language constructs available under C++

Among the C language constructs that cause an application to terminate are:

- The `abort()`, `raise(SIGABRT)`, and `exit()` function calls.

  If you call `abort()`, `raise(SIGABRT)`, or `exit()`, the T_I_S (Termination Imminent Due to STOP) condition is raised. After T_I_S has been processed and all user code has been removed from the stack, the C++ atexit list is honored.

PL/I

The PL/I language constructs that cause the application to terminate are:

- A STOP statement, or an EXIT statement

  If you code a STOP or EXIT statement, the T_I_S (Termination Imminent Due to STOP) condition is raised. After T_I_S has been processed and after all user code has been removed from the stack, the C atexit list is honored.
- A call to PLIDUMP with the S or E option

  If you call PLIDUMP with the S or E option, neither termination imminent condition is raised, and the C++ atexit list is not honored before the enclave is terminated. See *z/OS Language Environment Debugging Guide* for syntax of the PLIDUMP service.

Exception occurs in C++

This scenario describes the behavior of an application that contains a C++ and a PL/I routine. Refer to Figure 32 on page 163 throughout the following discussion. In this scenario, a PL/I main routine invokes a C++ subroutine. An exception occurs in the C++ subroutine.

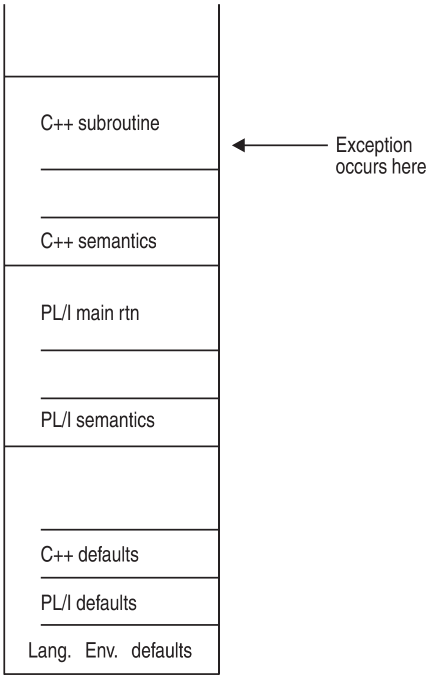


Figure 32. Stack contents when the exception occurs in C++

The actions taken are the following:

1. In the enablement step, it is determined whether the exception in the C++ routine should be enabled and treated as a condition. If any of the following are true, the exception is ignored, and processing continues at the next sequential instruction after where the exception occurred:
   - You specified `SIG_IGN` for the exception in a call to `signal()`.

     Note: The system or user abend corresponding to the `signal(SIGABND)` or the Language Environment message 3250 is not ignored. The enclave is terminated.
   - The exception is one of those listed as masked in Table 63 on page 249, and you have not enabled it using the CEE3SPM callable service.
   - You did not specify any action, but the default action for the condition is `SIG_IGN` (see Table 63 on page 249).
   - You are running under CICS and a CICS handler is pending.

   If you did none of these things, the condition is enabled and processed as a condition.
2. If a user-written condition handler has been registered on the stack frame using CEEHDLR, it is given control.

   If it issues a resume, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
3. If a C signal handler has been registered for the condition on the C++ stack frame, it is given control. If it successfully issues a resume or a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points.

   In this case, there is not a C signal handler registered for the condition.

4. The condition is still unhandled. If C++ does not recognize the condition, or if the C++ default action (listed in Table 63 on page 249) is to terminate, the condition is percolated.
5. Is a user-written condition handler has been registered on the PL/I stack frame using CEEHDLR, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine at the point where the resume cursor points. In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
6. If an ON-unit has been established for the condition being processed on the PL/I stack frame, it is given control. If it issues a GOTO out-of-block, the condition handling step ends. Execution resumes at the label of the GOTO. In this example, no ON-unit is established for the condition, so the condition is percolated.
7. What happens next depends on whether the condition is promotable to the PL/I ERROR condition. The following can happen:
   - If the condition is not promotable to the PL/I ERROR condition, then the Language Environment default actions take place, as described in Table 62 on page 249. Condition handling ends.
   - If the PL/I default action for the condition is to promote it to the PL/I ERROR condition, the condition is promoted, and another pass is made of the stack to look for ERROR ON-units or user-written condition handlers. If an ERROR ON-unit or user-written condition handler is found, it is invoked.
   - If either of the following occurs:
     - An ERROR ON-unit or user-written condition handler is found, but it does not issue a GOTO out of block or similar construct
     - No ERROR ON-unit or user-written condition handler is found

     then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled condition). Condition handling now enters the termination imminent step. Because T_I_U maps to the PL/I FINISH condition, a FINISH ON-unit or user-written condition handler is run if the stack frame in which it is established is reached.
   - If no condition handler moves the resume cursor and issued a resume, Language Environment terminates the thread.

Exception occurs in PL/I

This scenario describes the behavior of an application that contains a PL/I and a C++ routine. Refer to Figure 33 on page 165 throughout the following discussion. In this example, a C++ main routine invokes a PL/I subroutine. An exception occurs in the PL/I subroutine.

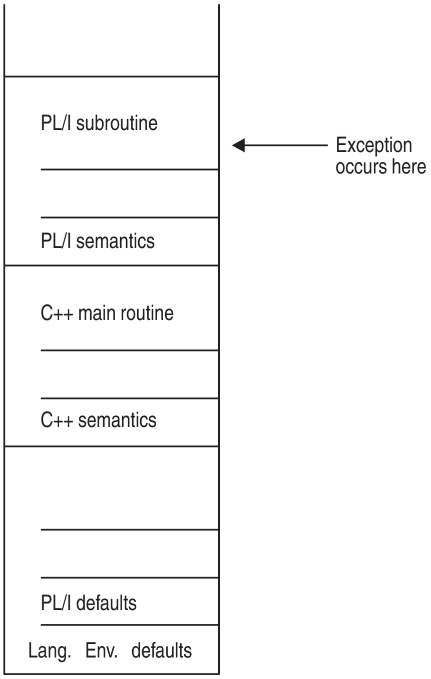


Figure 33. Stack contents when the exception occurs in PL/I

The actions taken are the following:

1. In the enablement step, PL/I determines if the exception that occurred should be handled as a condition according to the PL/I rules of enablement.
   - If the exception is to be ignored, control is returned to the next sequential instruction after where the exception occurred.
   - If the exception is to be enabled and processed as a condition, the condition handling step, described below, takes place.
2. Is a user-written condition handler has been registered on the PL/I stack frame using CEEHDLR, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine at the point where the resume cursor points. In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
3. If an ON-unit has been established for the condition being processed on the PL/I stack frame, it is given control. If it issues a GOTO out-of-block, the condition handling step ends. Execution resumes at the label of the GOTO. In this example, no ON-unit is established for the condition, so the condition is percolated.
4. If a user-written condition handler has been registered using CEEHDLR on the C++ stack frame, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   Note: There are special considerations for resuming from some IBM conditions of severity 2 or greater; see the chapter on coding user-written condition handlers in *z/OS Language Environment Programming Guide*.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
5. If a C signal handler has been registered for the condition, it is given control. If it successfully issues a resume or a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points.

In this example no C signal handler is registered for the condition, so the condition is percolated.

6. What happens next depends on whether the condition is promotable to the PL/I ERROR condition. The following can happen:
   - If the condition is not promotable to the PL/I ERROR condition, then the Language Environment default actions take place, as described in Table 62 on page 249. Condition handling ends.
   - If the PL/I default action for the condition is to promote it to the PL/I ERROR condition, the condition is promoted, and another pass of the stack is made to look for ERROR ON-units or user-written condition handlers. If an ERROR ON-unit or user-written condition handler is found, it is invoked.
   - If either of the following occurs:
     - An ERROR ON-unit or user-written condition handler is found, but it does not issue a GOTO out of block or similar construct
     - No ERROR ON-unit or user-written condition handler is found

     then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled condition). Condition handling now enters the termination imminent step. Because T_I_U maps to the PL/I FINISH condition, a FINISH ON-unit is run if the stack frame in which it is established is reached.
   - If no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Fixed-point overflow

The XL C++ compiler assumes that fixed-point overflow exceptions will be handled by S/390 hardware, while the PL/I compiler assumes that the runtime library will process these exceptions. Therefore, if a C++ to PL/I ILC application incurs a fixed-point overflow, Language Environment will try to handle this exception in the runtime library. This will result in increased CPU utilization compared to an application using only C++.

Sample C++ to PL/I ILC applications

PL/I main routine calling a C++ subroutine

```
/*COMPILATION UNIT: IBMPCX       */
/**Process lc(101),opt(0),s,map,list,stmt,a(f),ag;*/
CEPLI2C: PROC OPTIONS(MAIN);
/***************************************************************/
/* FUNCTION   :  Interlanguage communications call to         *
/*               a C++ program.                               *
/*                                                            *
/* This example illustrates an interlanguage call from        *
/* a PL/I main program to a C++ subroutine.                   *
/* The parameters passed across the call from PL/I to         *
/* C++ have the following declarations:                       *
/*                                                            *
/* PL/I fixed bin(15,0) to C++ short as pointer to BIN        *
/* PL/I fixed bin(31,0) to C++ int                            *
/* PL/I float bin(53)   to C++ double                         *
/* PL/I float bin(109)  to C++ long double                    *
/* PL/I characters      to C++ as pointer to pointer to CHAR  *
/***************************************************************/
/***************************************************************/
/* DECLARES FOR THE CALL TO C++                               *
/***************************************************************/
   DCL ADDR            BUILTIN;
   DCL J               FIXED BIN(31,0);
   DCL CECFPLI         EXTERNAL ENTRY RETURNS(FIXED BIN(31,0));
   DCL PL1_SHORT       FIXED BIN(15,0) INIT(15);
   DCL PL1_INT         FIXED BIN(31,0) INIT(31);
   DCL PL1_DOUBLE      FLOAT BIN(53) INIT (53.99999);
   DCL PL1_LONG_DOUBLE FLOAT BIN(109) INIT(3.14151617);
   DCL PL1_POINTER     PTR;
   DCL CHARSTRING      CHAR(23) INIT('PASSED CHARACTER STRING');
/**************************************************************/
/*  PROCESS STARTS HERE                                       *
/**************************************************************/
   PUT SKIP LIST ('**********************************');
   PUT SKIP LIST ('PL/I CALLING C++   EXAMPLE STARTED');
   PUT SKIP LIST ('**********************************');
   PL1_POINTER = ADDR(CHARSTRING);
   PUT SKIP LIST ('Calling C/370 subroutine');
   J = CECFPLI( ADDR(PL1_SHORT), PL1_INT, PL1_DOUBLE,
      PL1_LONG_DOUBLE, ADDR(PL1_POINTER));
   PUT SKIP LIST ('Returned from C/370 subroutine');
   IF (J ¬= 999) THEN
      PUT SKIP LIST ('Error in return code from C/370');
   PUT SKIP LIST ('**********************************');
   PUT SKIP LIST ('PL/I CALLING C++   EXAMPLE ENDED  ');
   PUT SKIP LIST ('**********************************');
END CEPLI2C;
```

C++ routine called by PL/I main routine

```
/*COMPILATION UNIT:  EDCPCX   */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern "PLI" int CECFPLI(short **c_short, int *c_int, double *c_double,
         long double *c_long_double, char ***c_character_string);
/**********************************************************************
 *This is an example of a C++ program invoked by a PL/I program.      *
 *CECFPLI is called from PL/I program CEPLI2C with the following      *
 *list of arguments:                                                  *
 *                                                                    *
 * PL/I fixed bin(15,0) to C/C++ short as pointer to BIN              *
 * PL/I fixed bin(31,0) to C/C++ int                                  *
 * PL/I float bin(53)   to C/C++ double                               *
 * PL/I float bin(109)  to C/C++ long double                          *
 * PL/I characters      to C/C++ as pointer to pointer to CHAR        *
 *                                                                    *
 * This example is using C++ as a better C. It is illustrating the    *
 * minimum number of changes required.                                *
 **********************************************************************/
int CECFPLI (short **c_short,
             int *c_int,
             double *c_double,
             long double *c_long_double,
             char *** c_character_string
              )
{
   int ret=999; /* pli is expecting 999 returned */

   fprintf(stderr,"CECFPLI STARTED\n");
/********************************************************************
 * Compare each passed argument against the C++ value.              *
 * Issue an error message for any incorrectly passed parameter.     *
 ********************************************************************/
   if (**c_short != 15)
   {
     fprintf(stderr,"**c_short not = 15\n");
     --ret;
   }

   if (*c_int != 31)
   {
     fprintf(stderr,"*c_int not = 31\n");
     --ret;
   }

   if ((53.99999 - *c_double) >1.0E-14)
   {
     fprintf(stderr,
          "53.99999 - *c_double not >1.0E-14\n");
     --ret;
   }
   if ((3.14151617 - *c_long_double) >1.0E-16)
   {
     fprintf(stderr,
          "3.14151617 - *c_long_double not >1.0E-16\n");
     --ret;
   }

   if (memcmp(**c_character_string,"PASSED CHARACTER STRING",23)
             != 0)
   {
     fprintf(stderr,"**c_character_string not %s\n",
     "\"PASSED CHARACTER STRING\"");
     --ret;
   }
/*******************************************************************
 * PL/I will check for a correct return code.                      *
 *******************************************************************/
   fprintf(stderr,"CECFPLI ENDED\n");
   return(ret);
}
```

Chapter 10. Communicating between COBOL and Fortran

This topic describes Language Environment's support for COBOL and Fortran ILC applications.

General facts about COBOL to Fortran ILC

- A load module consisting of object code compiled with any Fortran compiler link-edited with object code compiled in another language is not reentrant, regardless of whether the Fortran routine was compiled with the RENT compiler option.
- Language Environment does not support passing return codes between COBOL routines and Fortran routines.
- Fortran routines cannot operate under CICS.

Preparing for ILC

This section describe topics you might want to consider before writing an ILC application. To determine how different versions of HLLs work together, refer to the migration guides for the HLLs you plan to use.

Language Environment ILC support

Language Environment provides ILC support between the following combinations of COBOL and Fortran:

Table 45. Supported languages for Language Environment ILC

HLL pair	COBOL	Fortran
COBOL to Fortran	• VS COBOL II Version 1 Release 3 (static calls only) • COBOL/370 • COBOL for MVS & VM • COBOL for OS/390 & VM • Enterprise COBOL for z/OS	• FORTRAN IV G1 • FORTRAN IV H Extended • VS FORTRAN Version 1, except modules compiled with Release 2.0 or earlier and that either pass character arguments to, or receive character arguments from, subprograms. • VS FORTRAN Version 2, except modules compiled with Releases 5 or 6 and whose source contained any parallel language constructs or parallel callable services, or were compiled with either of the compiler options PARALLEL or EC.

Note: Dynamic calls from Fortran are only available from VS FORTRAN Version 2 Release 6.

Migrating ILC applications

All COBOL to Fortran ILC applications require a relink except those containing OS/VS COBOL or VS COBOL II programs that were compiled with the NORES compiler option. If an OS/VS COBOL program is relinked, it cannot call or be called by a Fortran routine; the OS/VS COBOL program would need to be upgraded.

Fortran provides a migration tool that replaces old library modules with Language Environment ones. For more information about Fortran's library module replacement tool, see *z/OS Language Environment Programming Guide*.

Determining the main routine

In Language Environment, only one routine can be the main routine; no other routine in the enclave can use syntax that indicates it is main.

A COBOL program is designated as main if it is the first program to run in an enclave. A Fortran routine is designated as a main routine with a PROGRAM statement, which indicates the name of the main routine. A main routine can also be designated when there are no PROGRAM, SUBROUTINE, or FUNCTION statements, in which case the name of the main routine is the default value of `MAIN` (or `MAIN#` for VS FORTRAN Version 2 Releases 5 and 6). The name of the main routine is the entry point into the load module.

Table 46 describes how COBOL and Fortran identify the main routine.

Table 46. How COBOL and Fortran main routines are determined

Language	When determined	Explanation
COBOL	Run time	Determined dynamically. If it is the first program to run, it is a main program.
Fortran	Compilation	Determined in the Fortran source by the name on the PROGRAM statement.

An entry point is defined for each supported HLL. Table 47 identifies the main and fetched entry point for each language. The table assumes that your code has been compiled using the Language Environment-conforming compilers.

Table 47. Determining the entry point

HLL	Main entry point	Fetched entry point
COBOL	Name of the first object program to get control in the object module	Program name
Fortran	Name on the PROGRAM statement, or `MAIN` (or `MAIN#` for VS FORTRAN Version 2 Releases 5 and 6), if no PROGRAM, SUBROUTINE, or FUNCTION statements are used	Subprogram name

Declaring COBOL to Fortran ILC

There are no special declarations needed in either COBOL or Fortran to use ILC between them.

Calling between COBOL and Fortran

This section describes the types of calls permitted between COBOL and Fortran, and considerations when using dynamic calls and fetch.

Types of calls permitted

Table 48 describes the types of calls between COBOL and Fortran that Language Environment allows:

Table 48. Calls permitted for COBOL and Fortran ILC

ILC direction	Static calls	Dynamic calls
COBOL to Fortran	Yes	Yes
Fortran to COBOL	Yes	Yes

Dynamic call/fetch considerations

This section describes the considerations for using dynamic calls and fetch in COBOL to Fortran ILC.

COBOL dynamically calling Fortran

Dynamic calls are made in COBOL by either a CALL statement with an identifier whose value is the called routine name, or a CALL statement with a literal in a routine that is compiled with the DYNAM compiler option. The dynamically called routine can be a Fortran routine that can statically call another COBOL program. Or it can be a COBOL program that statically calls a Fortran routine. A routine in the dynamically loaded module can then dynamically call other COBOL or Fortran routines.

You cannot use a COBOL CANCEL statement to delete a dynamically called Fortran routine.

Restriction: When a COBOL program dynamically calls a Fortran routine, the dynamically loaded module can contain only routines written in those languages that already exist in a previous load module. (The routine in the previous load module need not be called; it only needs to be present.) For a Fortran routine to be recognized, ensure that at least one of the following is present in a previous load module:

- A Fortran main program
- A Fortran routine that causes one or more Fortran runtime library routines to be link-edited into the load module. If the Fortran routine contains either an I/O statement, a mathematical function reference, or a call to any Fortran callable service (such as CPUTIME), then a library routine is included, and this requirement is satisfied.
- The Fortran signature CSECT, CEESG007. Use the following linkage editor statement to include CEESG007 if neither of the two previous conditions is true:

```
INCLUDE SYSLIB (CEESG007)
```

Fortran dynamically calling COBOL

Dynamic calls are made in Fortran by specifying the name of the routine to be loaded with the DYNAMIC compiler option, and then using the same name in a CALL statement. The dynamically called routine can be either COBOL or Fortran, and it can in turn statically call either a COBOL or Fortran routine. In the dynamically loaded module, a routine can dynamically call other COBOL or Fortran routines.

Neither a COBOL routine nor a Fortran routine can delete a dynamically loaded routine that was dynamically loaded in a Fortran routine.

Calling functions

Only a Fortran subroutine subprogram can be invoked from a COBOL program. A Fortran routine written as a function subprogram cannot be invoked from a COBOL program. Similarly, only Fortran CALL statements can be used to invoke a COBOL program. A COBOL program cannot be invoked with a function reference by a Fortran routine.

Passing data between COBOL and Fortran

Table 49 lists the data types that can be passed between COBOL and Fortran by reference.

Table 49. Supported data types between COBOL and Fortran

COBOL	Fortran
PIC S9(4) USAGE IS COMPUTATIONAL or PIC S9(4) USAGE IS BINARY	INTEGER*2
PIC S9(9) USAGE IS COMPUTATIONAL or PIC S9(9) USAGE IS BINARY	INTEGER*4
PIC S9(18) USAGE IS COMPUTATIONAL or PIC S9(18) USAGE IS BINARY	INTEGER*8
USAGE IS COMPUTATIONAL-1	REAL*4
USAGE IS COMPUTATIONAL -2	REAL*8
PIC X(n) USAGE IS DISPLAY	CHARACTER*n
USAGE IS POINTER	POINTER

Passing character data

Character data can be received by a Fortran routine only when the routine that receives the data declares the data to be of fixed length. Therefore, the following form cannot be used by a Fortran routine to receive character data:

```
CHARACTER*(*)
```

In addition, the **occurs depending on** clause cannot be specified in the COBOL declaration of the character data that is passed to the Fortran routine.

Mapping arrays

The COBOL equivalent of a Fortran array of one of the data types listed in Table 49 is a fixed-length table that includes one or more **occurs** clauses. A COBOL fixed-length table and Fortran array can be passed between COBOL and Fortran routines only if the number of repeating elements is a constant value and the elements are in contiguous storage locations. COBOL tables containing the **occurs depending on** clause and Fortran assumed-shape or assumed-size arrays cannot, therefore, be passed between COBOL and Fortran routines.

In COBOL, tables of more than one dimension are arranged in row major order, while in Fortran, arrays of more than one dimension are arranged in column major order. You can correspond elements of a Fortran array to a COBOL table simply by reversing the order of the subscripts.

Data equivalents

This section describes how COBOL and Fortran data types correspond to each other.

Equivalent data types for COBOL to Fortran

The following examples illustrate how COBOL and Fortran routines within a single ILC application might code the same data types.

16-bit signed binary integer

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBFC16I.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X  PIC S9(4) USAGE IS BINARY.
PROCEDURE DIVISION.
  MOVE 5 TO X.
  CALL "CBFF16I" USING X.
  DISPLAY "UPDATED VALUE IN COBOL: ", X.
  GOBACK.
END PROGRAM CBFC16I.
```

Fortran subroutine

```
SUBROUTINE CBFF16I( ARG )
INTEGER*2 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
ARG = ARG + 1
END
```

32-bit signed binary integer

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBFC32I.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X  PIC S9(9) USAGE IS BINARY.
PROCEDURE DIVISION.
  MOVE 5 TO X.
  CALL "CBFF32I" USING X.
  DISPLAY "UPDATED VALUE IN COBOL: ", X.
  GOBACK.
END PROGRAM CBFC32I.
```

Fortran subroutine

```
SUBROUTINE CBFF32I ( ARG )
INTEGER*4 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
ARG = ARG + 1
END
```

64-bit signed binary integer

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBFC64I.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X  PIC S9(18) USAGE IS BINARY.
PROCEDURE DIVISION.
  MOVE 5 TO X.
  CALL "CBFF64I" USING X.
  DISPLAY "UPDATED VALUE IN COBOL: ", X.
  GOBACK.
END PROGRAM CBFC64I.
```

Fortran subroutine

```
SUBROUTINE CBFF64I ( ARG )
INTEGER*8 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
ARG = ARG + 1
END
```

Short floating-point number

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBFCSFP.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X  USAGE IS COMPUTATIONAL-1.
PROCEDURE DIVISION.
  MOVE 5.0E0 TO X.
  CALL "CBFFSFP" USING X.
  DISPLAY "UPDATED VALUE IN COBOL: ", X.
  GOBACK.
END PROGRAM CBFCSFP.
```

Fortran subroutine

```
SUBROUTINE CBFFSFP ( ARG )
REAL*4 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
ARG = ARG + 1.0E0
END
```

Long floating-point number

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBFCLFP.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X  USAGE IS COMPUTATIONAL-2.
PROCEDURE DIVISION.
  MOVE 5.0E0 TO X.
  CALL "CBFFLFP" USING X.
  DISPLAY "UPDATED VALUE IN COBOL: ", X.
  GOBACK.
END PROGRAM CBFCLFP.
```

Fortran subroutine

```
SUBROUTINE CBFFLFP ( ARG )
REAL*8 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
ARG = ARG + 1.0D0
END
```

Fixed-length character data

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBFCFLC.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X  PIC X(10) USAGE IS DISPLAY.
1   Y  PIC X(10) USAGE IS DISPLAY.
PROCEDURE DIVISION.
  MOVE "1234567890" TO X.
  CALL "CBFFFLC" USING X, Y.
  DISPLAY "VALUE RETURNED TO COBOL: ", Y.
  GOBACK.
END PROGRAM CBFCFLC.
```

Fortran subroutine

```
SUBROUTINE CBFFFLC ( ARG1, ARG2 )
CHARACTER*10 ARG1, ARG2
PRINT *, 'FORTRAN ARG1 VALUE: ', ARG1
ARG2 = ARG1
END
```

Array

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBFCAF.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X.
    2  MATRIX    OCCURS 3 TIMES
                 USAGE IS COMPUTATIONAL-1.
PROCEDURE DIVISION.
  MOVE 0.0E0 TO MATRIX(1).
  MOVE 1.0E0 TO MATRIX(2).
  MOVE 2.0E0 TO MATRIX(3).
  CALL "CBFFAF" USING X.
  DISPLAY "UPDATED VALUES IN COBOL: ",
    MATRIX(1), MATRIX(2), MATRIX(3).
  GOBACK.
END PROGRAM CBFCAF.
```

Fortran subroutine

```
SUBROUTINE CBFFAF ( ARG )
REAL*4 ARG(3)
PRINT *, 'FORTRAN ARG VALUES:', ARG
DO J = 1, 3
  ARG(J) = ARG(J) + 1.0
ENDDO
END
```

Address of an array

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBFCAOA.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X.
    2 X1    OCCURS 3 TIMES
            PIC S9(9)
            USAGE IS BINARY.
1   P       USAGE IS POINTER.
PROCEDURE DIVISION.
  MOVE 0 TO X1(1).
  MOVE 1 TO X1(2).
  MOVE 2 TO X1(3).
  CALL "GETADDR" USING X, P.
  CALL "CBFFAOA" USING P.
  DISPLAY "UPDATED VALUES IN COBOL: ",
    X1(1), " ", X1(2), " ", X1(3).
  GOBACK.
IDENTIFICATION DIVISION.
PROGRAM-ID. GETADDR.
DATA DIVISION.
LINKAGE SECTION.
1   X.
    2  X1    OCCURS 3 TIMES
             PIC S9(9)
             USAGE IS BINARY.
1   P        USAGE IS POINTER.
PROCEDURE DIVISION USING X, P.
    SET P TO ADDRESS OF X.
    EXIT PROGRAM.
END PROGRAM GETADDR.
END PROGRAM CBFCAOA.
```

Fortran subroutine

```
SUBROUTINE CBFFAOA ( ARG )
POINTER*4 (ARG, Y)
INTEGER*4 Y(3)
PRINT *,
1  'FORTRAN ARRAY ARG VALUES:', Y
DO J = 1, 3
  Y(J) = Y(J) + 1
ENDDO
END
```

Equivalent data types for Fortran to COBOL

The following examples illustrate how COBOL and Fortran routines within a single ILC application might code the same data types.

