IBM Language Environment for z/VSE

Writing Interlanguage Communication Applications

Version 1 Release 4 Modification Level 6

SC33-6686-02

IBM Language Environment for z/VSE

Writing Interlanguage Communication Applications

Version 1 Release 4 Modification Level 6

SC33-6686-02

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

This edition applies to Version 1 Release 4 Modification Level 6 of IBM Language Environment for z/VSE, 5686-CF8, and to any subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC33–6686–01.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the addresses given below.

A form for readers’ comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Deutschland Research & Development GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

You may also send your comments by FAX or via the Internet:

Internet: s390id@de.ibm.com
FAX (Germany): 07031-16-3456
FAX (other countries): (+49)+7031-16-3456

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

Contents

Figures

Tables

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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 of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, U.S.A.

Any pointers in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement. IBM accepts no responsibility for the content or use of non-IBM Web sites specifically mentioned in this publication or accessed through an IBM Web site that is mentioned in this publication.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

```
IBM Deutschland Informationssysteme GmbH
Department 0215
Pascalstr. 100
70569 Stuttgart
Germany
```

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/VSE 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/VSE. Consult the assistive technology documentation for specific information when using such products to access z/VSE interfaces.

z/VSE Information

z/VSE information is accessible using screen readers with the BookServer/Library Server versions of z/VSE books in the Internet library at:

`http://www.ibm.com/servers/eserver/zseries/zos/bkserv/vse.html`

One exception is command syntax that is published in railroad track format. If required, screen-readable copies of z/VSE books with that syntax information are separately available in HTML zipped file form upon request to s390id@de.ibm.com.

Programming Interface Information

This book is intended to help with application programming. This book documents General-Use Programming Interface and Associated Guidance Information provided by IBM Language Environment for z/VSE.

General-Use programming interfaces allow the customer to write programs that obtain the services of IBM Language Environment for z/VSE.

Trademarks and Service Marks

IBM, the IBM logo, ibm.com, Lotus, and Notes are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at `www.ibm.com/legal/copytrade.shtml`

Linux is registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

About This Book

This book is written for application programmers and developers to create and run interlanguage communication (ILC) applications under LE/VSE.

LE/VSE improves ILC between LE/VSE-conforming high-level languages (HLLs) because it creates one common run-time environment and it defines data types and constructs that are equivalent across languages.

For application programming, you will need to use this book, the *LE/VSE Programming Guide* and the *LE/VSE Programming Reference*. Descriptions of these books are found below. You will also need to use the programming guides of the HLLs you are programming with; all are listed in "Where to Find More Information" on page xv.

This book is organized into pairwise chapters that discuss ILC between two languages. There is also a chapter that discusses applications developed in more than two languages (Chapter 5, "Communicating between Multiple HLLs," on page 79). ILC with assembler is discussed in Chapter 6, "Communicating between Assembler and HLLs," on page 87 and ILC under CICS® is discussed in Chapter 7, "ILC under CICS," on page 93.

Throughout this book, the term *routine* is used as a general term to describe a named external routine written in any one of the languages discussed, and with or without internal (contained) subroutines. Hence, in this book, a routine is analogous to an Assembler *CSECT*, a C *function*, a COBOL *program* and a PL/I *procedure*.

Please note that all information provided in this book should be treated as new.

What Is LE/VSE?

LE/VSE is a set of common services and language-specific routines that provide a single run-time environment for applications written in *LE/VSE-conforming* versions of the C, COBOL, and PL/I high level languages (HLLs), and for many applications written in previous versions of COBOL. (For a list of LE/VSE-conforming languages, and a description of compatibility with previous versions of COBOL, see "LE/VSE-Conforming Languages" on page xii.) LE/VSE also supports applications written in assembler language using LE/VSE-provided macros and assembled using High Level Assembler (HLASM).

Prior to LE/VSE, each programming language provided its own separate run-time environment. LE/VSE combines essential and commonly-used run-time services—such as message handling, condition handling, storage management, date and time services, and math functions—and makes them available through a set of interfaces that are consistent across programming languages. With LE/VSE, you can use one run-time environment for your applications, regardless of the application's programming language or system resource needs, because most system dependencies have been removed.

Services that work with only one language are available within language-specific portions of LE/VSE.

LE/VSE consists of:

- Basic routines for starting and stopping programs, allocating storage, communicating with programs written in different languages, and indicating and handling error conditions.
- Common library services, such as math services and date and time services, that are commonly needed by programs running on the system. These functions are supported through a library of callable services.
- Language-specific portions of the common run-time library.