16-bit signed binary integer

Sample Fortran usage

```
INTEGER*2 X
X = 5
CALL FCBC16I(X)
PRINT *,
1  'UPDATED VALUE IN FORTRAN:', X
END
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FCBC16I.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
1   X  PIC S9(4) USAGE IS BINARY.
PROCEDURE DIVISION USING X.
  DISPLAY "COBOL ARG VALUE: ", X.
  ADD 1 TO X.
  EXIT PROGRAM.
END PROGRAM FCBC16I.
```

32-bit signed binary integer

Sample Fortran usage

```
INTEGER*4 X
X = 5
CALL FCBC32I(X)
PRINT *,
1  'UPDATED VALUE IN FORTRAN:', X
END
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FCBC32I.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
1   X  PIC S9(9) USAGE IS BINARY.
PROCEDURE DIVISION USING X.
  DISPLAY "COBOL ARG VALUE: ", X.
  ADD 1 TO X.
  EXIT PROGRAM.
END PROGRAM FCBC32I.
```

64-bit signed binary integer

Sample Fortran usage

```
INTEGER*8 X
X = 5
CALL FCBC64I(X)
PRINT *,
1  'UPDATED VALUE IN FORTRAN:', X
END
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FCBC64I.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
1   X  PIC S9(18) USAGE IS BINARY.
PROCEDURE DIVISION USING X.
  DISPLAY "COBOL ARG VALUE: ", X.
  ADD 1 TO X.
  EXIT PROGRAM.
END PROGRAM FCBC64I.
```

Short floating-point number

Sample Fortran usage

```
REAL*4 X
X = 5.0
CALL FCBCSFP(X)
PRINT *,
1 'UPDATED VALUE IN FORTRAN:', X
END
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FCBCSFP.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
1   X  USAGE IS COMPUTATIONAL-1.
PROCEDURE DIVISION USING X.
  DISPLAY "COBOL ARG VALUE: ", X.
  ADD 1.0E0 TO X.
  EXIT PROGRAM.
END PROGRAM FCBCSFP.
```

Long floating-point number

Sample Fortran usage

```
REAL*8 X
X = 5.0D0
CALL FCBCLFP(X)
PRINT *,
1  'UPDATED VALUE IN FORTRAN:', X
END
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FCBCLFP.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
1   X  USAGE IS COMPUTATIONAL-2.
PROCEDURE DIVISION USING X.
  DISPLAY "COBOL ARG VALUE: ", X.
  ADD 1.0E0 TO X.
  EXIT PROGRAM.
END PROGRAM FCBCLFP.
```

Fixed-length character data

Sample Fortran usage

```
CHARACTER*10 X, Y
X = '1234567890'
CALL FCBCFLC(X, Y)
PRINT *,
1  'VALUE RETURNED TO FORTRAN: ', Y
END
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FCBCFLC.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
1   X  PIC X(10) USAGE IS DISPLAY.
1   Y  PIC X(10) USAGE IS DISPLAY.
PROCEDURE DIVISION USING X, Y.
  DISPLAY "COBOL ARG VALUE: ", X.
  MOVE X TO Y.
  EXIT PROGRAM.
END PROGRAM FCBCFLC.
```

Array

Sample Fortran usage

```
REAL*4 MATRIX(3) / 1.0, 2.0, 3.0 /
CALL FCBCAF(MATRIX)
PRINT *,
1  'UPDATED VALUES IN FORTRAN:', MATRIX
END
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FCBCAF.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  IX      PIC S9(9)
    USAGE IS BINARY.
LINKAGE SECTION.
1   X.
    2 MATRIX OCCURS 3 TIMES
      USAGE IS COMPUTATIONAL-1.
PROCEDURE DIVISION USING X.
     DISPLAY "COBOL ARG VALUES: ",
       MATRIX(1), MATRIX(2), MATRIX(3).
     PERFORM  VARYING IX
              FROM 1  BY 1
              UNTIL IX > 3
      SUBTRACT 1.0E0 FROM MATRIX(IX)
     END-PERFORM
     EXIT PROGRAM.
END PROGRAM FCBCAF.
```

Address of an array

Sample Fortran usage

```
POINTER*4 (P, I)
INTEGER*4 I(3)
INTEGER*4 J(3) / 1, 2, 3 /
P = LOC(J)
CALL FCBCAOA (P)
PRINT *,
1  'UPDATED VALUES IN FORTRAN:', I
END
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FCBCAOA.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 IX       PIC S9(9)
    USAGE IS BINARY.
LINKAGE SECTION.
1   X            USAGE IS POINTER.
1   Y.
    2  M         PIC S9(9)
                 USAGE IS BINARY
                 OCCURS 3 TIMES.
PROCEDURE DIVISION USING X.
     SET ADDRESS OF Y TO X.
     DISPLAY "COBOL ARG VALUES: ",
       M(1), " ", M(2), " ", M(3).
     PERFORM  VARYING IX
         FROM 1  BY 1
         UNTIL IX > 3
       SUBTRACT 1 FROM M(IX)
     END-PERFORM.
     EXIT PROGRAM.
END PROGRAM FCBCAOA.
```

External data

External data in Fortran and COBOL (common block in Fortran and data specified by the EXTERNAL clause in COBOL) cannot be shared among routines.

Directing output in ILC applications

COBOL and Fortran do not share files, except the Language Environment message file (the ddname specified in the Language Environment MSGFILE runtime option). You must manage all other files to ensure that no conflicts arise. Performing I/O operations on the same ddname might cause abnormal termination. If directed to the Language Environment message file, output from both COBOL and Fortran programs will be interspersed in the file.

Under COBOL, runtime messages and other related output are directed to the Language Environment message file. For COBOL programs, output from the DISPLAY UPON SYSOUT statement is also directed to the Language Environment message file if the ddname name in the MGSFILE runtime option matches that in the OUTDD compiler option. (The IBM-supplied default value for OUTDD is SYSOUT.)

Fortran runtime messages, output written to the print unit, and other output (such as output from the SDUMP callable service) are directed to the file specified by the MSGFILE runtime option. To direct this output to the file with the ddname FT*nn*F001, (where *nn* is the two-digit error message unit number), specify the runtime option MSGFILE(FT*nn*F001). If the print unit is different than the error message unit (if the PRTUNIT and the ERRUNIT runtime options have different values), output from a PRINT statement won't be directed to the Language Environment message file.

COBOL to Fortran condition handling

This section provides two scenarios of condition handling behavior in a COBOL to Fortran ILC application. If an exception occurs in a COBOL routine, the set of possible actions is as described in "Exception occurs in COBOL" on page 180. If an exception occurs in a Fortran program, the set of possible actions is as described in "Exception occurs in Fortran" on page 181.

Keep in mind that some conditions can be handled only by the HLL of the routine in which the exception occurred. Two examples are:

- In a COBOL program, if a statement has a condition handling clause added to a verb (such as ON EXCEPTION), the condition is handled within COBOL. For example, the ON SIZE clause of a COBOL DIVIDE verb (which includes the logical equivalent of the zero divide condition) is handled completely within COBOL.
- When the Fortran ERR or IOSTAT specifier is present on a Fortran I/O statement, and an error is detected while executing that statement, the Fortran language semantics take precedence over Language Environment condition handling. Control returns immediately to the Fortran routine and no condition is signaled to Language Environment.

See *z/OS Language Environment Programming Guide* for a detailed description of Language Environment condition handling. For information about Fortran condition handling semantics, see *VS FORTRAN Version 2 Language and Library Reference.*

Enclave-terminating language constructs

Enclaves can be terminated for reasons other than an unhandled condition of severity 2 or greater. HLL constructs that cause the termination of a single language enclave also cause the termination of a COBOL to Fortran enclave. In Language Environment ILC, you can issue the language construct to terminate the enclave from a COBOL or Fortran routine.

COBOL

The COBOL language constructs that cause the enclave to terminate are:

- STOP RUN

  COBOL's STOP RUN is equivalent to the Fortran stop statement. If you code a COBOL STOP RUN statement, the T_I_S (Termination Imminent Due to STOP) condition is raised.
- Call to ILBOABN0 or CEE3ABD

  Calling ILBOABN0 or CEE3ABD causes T_I_U to be signaled. Condition handlers are given a chance to handle the abend. If the abend remains unhandled, normal Language Environment termination activities occur. For example, the Fortran stop statement is honored and the Language Environment assembler user exit gains control.

  User-written condition handlers written in COBOL must be compiled with Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, or COBOL/370.

Fortran

The Fortran language constructs that cause the enclave to terminate are:

- A STOP statement
- An END statement in the main routine
- A call to EXIT or SYSRCX

- A call to DUMP or CDUMP

Except for executing the END statement in a main program all of the constructs listed above cause the T_I_S (termination imminent due to stop) condition to be signaled.

Exception occurs in COBOL

This scenario describes the behavior of an application that contains a COBOL and a Fortran routine. Refer to Figure 34 throughout the following discussion. In this scenario, a Fortran main routine invokes a COBOL subroutine. An exception occurs in the COBOL subroutine.

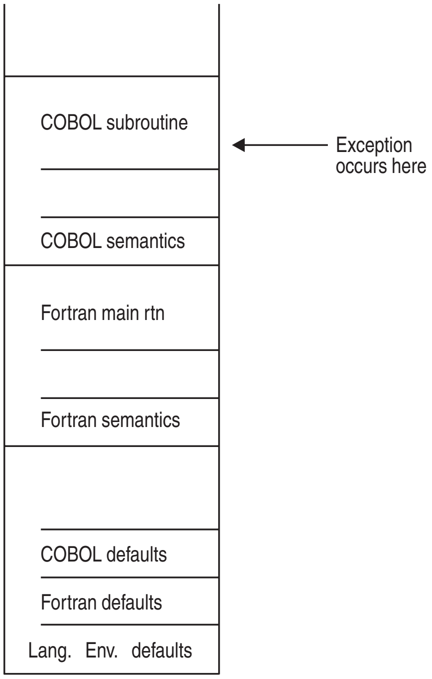


Figure 34. Stack contents when the exception occurs

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, COBOL determines whether the exception that occurred should be handled as a condition.
   - If the exception is to be ignored, control is returned to the next sequential instruction after where the exception occurred.
   - If the exception is to be enabled and processed as a condition, the condition handling step, described below, takes place.
2. If a user-written condition handler has been registered using CEEHDLR on the COBOL stack frame, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   Two areas to watch out for here are resuming from an IBM condition of severity 2 or greater (see the chapter on coding a user-written condition handler in *z/OS Language Environment Programming Guide*) and moving the resume cursor in an application that contains a COBOL program (see “GOTO out-of-block and move resume cursor” on page 182).

In this example, no user-written condition handler is registered for the condition, so the condition is percolated.

3. If the condition has a Facility_ID of IGZ, the condition is COBOL-specific. The COBOL default actions occur. If COBOL doesn't recognize the condition, condition handling continues.
4. There is no user-written condition handler on the Fortran stack frame (because CEEHDLR cannot be called from a Fortran routine), and the condition is percolated.
5. If the condition is of severity 0 or 1, Language Environment default actions take place, as described in Table 62 on page 249.
6. If the condition is of severity 2 or above, Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition) and redrive the stack. Condition handling now enters the termination imminent step.
7. If on the second pass of the stack no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Exception occurs in Fortran

This scenario describes the behavior of an application that contains a Fortran and a COBOL routine. Refer to Figure 35 throughout the following discussion. In this scenario, a COBOL main routine invokes a Fortran subroutine. An exception occurs in the Fortran subroutine.

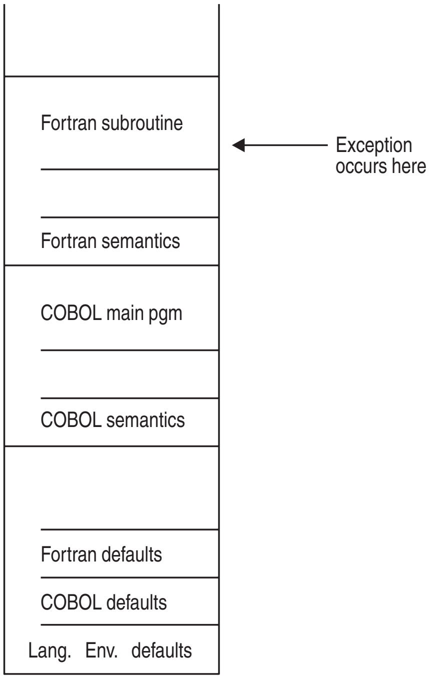


Figure 35. Stack contents when the exception occurs in Fortran

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. If an I/O error is detected on a Fortran I/O statement that contains an ERR or IOSTAT specifier, Fortran semantics take precedence. The exception is not signaled to the Language Environment condition handler.

2. In the enablement step, Fortran treats all exceptions as conditions. Processing continues with the condition handling step, described below.
3. There is no user-written condition handler on the Fortran stack frame (because CEEHDLR cannot be called from a Fortran routine), and the condition is percolated.
4. If a user-written condition handler registered using CEEHDLR is present on the COBOL stack frame, it is given control. (User-written condition handlers written in COBOL must be compiled with COBOL/370 or COBOL for MVS & VM.) If it successfully issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points. Note that you must be careful when moving the resume cursor in an application that contains a COBOL program. See "GOTO out-of-block and move resume cursor" for details.

   In this example, there is not a user-written condition handler registered for the condition, so the condition is percolated.
5. If the condition is of severity 0 or 1, Language Environment default actions take place, as described in Table 62 on page 249.
6. If the condition is of severity 2 or above, Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition) and redrive the stack. Condition handling now enters the termination imminent step.
7. If on the second pass of the stack no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

GOTO out-of-block and move resume cursor

When a GOTO out-of-block or a call to CEEMRCR causes a routine to be removed from the stack, a "non-return style" termination of the routine occurs. Multiple routines can be terminated by a non-return style termination independent of the number of ILC boundaries that are crossed. If one of the routines that is terminated by the non-return style is a COBOL program, the COBOL program can be re-entered via another call path.

If the terminated program is one of the following, the program is not deactivated. If the COBOL program does not specify RECURSIVE in the PROGRAM-ID, a recursion error is raised if you attempt to enter the routine again.

- A COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, or a VS COBOL II program compiled with the CMPR2 option
- A VS COBOL II program that is compiled with the NOCMPR2 option and contains nested programs
- A COBOL for OS/390 & VM, COBOL for MVS & VM, or COBOL/370 program that is compiled with the NOCMPR2 option and has the combination of the INITIAL attribute, nested programs, and file processing in the same compilation unit.
- An Enterprise COBOL for z/OS program that does not use the combination of the INITIAL attribute, nested routines, and file processing in the same compilation unit.

In addition, if the program is an Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, or a VS COBOL II program with the INITIAL attribute and if it contains files, the files are closed. (COBOL supports VSAM and QSAM files, and these files are closed.)

Sample ILC applications

```
@PROCESS LIST DYNAMIC(CBLFFOR)
      PROGRAM FOR2CB

*    Module/File Name:  AFHCBFOR */
****************************************************
*     Illustration of Interlanguage Communication  *
*     bewteen Fortran and COBOL.  This Fortran     *
*     program makes a dynamic call to the COBOL    *
*     routine named CBLFFOR.                       *
****************************************************
*
      INTEGER*2    INT_2   / 2 /
      INTEGER*4    INT_4   / 4 /
      REAL*8       REAL_8  / 8.0D0 /
      CHARACTER*23 CHAR_23 / ' ' /

      PRINT *, 'FOR2CB STARTED'

      CALL CBLFFOR (INT_2, INT_4, REAL_8, CHAR_23)
      IF (CHAR_23 /=  'PASSED CHARACTER STRING') THEN
         PRINT *, 'CHAR_23 NOT SET PROPERLY'
      ENDIF

      PRINT *, 'FOR2CB STARTED'
      END
```

Figure 36. Fortranprogram that dynamically calls COBOL program

```
CBL LIB,QUOTE
      *Module/File Name:  IGZTCBFO

       IDENTIFICATION DIVISION.
       PROGRAM-ID.  CBLFFOR

       DATA DIVISION.

       LINKAGE SECTION.
       77  int2                PIC S9(4) BINARY.
       77  int4                PIC S9(9) BINARY.
       77  float               COMP-2.
       77  char-string         PIC X(23).

       PROCEDURE DIVISION USING int2 int4 float char-string.

            DISPLAY "CBLFFOR STARTED".

            IF (int2 NOT = 2) THEN
              DISPLAY "INT2 NOT = 2".

            IF (int4 NOT = 4) THEN
              DISPLAY "INT4 NOT = 4".

            IF (float NOT = 8.0) THEN
              DISPLAY "FLOAT NOT = 8".

            MOVE "PASSED CHARACTER STRING" TO char-string.

            DISPLAY "CBLFFOR ENDED".
            GOBACK.
```

Figure 37. COBOLProgram dynamically called by Fortran program

Chapter 11. Communicating between COBOL and PL/I

This topic describes Language Environment's support for COBOL and PL/I ILC applications. If you are running a COBOL to PL/I ILC application under CICS, you should also consult Chapter 15, “ILC under CICS,” on page 241.

General facts about COBOL to PL/I ILC

- A COBOL program cannot be called as a function.
- The halfword prefix for PL/I varying strings is exposed, so you need to code the COBOL group data item with a halfword in front of the character string.
- PL/I supports access to COBOL files via the COBOL option of the ENVIRONMENT attribute.
- See “Using storage functions in C to PL/I ILC” on page 141 for information about how to use PL/I's storage facilities with Language Environment storage services.
- Language Environment supports the passing of return codes between COBOL and PL/I routines within an ILC application.

Preparing for ILC

This section describes topics you might want to consider before writing an application that uses ILC between COBOL and PL/I. For help in determining how different versions of HLLs work together, refer to the migration guides for the HLLs you plan to use.

Language Environment ILC support

Language Environment supports ILC between the following combinations of COBOL and PL/I:

Table 50. Supported languages for Language Environment ILC support

HLL pair	COBOL	PL/I
COBOL to PL/I	• VS COBOL II Release 3 or later • COBOL/370 Release 1 • COBOL for MVS & VM Release 2 • COBOL for OS/390 & VM • Enterprise COBOL for z/OS	• OS PL/I Version 1 Release 3 or later • OS PL/I Version 2 • PL/I for MVS & VM or later • Enterprise PL/I for z/OS

Note: Language Environment does not support ILC between OS/VS COBOL Releases 2.3 or 2.4 and PL/I. It also does not support ILC between COBOL and OS PL/I Version 1 Release 1 or 2.

Migrating ILC applications

You need to relink pre-Language Environment-conforming ILC applications in order to get Language Environment's ILC support.

If you link your VS COBOL II to OS PL/I ILC applications to the migration tool provided by OS PL/I Version 2 Release 3, you do not need to relink your applications to the Language Environment library routines. This migration tool also supports multitasking applications that contain COBOL. For more information about this migration tool, refer to the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735). The PTF numbers for the PL/I migration tool are:

- On MVS, UN76954 and UN76955

You don't need to recompile an existing ILC application unless the COBOL programs were compiled with OS/VS COBOL or the PL/I routines were compiled with OS PL/I Version 1 Release 1 or 2. In these cases, you must upgrade the source and compile with a newer version of the compilers.

Determining the main routine

In Language Environment, only one routine can be the main routine. The main routine should be presented to the linkage editor first. Because all potential main routines nominate the entry point through the END record, the correct entry point is chosen. If the main routine is not presented first, the entry point must be specified with a link-edit control card.

An entry point is defined for each supported HLL. Table 51 identifies the desired entry point. The table assumes that your code has been compiled using the Language Environment-conforming compilers.

Table 51. Determining the entry point

HLL	Main entry point	Fetched entry point
COBOL	Name of the first object program to get control in the object module	Program name
PL/I	CEESTART	CEESTART or routine name, if OPTIONS(FETCHABLE) is used.

Multitasking with PL/I and COBOL

In a multitasking ILC application, the main program, or target program of a CALL statement that creates a subtask, must be PL/I. Subsequent programs invoked from the first program can be COBOL.

COBOL programs can run in more than one PL/I subtask when all the COBOL programs in the application are Enterprise COBOL programs compiled with the THREAD compiler option. If one or more of the COBOL programs is not Enterprise COBOL compiled with the THREAD compiler option, then when a COBOL program has been invoked in a task (either the main task or a subtask), no other COBOL program can execute in any other task until the task used to invoke the COBOL program ends.

To run Enterprise COBOL for z/OS programs compiled THREAD in a PL/I multitasking application:

1. The COBOL program load modules must be link-edited RENT.
2. If a COBOL program is going to be FETCHed in a subtask, it must be FETCHed in the main task first (even though it may not be called in the main task).

Declaring COBOL to PL/I ILC

If a PL/I routine invokes a COBOL program or a COBOL program invokes a PL/I routine, you must specify entry declarations in the PL/I source code. No special declaration is required within the COBOL program.

When invoking a COBOL program from PL/I, you identify the COBOL entry point by using the OPTIONS attribute in the declaration of the entry in the calling PL/I routine. By specifying OPTIONS(COBOL) when calling a COBOL program, you request that the PL/I compiler generate a parameter list for the COBOL program in the style COBOL accepts.

In a PL/I routine that calls a COBOL program, the declaration of the COBOL entry point looks like the following:

```
DCL COBOLEP ENTRY OPTIONS(COBOL);
```

The entry points in a PL/I routine invoked from a COBOL program must be identified by the appropriate options in the corresponding PL/I PROCEDURE or ENTRY statement, as illustrated here:

```
PLIEP: PROCEDURE (parms) OPTIONS(COBOL);
```

`parms` specifies parameters that are passed from the calling COBOL program. `OPTIONS(COBOL)` specifies that the entry point can be invoked only by a COBOL program.

For more information about the COBOL option, see the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).

In addition to the COBOL option, other options suppress remapping of data aggregates. These are described in "Using aggregates" on page 190.

Only data types common to both languages can be passed or received.

Building a reentrant COBOL to PL/I application

PL/I and COBOL reentrancy

In Language Environment, the reentrancy schemes are maintained at their current support for either COBOL and PL/I. However, ILC applications in which PL/I calls COBOL are now reentrant (assuming that all COBOL and PL/I routines in the application are reentrant).

PL/I reentrancy

You should use PROC OPTIONS(REENTRANT) for all external procedures in a multitasking environment.

Reentrancy for PL/I multitasking applications

COBOL programs running in a PL/I multitasking application should be compiled with the RENT compiler option. COBOL programs compiled with the NORENT compiler option will run in an ILC application; however, once a COBOL NORENT program has run in one task, the same program cannot be used in another task.

Calling between COBOL and PL/I

This section describes the types of calls that are permitted between COBOL and PL/I as well as dynamic call/fetch considerations.

Types of calls permitted

Table 52 on page 188 describes the types of calls between COBOL and PL/I that Language Environment allows:

Table 52. Calls permitted for COBOL and PL/I

ILC direction	Static calls	Dynamic calls	Fetch/calls
COBOL to PL/I	Yes	Yes	N/A
PL/I to COBOL	Yes	N/A	Yes

Dynamic call/fetch considerations

This section describes the call/fetch differences between COBOL to PL/I dynamic CALLs and PL/I to COBOL fetches.

COBOL dynamically calling PL/I

Dynamically loaded modules that contain ILC cannot be released by using the COBOL CANCEL verb. The dynamically load module is instead released by Language Environment termination processing.

A COBOL program can dynamically CALL a PL/I routine; COBOL programs cannot dynamically CALL PL/I routines that were compiled by compilers previous to Language Environment-conforming version. Fetched PL/I routines must adhere to the restrictions listed in the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735). Enterprise PL/I for z/OS has lifted some of the fetching restrictions. See the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735) for more information.

If a PL/I procedure is to be dynamically loaded, you must do either of the following:

- Specify OPTIONS(COBOL) on the PROC statement of the program that you compile and use the routine name as the entry point when you link-edit it.
- Specify OPTIONS(COBOL FETCHABLE) on the PROC statement of the program that you compile.

In multitasking ILC applications, once a COBOL program is called by PL/I, it can dynamically CALL a PL/I subroutine by using the COBOL CALL identifier statement or the DYNAM compiler option.

PL/I fetching COBOL

PL/I routines can call Language Environment-conforming COBOL programs only. ILC between PL/I and COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL for z/OS is supported within the fetched load module.

When using PL/I for MVS & VM or earlier, you cannot use the PL/I RELEASE statement to release a COBOL program that was explicitly loaded by FETCH. A COBOL CANCEL statement cannot be issued against any routine dynamically loaded using the PL/I FETCH statement.

When using Enterprise PL/I, the fetch and release of a COBOL program and COBOL CANCEL of a program that was dynamically loaded using the PL/I FETCH statement is supported.

Passing data between COBOL and PL/I

This section lists the data types that are supported between COBOL and PL/I. It also includes information about mapping aggregates across the two languages.

Supported data types between COBOL and PL/I

The data types supported between COBOL and PL/I are listed in Table 53.

Table 53. Supported data types between COBOL and PL/I

COBOL	PL/I
PIC S9(4) USAGE IS BINARY	REAL FIXED BINARY(15,0)
PIC S9(9) USAGE IS BINARY	REAL FIXED BINARY(31,0)
PIC S9(18) USAGE IS BINARY	REAL FIXED BINARY(63,0)
PIC 9(4) USAGE IS BINARY	REAL FIXED BINARY(16,0) UNSIGNED
PIC 9(9) USAGE IS BINARY	REAL FIXED BINARY(32,0) UNSIGNED
PIC 9(18) USAGE IS BINARY	REAL FIXED BINARY(64,0) UNSIGNED
COMP-1	REAL FLOAT DECIMAL(6)
COMP-2	REAL FLOAT DECIMAL(16)
PIC S9(n) USAGE IS PACKED-DECIMAL	FIXED DECIMAL(n)
PIC S9(n) USAGE IS COMPUTATIONAL-3	FIXED DECIMAL(n)
PIC X(n) USAGE IS DISPLAY	CHARACTER(n)
PIC G(n) USAGE IS DISPLAY-1	GRAPHIC(n)
PIC N(n) USAGE IS DISPLAY-1	GRAPHIC(n)
PIC N(n) USAGE IS NATIONAL	WIDECHAR(n)
PIC X(n)	CHAR(n)
PIC 9(n) USAGE IS DISPLAY	PICTURE '(n)9'
groups	aggregates
numeric edited	numeric character
POINTER	POINTER
FUNCTION-POINTER	POINTER or LIMITED ENTRY

`01 PLISTRING.`
`   02 LEN  PIC 9(4)  BINARY.`
`   02 CHAR PIC X OCCURS 1 TO n`
`        DEPENDING ON LEN.`
	PLISTRING CHAR(n) VARYING

If you use binary data that contains more significant digits than is specified in the COBOL PICTURE clause, you must either use USAGE IS COMP-5 instead of USAGE IS BINARY, or you must use the compiler option TRUNC(BIN) to guarantee that truncation of high-order digits does not occur.

COBOL PIC N(n) without a USAGE clause maps to PL/I GRAPHIC(n) when the NSYMBOL(DBCS) compiler option is in effect. COBOL PIC N(n) without a USAGE clause maps to PL/I WIDECHAR(n) when the NSYMBOL(NATIONAL) compiler option is in effect.

For PL/I double-word binary support (REAL FIXED BINARY(63) or REAL FIXED BINARY(64) UNSIGNED), you must use the LIMITS(FIXEDBIN(31,63)) or the LIMITS(FIXEDBIN(63)) compiler option.

For PL/I 31-digit decimal support, you must use the LIMITS(FIXEDDEC(15,31)) or the LIMITS(FIXEDDEC(31)) compiler option. For COBOL 31-digit decimal support, you must use the ARITHS(EXTEND) compiler option.

PL/I program control data is used to control the execution of your routine. It consists of the area, entry, event, file, label, and locator data types. Program control data can be passed through a COBOL program to a PL/I routine.

COBOL represents the NULL pointer value as X'00000000'. PL/I represents the NULL pointer value as either X'00000000' using the SYSNULL built-in function or as X'FF000000' using the NULL built-in function. You are responsible for managing the different NULL values when passing pointers between COBOL and PL/I.

You must ensure that the physical layout of the data matches when passing data by pointers between PL/I and COBOL. This particularly applies when passing aggregates/groups and strings.

The non-address bits in all fullword pointers declared in PL/I source code should always be zero. If they are not, results are unpredictable.

Using aggregates

PL/I and COBOL map structures differently. In PL/I, the alignment of parameters is determined by the use of the ALIGNED and UNALIGNED attributes. For best results, all parameters passed between PL/I and COBOL routines should be declared using the ALIGNED attribute. The equivalent specification in COBOL is the SYNCHRONIZED clause. See the appropriate language reference and programming guide for details about the ALIGNED attribute.

COBOL and PL/I alignment requirements

COBOL alignment: COBOL structures are mapped as follows. Working from the beginning, each item is aligned to its required boundary in the order in which it is declared. The structure starts on a doubleword boundary.

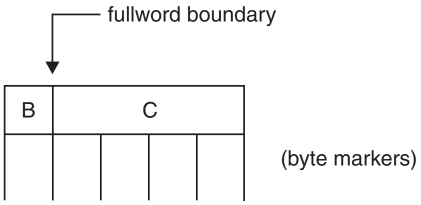
If you specify the SYNCHRONIZED phrase, then BINARY and floating-point data items are aligned on halfword, fullword, doubleword boundaries, depending on their length. If SYNCHRONIZED is not specified, then all data items are aligned on a byte boundary only.

PL/I alignment: PL/I structures are mapped by a method that minimizes the unused bytes in the structure. Simply put, the method used is to first align items in pairs, moving the item with the lesser alignment requirement as close as possible to the item with the greater alignment requirement. The method is described in full in the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).

Examples of alignment: Consider, for example, a structure consisting of a single character and a fullword fixed binary item. The fullword fixed binary item has a fullword alignment requirement; the character has a byte alignment requirement. In PL/I, ALIGNED is the default, and the structure is declared as follows:

```
DCL 1 A,
    2 B CHAR(1),
    2 C FIXED BINARY(31,0);
```

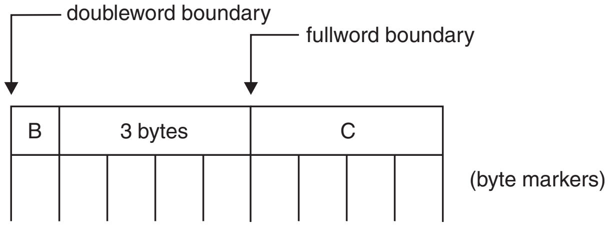
and is held like this:



In COBOL, using SYNCHRONIZED, the structure would be declared as follows:

```
01 A SYNCHRONIZED.
   02 B PIC X DISPLAY.
   02 C PIC S9(9) BINARY.
```

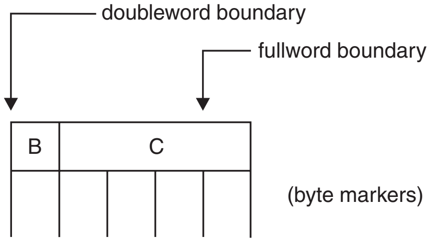
and is held like this:



In COBOL, without SYNCHRONIZED, the structure would be declared as follows:

```
01 A.
   02 B PIC X.
   02 C PIC S9(9) USAGE BINARY.
```

and is held like this:



Mapping aggregates

When passing aggregates between COBOL and PL/I, you should ensure that the storage layout matches in each HLL. Also, be sure to completely declare every byte in the aggregate so that there are no open fields.

HLL facilities provide listings of the aggregate elements to help you perform the mapping. The COBOL MAP compiler option and PL/I AGGREGATE compiler option provide a layout of aggregates.

Arrays in PL/I map to tables (OCCURS clause) in COBOL.

The options in the entry declaration that inhibit or restrict the remapping of data aggregates in PL/I are listed as follows:

NOMAP
Specifies that a dummy argument is not created by PL/I. The aggregate is passed by reference to the invoked routine.

NOMAPIN
Specifies that if a dummy argument is created by PL/I, it is not initialized with the values of the aggregate.

NOMAPOUT
Specifies that if a dummy argument is created by PL/I, its values are not assigned by the aggregate upon return to the invoking routine.

ARGn
Applies to the NOMAP, NOMAPIN, and NOMAPOUT options. It specifies arguments to which these options apply. If ARGn is omitted, a specified option applies to all arguments.

Data equivalents

This section describes how PL/I and COBOL data types correspond to each other.

Equivalent data types for COBOL to PL/I

The following examples illustrate how COBOL and PL/I routines within a single ILC application might code the same data types.

16-bit signed binary integer

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CSFB15.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X  PIC S9(4) USAGE IS BINARY.
PROCEDURE DIVISION.
    MOVE 16 to X.
    CALL "PTFB15" USING X.
    GOBACK.
END PROGRAM CSFB15.
```

PL/I procedure

```
PTFB15: Proc(X) Options(COBOL);
  Dcl SYSPRINT file;
  Dcl X Fixed Binary(15,0);

  Put Skip List( X );
End;
```

32-bit signed binary integer

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CSFB31.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1 X PIC S9(9) USAGE IS BINARY.
PROCEDURE DIVISION.
    MOVE 5 to X.
    CALL "PTFB31" USING X.
    GOBACK.
END PROGRAM CSFB31.
```

PL/I procedure

```
PTFB31: Proc (X) Options(COBOL);
  Dcl SYSPRINT file;
  Dcl X Fixed Binary(31);

  Put Skip List( X );
End;
```

Short floating-point number

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CSFTD6.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X  USAGE IS COMPUTATIONAL-1.
PROCEDURE DIVISION.
    MOVE 16 TO X.
    CALL "PTFTD6" USING X.
    GOBACK.
END PROGRAM CSFTD6.
```

PL/I procedure

```
PTFTD6: Proc ( X ) Options(COBOL);
  Dcl SYSPRINT file;
  Dcl X Float Decimal(6);

  Put Skip List( X );
End;
```

Long floating-point number

Sample COBOL usage

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CSFTD16.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
1   X  USAGE IS COMPUTATIONAL-2.
PROCEDURE DIVISION.
    MOVE 0 TO X.
    CALL "PTFTD16" USING X.
    GOBACK.
END PROGRAM CSFTD16.
```

PL/I procedure

```
PTFTD16: Proc (X) Options(COBOL);
  Dcl SYSPRINT file;
  Dcl X Float Decimal(16);

  Put Skip List( X );
End;
```

Equivalent data types for PL/I to COBOL

The following examples illustrate how COBOL and PL/I routines within a single ILC application might code the same data types.

16-bit signed binary integer

Sample PL/I usage

```
PSFB15: Proc Options(Main);
  Dcl SYSPRINT file;
  Dcl X Fixed Binary(15,0);
  Dcl CTFB15 external entry
      Options(COBOL);
    X=1;

    Call CTFB15( X );
End;
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CTFB15.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
1   X  PIC S9(4) USAGE IS BINARY.
PROCEDURE DIVISION USING X.
    DISPLAY X.
    GOBACK.
END PROGRAM CTFB15.
```

32-bit signed binary integer

Sample PL/I usage

```
PSFB31: Proc Options(Main);
  Dcl SYSPRINT file;
  Dcl X Fixed Binary(31);
  Dcl CTFB31 external entry
        Options(COBOL);
      X=1;

      Call CTFB31( X );
End;
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CTFB31.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
1   X  PIC S9(9) USAGE IS BINARY.
PROCEDURE DIVISION USING X.
    DISPLAY X.
    GOBACK.
END PROGRAM CTFB31.
```

Short floating-point number

Sample PL/I usage

```
PSFTD6: Proc Options(Main);
  Dcl SYSPRINT file;
  Dcl X Float Decimal(6);
  Dcl CTFTD6 external entry
        Options(COBOL);
      X=1;

      Call CTFTD6( X );
End;
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CTFTD6.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
1   X  USAGE IS COMP-1.
PROCEDURE DIVISION USING X.
    DISPLAY X.
    GOBACK.
END PROGRAM CTFTD6.
```

Long floating-point number

Sample PL/I usage

```
PSFTD16: Proc Options(Main);
  Dcl SYSPRINT file;
  Dcl X Float Decimal(16);
  Dcl CTFTD16 external entry
        Options(COBOL);
      X=1;

      Call CTFTD16( X );
End;
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CTFTD16.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
1   X  USAGE IS COMP-2.
    PROCEDURE DIVISION USING X.
    DISPLAY X.
    GOBACK.
END PROGRAM CTFTD16.
```

Fixed-length character data

Sample PL/I usage

```
PSFSTR: Proc Options(Main);
  Dcl SYSPRINT file;
  Dcl Str Char(80);
  Dcl CTFSTR external entry
        Options(COBOL);
      Str = 'Test PL/I-COBOL message.';

      Call CTFSTR( Str );
End;
```

COBOL subroutine

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CTFSTR.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 STR    PIC X(80).
PROCEDURE DIVISION USING STR.
    DISPLAY STR.
    GOBACK.
END PROGRAM CTFSTR.
```

Data type equivalents when TRUNC(BIN) is specified

If you specify the COBOL compiler option TRUNC(BIN), the following data types are equivalent between PL/I and COBOL:

Table 54. Equivalent data types between PL/I and COBOL when TRUNC(BIN) compiler option specified

PL/I	COBOL
REAL FIXED BINARY(31,0)	PIC S9(9) USAGE IS BINARY PIC S9(8) USAGE IS BINARY PIC S9(7) USAGE IS BINARY PIC S9(6) USAGE IS BINARY PIC S9(5) USAGE IS BINARY
REAL FIXED BINARY(15,0)	PIC S9(4) USAGE IS BINARY PIC S9(3) USAGE IS BINARY PIC S9(2) USAGE IS BINARY PIC S9(1) USAGE IS BINARY

Name scope of external data

In programming languages, the *name scope* is defined as the portion of an application within which a particular declaration applies or is known.

The name scope of external data under COBOL is the enclave. Under PL/I, it is the load module. Figure 38 and Figure 39 on page 196 illustrate these differences.

Due to the difference in name scope for COBOL and PL/I, PL/I external data and COBOL EXTERNAL data (both reentrant and non-reentrant) do not map to each other, regardless of whether you attempt to share the data within the same load module or across different load modules. External variables with the same name are considered separate between COBOL and PL/I.

If your application relies on the separation of external data, however, do not give the data the same name in both languages within a single application. By giving the data in each load module a different name, you can change the language mix in the application later, and your application will still behave as you expect it to.

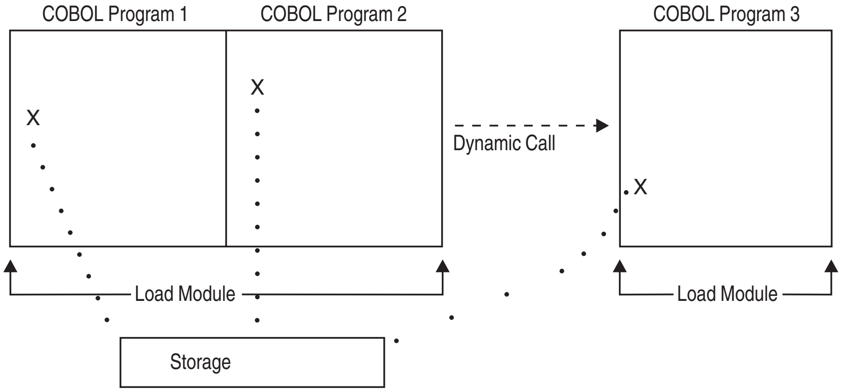


Figure 38. Name scope of external variables for COBOL dynamic call

In Figure 38, COBOL Programs 1, 2, and 3 comprise a COBOL run unit (a Language Environment enclave). EXTERNAL data declared in COBOL Program 1 maps to that declared in COBOL Program 2 in the same load module. When a dynamic call to COBOL Program 3 in another load module is made, the EXTERNAL data still maps, because the name scope of EXTERNAL data in

COBOL is the enclave.

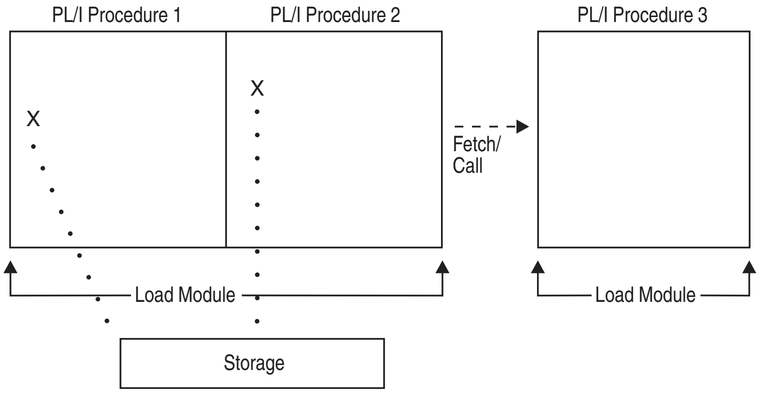


Figure 39. Name scope of external variables for PL/I fetch

The name scope of external data in PL/I is the load module. In Figure 39, external data declared in PL/I Procedure 1 maps to that declared in PL/I Procedure 2 in the same load module. The fetched PL/I Procedure 3 in another load module cannot have any external data in it.

Name space of external data

In programming languages, the *name space* is defined as the portion of a load module within which a particular declaration applies or is known. Figure 40 and Figure 41 on page 197 illustrates that, like the name scope, the name space of external data differs between PL/I and COBOL.

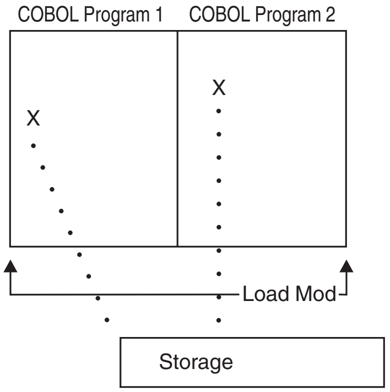


Figure 40. Name space of external data for COBOL static call to COBOL

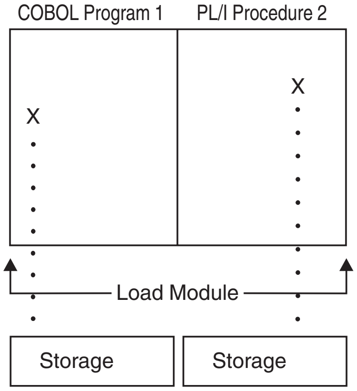


Figure 41. Name space of external data in COBOL static call to PL/I

Figure 40 on page 196 and Figure 41 illustrate that within the same load module, the name space of COBOL programs is the same. However, the name spaces of a COBOL program and a PL/I procedure within the same load module are not the same. If you give external data the same name in both languages, an incompatibility in external data mapping can occur in the future.

External data in multitasking applications

External data defined by a COBOL program persists until the enclave terminates. The external data, then, defined in a COBOL subtask, persists even if the subtask terminates; therefore, the same external data can be used by COBOL program in more than one task.

The following example illustrates how you might use external data in a multitasking ILC application:

1. COBOL PGMA is called in subtask 1. COBOL PGMA defines external PIC X data item as EXT1. PGMA sets EXT1 to 'Z'.
2. Subtask 1 terminates.
3. COBOL PGMB is called in subtask 2. PGMB has the same external data item EXT1. The value of EXT1 is 'Z', or the value that was set by PGMA in subtask 1.

Sharing data

This section describes how to share files between COBOL and PL/I, both in a standard environment, and under PL/I multitasking.

Sharing files between COBOL and PL/I

By specifying the COBOL option in the PL/I ENVIRONMENT attributes in a file declaration, files can be shared between COBOL and PL/I. However, if structures are used in a file, mapping can be different, as described in “Using aggregates” on page 190. When structures are in a file and you don't know whether the mapping is the same, both COBOL and PL/I structures are mapped. Then the object module transfers the data between structures immediately after reading the data for input and immediately before writing the data for output.

During compilation, the compiler examines the record variable to see if there are any structures. If there are no structures, no further action is taken. If there are structures, the compiler tests to see if the mapping of the structure(s) is the same in PL/I and COBOL. If the mapping is the same, no action is required. If the compiler cannot determine that the mapping is the same, or if the structure is adjustable, both structures will be mapped.

When the compiler reformats the data, and when a record I/O statement involving a file with the COBOL option is executed, the following actions take place:

INPUT
The data is read into a structure that has been mapped using the COBOL mapping algorithm and assigned to a PL/I mapped structure.

OUTPUT
Before the output takes place, the data in the PL/I structure is assigned to a structure mapped for COBOL. The output to the file then takes place from the second structure.

File sharing under PL/I multitasking

Files can be opened by COBOL programs that are running in the main task or in a subtask. However, when the task terminates, all open files, external and non-external, defined in COBOL programs within the task, are closed.

Directing output in ILC applications

Under COBOL, runtime messages and other related output are directed to the MSGFILE ddname. The output from DISPLAY goes to the MSGFILE ddname only when the OUTDD compiler option ddname matches the MSGFILE ddname. Output from the COBOL DISPLAY UPON SYSOUT statement is also directed to the default MSGFILE ddname. If you want to intersperse output from COBOL and PL/I, you must compile your COBOL program using OUTDD(SYSPRINT) to override the default OUTDD.

Under PL/I, runtime messages and other related output (such as ON condition SNAP output) are directed to the file specified in the Language Environment MSGFILE runtime option, instead of to the PL/I SYSPRINT STREAM PRINT file. User-specified output is still directed to the PL/I SYSPRINT STREAM PRINT file. To direct this output to the Language Environment MSGFILE file, specify the runtime option MSGFILE(SYSPRINT).

COBOL to PL/I condition handling

This section offers two scenarios of condition handling behavior in a COBOL to PL/I ILC application. If an exception occurs in a COBOL program, the set of possible actions is as described in “Exception occurs in COBOL” on page 199. If an exception occurs in a PL/I routine, the set of possible actions is as described in “Exception occurs in PL/I” on page 201.

If a PL/I routine is currently active on the stack, PL/I language semantics can be applied to handle conditions that occur in non-PL/I routines within an ILC application. For example, PL/I semantics apply to Language Environment hardware conditions that map directly to PL/I conditions such as ZERODIVIDE, even if they occur in a non-PL/I routine. Also, PL/I treats any unknown condition of severity 2 or greater as the ERROR condition. In a case in which a COBOL-specific condition of severity 2 or greater is passed to a PL/I stack frame, an ERROR ON-unit can handle it on the first pass of the stack.

However, some conditions can be handled only by the HLL of the routine in which the exception occurred. Two examples are:

- Conditions raised using the PL/I statement SIGNAL are PL/I-specific conditions and can be handled only by PL/I.
- In a COBOL program, if a statement has a condition handling clause added to a verb (such as ON EXCEPTION), the condition is handled within COBOL. For example, the ON SIZE clause of a COBOL DIVIDE verb (which includes the logical equivalent of zero divide condition) is handled completely within COBOL.

Multitasking ILC consideration

User-written condition handlers registered with the CEEHDLR callable service are not supported in PL/I multitasking applications.

For a detailed description of Language Environment condition handling, see *z/OS Language Environment Programming Guide*.

Enclave-terminating language constructs

Enclaves might be terminated due to reasons other than an unhandled condition of severity 2 or greater. In Language Environment ILC, you can issue an HLL language construct to terminate a COBOL to PL/I enclave from either COBOL or PL/I.

COBOL

The COBOL language constructs that cause the enclave to terminate are:

- A STOP RUN

  COBOL's STOP RUN is equivalent to the PL/I STOP statement. If you code a COBOL STOP RUN statement, the T_I_S (Termination Imminent Due to STOP) condition is raised.
- A call to ILBOABN0 or CEE3ABD

  Calling ILBOABN0 or CEE3ABD causes T_I_U to be signaled. Condition handlers are given a chance to handle the abend. If the abend remains unhandled, normal Language Environment termination activities occur. For example, the Language Environment assembler user exit gains control.

  User-written condition handlers written in COBOL must be compiled with Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, or COBOL/370.

PL/I

The PL/I language constructs that cause the enclave to terminate are:

- A STOP or an EXIT statement

  If you code a STOP or EXIT statement, the T_I_S (Termination Imminent Due to STOP) condition is raised.
- A call to PLIDUMP with the S or E option

  If you call PLIDUMP with the S or E option, neither termination imminent condition is raised before the enclave is terminated. See *z/OS Language Environment Debugging Guide* for syntax of the PLIDUMP service.

Exception occurs in COBOL

This scenario describes the behavior of an application that contains a COBOL and a PL/I routine. Refer to Figure 42 on page 200 throughout the following discussion. In this scenario, a PL/I main routine invokes a COBOL subroutine. An exception occurs in the COBOL subroutine.

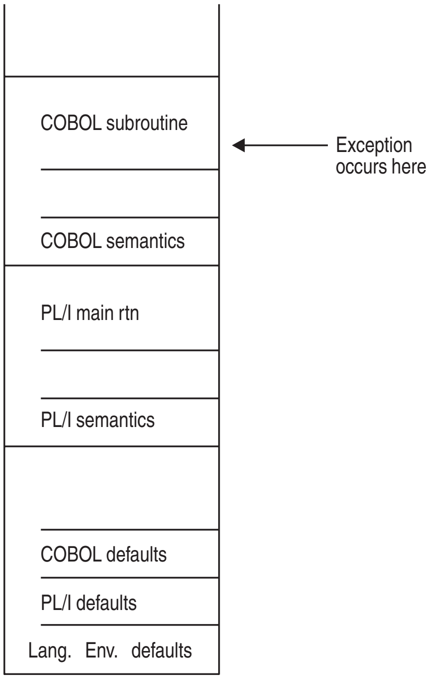


Figure 42. Stack contents when the exception occurs in COBOL

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, COBOL determines whether the exception that occurred should be handled as a condition.
   - If the exception is to be ignored, control is returned to the next sequential instruction after where the exception occurred.
   - If the exception is to be enabled and processed as a condition, the condition handling step takes place.
2. If a user-written condition handler has been registered using CEEHDLR on the COBOL stack frame, it is given control.

   If it issues a resume, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   Two areas to watch out for here are resuming from an IBM condition of severity 2 or greater (see the information about coding a user-written condition handler in *z/OS Language Environment Programming Guide*) and moving the resume cursor in an application that contains a COBOL program (see “GOTO out-of-block and move resume cursor” on page 203).

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
3. Is a user-written condition handler has been registered on the PL/I stack frame using CEEHDLR, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine at the point where the resume cursor points. In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
4. If an ON-unit has been established for the condition being processed on the PL/I stack frame, it is given control. If it issues a GOTO out-of-block, the condition handling step ends. Execution resumes at the label of the GOTO. In this example, no ON-unit is established for the condition, so the condition is percolated.

5. After all stack frames have been visited, and if the condition is COBOL-specific (with a facility ID of IGZ), the COBOL default action occurs. Otherwise, the Language Environment default actions take place.
6. What happens next depends on whether the condition is promotable to the PL/I ERROR condition. The following can happen:
   - If the condition is not promotable to the PL/I ERROR condition, then the Language Environment default actions take place, as described in Table 62 on page 249. Condition handling ends.
   - If the PL/I default action for the condition is to promote it to the PL/I ERROR condition, The condition is promoted, and another pass of the stack is made to look for ERROR ON-units or user-written condition handlers. If an ERROR ON-unit or user-written condition handler is found, it is invoked.
   - If either of the following occurs:
     - An ERROR ON-unit or user-written condition handler is found, but it does not issue a GOTO out of block or similar construct
     - No ERROR ON-unit or user-written condition handler is found

     then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled condition). Condition handling now enters the termination imminent step. Because T_I_U maps to the PL/I FINISH condition, both FINISH ON-units and user-written condition handlers can be run if the stack frames in which they are established is reached.
   - If no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Exception occurs in PL/I

This scenario describes the behavior of an application that contains a PL/I and a COBOL routine. Refer to Figure 43 on page 202 throughout the following discussion. In this scenario, a COBOL main program invokes a PL/I subroutine. An exception occurs in the PL/I subroutine.

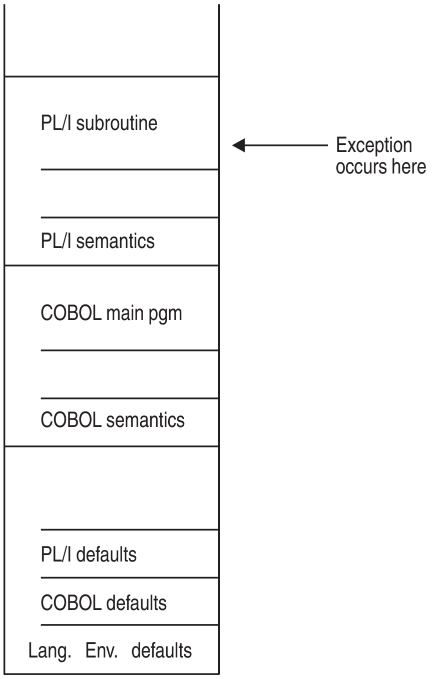


Figure 43. Stack contents when the exception occurs in PL/I

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, PL/I determines if the exception that occurred should be handled as a condition according to the PL/I rules of enablement.
   - If the exception is to be ignored, control is returned to the next sequential instruction after where the exception occurred.
   - If the exception is to be enabled and processed as a condition, the condition handling step takes place.
2. Is a user-written condition handler has been registered on the PL/I stack frame using CEEHDLR, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine at the point where the resume cursor points. In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
3. If an ON-unit has been established for the condition being processed on the PL/I stack frame, it is given control. If it issues a GOTO out-of-block, the condition handling step ends. Execution resumes at the label of the GOTO. In this example, no ON-unit is established for the condition, so the condition is percolated.
4. If a user-written condition handler registered using CEEHDLR is present on the COBOL stack frame, it is given control. (User-written condition handlers written in COBOL must be compiled with COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL for z/OS.) If it successfully issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points. Note that you must be careful when moving the resume cursor in an application that contains a COBOL program. See “GOTO out-of-block and move resume cursor” on page 203 for details.

   In this example, there is not a user-written condition handler registered for the condition, so the condition is percolated.

5. What happens next depends on whether the condition is promotable to the PL/I ERROR condition. The following can happen:
   - If the condition is not promotable to the PL/I ERROR condition, then the Language Environment default actions take place, as described in Table 62 on page 249. Condition handling ends.
   - If the PL/I default action for the condition is to promote it to the PL/I ERROR condition, The condition is promoted, and another pass is made of the stack to look for ERROR ON-units or user-written condition handlers. If an ERROR ON-unit or user-written condition handler is found, it is invoked.
   - If either of the following occurs:
     - An ERROR ON-unit or user-written condition handler is found, but it does not issue a GOTO out-of-block or similar construct
     - No ERROR ON-unit or user-written condition handler is found

     then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled condition). Condition handling now enters the termination imminent step. Because T_I_U maps to the PL/I FINISH condition, both FINISH ON-units and user-written condition handlers can be run if the stack frames in which they are established are reached.
   - If no condition handler moves the resume cursor and issued a resume, Language Environment terminates the thread.

GOTO out-of-block and move resume cursor

When a GOTO out-of-block or a call to CEEMRCR causes a routine to be removed from the stack, a “non-return style” termination of the routine occurs. Multiple routines can be terminated by a non-return style termination independent of the number of ILC boundaries that are crossed. If one of the routines that is terminated by the non-return style is a COBOL program, the COBOL program can be reentered via another call path.

If the terminated program is one of the following, the program is not deactivated. If the COBOL program does not specify RECURSIVE in the PROGRAM-ID, a recursion error is raised if you attempt to enter the routine again.

- A COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, or a VS COBOL II program compiled with the CMPR2 option
- A VS COBOL II program that is compiled with the NOCMPR2 option and contains nested programs
- A COBOL for OS/390 & VM, COBOL for MVS & VM, or COBOL/370 program that is compiled with the NOCMPR2 option and has the combination of the INITIAL attribute, nested programs, and file processing in the same compilation unit.
- A Enterprise COBOL for z/OS program that does not use the combination of the INITIAL attribute, nested routines, and file processing in the same compilation unit.

In addition, if the program is a COBOL program with the INITIAL attribute and if it contains files, the files are closed. (COBOL supports VSAM and QSAM files, and these files are closed.)

Sample PL/I to COBOL applications

PL/I routine calling COBOL subroutine