LE/VSE is the implementation of Language Environment on the VSE platform. Language Environment is also offered on platforms z/OS and VM, and on OS/400 as Integrated Language Environment.

LE/VSE-Conforming Languages

An LE/VSE-conforming language is any HLL that adheres to the LE/VSE common interface. Table 1 lists the LE/VSE-conforming language compiler products you can use to generate applications that run with LE/VSE Release 4.

Table 1. LE/VSE-Conforming Languages

Language	LE/VSE-Conforming Language	Minimum Release
C	IBM C for VSE/ESA	Release 1
COBOL	IBM COBOL for VSE/ESA	Release 1
PL/I	IBM PL/I for VSE/ESA	Release 1

Any HLL not listed in Table 1 is known as a *non-LE/VSE-conforming* or, alternatively, a *pre-LE/VSE-conforming* language. Some examples of non-LE/VSE-conforming languages are:

- C/370
- DOS/VS COBOL
- VS COBOL II
- DOS PL/I
- DOS/VS RPG II

Only the following products can generate applications that run with LE/VSE:

- LE/VSE-conforming languages
- HLASM using LE/VSE-provided macros (for details, see *LE/VSE Programming Guide*)
- DOS/VS COBOL and VS COBOL II, with some restrictions (see LE/VSE Compatibility with Previous Versions of COBOL below).

LE/VSE Compatibility with Previous Versions of COBOL

Although DOS/VS COBOL and VS COBOL II are non-LE/VSE-conforming languages, many applications generated with these compilers can run with LE/VSE without recompiling. For details about compatibility, see *LE/VSE Run-Time Migration Guide*.

However relinking under LE/VSE is the *minimum* effort in order to migrate run-time, and involve LE/VSE COBOL-compatibility routines (rather than the old and unsupported library routines of non-LE/VSE conforming COBOL compilers).

This particularily applies to NORES-compiled units or applications that involve former initialization techniques such as ILBDSET0. There are even restrictions with this approach, such as:

- No use of 4-digit dates.
- No exploitation of LE/VSE functionality.
- Interlanguage communication capabilities, and so on.

Therefore you are *strongly recommended* to carry out a (subsequent) full migration to a higher ANSI standard and LE/VSE-conforming COBOL compiler (COBOL for VSE/ESA).

VS COBOL II can also dynamically call some LE/VSE date and time callable services. For details, see *LE/VSE Programming Reference*.

Where to Find More Information

These are the manuals that describe LE/VSE:

Table 2. LE/VSE Publications

Publication	Form Number
LE/VSE Fact Sheet	GC33-6679
LE/VSE Concepts Guide	GC33-6680
LE/VSE Customization Guide	SC33-6682
LE/VSE Programming Guide	SC33-6684
LE/VSE Programming Reference	SC33-6685
LE/VSE C Run-Time Programming Guide	SC33-6688
LE/VSE C Run-Time Library Reference	SC33-6689
LE/VSE Debugging Guide and Run-Time Messages	SC33-6681
LE/VSE Writing Interlanguage Communication Applications	SC33-6686
LE/VSE Run-Time Migration Guide	SC33-6687
LE/VSE Licensed Program Specifications	GC33-6683

These are the z/VSE manuals to which you might need to refer:

Table 3. z/VSE Publications

Publication	Form Number
z/VSE Administration	SC33-8304
z/VSE Messages and Codes, Volume 1	SC33-8306
z/VSE Messages and Codes, Volume 2	SC33-8307
z/VSE Messages and Codes, Volume 3	SC33-8308
z/VSE Planning	SC33-8301
z/VSE System Control Statements	SC33-8305
z/VSE System Macros Reference	SC33-8405
z/VSE System Macros User's Guide	SC33-8407
z/VSE System Upgrade and Service	SC33-8303
VSE/VSAM User's Guide and Application Programming	SC33-8316
VSE/VSAM Commands	SC33-8315
TCP/IP for VSE/ESA IBM Program Setup and Supplementary Information	SC33-6601

These are the manuals that describe IBM C for VSE/ESA:

Table 4. IBM C for VSE/ESA Publications

Publication	Form Number
Licensed Program Specifications	GC09-2421
Installation and Customization Guide	GC09-2422
Migration Guide	SC09-2423

Table 4. IBM C for VSE/ESA Publications (continued)

Publication	Form Number
User's Guide	SC09-2424
Language Reference	SC09-2425
Diagnosis Guide	GC09-2426

These are the manuals that describe IBM COBOL for VSE/ESA:

Table 5. IBM COBOL for VSE/ESA Publications

Publication	Form Number
General Information	GC33-6679
Licensed Program Specifications	GC33-6680
Migration Guide	SC33-6682
Installation and Customization Guide	GC33-6680
Programming Guide	SC33-6684
Language Reference	SC33-6685
Diagnosis Guide	SC33-6684
Reference Summary	SX26-3834

These are the manuals that describe IBM PL/I for VSE/ESA:

Table 6. IBM PL/I for VSE/ESA Publications

Publication	Form Number
Fact Sheet	GC26-8052
Programming Guide	SC26-8053
Language Reference	SC26-8054
Licensed Program Specifications	GC26-8055
Migration Guide	SC33-6684
Installation and Customization Guide	SC26-8057
Diagnosis Guide	SC26-8058
Compile-Time Messages and Codes	SC26-8059
Reference Summary	SX26-3836

These are the manuals that describe Debug Tool for VSE/ESA:

Table 7. Debug Tool for VSE/ESA Publications

Publication	Form Number
User's Guide and Reference	SC26-8797
Installation and Customization Guide	SC26-8798
Fact Sheet	GC26-8925

You might also refer to the ...

z/VSE Home Page

z/VSE has a home page on the World Wide Web, which offers up-to-date information about VSE-related products and services, new z/VSE functions, and other items of interest to VSE users.

You can find the z/VSE home page at:

`http://www.ibm.com/servers/eserver/zseries/zvse/`

You can also find VSE User Examples (in zipped format) at:

`http://www.ibm.com/servers/eserver/zseries/zvse/downloads/samples.html`

Softcopy Publications

The following collection kit contains the LE/VSE and LE/VSE-conforming language product publications:
VSE Collection, SK2T-0060

Summary Of Changes

- Recommendations have been provided for defining interlanguage communication between COBOL and C routines. See "COBOL Dynamically Calling C" on page 8 and "Passing Data between C and COBOL" on page 8.

Chapter 1. Getting Started with LE/VSE 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 each language maps data, how conditions are handled, or how data can be called and received by each language.

This book helps you create ILC applications using LE/VSE-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, reentrancy, 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–COBOL and C–PL/I ILC. Then read Chapter 5, "Communicating between Multiple HLLs," on page 79 for additional information on developing multiple-language applications. If you have ILC with assembler or under CICS, see Chapter 6, "Communicating between Assembler and HLLs," on page 87 and Chapter 7, "ILC under CICS," on page 93.

The Benefits of ILC under LE/VSE

Performance improves under the single run time environment. LE/VSE ILC applications run in one environment, giving you cooperative ILC support for running mixed-language applications, without the overhead of multiple libraries and library initializations.

Environment tailored to HLLs at initialization. When you run your ILC applications in LE/VSE, the initialization process establishes the LE/VSE environment, tailored to the set of HLLs in the main phase. ILC applications follow the LE/VSE program model, making program execution consistent and predictable.

Coordinated cleanup at termination. LE/VSE 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. All languages participating in the ILC application handle conditions cooperatively, making exception and condition handling consistent and predictable.

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 *LE/VSE 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 LE/VSE-conforming**.

   Each chapter gives the basics of what you need to do to make your ILC applications LE/VSE-conforming. For detailed information on migration, see the language migration guides, as listed in "Where to Find More Information" on page xv.

3. **Decide which language will have the main routine**.

   LE/VSE 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 5, "Communicating between Multiple HLLs," on page 79 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 LE/VSE 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 between C and COBOL

This chapter describes LE/VSE's support for C and COBOL ILC applications. If you are running a C–COBOL ILC application under CICS, you should also consult Chapter 7, "ILC under CICS," on page 93.

General Facts about C–COBOL ILC

- `#pragma linkage(...,COBOL)` is required in all *statically-called* C routines.
- A C–COBOL application can be constructed to be reentrant.
- LE/VSE does not support the passing of return codes between C and COBOL routines in an ILC application. See "Return Codes" on page 11 for more information.

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.

LE/VSE ILC Support

LE/VSE provides ILC support between the following combinations of C and COBOL:

Table 8. Supported Languages for LE/VSE ILC

HLL Pair	C	COBOL
C–COBOL	• C/VSE	• COBOL/VSE Release 1 • VS COBOL II Release 3 and later

Migrating ILC Applications

Relinking

You must relink ILC applications that contain phase(s) with VS COBOL II programs. The relink is necessary in order to remove old VS COBOL II library routines.

If you relink a VS COBOL II NORES program with LE/VSE, you should explicitly include IGZENRI to ensure the correct versions of the library routines are included. If you do not include IGZENRI, the link-edited phase will be unnecessarily large and the linkage editor will produce messages indicating duplicate sections.