```
*PROCESS MACRO;
 PL1CBL: PROC OPTIONS(MAIN);
 /*Module/File Name: IBMPCB
 /*********************************************************************/
 /* FUNCTION   :  Interlanguage communications call to a              *
 /*               a COBOL program.                                    *
 /*                                                                   *
 /* This example illustrates an interlanguage call from               *
 /* a PL/I main program to a COBOL subroutine.                        *
 /* The parameters passed across the call from PL/I to                *
 /* COBOL have the following characteristics:                        *
 /*                                                                   *
 /* Data Type         PL/I Attributes       COBOL Data Description    *
 /* ----------------  --------------------  ----------------------    *
 /* Halfword Integer  REAL FIXED BIN(15,0)  PIC S9999 USAGE COMP      *
 /* Fullword Integer  REAL FIXED BIN(31,0)  PIC S9(9) USAGE COMP      *
 /* Packed Decimal    REAL FIXED DEC(m,n)   PIC S9(m-n).9(n) COMP-3   *
 /* Short Floating    REAL FLOAT DEC(6)     USAGE COMP-1              *
 /*                    or REAL FLOAT BIN(21)                          *
 /* Long Floating     REAL FLOAT DEC(16)    USAGE COMP-2              *
 /*                   or REAL FLOAT BIN(53)                           *
 /* Character string  CHARACTER(n)          PIC X(n) USAGE DISPLAY    *
 /* DBCS string       GRAPHIC(n)            PIC G(n) USAGE DISPLAY-1  *
 /*                                                                   *
 /* Note 1:  in COBOL, the usages COMPUTATIONAL-1 and COMP-1          *
 /*          are equivalent.                                          *
 /* Note 2:  in COBOL, the usages COMPUTATIONAL-2 and COMP-2          *
 /*          are equivalent.                                          *
 /* Note 3:  in COBOL, the usages FIXED-DECIMAL, COMP-3, and          *
 /*          COMPUTATIONAL-3 are all equivalent.                      *
 /* Note 4:  in COBOL, the usages COMP, COMPUTATIONAL, COMP-4,        *
 /*          COMPUTATIONAL-4, and BINARY are all equivalent.          *
 /* Note 5:  character strings passed must NOT have the VARYING       *
 /*          attribute in PL/I (both SBCS and DBCS).                  *
 /* Note 6:  in COBOL, the reserved word USAGE is optional.           *
 /* Note 7:  in PL/I, the attributes BIN and BINARY are equivalent.   *
 /* Note 8:  in PL/I, the attributes DEC and DECIMAL are equivalent.  *
 /* Note 9:  in PL/I, attributes CHAR and CHARACTER are equivalent.   *
 /*                                                                   *
 /*********************************************************************/
     %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    /**********************************************************/
    /* DECLARE ENTRY FOR THE CALL TO COBOL                    */
    /**********************************************************/
    DCL PL1CBSB EXTERNAL ENTRY(
             /*1*/ FIXED BINARY(15,0),
             /*2*/ FIXED BINARY(31,0),
             /*3*/ FIXED DECIMAL(5,3),
             /*4*/ FLOAT DECIMAL(6),
             /*5*/ FLOAT DECIMAL(16),
             /*6*/ CHARACTER(23),
             /*7*/ GRAPHIC(2) )
          OPTIONS(COBOL);


    /**********************************************************/
    /* Declare parameters:                                    */
    /**********************************************************/
    DCL PLI_INT2    FIXED BINARY(15,0) INIT(15);
    DCL PLI_INT4    FIXED BINARY(31,0) INIT(31);
    DCL PLI_PD53    FIXED DECIMAL(5,3) INIT(-12.345);
    DCL PLI_FLOAT4  FLOAT DECIMAL(6)   INIT(53.99999);
    DCL PLI_FLOAT8  FLOAT DECIMAL(16)  INIT(3.14151617);
    DCL PLI_CHAR23  CHARACTER(23) INIT('PASSED CHARACTER STRING');
    DCL PLI_DBCS    GRAPHIC(2)    INIT('40404040'GX);

    /**********************************************************/
    /*  PROCESS STARTS HERE                                   */
    /**********************************************************/
    PUT SKIP LIST( '*******************************************');
    PUT SKIP LIST( 'PL/I Calling COBOL example is now in motion');
    PUT SKIP LIST( '*******************************************');
    PUT SKIP;
    CALL PL1CBSB( PLI_INT2, PLI_INT4, PLI_PD53,
                  PLI_FLOAT4, PLI_FLOAT8, PLI_CHAR23, PLI_DBCS);
    PUT SKIP LIST( 'PL/I calling COBOL subroutine example ended');

 END PL1CBL;
```

COBOL program called by a PL/I main

```
CBL LIB,QUOTE,NODYNAM
      *************************************************
      *                                               *
      *  IBM Language Environment for MVS & VM    *
      *                                               *
      *  Licensed Materials - Property of IBM         *
      *                                               *
      *  5645-001 5688-198                            *
      *  (C) Copyright IBM Corp. 1991, 1998           *
      *  All Rights Reserved                          *
      *                                               *
      *  US Government Users Restricted Rights - Use, *
      *  duplication or disclosure restricted by GSA  *
      *  ADP Schedule Contract with IBM Corp.         *
      *                                               *
      *************************************************
      *Module/File Name: IGZTPCB
      ****************************************************************
      **  PL1CBSB - COBOL language subroutine invoked by the       ***
      **            PL/I program PL1CBL.                           ***
      **                                                           ***
      ** This is an example of a COBOL subroutine that is called   ***
      ** from a PL/I main program.  See the calling PL/I program   ***
      ** for a table of the PL/I data formats and corresponding    ***
      ** COBOL data formats.  The arguments received are compared  ***
      ** to their expected values, and any discrepancies reported. ***
      **                                                           ***
      ****************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    PL1CBSB.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       77   COBOL-INT2         PIC S9999 BINARY VALUE 15.
       77   COBOL-INT4         PIC S9(9) BINARY VALUE 31.
       77   COBOL-PD53         PIC S9(2)V9(3) COMP-3 VALUE -12.345.
       77   COBOL-FLOAT4       COMP-1 VALUE 53.99999E0.
       77   COBOL-FLOAT8       COMP-2 VALUE 3.14151617E0.
       77   COBOL-CHAR23       PIC X(23) DISPLAY
                                   VALUE "PASSED CHARACTER STRING".
       77   COBOL-DBCS         PIC G(2) DISPLAY-1 VALUE SPACES.
       77   FLOAT8-DIFF        COMP-2.
       LINKAGE SECTION.
       01   INT2-ARG           PIC S9999 BINARY.
       01   INT4-ARG           PIC S9(9) BINARY.
       01   PD53-ARG           PIC S9(2)V9(3) COMP-3.
       01   FLOAT4-ARG         COMP-1.
       01   FLOAT8-ARG         COMP-2.
       01   CHAR23-ARG         PIC X(23) DISPLAY.
       01   DBCS-ARG           PIC G(2) DISPLAY-1.
      **

       PROCEDURE DIVISION USING INT2-ARG, INT4-ARG, PD53-ARG,
                                FLOAT4-ARG, FLOAT8-ARG,
                                CHAR23-ARG, DBCS-ARG.

       0001-ENTRY-FROM-PL1.
           DISPLAY "***************************************".
           DISPLAY "COBOL PROGRAM ENTERED FROM PL/I PROGRAM".
           DISPLAY "***************************************".
      ****************************************************************
      ** Compare passed arguments to initialized values.            **
      ****************************************************************
           IF (INT2-ARG NOT = COBOL-INT2)  THEN
               DISPLAY "Error passing PL/I FIXED BIN(15,0) to COBOL:"
               DISPLAY "Actual argument value is " INT2-ARG
               DISPLAY "Expected        value is " COBOL-INT2
           END-IF.

           IF (INT4-ARG NOT = COBOL-INT4)  THEN
               DISPLAY "Error passing PL/I FIXED BIN(31,0) to COBOL:"
               DISPLAY "Actual argument value is " INT4-ARG
               DISPLAY "Expected        value is " COBOL-INT4
           END-IF.

           IF (PD53-ARG NOT = COBOL-PD53)  THEN
               DISPLAY "Error passing PL/I FIXED DEC(5,3) to COBOL:"
               DISPLAY "Actual argument value is " PD53-ARG
```

```
         DISPLAY "Expected        value is " COBOL-PD53
     END-IF.

    IF (FLOAT4-ARG NOT = COBOL-FLOAT4)  THEN
****************************************************************
*      Calculate absolute difference between short float value *
****************************************************************
         COMPUTE FLOAT8-DIFF = COBOL-FLOAT4 - FLOAT4-ARG
         IF (FLOAT8-DIFF < 0)  THEN
             COMPUTE FLOAT8-DIFF = - FLOAT8-DIFF
         END-IF
         IF (FLOAT8-DIFF > .00001E0)  THEN
             DISPLAY "Error passing PL/I FLOAT DEC(6) to COBOL:"
         ELSE
             DISPLAY "Warning:  slight difference found when "
                     "passing PL/I FLOAT DEC(6) to COBOL:"
         END-IF
         DISPLAY "Actual argument value is " FLOAT4-ARG
         DISPLAY "Expected        value is " COBOL-FLOAT4
     END-IF.

     IF (FLOAT8-ARG NOT = COBOL-FLOAT8) THEN
                 ****************************************************************
*      Calculate absolute difference between long float values *
****************************************************************
         COMPUTE FLOAT8-DIFF = COBOL-FLOAT8 - FLOAT8-ARG
         IF (FLOAT8-DIFF < 0)  THEN
             COMPUTE FLOAT8-DIFF = - FLOAT8-DIFF
         END-IF
         IF (FLOAT8-DIFF > .000000001E0)  THEN
             DISPLAY "Error passing PL/I FLOAT DEC(16) to COBOL:"
         ELSE
             DISPLAY "Warning:  slight difference found when "
                     "passing PL/I FLOAT DEC(16) to COBOL:"
         END-IF
         DISPLAY "Actual argument value is " FLOAT8-ARG
         DISPLAY "Expected        value is " COBOL-FLOAT8
     END-IF.

     IF (CHAR23-ARG NOT = COBOL-CHAR23) THEN
         DISPLAY "Error passing PL/I CHAR(23) to COBOL:"
         DISPLAY "Actual argument value is '" CHAR23-ARG "'"
         DISPLAY "Expected        value is '" COBOL-CHAR23 "'"
     END-IF.

     IF (DBCS-ARG NOT = COBOL-DBCS) THEN
         DISPLAY "Error passing PL/I GRAPHIC(23) to COBOL:"
         DISPLAY "Actual argument value is '" DBCS-ARG "'"
         DISPLAY "Expected        value is '" COBOL-DBCS "'"
     END-IF.

     GOBACK.
```

Chapter 12. Communicating between Fortran and PL/I

This topic describes Language Environment's support for Fortran and PL/I applications.

General facts about Fortran to PL/I ILC

- Fortran object code link-edited with that of another HLL produces a non-reentrant load module, regardless of whether the Fortran routine was compiled with the RENT compiler option. See "Building a reentrant Fortran to PL/I application" on page 209 for more information about reentrancy.
- Language Environment does not support passing return codes between Fortran routines and PL/I routines.
- Fortran routines cannot operate under CICS.
- Support for Fortran on VM is not provided by Language Environment.
- In PL/I multitasking applications, once a Fortran routine is called in a task, no other task can call a Fortran routine until the calling task terminates.

Preparing for ILC

This section describes topics you might want to consider before writing an application that uses ILC between Fortran and PL/I. For help in determining how different versions of HLLs work together, refer to the migration guides for the HLLs you plan to use.

Language Environment ILC support

Language Environment supports ILC between the following combinations of Fortran and PL/I:

Table 55. Supported languages for Language Environment ILC support

HLL pair	Fortran	PL/I
Fortran- to-PL/I static calls	• FORTRAN IV G1 • FORTRAN IV H Extended • VS FORTRAN Version 1, except modules compiled with Release 2.0 or earlier and that either pass character arguments to, or receive character arguments from, subprograms. • VS FORTRAN Version 2, except modules compiled with Releases 5 or 6 and whose source contained any parallel language constructs or parallel callable services, or were compiled with either of the compiler options PARALLEL or EC.	• OS PL/I Version 1 Releases 3.0 through 5.1 • OS PL/I Version 2 • PL/I for MVS & VM • Enterprise PL/I for z/OS
PL/I-to-Fortran dynamic calls	See the preceding list above for Fortran support.	• OS PL/I Version 1 Releases 3.0 through 5.1 • OS PL/I Version 2 • PL/I for MVS & VM • Enterprise PL/I for z/OS

Table 55. Supported languages for Language Environment ILC support (continued)

HLL pair	Fortran	PL/I
Fortran-to-PL/I dynamic calls	VS FORTRAN Version 2 Release 6, with the exceptions listed above	• OS PL/I Version 2, Release 2 or later • PL/I for MVS & VM • Enterprise PL/I for z/OS

Migrating ILC applications

You need to relink Fortran to PL/I ILC applications in order to get Language Environment's ILC support.

Both Fortran and PL/I provide migration tools that replace old library modules with Language Environment ones. For more information about Fortran's library module replacement tool, see *z/OS Language Environment Programming Guide*. For more information about the PL/I migration tool, see the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).

Determining the main routine

In Language Environment, only one routine can be the main routine. The main routine should be presented to the linkage editor first.

A Fortran routine is designated as a main routine with a PROGRAM statement, which indicates the name of the main routine. A Fortran routine can also be designated as a main routine in the absence of the PROGRAM, SUBROUTINE, and FUNCTION statements, in which case the name of the main routine has a default value of `MAIN` (or `MAIN#` for VS FORTRAN Version 2 Releases 5 and 6).

A PL/I routine is designated as a main routine with the OPTIONS(MAIN) option on the PROCEDURE statement.

An entry point is defined for each supported HLL. Table 56 identifies the desired entry point. The table assumes that your code was compiled using the Language Environment-conforming compilers.

Table 56. Determining the entry point

HLL	Main entry point	Fetched entry point
Fortran	Name on the PROGRAM statement or `MAIN` (or `MAIN#` for VS FORTRAN Version 2 Releases 5 and 6).	Name on a SUBROUTINE or FUNCTION statement
PL/I	CEESTART	CEESTART or routine name, if OPTIONS(FETCHABLE) is used.

Declaring Fortran to PL/I ILC

You must specify entry declarations in PL/I source code when a PL/I routine invokes a Fortran routine and when a Fortran routine invokes a PL/I routine. The special declarations cause the PL/I compiler to generate an internal argument list that the Fortran routine accepts. No special declaration is required within the Fortran program.

In the PL/I routine that invokes Fortran, identify the entry point as Fortran in the OPTIONS attribute. The following example illustrates a PL/I routine that identifies a Fortran entry point FORTEP.

```
DCL FORTEP ENTRY OPTIONS(FORTRAN);
```

In the PL/I routine called by Fortran, declare the entry point in the OPTIONS attribute in the PROCEDURE or ENTRY statement. The following illustrates a PL/I routine FORTPLI that will be called by Fortran:

```
FORTPLI: PROCEDURE (parms) OPTIONS(FORTRAN);
```

`parms` specifies the parameters that are passed from the Fortran routine. In previous PL/I compilers, the OPTIONS attribute also needed the INTER keyword to control condition handling between Fortran and PL/I. The INTER keyword is ignored under Language Environment.

Invoking functions

Functions can also be declared as described above. A procedure written in PL/I as a function can be invoked through a function reference in a Fortran routine. Similarly, a routine written in Fortran as a function subprogram can be invoked through a function reference in a PL/I routine.

Character data cannot be used as function return values in functions invoked by either Fortran or PL/I in a Fortran to PL/I ILC application. However, character data can be used as arguments for either functions or subroutines. See "Passing data between Fortran and PL/I" on page 211 for information about character data in Fortran to PL/I ILC.

Building a reentrant Fortran to PL/I application

Fortran object code link-edited with that of another HLL produces a non-reentrant load module, regardless of whether the Fortran routine was compiled with the RENT compiler option. If you need to call a Fortran routine from a reentrant routine written in another language, link-edit the Fortran routine into a separate load module and invoke it with a dynamic call from another language.

Calling between Fortran and PL/I

This section describes the types of calls permitted in Fortran to PL/I ILC applications and considerations when using dynamic call/fetch mechanisms.

Types of calls permitted

Table 57 describes the types of calls between Fortran and PL/I that Language Environment allows:

Table 57. Calls permitted for Fortran and PL/I

ILC direction	Static calls	Dynamic calls	Fetch/Calls
Fortran to PL/I	Yes	Yes	N/A
PL/I to Fortran	Yes	N/A	Yes

See Table 55 on page 207 for exceptions to ILC support.

Dynamic call/fetch considerations

This section describes how to perform dynamic calls in Fortran to PL/I ILC applications.

Fortran Dynamically calling PL/I

For Fortran to dynamically call a PL/I routine, the Fortran routine uses the same name in a CALL statement or function reference as is specified on the DYNAMIC compiler option. In the Fortran routine, the dynamically called PL/I routine (FORTPLI in the following example), would be declared as follows:

```
@PROCESS DYNAMIC(FORTPLI)
```

In the PL/I routine, the Fortran calling routine is declared the same as for a Fortran-to-PL/I static call, as follows:

```
FORTPLI: PROCEDURE (...) OPTIONS(FORTRAN);
```

The dynamically called routine, whether Fortran or PL/I, can statically call another Fortran or PL/I routine.

The dynamically called routine, whether Fortran or PL/I, can dynamically call another Fortran or PL/I routine. However, only two of the dynamically loaded modules can contain PL/I routines (including the load module dynamically loaded by the operating system or subsystem).

There is no Fortran facility to delete a dynamically loaded routine.

PL/I Dynamically calling Fortran

For PL/I to dynamically call a Fortran routine, the PL/I routine must declare the Fortran entry point, such as in the following example:

```
DCL FORTEP ENTRY OPTION(FORTRAN);
```

PL/I could then dynamically call the Fortran routine with the following code:

```
FETCH FORTEP;
CALL FORTEP (...);
```

The dynamically called routine, whether Fortran or PL/I, can statically call another Fortran or PL/I routine. However, the dynamically called Fortran routine cannot contain a PL/I routine in the dynamically loaded module unless there was also a PL/I routine in a previously executed load module.

The dynamically called routine, whether Fortran or PL/I, can dynamically call another Fortran or PL/I routine. However, only two of the dynamically loaded modules can contain PL/I routines (including the load module dynamically loaded by the operating system or subsystem).

You cannot use the PL/I RELEASE statement to release a Fortran program.

Restriction: When a PL/I routine fetches a Fortran routine, the dynamically loaded module can contain only routines written in those languages that already exist in a previous load module. (The routine in the previous load module need not be called; it only needs to be present.) For a Fortran routine to be recognized, ensure that at least one of the following is present in a previous load module:

- A Fortran main program
- A Fortran routine that causes one or more Fortran runtime library routines to be link-edited into the load module. If the Fortran routine contains either an I/O statement, a mathematical function reference, or a call to any Fortran callable service (such as CPUTIME), then a library routine is included, and this requirement is satisfied.
- The Fortran signature CSECT, CEESG007. Use the following linkage editor statement to include CEESG007 if neither of the two previous conditions is true:

  ```
  INCLUDE SYSLIB(CEESG007)
  ```

Passing data between Fortran and PL/I

This section describes the data types that are compatible between Fortran and PL/I. It also includes information about how to map aggregates across the two languages.

Supported data types between Fortran and PL/I

The data types supported between Fortran and PL/I are listed in Table 58.

Table 58. Supported data types between Fortran and PL/I

Fortran	PL/I
INTEGER*2	REAL FIXED BINARY(15,0)
INTEGER*4	REAL FIXED BINARY(31,0)
REAL*4	REAL FLOAT DEC(6,0)
REAL*8	REAL FLOAT DEC(16,0)
REAL*16	REAL FLOAT DEC(33,0)
COMPLEX*8	COMPLEX FLOAT DEC(6,0)
COMPLEX*16	COMPLEX FLOAT DEC(16,0)
COMPLEX*32	COMPLEX FLOAT DEC(33,0)
CHARACTER**n*	CHARACTER(*n*)
POINTER	POINTER

Passing character data

Character data can be received by a Fortran or PL/I routine only when the routine that receives the data declares the data as fixed length. Therefore, the Fortran form CHARACTER*(*) and the PL/I form CHARACTER(*) cannot be used to receive character data. The VARYING attribute cannot be specified in the PL/I declaration of character data.

Using aggregates

An array can be passed between Fortran and PL/I routines only when the array has its elements in contiguous storage locations and when the called routine specifies a constant number of elements along each dimension. In Fortran, arrays of more than one dimension are arranged in storage in column major order; in PL/I they are in row major order. Unless you specifically override remapping with the PL/I attributes NOMAP, NOMAPIN, or NOMAPOUT (for overriding remapping in both called and calling Fortran routines), a temporary remapped array is created for the called routine. When an array is remapped, an element can be referenced in both Fortran and PL/I with subscripts in the same order.

Structures are not supported in Fortran to PL/I ILC.

Data equivalents

This section describes how Fortran and PL/I data types correspond to each other.

Equivalent data types for Fortran to PL/I

The following examples illustrate how PL/I and Fortran routines within a single ILC application might code the same data types.

16-bit signed binary integer

Sample Fortran usage

```
INTEGER*2 X, Y, F2PP16I
X = 5
Y = F2PP16I(X)
PRINT *,
1 'VALUE RETURNED TO FORTRAN:', Y
END
```

PL/I function

```
F2PP16I: PROC (X) OPTIONS(FORTRAN)
      RETURNS(FIXED BIN(15));
  DCL X       FIXED BIN(15);
  PUT SKIP
      LIST('PL/I ARG VALUE:', X);
  RETURN (X);
END F2PP16I;
```

32-bit signed binary integer

Sample Fortran usage

```
INTEGER*4 X, Y, F2PP32I
X = 5
Y = F2PP32I(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

PL/I Function

```
F2PP32I: PROC (X) OPTIONS(FORTRAN)
      RETURNS(FIXED BIN(31));
  DCL X       FIXED BIN(31);
  PUT SKIP
      LIST('PL/I ARG VALUE:', X);
  RETURN (X);
END F2PP32I;
```

Short floating-point number

Sample Fortran usage

```
REAL*4 X, Y, F2PPSFP
X = 5.0
Y = F2PPSFP(X)
PRINT *,
1 'VALUE RETURNED TO FORTRAN:', Y
END
```

PL/I Function

```
F2PPSFP: PROC (X) OPTIONS(FORTRAN)
      RETURNS(FLOAT DEC(6));
  DCL X       FLOAT DEC(6);
  PUT SKIP
      LIST('PL/I ARG VALUE:', X);
  RETURN (X);
END F2PPSFP;
```

Long floating-point number

Sample Fortran usage

```
REAL*8 X, Y, F2PPLFP
X = 12.5D0
Y = F2PPLFP(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

PL/I Function

```
F2PPLFP: PROC (X) OPTIONS(FORTRAN)
      RETURNS(FLOAT DEC(16));
  DCL X       FLOAT DEC(16);
  PUT SKIP
      LIST('PL/I ARG VALUE:', X);
  RETURN (X);
END F2PPLFP;
```

Extended floating-point number

Sample Fortran usage

```
REAL*16 X, Y, F2PPEFP
X = 12.1Q0
Y = F2PPEFP(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

PL/I function

```
F2PPEFP: PROC (X) OPTIONS(FORTRAN)
      RETURNS(FLOAT DEC(33));
  DCL X       FLOAT DEC(33);
  PUT SKIP
      LIST('PL/I ARG VALUE:', X);
  RETURN (X);
END F2PPEFP;
```

Complex: two adjacent short floating-point numbers

Sample Fortran usage

```
COMPLEX*8 X, Y, F2PPSCP
X = (5.0, 15.0)
Y = F2PPSCP(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

PL/I function

```
F2PPSCP: PROC (X) OPTIONS(FORTRAN)
      RETURNS(COMPLEX FLOAT DEC(6));
  DCL X       COMPLEX FLOAT DEC(6);
  PUT SKIP
      LIST('PL/I ARG VALUE:', X);
  RETURN (X);
END F2PPSCP;
```

Complex: two adjacent long floating-point numbers

Sample Fortran usage

```
COMPLEX*16 X, Y, F2PPLCP
X = (5.0D0, 15.0D0)
Y = F2PPLCP(X)
PRINT *,
1 'VALUE RETURNED TO FORTRAN:', Y
END
```

PL/I function

```
F2PPLCP: PROC (X) OPTIONS(FORTRAN)
      RETURNS(COMPLEX FLOAT DEC(16));
  DCL X       COMPLEX FLOAT DEC(16);
  PUT SKIP
      LIST('PL/I ARG VALUE:', X);
  RETURN (X);
END F2PPLCP;
```

Complex: two adjacent extended floating-point numbers

Sample Fortran usage

```
COMPLEX*32 X, Y, F2PPECP
X = (5.0Q0, 15.0Q0)
Y = F2PPECP(X)
PRINT *,
1  'VALUE RETURNED TO FORTRAN:', Y
END
```

PL/I function

```
F2PPECP: PROC (X) OPTIONS(FORTRAN)
      RETURNS(COMPLEX FLOAT DEC(33));
  DCL X       COMPLEX FLOAT DEC(33);
  PUT SKIP
      LIST('PL/I ARG VALUE:', X);
  RETURN (X);
END F2PPECP;
```

Fixed-length character data

Sample Fortran usage

```
X = '1234567890'
CALL F2PPFLC(X, Y)
1  'VALUE RETURNED TO FORTRAN: ', Y
END
```

PL/I subroutine

```
F2PPFLC: PROC (X, Y) OPTIONS(FORTRAN);
  DCL X       CHARACTER(10);
  DCL Y       CHARACTER(10);
  PUT SKIP
      LIST('PL/I ARG VALUE:', X);
  Y = X;
  RETURN;
END F2PPFLC;
```

Array

Sample Fortran usage

```
REAL*4 MATRIX(3) / 1.0, 2.0, 3.0 /
CALL F2PPAF(MATRIX)
PRINT *,
1  'UPDATED VALUES IN FORTRAN:', MATRIX
END
```

PL/I subroutine

```
F2PPAF: PROC (X) OPTIONS(FORTRAN);
  DCL X(3)    FLOAT DEC(6);
  DCL IX      FIXED BIN(31);
  PUT SKIP
      LIST('PL/I ARG VALUE:', X);
  DO IX = 1  TO 3  BY 1;
      X(IX) = X(IX) - 1.0E0;
  END;
  RETURN;
END F2PPAF;
```

Address of an array

Sample Fortran usage

```
POINTER*4 (P, I)
INTEGER*4 I(3)
INTEGER*4 J(3) / 1, 2, 3 /
P = LOC(J)
CALL F2PPAOA (P)
PRINT *,
1  'UPDATED VALUES IN FORTRAN:', I
END
```

PL/I subroutine

```
F2PPAOA: PROC (X) OPTIONS(FORTRAN);
  DCL X       POINTER;
  DCL Y(3)    FIXED BIN(31)  BASED(X);
  PUT SKIP
      LIST('PL/I ARG VALUES:', Y);
  DO IX = 1  TO 3  BY 1;
      Y(IX) = Y(IX) - 1;
  END;
  RETURN;
END F2PPAOA;
```

Equivalent data types for PL/I to Fortran

The following examples illustrate how PL/I and Fortran routines within a single ILC application might code the same data types.

16-bit signed binary integer

Sample PL/I usage

```
P2FP16I: PROC OPTIONS(MAIN);
  DCL PLFF16I ENTRY OPTIONS(FORTRAN)
      RETURNS(FIXED BIN(15));
  DCL X       FIXED BIN(15);
  DCL Y       FIXED BIN(15);
  X = 5;
  Y = P2FF16I(X);
  PUT SKIP
      LIST('Value Returned to PL/I:', Y);
END P2FP16I;
```

Fortran Function

```
FUNCTION P2FF16I ( ARG )
P2FF16I = ARG
END
```

32-bit signed binary integer

Sample PL/I usage

```
P2FP32I: PROC OPTIONS(MAIN);
  DCL P2FF32I ENTRY OPTIONS(FORTRAN)
      RETURNS(FIXED BIN(31));
  DCL X       FIXED BIN(31);
  DCL Y       FIXED BIN(31);
  X = 5;
  Y = P2FF32I(X);
  PUT SKIP
      LIST('VALUE RETURNED TO PL/I:', Y);
END P2FP32I;
```

Fortran function

```
FUNCTION P2FF32I ( ARG )
INTEGER*4 P2FF32I
INTEGER*4 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
P2FF32I = ARG
END
```

Short floating-point number

Sample PL/I usage

```
P2FPSFP: PROC OPTIONS(MAIN);
  DCL P2FFSFP ENTRY OPTIONS(FORTRAN)
      RETURNS(FLOAT DEC(6));
  DCL X       FLOAT DEC(6);
  DCL Y       FLOAT DEC(6);
  X = 5.0E0;
  Y = P2FFSFP(X);
  PUT SKIP
      LIST('VALUE RETURNED TO PL/I:', Y);
END P2FPSFP;
```

Fortran Function

```
FUNCTION P2FFSFP ( ARG )
REAL*4 P2FFSFP
REAL*4 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
P2FFSFP = ARG
END
```

Long floating-point number

Sample PL/I usage

```
P2FPLFP: PROC OPTIONS(MAIN);
  DCL P2FFLFP ENTRY OPTIONS(FORTRAN)
      RETURNS(FLOAT DEC(16));
  DCL X       FLOAT DEC(16);
  DCL Y       FLOAT DEC(16);
  X = 5.000000000000000E0;
  Y = P2FFLFP(X);
  PUT SKIP
      LIST('VALUE RETURNED TO PL/I:', Y);
END P2FPLFP;
```

Fortran Function

```
FUNCTION P2FFLFP ( ARG )
REAL*8 P2FFLFP
REAL*8 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
P2FFLFP = ARG
END
```

Extended floating-point number

Sample PL/I usage

```
P2FPEFP: PROC OPTIONS(MAIN);
  DCL P2FFEFP ENTRY OPTIONS(FORTRAN)
      RETURNS(FLOAT DEC(33));
  DCL X       FLOAT DEC(33);
  DCL Y       FLOAT DEC(33);
  X = 5.00000000000000000000000E0;
  Y = P2FFEFP(X);
  PUT SKIP
      LIST('VALUE RETURNED TO PL/I:', Y);
END P2FPEFP;
```

Fortran Function

```
FUNCTION P2FFEFP ( ARG )
REAL*16 P2FFEFP
REAL*16 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
P2FFEFP = ARG
END
```

Complex: two adjacent short floating-point numbers

Sample PL/I usage

```
P2FPSCP: PROC OPTIONS(MAIN);
  DCL P2FFSCP ENTRY OPTIONS(FORTRAN)
      RETURNS(COMPLEX FLOAT DEC(6));
  DCL X       COMPLEX FLOAT DEC(6);
  DCL Y       COMPLEX FLOAT DEC(6);
  X = 5.0E0 + 15.0E0I;
  Y = P2FFSCP(X);
  PUT SKIP
      LIST('VALUE RETURNED TO PL/I:', Y);
END P2FPSCP;
```

Fortran Function

```
FUNCTION P2FFSCP ( ARG )
COMPLEX*8 P2FFSCP
COMPLEX*8 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
P2FFSCP = ARG
END
```

Complex: two adjacent long floating-point numbers

Sample PL/I usage

```
P2FPLCP: PROC OPTIONS(MAIN);
  DCL P2FFLCP ENTRY OPTIONS(FORTRAN)
      RETURNS(COMPLEX FLOAT DEC(16));
  DCL X       COMPLEX FLOAT DEC(16);
  DCL Y       COMPLEX FLOAT DEC(16);
  X = 5.000000000000000E0
      + 15.00000000000000E0I;
  Y = P2FFLCP(X);
  PUT SKIP
      LIST('VALUE RETURNED TO PL/I:', Y);
END P2FPLCP;
```

Fortran Function

```
FUNCTION P2FFLCP ( ARG )
COMPLEX*16 P2FFLCP
COMPLEX*16 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
P2FFLCP = ARG
END
```

Complex: two adjacent extended floating-point numbers

Sample PL/I usage

```
P2FPECP: PROC OPTIONS(MAIN);
  DCL P2FFECP ENTRY OPTIONS(FORTRAN)
      RETURNS(COMPLEX FLOAT DEC(33));
  DCL X       COMPLEX FLOAT DEC(33);
  DCL Y       COMPLEX FLOAT DEC(33);
  X = 5.0000000000000000000E0
      + 15.000000000000000000E0I;
  Y = P2FFECP(X);
  PUT SKIP
      LIST('VALUE RETURNED TO PL/I:', Y);
END P2FPECP;
```

Fortran Function

```
FUNCTION P2FFECP ( ARG )
COMPLEX*32 P2FFECP
COMPLEX*32 ARG
PRINT *, 'FORTRAN ARG VALUE:', ARG
P2FFECP = ARG
END
```

Fixed-length character data

Sample PL/I usage

```
P2FPFLC: PROC OPTIONS(MAIN);
  DCL P2FFFLC ENTRY OPTIONS(FORTRAN);
  DCL X       CHARACTER(10);
  DCL Y       CHARACTER(10);
  X = '1234567890';
  CALL P2FFFLC(X, Y);
  PUT SKIP
      LIST('VALUE RETURNED TO PL/I:', Y);
END P2FPFLC;
```

Fortran subroutine

```
SUBROUTINE P2FFFLC ( ARG1, ARG2 )
CHARACTER*10 ARG1, ARG2
PRINT *, 'FORTRAN ARG1 VALUE: ', ARG1
ARG2 = ARG1
END
```

Array

Sample PL/I usage

```
P2FPAF: PROC OPTIONS(MAIN);
  DCL P2FFAF ENTRY OPTIONS(FORTRAN);
  DCL X(3)    FLOAT DEC(6);
  X(1) = 0E0;
  X(2) = 1E0;
  X(3) = 2E0;
  CALL P2FFAF(X);
  PUT SKIP
      LIST('UPDATED VALUES IN PL/I:', X);
END P2FPAF;
```

Fortran subroutine

```
SUBROUTINE P2FFAF ( ARG )
REAL*4 ARG(3)
PRINT *, 'FORTRAN ARG VALUES:', ARG
DO J = 1, 3
  ARG(J) = ARG(J) + 1.0
ENDDO
END
```

Address of an array

Sample PL/I usage

```
P2FPAOA: PROC OPTIONS(MAIN);
  DCL P2FFAOA ENTRY OPTIONS(FORTRAN);
  DCL X(3)    FIXED BIN(31);
  DCL Y(3)    FIXED BIN(31)  BASED(P);
  P = ADDR(X);
  Y(1) = 0;
  Y(2) = 1;
  Y(3) = 2;
  CALL P2FFAOA(P);
  PUT SKIP
      LIST('UPDATED VALUES IN PL/I:', Y);
END P2FPAOA;
```

Fortran subroutine

```
SUBROUTINE P2FFAOA ( ARG )
POINTER*4 (ARG, Y)
INTEGER*4 Y(3)
PRINT *,
1  'FORTRAN ARRAY ARG VALUES:', Y
DO J = 1, 3
Y(J) = Y(J) + 1
ENDDO
END
```

External data

Fortran static common blocks and PL/I static external data of the same name can be shared among routines in a load module under the following conditions:

- The Fortran static common blocks are either used only with one load module in an application or they are declared as private common blocks. A private common block is not shared across load modules and is created in any of the following ways:
  - Specified or implied by the PC compiler option
  - Referenced by Fortran object code produced by the VS FORTRAN Version 2 Release 4 compiler or earlier
  - In an application that executes with the PC runtime option.

Directing output from ILC applications

Fortran runtime messages, output written to the print unit, and other output (such as output from the SDUMP callable service) are directed to the file specified by the MSGFILE runtime option. To direct this output to the file with the ddname FT*nn*F001 (where *nn* is the two-digit error message unit number), specify the runtime option MSGFILE(FT*nn*F001). If the print unit is different from the error message unit (if the PRTUNIT and the ERRUNIT runtime options have different values), output from a PRINT statement will not be directed to the Language Environment message file.

PL/I runtime messages and other related output (such as ON condition SNAP output) are directed to the file specified in the MSGFILE runtime option instead of to the PL/I SYSPRINT STREAM PRINT file. User-written output is still directed to the PL/I SYSPRINT STREAM PRINT file; to direct user-written output to the Language Environment message file, specify the MSGFILE(SYSPRINT) runtime option.

Running Fortran routines in the PL/I multitasking facility

This section describes different considerations and restrictions of running Fortran in the PL/I Multitasking Facility.

In a PL/I multitasking ILC application, Fortran routines can be executed in either the main task or in a subtask. If a Fortran routine was invoked in a task (either main task or subtask), no other Fortran routine can execute in any other task until the task used to invoke the Fortran routine ends. Therefore, in a multitasking environment, you cannot invoke a Fortran routine in a subtask if a Fortran routine has already been invoked in the main task.

Reentrancy in a multitasking application

You need to be careful about calling the routine in a different enclave or in a PL/I subtask, because any load module containing a Fortran routine is non-reentrant. The following restrictions apply:

- Once a Fortran routine in a load module has been invoked, no Fortran routine in the same copy of the load module can be invoked from a different task, even after the first task ends.
- If there are Fortran routines in a load module, then no Fortran routine in the same copy of that load module can be used in an enclave other than in the first enclave in which a Fortran routine is called.

Common blocks in a PL/I multitasking application

In a PL/I multitasking application, Fortran dynamic common blocks are always maintained for the whole enclave, even though subtasks that use them can start and end.

Fortran static common blocks persist only as long as the load module that contains them is in storage. When a load module is loaded from within a subtask, Fortran common blocks persist only within that subtask because the load module is deleted when the task ends. If, in a later subtask, the same load module is loaded, a fresh copy is loaded with fresh contents of the common block.

If, however, the load module is loaded in the main task (with the FETCH statement), and is also specified in a CALL statement to run a subtask, the load module (and the Fortran common blocks) is retained in storage after the subtask ends. The same copy of the load module can be used in another CALL statement to create another subtask, with the same contents of the common block. The following code illustrates this method:

```
FETCH ABC;
CALL ABC (...) TASK;
```

Data-in-virtual data objects in PL/I multitasking applications

You can use data objects that are accessed by the data-in-virtual (DIV) callable services in different subtasks. If you want to use the same data object or a different data object in different subtasks, you need to establish their use in the DIVINF or DIVINV callable services. Otherwise, at the end of each subtask, the data object is terminated.

Files and print units in a multitasking application

Fortran routines running in either the main task or a subtask can open files. However, when the task terminates, all the open files except the Language Environment message file are automatically closed.

If the standard print unit is the same unit number as the error message unit, it is in effect the Language Environment message file, and will stay open when the task terminates. If the standard print unit is different from the error message unit, the print unit will close, along with other Fortran files, at task termination. The unit numbers are set with the PRTUNIT and ERRUNIT runtime options.

Fortran to PL/I condition handling

This section offers two scenarios of condition handling behavior in a Fortran to PL/I ILC application. If an exception occurs in a Fortran program, the set of possible actions is as described in “Exception occurs in Fortran” on page 219. If an exception occurs in a PL/I routine, the set of possible actions is as described in “Exception occurs in PL/I” on page 221.

Some conditions can be handled only by the HLL of the routine in which the exception occurred. For example, when ERR and IOSTAT specifiers are present on a Fortran I/O statement and an error is detected while executing that statement, Fortran condition handling semantics take precedence over Language Environment condition handling. Control returns immediately to the Fortran program and no condition is signaled to Language Environment.

If there is a PL/I routine currently active on the stack, PL/I language semantics can be applied to handle conditions that occur in non-PL/I routines within an ILC application. For example, PL/I semantics apply to Language Environment hardware conditions that map directly to PL/I conditions such as ZERODIVIDE, even if they occur in a non-PL/I routine. Also, PL/I treats any unknown condition of severity 2 or greater as the ERROR condition. In a case in which a Fortran-specific condition of severity 2 or greater is passed to a PL/I stack frame, an ERROR ON-unit can handle it on the first pass of the stack.

PL/I Multitasking ILC considerations

User-written condition handlers registered with the CEEHDLR callable service are not supported in PL/I multitasking applications.

When a Fortran enclave-terminating construct, such as a STOP statement, is executed from a Fortran routine in a PL/I subtask, the entire enclave is terminated.

See *z/OS Language Environment Programming Guide* for a detailed description of Language Environment condition handling. For information about Fortran condition handling semantics, see *VS FORTRAN Version 2 Programming Guide for CMS and MVS.*

Enclave-terminating language constructs

Enclaves can be terminated due to reasons other than an unhandled condition of severity 2 or greater. In Language Environment ILC, you can issue an HLL language construct to terminate a Fortran to PL/I enclave from either a Fortran or PL/I routine.

Fortran

The Fortran language constructs that cause the enclave to terminate are:

- A STOP statement
- An END statement in the main routine
- A call to EXIT or SYSRCX
- A call to DUMP or CDUMP

PL/I

The PL/I language constructs that cause the enclave to terminate are:

- A STOP or EXIT statement

  If you code a STOP or EXIT statement, the T_I_S (Termination Imminent Due to STOP) condition is raised.
- An END or RETURN statement in the main routine
- A call to PLIDUMP with the S or E option

  If you call PLIDUMP with the S or E option, neither termination imminent condition is raised before the enclave is terminated. See *z/OS Language Environment Debugging Guide* for syntax of the PLIDUMP service.

Exception occurs in Fortran

This scenario describes the behavior of an application that contains a Fortran and a PL/I routine. Refer to Figure 44 on page 220 throughout the following discussion. In this scenario, a PL/I main routine invokes a Fortran subroutine. An exception occurs in the Fortran subroutine.

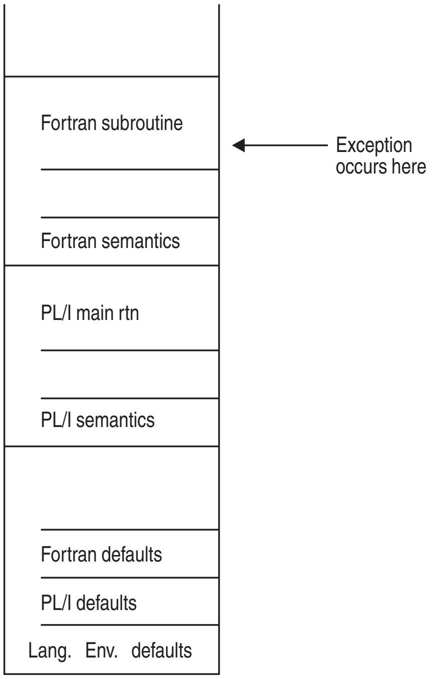


Figure 44. Stack contents when the exception occurs in Fortran

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. If an I/O error is detected on a Fortran I/O statement that contains an ERR or IOSTAT specifier, Fortran semantics take precedence. The exception is not signaled to the Language Environment condition handler.
2. In the enablement step, Fortran treats all exceptions as conditions. Processing continues with the condition handling step.
3. There is no user-written condition handler on the Fortran stack frame (because CEEHDLR cannot be called from a Fortran routine), and the condition is percolated.
4. If a user-written condition handler has been registered on the PL/I stack frame using CEEHDLR, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine at the point where the resume cursor points.

   In this example, no user-written condition handler is registered for the condition, so the condition is percolated.
5. If an ON-unit has been established for the condition being processed on the PL/I stack frame, it is given control. If it issues a GOTO out-of-block, the condition handling step ends. Execution resumes at the label of the GOTO. In this example, no ON-unit is established for the condition, so the condition is percolated.
6. What happens next depends on whether the condition is promotable to the PL/I ERROR condition. The following can happen:
   - If the condition is not promotable to the PL/I ERROR condition, then the Language Environment default actions take place, as described in Table 62 on page 249. Condition handling ends.
   - If the PL/I default action for the condition is to promote it to the PL/I ERROR condition, The condition is promoted, and another pass of the stack is made to look for ERROR ON-units or user-written condition handlers. If an ERROR ON-unit or user-written condition handler is found, it is invoked.

- If either of the following occurs:
  - An ERROR ON-unit or user-written condition handler is found, but it does not issue a GOTO out of block or similar construct
  - No ERROR ON-unit or user-written condition handler is found

  then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled condition). Condition handling now enters the termination imminent step. Because T_I_U maps to the PL/I FINISH condition, both FINISH ON-units and user-written condition handlers can be run if the stack frames in which they are established is reached.
- If no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

Exception occurs in PL/I

This scenario describes the behavior of an application that contains a PL/I and a Fortran routine. Refer to Figure 45 throughout the following discussion. In this scenario, a Fortran main routine invokes a PL/I subroutine. An exception occurs in the PL/I subroutine.

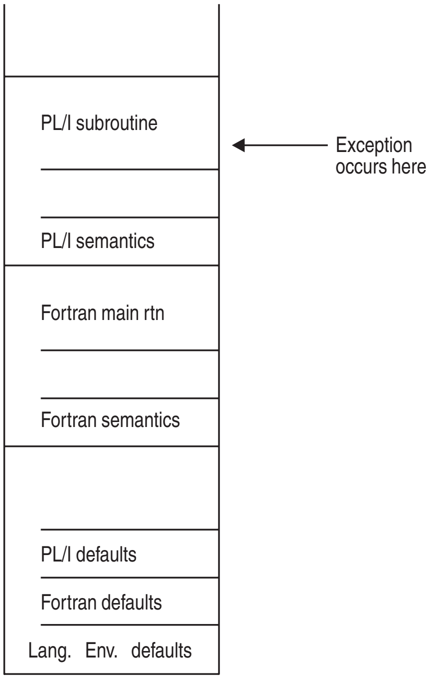


Figure 45. Stack contents when the exception occurs in PL/I

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, PL/I determines if the exception that occurred should be handled as a condition according to the PL/I rules of enablement.
   - If the exception is to be ignored, control is returned to the next sequential instruction after where the exception occurred.
   - If the exception is to be enabled and processed as a condition, the condition handling step takes place.
2. If a user-written condition handler has been registered on the PL/I stack frame using CEEHDLR, it is given control. If it issues a resume, the condition handling step ends. Processing continues in the routine at the point where the

resume cursor points. In this example, no user-written condition handler is registered for the condition, so the condition is percolated.

3. If an ON-unit has been established for the condition being processed on the PL/I stack frame, it is given control. If it issues a GOTO out-of-block, the condition handling step ends. Execution resumes at the label of the GOTO. In this example, no ON-unit is established for the condition, so the condition is percolated.
4. This is no user-written condition handler on the Fortran stack frame (because CEEHDLR cannot be called from a Fortran routine), and the condition is percolated.
5. What happens next depends on whether the condition is promotable to the PL/I ERROR condition. The following can happen:
   - If the condition is not promotable to the PL/I ERROR condition, then the Language Environment default actions take place, as described in Table 62 on page 249. Condition handling ends.
   - If the PL/I default action for the condition is to promote it to the PL/I ERROR condition, The condition is promoted, and another pass is made of the stack to look for ERROR ON-units or user-written condition handlers. If an ERROR ON-unit or user-written condition handler is found, it is invoked.
   - If either of the following occurs:
     - An ERROR ON-unit or user-written condition handler is found, but it does not issue a GOTO out of block or similar construct
     - No ERROR ON-unit or user-written condition handler is found

     then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled condition). Condition handling now enters the termination imminent step. Because T_I_U maps to the PL/I FINISH condition, both FINISH ON-units and user-written condition handlers can be run if the stack frames in which they are established are reached.
   - If no condition handler moves the resume cursor and issued a resume, Language Environment terminates the thread.

Sample ILC applications

```
@PROCESS LIST
      PROGRAM FOR2PLI

*    Module/File Name:  AFHPLFOR */
****************************************************
*     Illustration of Interlanguage Communication  *
*     between Fortran and PL/I.                    *
****************************************************
*
      INTEGER*2    INT_2   / 2 /
      INTEGER*4    INT_4   / 4 /
      REAL*8       PI      / 3.14159265358979312D0 /
      CHARACTER*23 CHAR_23 / ' ' /

      PRINT *, 'FOR2PLI STARTED'

      CALL PLIFFOR (INT_2, INT_4, PI, CHAR_23)
      IF (CHAR_23 /=  'PASSED CHARACTER STRING') THEN
         PRINT *, 'CHAR_23 NOT SET PROPERLY'
      ENDIF

      PRINT *, 'FOR2PLI ENDED'
      END
```

Figure 46. Fortran routine that calls a PL/I routine

```
 PLIFFOR: PROCEDURE (INT_2, INT_4, REAL_8, CHAR_STRING)
                         OPTIONS(FORTRAN);
 /* Module/File Name:   IBMPLFOR                   */


  DCL ABS          BUILTIN;
  DCL INT_2        FIXED BIN(15,0);
  DCL INT_4        FIXED BIN(31,0);
  DCL REAL_8       FLOAT BIN(53);
  DCL CHAR_STRING  CHAR(23);

  DCL PI           FLOAT BIN(53)
                    INIT (3.141592653589793E0);


  PUT SKIP LIST ('PLIFFOR STARTED');

  IF (INT_2 ¬= 2) THEN
     PUT SKIP LIST ('INT_2 NOT 2');

  IF (INT_4 ¬= 4) THEN
     PUT SKIP LIST ('INT_4 NOT 4');

  IF (ABS(REAL_8 - PI) > 1.0E13) THEN
     PUT SKIP LIST ('REAL_8 NOT PI');

  CHAR_STRING = 'PASSED CHARACTER STRING';

  PUT SKIP LIST ('PLIFFOR ENDED');

 END PLIFFOR;
```

Figure 47. PL/I routine called by Fortran

Chapter 13. Communicating between multiple HLLs

This section describes considerations for writing ILC applications comprised of three or more languages. One approach to writing an *n-way* ILC application is to treat it as several pairwise ILC groupings within a single application. For any call between routines written in two different HLLs, you must, at a minimum, adhere to the restrictions described for that pair, as documented in the pairwise ILC descriptions.

The considerations in this section apply to any combination of supported languages within an ILC application. These common considerations are summarized here to provide a convenient overview of writing an *n*-way ILC application. For specific details, refer to the appropriate pairwise considerations described previously.

If you are running any ILC application under CICS, you should also consult Chapter 15, "ILC under CICS," on page 241.

Supported data types across HLLs

Table 59 lists those data types that are common across all supported HLLs when passed without using a pointer. There are, in addition to those listed in this table, additional data types supported across specific ILC pairs; these are listed in the applicable pairwise ILC descriptions.

Table 59. Data types common to all supported HLLs

C	C++	COBOL	PL/I
signed long int	signed long int	PIC S9(9) USAGE IS BINARY	Real Fixed Bin(31,0)
double	double	COMP-2	Real Float Dec(16)

External data

The following list describes how external data maps across the languages, as well as how mapping is restricted:

C to C++
External variables map if: 1) C has constructed reentrancy, or 2) C++ uses `#pragma variable(...,norent)` to make the specific variable non-reentrant.

C to COBOL
C and COBOL static external variables do not map to each other.

C to PL/I
If C is non-reentrant, then C and PL/I static external variables map by name. If the C routine has constructed reentrancy, the C and PL/I static external variables will map if the C routine uses `#pragma variable(...,norent)` to make the specific variable non-reentrant.

C++ to COBOL
C++ and COBOL static external variables do not map to each other.

C++ to PL/I
External variables map if C++ uses `#pragma variable(...,norent)` to make specific variable non-reentrant.

COBOL to PL/I

COBOL and PL/I static external variables do not map to each other.

Thread management

POSIX-conforming C/C++ applications can communicate with Enterprise COBOL for z/OS programs compiled THREAD on any thread.

POSIX-conforming C/C++ applications can communicate with the following COBOL programs on only one thread:

- Enterprise COBOL for z/OS (compiled NOTHREAD)
- COBOL for OS/390 & VM
- COBOL for MVS & VM
- COBOL/370
- VS COBOL II

POSIX-conforming C/370 applications can communicate with assembler routines on any thread when the assembler routines use the CEEENTRY/CEETERM macros or the EDCPRLG/EDCEPIL macros provided by C/C++.

OS/390-conforming C applications can communicate with PL/I on any thread created by C routines. PL/I routines, however, must follow the rules described in *z/OS Language Environment Programming Guide*.

If an asynchronous signal is being delivered to a thread, the thread is interrupted for the signal only when the execution is:

- In a user C routine,
- Just prior to a return to a C routine
- In an Enterprise COBOL for z/OS program compiled with the THREAD option
- Just prior to return to Enterprise COBOL for z/OS program compiled with the THREAD option, or
- Just prior to an invocation of a Language Environment library from a user routine

C routines or COBOL routines compiled with the THREAD compiler option may need to protect against asynchronous signals based on the application logic including the possible use of the POSIX signal-blocking function that is available. For all other routines, it does not have to be concerned about being interrupted during its execution for an asynchronous signal.

Condition handling

This section describes what happens during Language Environment condition handling and enclave termination.

C++ exception handling constructs `try/throw/catch` cannot be used with Language Environment and HLL condition handling. If you use C exception handling constructs (`signal/raise`) in your C++ routine, condition handling will proceed as described in this section and in the other C chapters. Otherwise, you will get undefined behavior in your programs if you mix the C constructs with the C++ constructs.

See *z/OS Language Environment Programming Guide* for a detailed description of Language Environment condition handling.

Enclave-terminating constructs

Enclave termination can occur due to reasons other than an unhandled condition of severity 2, 3, or 4. These include:

- A language STOP-like construct such as a C `abort()`, `raise(SIGABRT)`, `exit()` function call, COBOL STOP RUN, or PL/ISTOP or EXIT statement.

  When one of these statements is encountered, the T_I_S (Termination Imminent Due to STOP) condition is signaled.
- A return from the main routine
- A Language Environment-initiated abend
- A user-requested abend (call to CEE3ABD or ILBOABN0)

  You can call CEE3ABD to request an abend either with or without clean-up. If the abend is issued without clean-up, T_I_U (Termination Imminent due to an Unhandled condition) is not raised. See *z/OS Language Environment Programming Reference* for more information about the CEE3ABD callable service.

  If you call CEE3ABD and request an abend with clean-up, or you call ILBOABN0 from a COBOL program, T_I_U is signaled. Condition handlers are given a chance to handle the abend. If the abend remains unhandled, normal Language Environment termination activities occur. For example, the C `atexit` list is honored if a C routine is present on the stack, and the Language Environmentassembler user exit gains control.

C, COBOL, and PL/I scenario: exception occurs in C

This scenario describes the behavior of an application that contains C, COBOL, and PL/I. Refer to Figure 48 on page 228 throughout the following discussion.

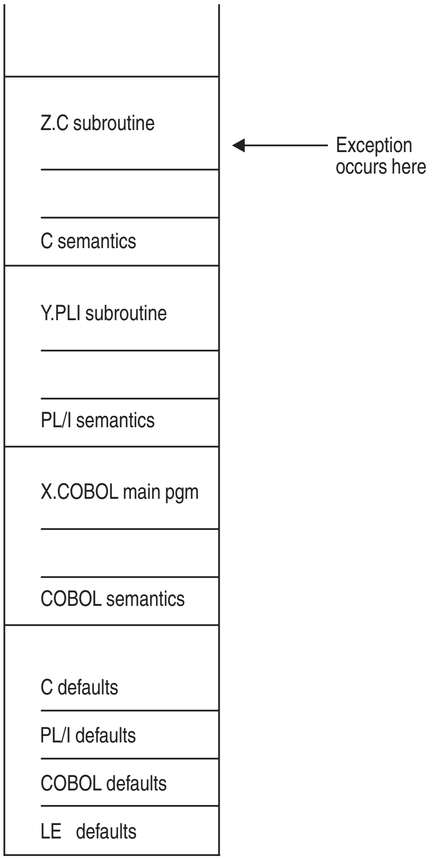


Figure 48. Stack contents when the exception occurs in C

In this example, X.COBOL invokes Y.PLI, which invokes Z.C. A condition is raised in Z.C. The stack contains what is shown in Figure 48. No user-written condition handlers have been registered using CEEHDLR for any stack frame, and no PL/I ON-units have been established.

The actions taken follow the three Language Environment condition handling steps: enablement, condition, and termination imminent.

1. In the enablement step, it is determined whether the exception in the C routine should be enabled and treated as a condition. If any of the following are true, the exception is ignored, and processing continues at the next sequential instruction after where the exception occurred:
   - You specified `SIG_IGN` for the exception in a call to `signal()`.

     However, the system or user abend represented by the Language Environment message 3250 and the `signal(SIGABND)` is not ignored. The enclave is terminated.
   - The exception is one of those listed as masked in Table 63 on page 249.
   - You do not specify any action, but the default action for the condition is `SIG_IGN` (see Table 63 on page 249).
   - You are running under CICS and a CICS handler is pending.

   If you do none of these things, the condition is enabled and processed as a condition.

2. If a user-written condition handler is registered using CEEHDLR on the Z.C stack frame, it receives control. If it issues a resume, the condition handling step ends. Processing continues in the routine to which the resume cursor points.

   In this example, there is no user-written condition handler registered, so the condition is percolated.
3. If a C signal handler has been registered for the condition on the Z.C stack frame, it is given control. If it issues a resume or a call to `longjmp()`, the condition handling step ends. Processing resumes in the routine to which the resume cursor points.

   In this example, no signal handler is registered, so the condition is percolated.
4. The condition is still unhandled. If C does not recognize the condition, or if the C default action (listed in Table 63 on page 249) is to terminate, the condition is percolated.
5. No user-written condition handlers can be registered using CEEHDLR on the Y.PLI stack frame, because they cannot be written in PL/I. If an ON-unit that corresponds to the condition being processed exists on the Y.PLI stack frame, however, it is given control. If it issues a GOTO out of block, the condition handling step ends. Execution resumes at the label of the GOTO.

   In this example, no ON-unit has been established for the condition on the Y.PLI stack frame, so the condition is percolated.
6. If a user-written condition handler has been registered using CEEHDLR on the X.COBOL stack frame, it is given control. (User-written condition handlers written in COBOL must be compiled with COBOL/370, COBOL for MVS & VM, or COBOL for OS/390 & VM.) If it issues a resume, with or without moving the resume cursor, the condition handling step ends. Processing continues in the routine to which the resume cursor points.
7. What happens next depends on whether the condition is promotable to the PL/I ERROR condition. The following can happen:
   - If the condition is not promotable to the PL/I ERROR condition, then Language Environment default actions take place, as described in Table 62 on page 249. Condition handling ends.
   - If the PL/I default action for the condition is to promote it to the PL/I ERROR condition, the condition is promoted, and another pass of the stack is made to look for ERROR ON-units or user-written condition handlers. If an ERROR ON-unit or user-written condition handler is found, it is invoked.
   - If either of the following occurs:
     - An ERROR ON-unit or user-written condition handler is found, but it does not issue a GOTO out of block or similar construct
     - No ERROR ON-unit or user-written condition handler is found

     then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled condition). Condition handling now enters the termination imminent step. Because T_I_U maps to the PL/I FINISH condition, a FINISH ON-unit or user-written condition handler is run if the stack frame in which it is established is reached.
   - If no condition handler moves the resume cursor and issues a resume, Language Environment terminates the thread.

User handlers that you register using CEEHDLR must be written in the same language you are using to do the registration.

Sample N-Way ILC applications

```
*Process lc(101),s,map,list,stmt,a(f),ag;
  NWAYILC: PROC OPTIONS(MAIN);
  /*Module/File Name: IBMNWAY
  /***********************************************************/
  /* FUNCTION   :  Interlanguage communications call to a    *
  /*              C program which in turn calls a           *
  /*              COBOL program.                            *
  /*                                                         *
  /* Our example illustrates a 3-way interlanguage call from *
  /* a PL/I main program to a C program and from C to        *
  /* a COBOL subroutine. PL/I initializes an array to zeros. *
  /* PL/I passes the array and an empty character string to  *
  /* C program NWAY2C. NWAY2C fills the numeric array with   *
  /* random numbers and a C character array with lowercase   *
  /* letters.  A COBOL program, NWAY2CB, is called to convert*
  /* the characters to uppercase. The random numbers array   *
  /* and the string of uppercase characters are returned     *
  /* to PL/I main program and printed.                       *
  /***********************************************************/
  /***********************************************************/
  /* DECLARES FOR THE CALL TO C                              *
  /***********************************************************/
  DCL J FIXED BIN(31,0);
  DCL NWAY2C EXTERNAL ENTRY RETURNS(FIXED BIN(31,0));
  DCL RANDS(6) FIXED BIN(31,0);
  DCL STRING CHAR(80) INIT('Initial String Value');
  DCL ADDR BUILTIN;
  RANDS = 0;

  PUT SKIP LIST('NWAYILC STARTED');
  /**********************************************/
  /*Pass array and an empty string to C.        */
  /**********************************************/
  J = NWAY2C( ADDR( RANDS ), ADDR( STRING ) );
  PUT SKIP LIST ('Returned from C and COBOL subroutines');
  IF (J = 999) THEN DO;
   PUT EDIT (STRING) (SKIP(1) , A(80));
   PUT EDIT ( (RANDS(I) DO I = 1 TO HBOUND(RANDS,1)) )
               (SKIP(1) , F(10) );
  END;
  ELSE DO;
    PUT SKIP LIST('BAD RETURN CODE FROM C');
  END;
  PUT SKIP LIST('NWAYILC ENDED');
 END NWAYILC;
```

Figure 49. PL/I main routine of ILC application

```
/*Module/File Name:  EDCNWAY  */
 /****************************************************************
  * NWAY2C is invoked by a PL/I program. The PL/I program passes   *
  * an array of zeros and an UNINITIALIZED character string.       *
  * NWAY2C fills the array with random numbers. It fills the       *
  * character string with lowercase letters, calls a COBOL         *
  * subroutine to convert them to uppercase (NWAY2CB), and returns *
  * to PL/I. The by reference parameters are modified.             *
  ****************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#ifdef __cplusplus
  extern "PLI" int NWAY2C (int *c_array 6 , char *chrptr 80   );
  extern "COBOL" void NWAY2CB(char *);
#else
  #pragma linkage (NWAY2C,PLI)
  #pragma linkage (NWAY2CB,COBOL)
  void NWAY2CB(char *);
#endif
  int NWAY2C (int *c_array 6 , char *chrptr 80   )
   {
    int  *pRns ;
    char *pChr ;
    char string  80  = "the random numbers are";
    int i, ret=999;
    fprintf(stderr,"NWAY2C STARTED\n");
    /********************************************************/
    /* Check chrptr from PLI to verify we got what expected *
    /********************************************************/
   if(strncmp(*chrptr, "Initial String Value", 20))
   {
     fprintf(stderr,
       "NWAY2C: chrptr not what expected.\n \"%s\"\n", *chrptr);
     --ret;
   }
   /********************************************************/
   /*Fill numeric array parameter with random numbers.     *
   /*Adjust for possible array element size difference.    *
   /********************************************************/
   pRns = *c_array;
   for (i=0; i < 6; ++i)
   {
      pRns i  = rand() ;
      fprintf(stderr,"pRns %d  = %d\n",i,pRns i  );
   }
   /********************************************************/
   /* Call COBOL to change lowercase characters to upper.  *
   /********************************************************/
   NWAY2CB(*chrptr);
   if(strncmp(*chrptr, "INITIAL STRING VALUE", 20))
   {
     fprintf(stderr,
       "NWAY2C: string not what expected.\n \"%s\"\n", *chrptr);
     --ret;
   }
   fprintf(stderr,"NWAY2C ENDED\n");
   return(ret);
  }
```

Figure 50. C routine called by PL/I in a 3-way ILC application

```
  CBL QUOTE
*Module/File Name: IGZTNWAY
*************************************************
** NWAY2CB is called and passed an 80-character *
*  lowercase character string by reference.     *
*  The string is converted to uppercase and     *
*  control returns to the caller.               *
*************************************************
 ID DIVISION.
 PROGRAM-ID. NWAY2CB.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 77  STRING-VAL PIC X(80).
 PROCEDURE DIVISION USING STRING-VAL.

     DISPLAY "NWAY2CB STARTED".
     MOVE FUNCTION UPPER-CASE (STRING-VAL)
          TO STRING-VAL.
     DISPLAY "NWAY2CB ENDED".

     GOBACK.
```

Figure 51. COBOL program called by C in a 3-way ILC application

Chapter 14. Communicating between assembler and HLLs

This chapter describes Language Environment's support for assembler–HLL ILC applications. For more information about using assembler under Language Environment, see *z/OS Language Environment Programming Guide*.

Calling between assembler and an HLL

Language Environment-conforming assembler routines can be dynamically called/fetched from any Language Environment-conforming HLL. In addition, Language Environment-conforming assembler routines can dynamically load another routine by using the Language Environment CEEFETCH macro or CEELOAD macro. Using CEEFETCH to load the routine allows the routine to be deleted using CEERELES.

Note: CEEFETCH/CEERELES and CEELOAD should **not** be used for DLLs.

Using the CEEFETCH macro

CEEFETCH dynamically loads a routine and is preferred over CEELOAD for loading routines. However, it does not create a nested enclave. CEEFETCH is the only supported method for loading a routine that was created using the DFSMS Binder when omitting the use of the Language Environment Prelinker Utility. The Language Environment Prelinker Utility function is provided in the DFSMS Binder eliminating the requirement for the prelink step to create the routine.

System services should not be used to delete modules loaded with CEEFETCH; instead, the CEERELES macro should be used to delete routines loaded with CEEFETCH. During thread (if SCOPE=THREAD) or enclave (if SCOPE=ENCLAVE) termination, Language Environment deletes modules loaded by CEEFETCH.

If CEEFETCH completes successfully, the address of the loaded routine is found in R15. The routine can then be invoked using BALR 14,15 (or BASSM 14,15). The syntax of CEEFETCH is described in *z/OS Language Environment Programming Guide*.

CEEFETCH can be used to dynamically load a routine that has been compiled XPLINK, as long as the entry point has been defined using `#pragma linkage (..., fetchable)`.

Using the CEERELES macro

CEERELES dynamically deletes a Language Environment-conforming routine. The syntax of CEERELES is described in *z/OS Language Environment Programming Guide*.

Using the CEELOAD macro

CEELOAD is used to dynamically load a Language Environment-conforming routine. It does not create a nested enclave, so the target of CEELOAD must be a subroutine.

Some restrictions apply when using CEELOAD to call, fetch, or dynamic load:

- You cannot fetch or dynamically call a routine that has been fetched by or dynamically called by another language, or has been dynamically loaded by CEELOAD.
- You cannot dynamically load a routine with CEELOAD that has already been dynamically loaded by CEELOAD, or has been fetched or dynamically called by another language.
- You cannot dynamically load a routine with CEELOAD that has been compiled XPLINK.
- The loading of a DFSMS program object with Deferred Load Classes is not supported by CEELOAD.

The syntax of CEELOAD is described in *z/OS Language Environment Programming Guide*.

While CEELOAD supports dynamic loading, in order to provide for loading and deleting a module, the CEEFETCH and CEERELES macros should be used.

Passing arguments between HLL and assembler routines

Arguments are passed between HLL and assembler routines in a list of addresses. Each address in a parameter list occupies a fullword in storage. The last fullword in the list must have its high-order bit turned on for the last parameter address to be recognized. Each address in a parameter list is either the address of a data item or the address of a control block that describes a data item.

POSIX

With POSIX(ON), calls to assembler can occur on any thread as long as the assembler routines use:

- The CEEENTRY/CEETERM macros provided by Language Environment, or
- The EDCPRLG/EDCEPIL macros provided by C/370

C and C++

To facilitate calls between assembler and C or C++, include the following directives in your C or C++ code:

In C `#pragma linkage(,OS)`

In C++

```
extern "OS"
```

Using either of these directives will result in the compiler generating code that uses the OS linkage convention. In this case, register 1 will contain the address of a list of addresses of the actual arguments. The last address in this list has its high order bit set. For more information, refer to the descriptions of these directives in *z/OS XL C/C++ Language Reference*.

COBOL

For COBOL to use z/OS parameter passing, use the USING BY REFERENCE phrase for CALL and PROCEDURE DIVISION statements:

```
77 X PIC S9(9) BINARY VALUE 32767.
77 Y PIC X(4)  VALUE 'PRM2'.
77 Z POINTER.
```

```
SET Z TO ADDRESS OF Y.
CALL 'SUB1' USING BY REFERENCE X,Y,Z
```

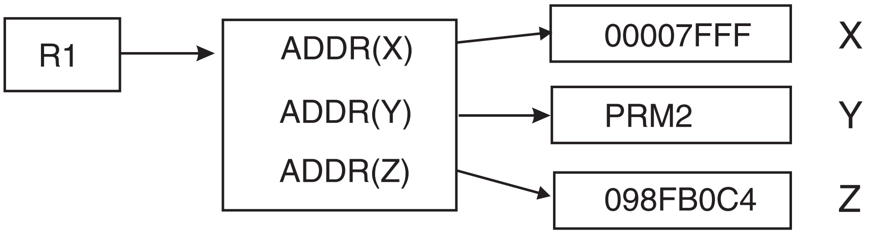


Figure 52. Parameter passing by reference

To pass a pointer to a copy of the parameters in COBOL, use the USING BY CONTENT phrase.