Recompiling

You must recompile all pre-LE/VSE-conforming C programs with an LE/VSE-conforming C compiler.

Compiling and Linking Considerations

Compiling

Compile your COBOL/VSE program using the RMODE(ANY) compiler option. Compile your VS COBOL II program using the RES and RENT compiler options.

Linking

When link-editing ILC applications, you can have only one main routine, or one fetchable entry point. You should present your main routine to the linkage editor first in order to avoid an incorrectly chosen entry point. When you are linking modules to be fetched, you should present your fetchable entry point first. See "Determining the Main Routine" for information on how to identify the main routine.

Determining the Main Routine

In LE/VSE, 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 as a subroutine 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. Within any single VS COBOL II or COBOL/VSE enclave, a recursively-called main program is not permitted

Table 9 describes how C and COBOL identify the main routine.

Table 9. 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 in the enclave to run, it is a main program. The main program cannot be called recursively within a single enclave in VS COBOL II.

An entry point is defined for each supported HLL. Table 10 identifies the desired entry point. The table assumes that your code has been compiled using the LE/VSE-conforming compilers.

Table 10. Determining the Entry Point

HLL	Main Entry Point	Fetched Entry Point
C	CEESTART	CEESTART or routine name if `#pragma linkage(...,FETCHABLE)` is used without pre-linking
COBOL	Name of the first object program to get control in the object module	Program name

C and COBOL routines that make up an ILC application are executed together in a single run unit (the equivalent of an LE/VSE enclave). However, unlike in earlier versions of COBOL (VS COBOL II and DOS/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 LE/VSE enclave.

Declarations

A C `#pragma linkage(...,COBOL)` directive is required for both static calls and dynamic calls of mixed C and COBOL load modules. For dynamic calls of C-only load modules, a `#pragma linkage(...,FETCHABLE)` is required instead.

All entry declarations are made in the C code, both in the case where C calls COBOL and vice versa. The C `#pragma linkage(...,COBOL)` directive lets the C compiler generate parameter lists for COBOL or accept them from COBOL.

C-only (not mixed languages) routines or mixed COBOL/C subroutines that:
- have a C routine as the entry point that are called dynamically, and
- receive parameters from a calling COBOL program,

must successfully handle parameter-addressing themselves. This also involves taking into account the way in which the COBOL CALL statement has been coded.

The `#pragma linkage(...,COBOL)` directive has the following format:

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

routine_name can be up to eight characters. *routine_name* is either the COBOL program being called by C, or the C function being called by COBOL.

Declaring C–COBOL ILC

Declaration for C Calling COBOL:

C Function	COBOL Subroutine

`#pragma linkage(CBLRTN,COBOL)`
`void CBLRTN(int);`
`main() {`
`  int p1;`
`  CBLRTN(p1);`
`};`
	`CBL RMODE(ANY)`
	`       IDENTIFICATION DIVISION.`
	`       PROGRAM-ID. CBLRTN.`
	`       ENVIRONMENT DIVISION.`
	`       DATA DIVISION.`
	`       LINKAGE SECTION.`
	`       01 P1 PIC S9(9) BINARY.`
	`       PROCEDURE DIVISION USING P1.`
	`           DISPLAY P1`
	`           GOBACK.`
	`       END PROGRAM CBLRTN.`

Declaration for COBOL Calling C:

COBOL Program	C Subroutine

`CBL RMODE(ANY),APOST`
`       IDENTIFICATION DIVISION.`
`       PROGRAM-ID. COBRTN.`
`       ENVIRONMENT DIVISION.`
`       DATA DIVISION.`
`       WORKING-STORAGE SECTION.`
`       01 P1 PIC S9(9) USAGE IS BINARY.`
`       PROCEDURE DIVISION.`
`           CALL 'CFUNC' USING BY CONTENT P1.`
`           GOBACK.`
`       END PROGRAM COBRTN.`
	`#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

Table 11 describes the types of calls between C and COBOL that LE/VSE allows:

Table 11. Calls Permitted for C and COBOL

ILC Direction	Static Call	Dynamic Call/Fetch
C to COBOL	Yes[1]	Yes[1]
COBOL to C	Yes	Yes[2]

Notes:

[1] The C return type from all calls to COBOL must be `void`.

[2] A C-only routine can be naturally reentrant or constructed reentrant. If you are using constructed reentrancy:
- Entry-point C modules that have COBOL statically linked to them *cannot* be non-reentrant.
- The COBOL subroutine must be reentrant and prelinked together with the C routine(s).