```
77 X PIC S9(9) BINARY VALUE 32767.
77 Y PIC X(4)   VALUE 'PRM2'.
77 Z POINTER.
```

```
SET Z TO ADDRESS OF Y.
CALL 'SUB1' USING BY CONTENT X,Y,Z
```

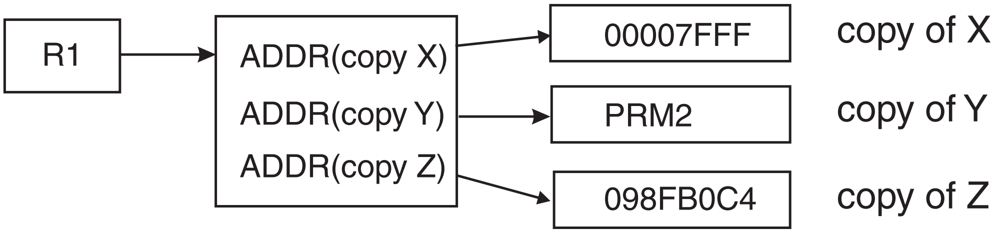


Figure 53. Parameter passing by content

To pass a parameter value directly in the parameter list in COBOL, as C does, use the USING BY VALUE phrase.

```
77 X PIC S9(9) BINARY VALUE 32767.
77 Y PIC X(4)   VALUE 'PRM2'.
77 z POINTER.
```

```
SET Z TO ADDRESS OF Y.
CALL 'SUB1' USING BY VALUE X,Y,Z
```

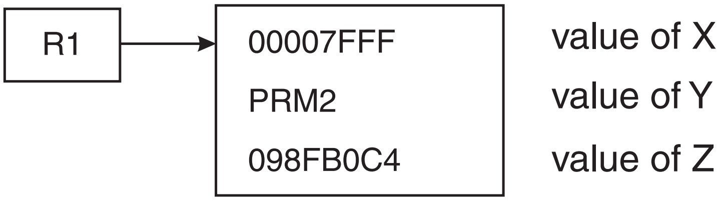


Figure 54. Parameter passing by value

To pass and/or receive a function value in COBOL, as C does, use the RETURNING phrase. The returned function value is accessed using register 15, so the COBOL RETURN-CODE special register cannot be used.

```
77 X PIC S9(9) BINARY VALUE 32767.
77 Y PIC X(4)   VALUE 'PRM2'.
77 R PIC S9(9) BINARY
```

```
SET Z TO ADDRESS of Y.
CALL 'SUB1' USING BY VLUE X,Y RETURNING R
```

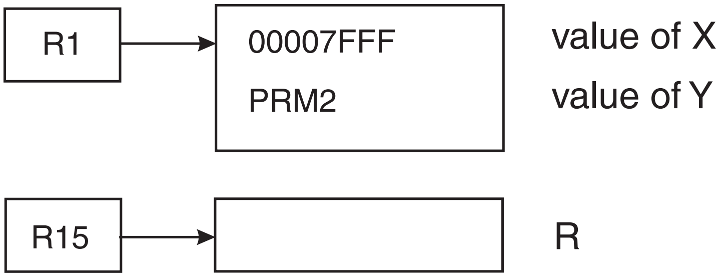


Figure 55. Parameter passing returning R

PL/I

For an assembler program to call PL/I, specify OPTIONS(ASM) on the PROC statement of the PL/I routine to be called. For PL/I to call an assembler program, your PL/I program should have an ENTRY declaration for the assembler routine it calls, and OPTIONS(ASM) should usually be specified on the ENTRY declaration.

How the parameters are passed should match. Generally, assembler routines pass and receive parameters using a list of addresses. This corresponds to passing parameters BYADDR in PL/I. For details on how to pass parameters in PL/I, see the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735).

Fortran

Assembler programs that initialized the Fortran runtime environment to call Fortran subroutines may need to be restructured to initialize Language Environment.

Canceling or releasing assembler

An assembler routine must be released using the same language that fetched it. An ILC module which has been loaded using CEELOAD cannot be deleted. An ILC module which has been fetched using CEEFETCH can be released using CEERELES. COBOL, C, C++, and PL/I can only CANCEL or release the assembler routine if there is no ILC with PL/I and Fortran in the target load modules.

Calling COBOL from assembler

AMODE considerations

When a called COBOL program returns control to a calling assembler program, the AMODE won't be reset to the AMODE of the calling program. Upon return to the calling assembler program, the AMODE will be the same as when the COBOL program was invoked. Therefore, when an assembler program calls a COBOL program that has a different AMODE, the calling program must save its own AMODE before calling. When control returns from the COBOL program, the calling assembler program must then restore its own AMODE.

The following instruction sequence illustrates the previous discussion:

```
     LA     2,RESET  SAVE BRANCH ADDRESS AND CURRENT
     BSM    2,0      AMODE IN REGISTER 2
     BASSM 14,15     CALL COBOL PROGRAM
     BSM    0,2      BRANCH AND RESTORE AMODE FROM REG. 2
RESET DS     OH
```

If an assembler program that is AMODE 31 calls a COBOL program that is AMODE 24, the assembler program must also be RMODE 24 in order for COBOL to return to the assembler program. If the assembler program is AMODE ANY, in this case, an abend may result from the COBOL program as a result of branching to an invalid address since R14 will contain a 31–bit address from the assembler program's save area, but COBOL will return to the assembler program in AMODE 24.

When you have an application with COBOL subprograms, some of the COBOL subprograms can be AMODE(31) and some can be AMODE(24). If your application consists of only COBOL programs and you are only using static and dynamic calls, each COBOL subprogram will always be entered in the proper AMODE. For example, if you are using a COBOL dynamic call from an AMODE(31) COBOL program to an AMODE(24) COBOL program, automatic AMODE switching is done.

However, if you are using assembler programs along with other HLL programs that call COBOL subprograms, you must ensure that when a COBOL subprogram is called more than once in an enclave, it is entered in the same AMODE each time it is called.

Canceling COBOL programs

Any COBOL subprograms compiled with the RENT compiler option that have been loaded by assembler routines must not be deleted by assembler routines. (This restriction does not apply to COBOL main programs loaded and deleted by assembler drivers.)

A COBOL program that has been fetched using CEEFETCH can be deleted using CEERELES.

Non-Language Environment-conforming assembler invoking an HLL main routine

When a C, C++, COBOL, or PL/I main routine is called from a non-Language Environment-conforming assembler program, the actions in Table 60 take place.

Table 60. What occurs when non-Language Environment-conforming assembler invokes an HLL main routine

Type of assembler invocation	Language Environment is not up	Language Environment is up
EXEC CICS LINK and EXEC CICS XCTL	Initial enclave is created.	Nested enclave is created. The COBOL program could be a main in this case.
EXEC CICS LOAD and BALR	This is not supported.	This is not supported.
LINK	Initial enclave is created.	Nested enclave is created. The COBOL program could be a main in this case.
LOAD and BALR	Initial enclave is created.	CEE393 is signaled. You cannot LOAD and BALR a main routine under Language Environment. However, in COBOL, this is supported because the COBOL program would be a subroutine, not a main.

Language Environment-conforming assembler invoking an HLL main routine

When a C, C++, or PL/I main routine is called from Language Environment-conforming assembler, the actions in Table 61 take place.

Note: Unlike C, C++, and PL/I, COBOL has no mechanism to statically declare a program “main”; rather, a "main" program is determined dynamically when a COBOL program is the first program in an enclave. Therefore, it is meaningful to call a COBOL "main", only in the context of creating a new enclave in which a COBOL program is the first to run. Only parts of the following tables apply to a COBOL main.

Table 61. What occurs when Language Environment-conforming assembler invokes an HLL main routine

Type of assembler invocation	Language Environment is up
CEELOAD macro	CEE393 is signaled. CEELOAD cannot load a main routine.
CEEFETCH macro	Nested enclave is not created. The COBOL program could be a main in this case.
EXEC CICS LINK and EXEC CICS XCTL	Nested enclave is created. The COBOL program could be a main in this case.
EXEC CICS LOAD and BALR	This is not supported.
LINK	Nested enclave is created. The COBOL program could be a main in this case.
LOAD and BALR	CEE393 is signaled. You cannot LOAD and BALR a main routine under Language Environment. This is supported in COBOL because the COBOL program would be a subroutine, not a main.

Note: See *z/OS Language Environment Programming Guide* for information about nested enclaves.

Assembler main routine calling HLL subroutines for better performance

To improve performance of a C, C++, COBOL, or PL/I routine called repeatedly from assembler, use a Language Environment-conforming assembler routine, because the Language Environment environment is maintained across calls. If the assembler routine is not Language Environment-conforming, the Language Environment environment is initialized and terminated at every call. To improve the performance for an assembler routine that is not Language Environment-conforming, use preinitialization services. (See *z/OS Language Environment Programming Guide* for information about using preinitialization services.)

The call can be either a static call (the HLL routine is linked with the assembler routine) or a dynamic load (using the CEELOAD or the CEEFETCH macro). The assembler routine is a main routine and the called HLL program is a subroutine.

For example, see Figure 56 on page 239, which demonstrates a Language Environment-conforming assembler routine statically calling a COBOL program.

```
*COMPILATION UNIT: LEASMCB
* ====================================================================
*   Bring up the LE/370 environment
* ====================================================================
CEE2COB  CEEENTRY PPA=MAINPPA,AUTO=WORKSIZE
         USING WORKAREA,13
*
*   Call the COBOL program
*
         CALL  ASMCOB,(X,Y)       Invoke COBOL subroutine
*
*   Call the CEEMOUT service
*
         CALL  CEEMOUT,(MESSAGE,DESTCODE,FC)  Dispatch message
         CLC   FC(8),CEE000       Was MOUT successful?
         BE    GOOD               Yes.. skip error logic
         LH    2,MSGNO            No.. Get message number
         ABEND (2),DUMP                LIGHTS OUT!
*
*   Terminate the LE/370 environment
*
GOOD     CEETERM  RC=0            Terminate with return code zero
*
* ---------------------------------------------------------------
*
*   Data Constants and Static Variables
*
Y        DC    PL3'+200'          2nd parm to COBOL program (input)
MESSAGE  DS    0H
MSGLEN   DC    Y(MSGEND-MSGTEXT)
MSGTEXT  DC    C'AFTER CALL TO COBOL: X='
X        DS    ZL6                1st parm for COBOL program (output)
MSGEND   EQU   *
DESTCODE DC    F'2'               Directs message to MSGFILE
CEE000   DC    3F'0'              Success condition token
FC       DS    0F                 12-byte feedback/condition code
SEV      DS    H                  severity
MSGNO    DS    H                  message number
CSC      DS    X                  flags - case/sev/control
CASE     EQU   X'C0'  11.....     case (1 or 2)
SEVER    EQU   X'38'  ..111..     severity (0 thru 4)
CNTRL    EQU   X'03'  .....11     control (1=IBM FACID, 0=USER)
FACID    DS    CL3                facility ID
ISI      DS    F                  index into ISI block
*
MAINPPA  CEEPPA                   Constants describing the code block
* ===================================================================
*   Workarea
* ===================================================================
WORKAREA DSECT
         CEEDSA  ,                Mapping of the Dynamic Save Area
         CEECAA  ,                Mapping of the Common Anchor Area
         CEEEDB  ,                Mapping of the Enclave Data Block
*
         END   CEE2COB
```

Figure 56. Language Environment-conforming assembler routine calling COBOL routine

```
*Module/File Name: IGZTASM
 IDENTIFICATION DIVISION.
   PROGRAM-ID. ASMCOB.
*
 ENVIRONMENT DIVISION.
 DATA DIVISION.
   WORKING-STORAGE SECTION.
*
 LINKAGE SECTION.
   01 X PIC +9(5).
   01 Y PIC S9(5) COMP-3.
*
 PROCEDURE DIVISION USING X Y.
     COMPUTE X = Y + 1.
*
     GOBACK.
```

Figure 57. COBOL routine called from Language Environment-conforming assembler

Chapter 15. ILC under CICS

In general, Language Environment provides the same ILC support for applications running under CICS as for those running in a non-CICS environment. If there is any ILC within a run unit under CICS, each compile unit must be compiled with a Language Environment-conforming compiler.

If you are using ILC in CICS DL/I applications, EXEC CICS DLI and CALL xxxTDLI can only be used in programs with the same language as the main program. CEETDLI is not supported on CICS.

XPLINK-compiled functions cannot run under CICS Transaction Server prior to CICS TS 3.1.

Fortran cannot run under CICS.

Language pairs supported in ILC under CICS

To understand what support Language Environment offers your ILC application, see the description for the specific language pair, and the applicable ILC chapter. If your ILC application involves multiple HLLs, see Chapter 13, “Communicating between multiple HLLs,” on page 225.

Note: The term ILC refers to the ILC that occurs within a Language Environment enclave.

Enclaves

A key feature of the program management model is the enclave, which consists of one or more load modules, each containing one or more separately compiled, bound routines. A load module can include HLL routines, assembler routines, and Language Environment routines.

By definition, the scope of a language statement is that portion of code in which it has semantic effect. The enclave defines the scope of the language semantics for its component routines.

Enclave boundary

The enclave defines the scope of the definition of the main routine and subroutines. The enclave boundary defines whether a routine is a main routine or a subroutine. The first routine to run in the enclave is known as the main routine in Language Environment. All others are designated subroutines of the main routine. The first routine invoked in the enclave must be capable of being designated main according to the rules of the language of the routine. All other routines invoked in the enclave must be capable of being a subroutine according to the rules of the languages of the routines.

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

Program mask conventions

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

When running an ILC application, the subroutines might involve multiple HLLs. The characteristics of these HLLs, such as program mask attributes, will be shared across the enclave.

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

The runtime option XUFLOW indicates the initial setting of the mask for exponent underflow. You can alter this setting by using the callable service CEE3SPM.

In summary, the initial setting of the program mask is determined by the requirements of the members within the main load module and by the setting of the XUFLOW runtime option.

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

C/C++ and COBOL

Language Environment supports ILC between routines written in C/C++ and COBOL under CICS as follows:

- Calls supported as documented in the sections "Calling between C++ and COBOL" and "Calling between C and COBOL" with the exception that there is no support for ILC calls to or from routines written in pre-Language Environment-conforming versions of C or COBOL.
- There is no support for ILC calls to or from routines written in pre-Language Environment-conforming versions of C or COBOL.

All components of your C/C++ to COBOL ILC application must be reentrant.

If there is any ILC with a run unit under CICS, each compile unit must be compiled with a Language Environment-conforming compiler.

For more information about ILC between C/C++ and COBOL, see Chapter 4, “Communicating between C and COBOL,” on page 27 and "Communicating between C++ and COBOL" on page 53.

z/OS XL C/C++ and PL/I

Language Environment supports ILC between routines written in z/OS XL C/C++ and PL/I for MVS & VM or Enterprise PL/I for z/OS under CICS as follows:

- z/OS XL C/C++ routines can statically call PL/I routines.
- PL/I routines can statically call z/OS XL C/C++ routines.

- z/OS XL C/C++ routines can `fetch()` PL/I routines that have OPTIONS(FETCHABLE) specified.
- PL/I routines can FETCH only those z/OS XL C/C++ routines that have not been run through the CICS translator. A PL/I routine cannot dynamically call an z/OS XL C/C++ routine that has been translated because the CICS translator introduces writable static data elements that are not capable of being initialized when the dynamic call is made.

  In addition, during the FETCH of z/OS XL C/C++ from PL/I, the static read/write pointer is not swapped.
- z/OS XL C/C++ routines calling PL/I routines must pass the EIB and COMMAREA as the first two parameters if the called routine contains any EXEC CICS commands.
- There is no support under CICS for ILC calls to or from routines written in pre-Language Environment-conforming versions of C or PL/I.

All components of your z/OS XL C/C++ to PL/I ILC application must be reentrant.

If there is any ILC with a run unit under CICS, each compile unit must be compiled with a Language Environment-conforming compiler.

For more information about ILC between PL/I and C, see Chapter 8, “Communicating between C and PL/I,” on page 127.

COBOL and PL/I

Language Environment supports ILC between routines compiled with Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, or PL/I for MVS & VM or Enterprise PL/I for z/OS under CICS as follows:

- Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, and COBOL/370 programs can statically call PL/I routines.
- PL/I routines can statically call Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, and COBOL/370 programs.
- Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, and COBOL/370 programs can dynamically CALL PL/I routines that have OPTIONS(FETCHABLE) specified.
- PL/I routines can FETCH Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, and COBOL/370 programs.
- COBOL routines calling PL/I routines must pass EIB and COMMAREA as the first two parameters if the called routine contains any EXEC CICS commands.
- PL/I routines calling Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, or COBOL/370 programs must pass the EIB and COMMAREA as the first two parameters if the called program contains any EXEC CICS commands.

For more information about ILC between COBOL and PL/I, see Chapter 11, “Communicating between COBOL and PL/I,” on page 185.

If there is any ILC with a run unit under CICS, each compile unit must be compiled with a Language Environment-conforming compiler.

Assembler

There is no support for Language Environment-conforming assembler main routines under CICS prior to z/OS V1R4, or prior to CICS Transaction Server for z/OS Version 3.1.

COBOL considerations

Static and dynamic calls are allowed in VS COBOL II, COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL for z/OS **to** but not **from** routines written in non-Language Environment-conforming assembler routines.

Calls are allowed from Language Environment-conforming assembler subprograms to COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL for z/OS.

Calls are allowed from COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL for z/OS to Language Environment-conforming assembler routines.

Calls are allowed from VS COBOL II programs to Language Environment-conforming assembler routines if the NAB=NO option is used on the CEEENTRY macro.

PL/I considerations

PL/I routines can statically call assembler routines declared with OPTIONS(ASSEMBLER). When you declare a routine with OPTIONS(ASSEMBLER), arguments are passed according to standard linking conventions.

Called assembler subroutines can invoke CICS services if they were passed the appropriate CICS control blocks.

See the CICS documentation for information about the use of CICS commands in an assembler language subroutine.

Link-editing ILC applications under CICS

You must link ILC applications with the CICS stub, DFHELII, in order to get ILC support under Language Environment.

ILC applications in which C/C++ is one of the participating languages must be link-edited AMODE(31).

CICS ILC application

The following examples illustrate how you can use ILC under CICS. A COBOL main program, COBCICS, dynamically CALLs a PL/I routine, PLICICS, which does the following:

- Writes a message to the operator
- Establishes a ZERODIVIDE ON-unit
- Generates a divide-by-zero
- Writes another message to the operator
- Returns to the COBOL main program

COBCICS then calls CUCICS, a statically linked C routine, and passes a message character string and a length field to the subroutine. This routine then calls the Language Environment service CEEMOUT to write the message to the CESE transient data queue.

```
 CBL  XOPTS(COBOL2),LIB,APOST
*Module/File Name: IGZTCICS
******************************************************
*  TRANSACTION: COBC.                                *
*  FUNCTION:                                         *
*                                                    *
*       A CICS COBOL main dynamically calls a PL/I   *
*       subroutine, and statically calls a C         *
*       subroutine. COBCICS passes a message to      *
*       the C subroutine to output to the            *
*       transient data queue.                        *
*                                                    *
******************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. COBCICS.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77  STARTMSG     PIC X(16) VALUE 'STARTING COBCICS'.
 77  DTVAL        PIC X(14) VALUE 'ENDING COBCICS'.
 77  RUNNING      PIC X(80) VALUE 'STARTING CUCICS'.
 77  RUNLENGTH    PIC S9(4) BINARY VALUE 15.
 77  PLISUBR      PIC X(8) VALUE 'PLISUBR'.

 PROCEDURE DIVISION.


     EXEC CICS SEND FROM(STARTMSG) ERASE END-EXEC.
     CALL PLISUBR USING DFHEIBLK DFHCOMMAREA.
     CALL 'CUCICS'  USING RUNLENGTH  RUNNING.


     EXEC CICS SEND FROM(DTVAL) ERASE END-EXEC.

     EXEC CICS RETURN END-EXEC.
```

Figure 58. COBOL CICS main program that calls C and PL/I subroutines

```
 /*module/file name: ibmcics                        */
 /***************************************************/
 /**                                                *
 /**  function:                                     *
 /**                                                *
 /**  plicics is a pl/i cics subroutine that is     *
 /**  called from a cobol main program, cobcics.    *
 /**  plicics writes a startup message to the       *
 /**  terminal operator and establishes a           *
 /**  zerodivide on-unit. a zerodivide is           *
 /**  generated and the zerodivide on-unit is       *
 /**  called to notify the terminal operator. the   *
 /**  zerodivide performs a normal return to the    *
 /**  program and the control returns to cobol.     *
 /**                                                *
 /***************************************************/

 plicics : procedure(dfheiptr) options(fetchable);

  dcl  running  char(20) init ( 'plicics entered' ) ;
  dcl  msg  char(30);

  msg = 'plicics entered';
  exec cics send from(msg) length(15) erase;
  on zdiv begin;
    msg = 'inside of zdiv on unit';
    put skip list(msg);
    exec cics send from(msg) length(30) erase;
  end;
  a = 10;
  a = a/0;

end plicics;
```

Figure 59. PL/I routine called by COBOL CICS main program

```
 /*Module/File Name: EDCCICS                                          */
 /*********************************************************************/
 /**                                                                  */
 /**Function: CEEMOUT: write message to transient data queue.         */
 /*                                                                   */
 /*  This example illustrates a C CICS subroutine that is             */
 /*  statically linked to a COBOL main routine, COBCICS. COBCICS      */
 /*  passes a message character string and a length field to the      */
 /*  subroutine.  This routine then calls the CEEMOUT service         */
 /*  to write the message to the transient data queue, CESE.          */
 /*                                                                   */
 /*********************************************************************/
#ifndef __cplusplus
#pragma linkage(CUCICS,COBOL)
#else
 extern "COBOL" void CUCICS(unsigned short *len, char (* running) 80 );
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <leawi.h>
#include <ceeedcct.h>

  _VSTRING message;
  _INT4 dest;
  _CHAR80 msgarea;
  _FEEDBACK fc;
 /*                                                                */
 /*  mainline.                                                     */
 /*                                                                */
void CUCICS(unsigned short *len, char (* running) 80  )
{
  /* Send a message to the CICS terminal operator.                 */
  char * startmsg = "CUCICS STARTED\n";
  unsigned short I1;
  I1 = strlen(startmsg);
  EXEC CICS SEND FROM(startmsg) LENGTH(I1) ERASE;

  /* set output area to nulls                   */
  memset(message.string,'\0',sizeof(_CHAR80) );
  if (*len >= sizeof(_CHAR80) )
      *len  = sizeof(_CHAR80)-1 ;

  /* copy message to output area */
  memcpy(message.string, running,(unsigned int) *len);

  message.length = (unsigned int) *len;
  dest = 2;
  /**************************************************************
   *   Call CEEMOUT to place copy of operator message in        *
   *   transient data queue CESE.                               *
   **************************************************************
  CEEMOUT(&message,&dest,&fc);

  if ( _FBCHECK (fc , CEE000) != 0 ) {
     /* put the message if CEEMOUT failed */
     dest = 2;
     CEEMSG(&fc,&dest,NULL);
     exit(2999);
  }
}
```

Figure 60. C routine called by COBOL CICS main program

Appendix A. Condition-handling responses

Table 62 and Table 63 list condition-handling responses, as referenced in the condition handling sections of the pairwise chapters.

Table 62. Language Environmentdefault responses to unhandled conditions. Language Environment's default responses to unhandled conditions fall into one of two types, depending on whether the condition was signaled using CEESGL and an fc parameter, or it came from any other source.

Severity of condition	Condition signaled by user in a call to CEESGL with an fc	Condition came from any other source
0 (Informative message)	Return CEE069 condition token, and resume processing at the next sequential instruction. See the fc table for CEESGL (*z/OS Language Environment Programming Reference*) for a description of the CEE069 condition token.	Resume without issuing message.
1 (Warning message)	Return CEE069 condition token, and resume processing at the next sequential instruction.	If the condition occurred in a stack frame associated with a COBOL program, resume and issue the message. If the condition occurred in a stack frame associated with a non-COBOL program, resume without issuing message.
2 (Program terminated in error)	Return CEE069 condition token, and resume processing at the next sequential instruction.	Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.
3 (Program terminated in severe error)	Return CEE069 condition token, and resume processing at the next sequential instruction.	Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.
4 (Program terminated in critical error)	Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.	Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.

Table 63 contains default C language error handling semantics.

Table 63. Cconditions and default system actions

C Condition	Origin	Default action
SIGILL	Execute exception operation exception privileged operation `raise(SIGILL)`	Abnormal termination (return code=3000)
SIGSEGV	Addressing exception protection exception specification exception `raise(SIGSEGV)`	Abnormal termination (return code=3000)

Table 63. Cconditions and default system actions (continued)

C Condition	Origin	Default action
SIGFPE	Data exception decimal divide exponent overflow fixed point divide floating point divide `raise(SIGFPE)`	Abnormal termination (return code=3000)
SIGABRT	`abort()` function `raise(SIGABRT)`	Abnormal termination (return code=2000)
SIGABND	Abend the function	Abnormal termination (return code=3000)
SIGTERM	Termination request `raise(SIGTERM)`	Abnormal termination (return code = 3000)
SIGINT	Attention condition	Abnormal termination (return code = 3000)
SIGIOERR	I/O errors	Ignore the condition
SIGUSR1	User-defined condition	Abnormal termination (return code=3000)
SIGUSR2	User-defined condition	Abnormal termination (return code=3000)
Masked	Exponent overflow fixed-point underflow significance	These exceptions are disabled. They are ignored during the condition handling process, even if you try to enable them using the CEE3SPM callable service.

Appendix B. Using nested enclaves

An enclave is a logical runtime structure that supports the execution of a collection of routines.

z/OS Language Environment explicitly supports the execution of a single enclave within a Language Environment process. However, by using the system services and language constructs described in this chapter, you can create an additional, or nested, enclave and initiate its execution within the same process.

The enclave that issues a call to system services or language constructs to create a nested enclave is called the *parent* enclave. The nested enclave that is created is called the *child* enclave. The child must be a main routine; a link to a subroutine by commands and language constructs is not supported under Language Environment.

If a process contains nested enclaves, none or only one enclave can be running with POSIX(ON).

Understanding the basics

In Language Environment, you can use the following methods to create a child enclave:

- Under CICS, the EXEC CICS LINK and EXEC CICS XCTL commands
- Under z/OS, the SVC LINK command
- Under z/OS, the C `system()` function (see *z/OS Language Environment Programming Guide* for more information about `system()`)
- Under z/OS, the PL/I FETCH and CALL to any of the following PL/I routines with PROC OPTIONS(MAIN) specified:
  - – Enterprise PL/I for z/OS
  - – PL/I for MVS & VM
  - – OS PL/I Version 2
  - – OS PL/I Version 1 Release 5.1
  - – Relinked OS PL/I Version 1 Release 3.0–5.1

  Such a routine, called a *fetchable main* in this book, can only be introduced by a FETCH and CALL from a PL/I routine.

  The routine performing the FETCH and CALL must be compiled with the PL/I for MVS & VM or Enterprise PL/I compiler or be a relinked OS PL/I routine.

If the target routine of any of these commands is not written in a Language Environment-conforming HLL or Language Environment-conforming assembler, no nested enclave is created.

COBOL considerations

In a non-CICS environment, OS/VS COBOL routines are supported in a single enclave only.

PL/I considerations

PL/I MTF is supported in the initial enclave only. If PL/I MTF is found in a nested enclave, Language Environment diagnoses it as an error. If a PL/I MTF application contains nested enclaves, the initial enclave must contain a single task. Violation of this rule is not diagnosed and is likely to cause unpredictable results.

Determining the behavior of child enclaves

If you want to create a child enclave, you need to consider the following factors:

- The language of the main routine in the child enclave
- The sources from which each type of child enclave gets runtime options
- The default condition handling behavior of each type of child enclave
- The setting of the TRAP runtime option in the parent and the child enclave

All of these interrelated factors affect the behavior, particularly the condition handling, of the created enclave. The sections that follow describe how the child enclaves created by each method (EXEC CICS LINK, EXEC CICS XCTL, SVC LINK, CMSCALL, C `system()` function, and PL/I FETCH and CALL of a fetchable main) will behave.

Creating child enclaves using EXEC CICS LINK or EXEC CICS XCTL

If your C, C++, COBOL, or PL/I application uses EXEC CICS commands, you must also link-edit the EXEC CICS interface stub, DFHELII, with your application. To be link-edited with your application, DFHELII must be available in the link-edit SYSLIB concatenation.

How runtime options affect child enclaves

The child enclave gets its runtime options from one of the sources discussed in *z/OS Language Environment Programming Guide*. The runtime options are completely independent of the creating enclave, and can be set on an enclave-by-enclave basis.

Some of the methods for setting runtime options might slow down your transaction. Follow these suggestions to improve performance:

- If you need to specify options in CEEUOPT, specify only those options that are different from system defaults.
- Before putting transactions into production, request a storage report (using the RPTSTG runtime option) to minimize the number of GETMAINs and FREEMAINs required by the transactions.
- Ensure that VS COBOL II transactions are not link-edited with IGZETUN, which is no longer supported and which causes an informational message to be logged. Logging this message for every transaction inhibits system performance.

How conditions arising in child enclaves are handled

This section describes the default condition handling for child enclaves created by EXEC CICS LINK or EXEC CICS XCTL.

Condition handling varies depending on the source of the condition, and whether or not an EXEC CICS HANDLE ABEND is active:

- If a Language Environment or CEEBXITA-initiated (generated by setting the CEEAUE_ABND field of CEEBXITA) abend occurs, the CICS thread is terminated. This occurs even if a CICS HANDLE ABEND is active, because CICS HANDLE ABEND does not gain control in the event of a Language Environment abend.
- If a software condition of severity 2 or greater occurs, Language Environment condition handling takes place. If the condition remains unhandled, the problem

is not percolated to the parent enclave. The CICS thread is terminated with an abend. These actions take place even if a CICS HANDLE ABEND is active, because CICS HANDLE ABEND does not gain control in the event of a Language Environment software condition.

- If a user abend or program check occurs, the following actions take place:
  - If no EXEC CICS HANDLE ABEND is active, and TRAP(ON) is set in the child enclave, Language Environment condition handling takes place. If the abend or program check remains unhandled, the problem is not propagated to the parent enclave. The CICS thread is terminated with an abend.
  - An active EXEC CICS HANDLE ABEND overrides the setting of TRAP. The action defined by the EXEC CICS HANDLE ABEND takes place.

Creating child enclaves by calling a second main without an RB crossing

The behavior of a child enclave created by an calling a second main program is determined by the language of its main routine: C, C++, COBOL, Fortran, PL/I, or Language Environment-conforming assembler (generated by use of the CEEENTRY and associated macros).

How runtime options affect child enclaves

Runtime options are processed in the normal manner for enclaves created because of a call to a second main. That is, programmer defaults present in the load module are merged, options in the command-line equivalent are also processed, as are options passed by the assembler user exit if present.

How conditions arising in child enclaves are handled

The command-line equivalent is determined in the same manner as for a SVC LINK for both VM and MVS.

Creating child enclaves using SVC LINK or CMSCALL

The behavior of a child enclave created by an SVC LINK or CMSCALL is determined by the language of its main routine: C, C++, COBOL, Fortran, PL/I, or Language Environment-conforming assembler (generated by use of the CEEENTRY and associated macros).

MVS considerations: When issuing a LINK to a routine, the high-order bit must be set on for the last word of the parameter list. To do this, set VL=1 on the LINK assembler macro.

How runtime options affect child enclaves

Child enclaves created by an SVC LINK or CMSCALL get runtime options differently, depending on the language that the main routine of the child enclave is written in.

Child enclave has a C, C++, Fortran, PL/I, or Language Environment-conforming assembler main routine: If the main routine of the child enclave is written in C, C++, Fortran, PL/I, or in Language Environment-conforming assembler, the child enclave gets its runtime options through a merge from the usual sources. Therefore, you can set runtime options on an enclave-by-enclave basis. However, you cannot pass command-line parameters to nested enclaves.

Child enclave has a COBOL main routine: If the main routine of the child enclave is written in COBOL, the child enclave inherits the runtime options of the creating enclave. Therefore, you cannot set runtime options on an enclave-by-enclave basis.

How conditions arising in child enclaves are handled

If a Language Environment or CEEBXITA-initiated (generated by setting the CEEAUE_ABND field of CEEBXITA) abend occurs in a child enclave created by SVC LINK or CMSCALL, regardless of the language of its main, the entire process is terminated.

Condition handling in child enclaves created by SVC LINK or CMSCALL varies, depending on the language of the child's main routine, the setting of the TRAP runtime option in the parent and child enclaves, the type of condition, and whether the routine is running under MVS. Refer to one of the following tables to see what happens when a condition remains unhandled in a child enclave.

Table 64. Handling conditions in child enclaves

If the child enclave was created by:	See:
An SVC LINK under MVS and has a C, C++, or Language Environment-conforming assembler main routine	Table 65
An SVC LINK or CMSCALL under CMS and has a C or Language Environment-conforming assembler main routine	Table 66 on page 255
An SVC LINK under MVS and has a COBOL main program	Table 67 on page 255
An SVC LINK under MVS and has a Fortran or PL/I main routine	Table 68 on page 255

You should always run your applications with TRAP(ON) or your results might be unpredictable.

Child enclave has a C, C++, or Language Environment-conforming assembler main routine:

Table 65. Unhandled condition behavior in a C, C++, or assembler child enclave, under MVS

	Parent enclave TRAP(ON) Child enclave TRAP(ON)	Parent enclave TRAP(ON) Child enclave TRAP(OFF)	Parent enclave TRAP(OFF) Child enclave TRAP(ON)	Parent enclave TRAP(OFF) Child enclave TRAP(OFF)
Unhandled condition severity 0 or 1	Resume child enclave	Resume child enclave	Resume child enclave	Resume child enclave
Unhandled condition severity 2 or above	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition
Non-Language Environment abend	Resume parent enclave, and ignore condition	Process terminated with original abend code	Resume parent enclave, and ignore condition	Process terminated with original abend code
Program check	Resume parent enclave, and ignore condition	Process terminated with abend U4036, Reason Code=2	Resume parent enclave, and ignore condition	Process terminated with abend S0Cx

Table 66. Unhandled condition behavior in a C or assembler child enclave, under CMS

	Parent enclave TRAP(ON) Child enclave TRAP(ON)	Parent enclave TRAP(ON) Child enclave TRAP(OFF)	Parent enclave TRAP(OFF) Child enclave TRAP(ON)	Parent enclave TRAP(OFF) Child enclave TRAP(OFF)
Unhandled condition severity 0 or 1	Resume child enclave	Resume child enclave	Resume child enclave	Resume child enclave
Unhandled condition severity 2 or above	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition
Non-Language Environment abend	Process terminated with original abend code	Process terminated with original abend code	Process terminated with original abend code	Process terminated with original abend code
Program check	Resume parent enclave, and ignore condition	Process terminated with abend U4036, Reason Code=2	Resume parent enclave, and ignore condition	Process terminated with CMS message

Child enclave has a COBOL main routine: Child enclaves created by SVC LINK or CMSCALL that have a COBOL main program inherit the runtime options of the parent enclave that created them. Therefore, the TRAP setting of the parent and child enclaves is always the same.

Table 67. Unhandled condition behavior in a COBOL child enclave, under MVS

	Parent enclave TRAP(ON) Child enclave TRAP(ON)	Parent enclave TRAP(OFF) Child enclave TRAP(OFF)
Unhandled condition severity 0 or 1	Resume child enclave	Resume child enclave
Unhandled condition severity 2 or above	Signal CEE391 (Severity=1, Message Number=3361) in parent enclave	Process terminated with abend U4094 RC=40
Non-Language Environment abend	Signal CEE391 in parent enclave	Process terminated with original abend code
Program check	Signal CEE391 in parent enclave	Process terminated with abend S0Cx

Child enclave has a Fortran or PL/I main routine:

Table 68. Unhandled condition behavior in a Fortran or PL/I child enclave, under MVS

	Parent enclave TRAP(ON) Child enclave TRAP(ON)	Parent enclave TRAP(ON) Child enclave TRAP(OFF)	Parent enclave TRAP(OFF) Child enclave TRAP(ON)	Parent enclave TRAP(OFF) Child enclave TRAP(OFF)
Unhandled condition severity 0 or 1	Resume child enclave	Resume child enclave	Resume child enclave	Resume child enclave
Unhandled condition severity 2 or above	Signal CEE391 (Severity=1, Message Number=3361) in parent enclave	Signal CEE391 in parent enclave	Process terminated with abend U4094 RC=40	Process terminated with abend U4094 RC=40
Non-Language Environment abend	Signal CEE391 in parent enclave	Process terminated with original abend code	Process terminated with abend U4094, Reason Code=40	Process terminated with original abend code
Program check	Signal CEE391 in parent enclave	Process terminated with abend U4036, Reason Code=2	Process terminated with abend U4094 RC=40	Process terminated with abend S0Cx

Creating child enclaves using the C system() function

Child enclaves created by the C `system()` function get runtime options through a merge from the usual sources (see *z/OS Language Environment Programming Guide* for more information). Therefore, you can set runtime options on an enclave-by-enclave basis. See *z/OS XL C/C++ Runtime Library Reference* for information about the `system()` function when running with POSIX(ON).

Under MVS, when you perform a `system()` function to a COBOL program, in the form:

```
system("PGM=program_name,PARM='...'")
```

the runtime options specified in the PARM= portion of the `system()` function are ignored. However, runtime options are merged from other valid sources.

z/OS UNIX considerations

In order to create a nested enclave under z/OS UNIX, you must either:

- Be running with POSIX(OFF) and issue `system()`, or
- Be running with POSIX(ON) and have set the environment variables to signal that you want to establish a nested enclave. You can use the `__POSIX_SYSTEM` environment variable to cause a `system()` to establish a nested enclave instead of performing a `fork()/exec()`. `__POSIX_SYSTEM` can be set to `NO`, `No`, or `no`.

In a multiple enclave environment, the first enclave must be running with POSIX(ON) and all other nested enclaves must be running with POSIX(OFF).

How conditions arising in child enclaves are handled

If a Language Environment- or CEEBXITA-initiated (generated by setting the CEEAUE_ABND field of CEEBXITA) abend occurs in a child enclave created by a call to `system()`, the entire process is terminated.

Depending on what the settings of the TRAP runtime option are in the parent and child enclave, the following might cause the child enclave to terminate:

- Unhandled user abend
- Unhandled program check

TRAP(ON | OFF) effects for enclaves created by system()

Table 69. Unhandled condition behavior in a system()-created child enclave, under MVS

	Parent enclave TRAP(ON) Child enclave TRAP(ON)	**Parent enclave TRAP(ON) Child enclave TRAP(OFF)**	**Parent enclave TRAP(OFF) Child enclave TRAP(ON)**	**Parent enclave TRAP(OFF) child enclave TRAP(OFF)**
Unhandled condition severity 0 or 1	Resume child enclave	Resume child enclave	Resume child enclave	Resume child enclave
Unhandled condition severity 2 or above	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition
Non-Language Environment abend	Resume parent enclave, and ignore condition	Process terminated with original abend code	Resume parent enclave, and ignore condition	Process terminated with original abend code
Program check	Resume parent enclave, and ignore condition	Process terminated with abend U4036, Reason Code=2	Resume parent enclave, and ignore condition	Process terminated with abend S0Cx

Creating child enclaves that contain a PL/I fetchable main

Under VM, the target load module can only be a member of a LOADLIB or be in a saved segment or relocatable load module. The target load module cannot be on a text deck or be a member of a TXTLIB.

Additional fetch and call considerations of PL/I fetchable mains are discussed in "Special fetch and call considerations."

How runtime options affect child enclaves

Child enclaves created when you issue a FETCH and CALL of a fetchable main get runtime options through a merge from the usual sources. Therefore, you can set runtime options on an enclave-by-enclave basis.

How conditions arising in child enclaves are handled

If a Language Environment or CEEBXITA-initiated (generated by setting the CEEAUE_ABND field of CEEBXITA) abend occurs in a child enclave that contains a fetchable main, the entire process is terminated.

Depending on what the settings of the TRAP runtime option are in the parent and child enclave, the following might cause the child enclave to terminate:

- Unhandled user abend
- Unhandled program check

Table 70. Unhandled condition behavior in a child enclave that contains a PL/I fetchable main, under MVS

	Parent enclave TRAP(ON) Child enclave TRAP(ON)	**Parent enclave TRAP(ON) Child enclave TRAP(OFF)**	**Parent enclave TRAP(OFF) Child enclave TRAP(ON)**	**Parent enclave TRAP(OFF) child enclave TRAP(OFF)**
Unhandled condition severity 0 or 1	Resume child enclave	Resume child enclave	Resume child enclave	Resume child enclave
Unhandled condition severity 2 or above	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition
Non-Language Environment abend	Resume parent enclave, and ignore condition	Process terminated with original abend code	Resume parent enclave, and ignore condition	Process terminated with original abend code
Program check	Resume parent enclave, and ignore condition	Process terminated with abend U4036, Reason code=2	Resume parent enclave, and ignore condition	Process terminated with abend S0Cx

Special fetch and call considerations

Do not recursively fetch and call the fetchable main from within the child enclave; results are unpredictable if you do.

The load module that is the target of the FETCH and CALL is reentrant if all routines in the load module are reentrant.

Language Environment relies on the underlying operating system for the management of load module attributes. In general, multiple calls of the same load module are supported for load modules that are reentrant, nonreentrant but serially reusable, or nonreentrant and non-serially reusable.

- Reentrant

  It is recommended that your target load module be reentrant.
- Nonreentrant but serially reusable

You should ensure that the main procedure of a nonreentrant but serially reusable load module is self-initializing. Results are unpredictable otherwise.

- Nonreentrant and non-serially reusable

  If a nonreentrant and non-serially reusable load module is called multiple times, each new call brings in a fresh copy of the load module. That is, there are two copies of the load module in storage: one from FETCH and one from CALL. Even though there are two copies of the load module in storage, you need only one PL/I RELEASE statement because upon return from the created enclave the load module loaded by CALL is deleted by the operating system. You need only release the load module loaded by FETCH.

Other nested enclave considerations

The following sections contain other information you might need to know when creating nested enclaves. The topics include:

- The string that CEE3PRM returns for each type of child enclave (see *z/OS Language Environment Programming Reference* for more information about the CEE3PRM callable service)
- The return and reason codes that are returned on termination of the child enclave
- How the assembler user exit handles nested enclaves
- Whether or not the message file is closed on return from a child enclave
- z/OS UNIX considerations
- AMODE considerations

What the enclave returns from CEE3PRM

CEE3PRM returns to the calling routine the user parameter string that was specified at program invocation. Only program arguments are returned.

See Table 71 to determine whether a user parameter string was passed to your routine, and where the user parameter string is found. This depends on the method you used to create the child enclave, the language of the routine in the child enclave, and the PLIST, TARGET, or SYSTEM setting of the main routine in the child enclave. If a user parameter string was passed to your routine, the user parameter string is extracted from the command-line equivalent for your routine (shown in Table 72 on page 259) and returned to you.

Note: Under CICS, CEE3PRM always returns a blank string.

Table 71. Determining the command-line equivalent

Language	Option	Suboption	`system()` on MVS	SVC LINK on MVS	Fetch/call of a PL/I main
C	`#pragma runopts(PLIST)`	HOST, CMS, MVS	PARM =, or the parameter string from the command string passed to `system()`	Halfword length-prefixed string pointed to by R1	Not allowed
		CICS,IMS,OS, or TSO	Not available	Not available	Not allowed

Table 71. Determining the command-line equivalent (continued)

Language	Option	Suboption	system() on MVS	SVC LINK on MVS	Fetch/call of a PL/I main
C++	PLIST and TARGET compiler options	Default	PARM =, or the parameter string from the command string passed to system()	Halfword length-prefixed string pointed to by R1	Not allowed
		PLIST(OS) and/or TARGET(IMS)	Not available	Not available	Not allowed
COBOL			Null	Null	Not allowed
Fortran			PARM =, or the parameter string from the command string passed to system()	Halfword length-prefixed string pointed to by R1	Not allowed
PL/I	SYSTEM compiler option	MVS	PARM = or the parameter string from the command string passed to system()	Halfword length-prefixed string pointed to by R1	User parameters passed through CALL
		CMS	Abend 4093-16	Abend 4093-16	User parameters passed through CALL
		CICS, CMSTPL, IMS, TSO	Not available	Not available	SYSTEM(CICS) not supported. Others not available.
Language Environment-conforming assembler	CEENTRY PLIST=	HOST, CMS, MVS	PARM = or the parameter string from the command string passed to system()	Halfword length-prefixed string pointed to by R1	Not allowed
		CICS, IMS, OS, or TSO	Not available	Not available	Not allowed

If Table 71 on page 258 indicates that a parameter string was passed to your routine at invocation, the string is extracted from the command-line equivalent listed in the right-hand column of Table 77. The command-line equivalent depends on the language of your routine and the runtime options specified for it.

Table 72. Determining the order of runtime options and program arguments

Language of routine	Runtime options in effect?	Order of runtime options and program arguments
C	`#pragma runopts(EXECOPS)`	runtime options / user parms
	`#pragma runopts(NOEXECOPS)`	entire string is user parms
C++	Compiled with EXECOPS (default)	runtime options / user parms
	Compiled with NOEXECOPS	entire string is user parms
COBOL	CBLOPTS(ON)	runtime options / user parms
	CBLOPTS(OFF)	user parms / runtime options
Fortran		runtime options / user parms

Table 72. Determining the order of runtime options and program arguments (continued)

Language of routine	Runtime options in effect?	Order of runtime options and program arguments
PL/I	Neither PROC OPTIONS(NOEXECOPS) nor SYSTEM(CICS ∣ IMS ∣ TSO) is specified.	runtime options / user parms
	Either PROC OPTIONS(NOEXECOPS) is specified, or NOEXECOPS is not specified but SYSTEM (CICS ∣ IMS ∣ TSO) is.	entire string is user parms
Language Environment-conforming assembler	CEENTRY EXECOPS=ON	runtime options / user parms
	CEENTRY EXECOPS=OFF	entire string is user parms

Finding the return and reason code from the enclave

The following list tells where to look for the return and reason codes that are returned to the parent enclave when a child enclaves terminates:

- EXEC CICS LINK or EXEC CICS XCTL

  If the CICS thread was not terminated, the return code is placed in the optional RESP2 field of EXEC CICS LINK or EXEC CICS XCTL. The reason code is discarded.
- SVC LINK or CMSCALL to a child enclave with a main routine written in any Language Environment-conforming language

  If the process was not terminated, the return code is reported in R15. The reason code is discarded.
- C's `system()` function

  Under MVS, if the target command or program of `system()` cannot be started, “-1” is returned as the function value of `system()`. Otherwise, the return code of the child enclave is reported as the function value of `system()`, and the reason code is discarded. (See *z/OS XL C/C++ Programming Guide* for more information about the `system()` function.)
- FETCH and CALL of a fetchable main

  Normally, the enclave return code and reason code are discarded when control returns to a parent enclave from a child enclave. However, in the parent enclave, you can specify the OPTIONS(ASSEMBLER RETCODE) option of the entry constant for the main procedure of the child enclave. This causes the enclave return code of the child enclave to be saved in R15 as the PL/I return code. You can then interrogate that value by using the PLIRETV built-in function in the parent enclave.

Assembler user exit

An assembler user exit (CEEBXITA) is driven for enclave initialization and enclave termination regardless of whether the enclave is the first enclave created in the process or a nested enclave. The assembler user exit differentiates between first and nested enclave initialization.

Message file

Under MVS, the message file is not closed when control returns from a child enclave.

AMODE considerations

ALL31 should have the same setting for all enclaves within a process. You cannot invoke a nested enclave that requires ALL31(OFF) from an enclave running with ALL31(ON).

Appendix C. Accessibility

Accessible publications for this product are offered through the z/OS Information Center, which is available at www.ibm.com/systems/z/os/zos/bkserv/.

If you experience difficulty with the accessibility of any z/OS information, please send a detailed message to mhvrcfs@us.ibm.com or to the following mailing address:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Accessibility features

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size.

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the z/OS Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1!

(KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

  Note:

  1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
  2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
  3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.
- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

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

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (http://www.ibm.com/software/support/systemsz/lifecycle/)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of Language Environment in z/OS.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Index

Product Number: 5650-ZOS

Printed in USA

SA38-0684-00