For an understanding of C reentrancy and "writable static", see *IBM C for VSE/ESA User's Guide*

For information about calls between COBOL/VSE programs and programs compiled under previous versions of COBOL, see *IBM COBOL for VSE/ESA Migration Guide*

Dynamic Call/Fetch Considerations

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 phase that you specify into main storage. The phase can be invoked subsequently from a C application (see *LE/VSE C Run-Time Library Reference* for more information about `fetch()`). In COBOL, you can use the dynamic CALL statement to dynamically load a phase into main storage (see *IBM COBOL for VSE/ESA Programming Guide* for more information about the CALL statement).

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 forth).

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 phase and a fetched C phase, when the ILC routines adhere to the rules stated in the C programming guides. Figure 1 on page 7 shows a C phase fetching a C–COBOL phase.

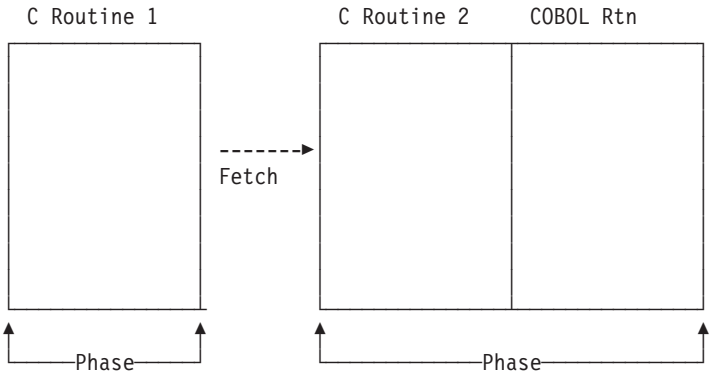


Figure 1. C Fetching C–COBOL Phase

If any ILC occurs within the fetched phase, the fetched phase should not be released. It is released by LE/VSE termination processing.

C Fetching COBOL

You can use the C `fetch()` function to fetch a COBOL routine and invoke it later using a function pointer. The declaration of a COBOL fetched routine within a C routine is shown in Figure 2.

```
typedef void CBL_FUNC();
#pragma linkage (CBL_FUNC, COBOL)
   :
CBL_FUNC *fetch_ptr;
fetch_ptr = (CBL_FUNC*) fetch("COBEP");   /* fetch the routine */
fetch_ptr(args);                          /* call COBEP        */
```

Figure 2. C Fetching a COBOL Routine

You cannot `release()` a COBOL routine 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 phase and a dynamically called COBOL phase, when the ILC routines adhere to the rules stated in *IBM COBOL for VSE/ESA Programming Guide*. Figure 3 on page 8 shows a COBOL phase fetching a COBOL–C phase.

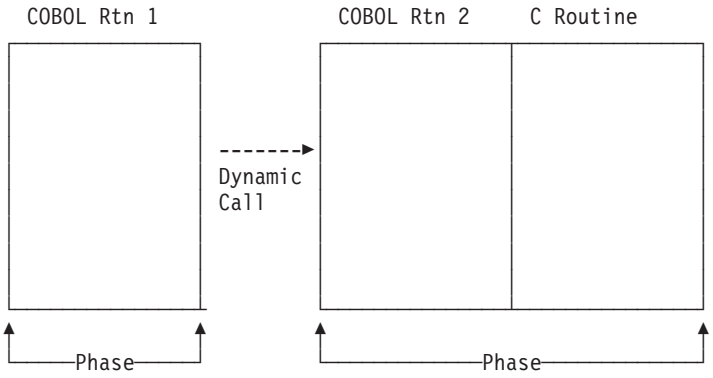


Figure 3. COBOL Dynamically Calling COBOL–C Phase

- The called PHASE can be CANCELed providing:
  - The PHASE has been prelinked (both C and COBOL routines simultaneously).
  - The COBOL subroutine has been compiled RENT.

  Otherwise, a COBOL CANCEL should not be used.
- If the C subroutine is the entry point, then that routine must have the `#pragma linkage(...,fetchable)` directive specified.
- You cannot successfully issue a C release() against a fetch'd COBOL/C or COBOL PHASE.

COBOL Dynamically Calling C

A COBOL/VSE routine can dynamically call a C routine that has been compiled with the `#pragma linkage(...,fetchable)` C/VSE compiler directive. This C routine is the entry point of the called PHASE.

These C routine(s):
- Can be non-reentrant.
- Can be naturally reentrant.
- Can use constructed reentrancy.

When compiling multiple functions in one compilation that will be dynamically called by COBOL, only the C function specified as the entry point requires the `#pragma linkage(...,fetchable)` directive.

Any COBOL dynamically-called C module that requires *writable static support* and *use of the COBOL CANCEL verb* must:
- have been prelinked,
- specify the `#pragma linkage(...,fetchable)` compiler directive on the appropriate C function.

Passing Data between C and COBOL

In VS COBOL II and COBOL/VSE you can pass parameters in two ways:

By reference (indirect):
COBOL BY REFERENCE

By value (indirect):
COBOL BY CONTENT

The term *by value* means that a temporary copy of the argument is passed to the called routine. Any changes to the parameter made by the called routine will not alter the original parameter passed by the calling routine.

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.

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

When data is passed between C and COBOL, the `#pragma linkage(...,COBOL)` directive causes the C compiler to generate 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.

You should be aware that the current C/VSE compiler does *not* support the use of both the COBOL *and* fetchable options on the `#pragma linkage` statement relating to the *same function*. This means that any COBOL applications calling C functions that are not fetchable *cannot use*:

- the writable static support, or
- the COBOL CANCEL verb on any dynamically-loaded C subroutines.

If "fetchable" *is* required, the C subroutine should be written so it can address the parameters passed from the COBOL caller *without* relying on the behavior of the `#pragma linkage(xxxx,COBOL)` compiler directive.

Passing Data by Value (Indirect) between C and COBOL

Copies of variables can be passed between C and COBOL routines. On return, the original value of the variables remains unchanged regardless of how the copy might 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(...,COBOL)` 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 on page 10 shows the supported data types for passing by value (indirect). For examples illustrating how these are passed between C and COBOL, refer to "Equivalent Data Types—C to COBOL" on page 11.

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 routine; any changes to the parameter made by the called routine will 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). For examples illustrating how these are passed between C and COBOL, refer to "Equivalent Data Types—C to COBOL" on page 11.

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)

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)

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)

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

Table 13. Supported Data Types Passed by Reference (Indirect) (continued)

C	COBOL
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.

Refer to "Sample ILC Application" on page 26 to see how string data is passed between C and COBOL.

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 C compile-time option AGGREGATE and the COBOL compiler option MAP provide a layout of structures to help you perform the mapping.

Return Codes

The passing of return codes between C (using the `return()` statement) and COBOL (using the RETURN-CODE special register) is not supported within an ILC application under LE/VSE.

However, it is possible to return data from a routine in one language to a routine in the other language using a suitably declared/defined argument which is passed by reference. For example, in the illustration "Fullword Integer" on page 12, the argument "Y" could be used to communicate the contents of the COBOL RETURN-CODE special register back to C.

Data Equivalents

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

Equivalent Data Types—C to COBOL

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

Char

Sample C Usage

```
#pragma linkage (cobrtn,COBOL)
#include <stdio.h>
void cobrtn (char*);
int main()
{
  char x;
  x='a';
  cobrtn(&x);    /* x by reference */
  printf("x = %c\n", x);
}
```

COBOL Subroutine

```
CBL RMODE(ANY),APOST
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBRTN.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       LINKAGE SECTION.
       01 X PIC X.
       PROCEDURE DIVISION USING X.
           DISPLAY X
           MOVE 'B' TO X.
           GOBACK.
       END PROGRAM COBRTN.
```

Output:

```
a
x = B
```

Short Integer

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 */
  printf("x = %d\n", x);
}
```

COBOL Subroutine

```
CBL RMODE(ANY)
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBRTN.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       LINKAGE SECTION.
       01 X PIC S9(4) BINARY.
       PROCEDURE DIVISION USING X.
           DISPLAY X
           COMPUTE X = X + 1.
           GOBACK.
       END PROGRAM COBRTN.
```

Output:

```
0005
x = 57
```

Fullword Integer

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 */
  printf("x = %d, y = %d\n", x, y);
}
```

COBOL Subroutine

```
CBL RMODE(ANY)
       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
           COMPUTE X = X + 1.
           COMPUTE Y = Y + 1.
           GOBACK.
       END PROGRAM COBRTN.
```

Output:

```
000000005000000006
x = 5, y = 7
```

Double-Precision Floating Point

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 */
  printf("x = %f, y = %f\n", x, y);
}
```

COBOL Subroutine

```
CBL RMODE(ANY)
       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
           COMPUTE X = X + 1.
           COMPUTE Y = Y + 1.
           GOBACK.
       END PROGRAM COBRTN.
```

Output:

```
 .31415926500000000E 01 .41415926500000000E 01
x = 3.141593, y = 5.141593
```

Structure

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 */
  printf("struc1.s1 = %d\n", struc1.s1);
  printf("struc2.s1 = %d\n", struc2.s1);
}
```

COBOL Subroutine

```
CBL RMODE(ANY)
       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
           COMPUTE S11 = S11 + 1.
           COMPUTE S21 = S21 + 1.
           GOBACK.
       END PROGRAM COBRTN.
```

Output:

```
00000000100000000200000003000000004
struc1.s1 = 1
struc2.s1 = 4
```

Array

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 */
  printf("array[0] = %d\n", array[0]);
}
```

COBOL Subroutine

```
CBL RMODE(ANY)
       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)
           COMPUTE ELE(1) = ELE(1) + 1.
           GOBACK.
       END PROGRAM COBRTN.
```

Output:

```
000000001000000002
array[0] = 2
```

Fixed Decimal

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 */
  printf("x = %D(5,2)\n", x);
}
```

COBOL Subroutine

```
CBL RMODE(ANY)
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  COBRTN.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       LINKAGE SECTION.
       01 X PIC 999V99 COMP-3.
       PROCEDURE DIVISION USING X.
           DISPLAY X
           COMPUTE X = X + 1.
           GOBACK.
       END PROGRAM COBRTN.
```

Output:

```
12345
x = 124.45
```

Equivalent Data Types—COBOL to C

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

Fullword Integer

Sample COBOL Usage

```
CBL RMODE(ANY),APOST
       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.
           DISPLAY X Y
           GOBACK.
       END PROGRAM COBRTN.
```

C Subroutine

```
#pragma linkage (centry,COBOL)
#include <stdio.h>
void centry (int x, int *y)
{
  printf("x = %d, y = %d\n",x,*y);
  ++x, ++*y;
  return;
}
```

Output:

```
x = 1, y = 2
000000001000000003
```

Double-Precision Floating Point

Sample COBOL Usage

```
CBL RMODE(ANY),APOST
       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.
           DISPLAY X Y
           GOBACK.
       END PROGRAM COBRTN.
```

C Subroutine

```
#pragma linkage (centry,COBOL)
#include <stdio.h>
void centry (double x, double *y)
{
  printf("x = %f, y = %f\n",x,*y);
  ++x, ++*y;
  return;
}
```

Output:

```
x = 3.141593, y = 4.141593
 .31415926500000000E 01 .51415926500000000E 01
```

Structure

Sample COBOL Usage

```
CBL RMODE(ANY),APOST
       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.
           DISPLAY S11 S12 S21 S22
           GOBACK.
       END PROGRAM COBRTN.
```

C Subroutine

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

Output:

```
a=1, b=2, c=3, d=4
00000000100000000200000004000000004
```

Fixed Decimal

Sample COBOL Usage

```
CBL RMODE(ANY),APOST
       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.
           DISPLAY X
           GOBACK.
       END PROGRAM COBRTN.
```

C Subroutine

```
#pragma linkage (centry,COBOL)
#include <stdio.h>
#include <decimal.h>
void centry (decimal(5,2) x)
{
  printf("x = %D(5,2)\n",x);
  ++x;
  return;
}
```

Output:

```
x = 123.45
12445
```

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 phase; under COBOL, it is the enclave (or run unit). Figure 4 on page 17 and Figure 5 on page 17 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 phase a different name, you can change the language mix in the application later, and your application still behaves as you expect it to.

Name Scope of External Data in a C Application

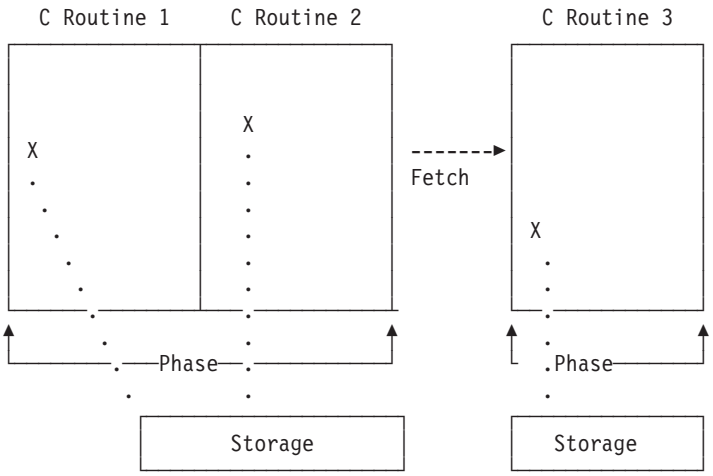


Figure 4. Name Scope of External Variables for C Fetch

In Figure 4, external data declared in C Routine 1 maps to that declared in C Routine 2 in the same phase. When a fetch to C Routine 3 in another phase is made, the external data does not map, because the name scope of external data in C is the phase.

Name Scope of External Data in a COBOL Run Unit

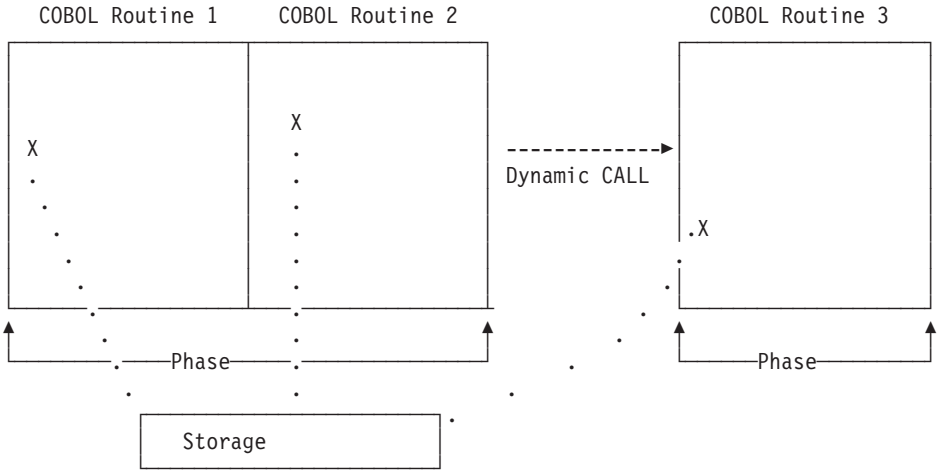


Figure 5. Name Scope of External Variables for COBOL Dynamic CALL

In Figure 5, Routines 1, 2, and 3 comprise a COBOL run unit. External data declared in COBOL Routine 1 maps to that declared in COBOL Routine 2 in the

same phase. When a dynamic CALL to COBOL Routine 3 in another phase is made, the external data still maps, because the name scope of external data in COBOL is the enclave.

The scope of non-reentrant external variables within COBOL routines is the phase.

Name Space of External Data

In programming languages, the *name space* is defined as the portion of a phase 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 6 and Figure 7 illustrate that within the same phase, the name space of COBOL routines is the same. However, the name spaces of a COBOL routine and a C routine within the same phase are not the same. If you give external data the same name in both languages, an incompatibility in external data mapping can occur.

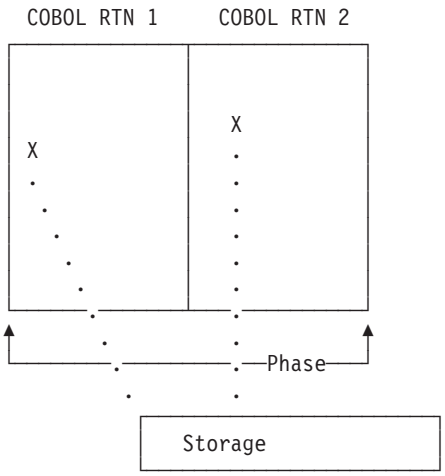


Figure 6. Name Space of External Data for COBOL Static CALL to COBOL

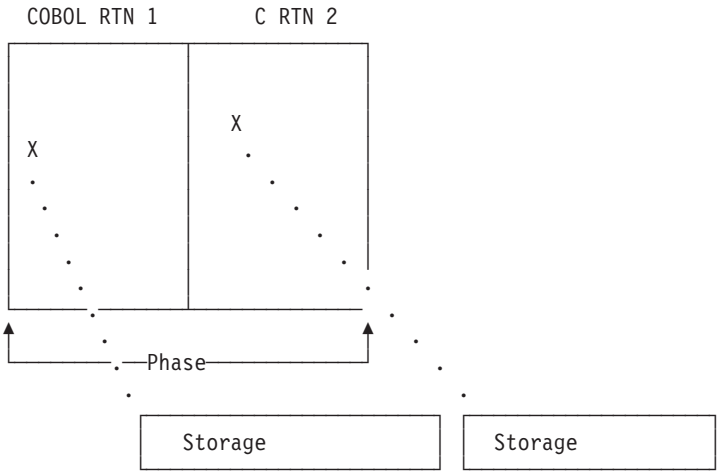


Figure 7. Name Space of External Data in COBOL Static CALL to C

File Sharing

Except for the file specified by the LE/VSE MSGFILE run-time option, LE/VSE provides no support for files that are open in C and COBOL at the same time. You must manage all such files to ensure that no conflicts arise. Performing I/O operations on the same filename, other than the one specified by the LE/VSE MSGFILE run-time option, might cause abnormal termination or data corruption of the file.

LE/VSE provides no support for using C to read from or write to a file created using COBOL, or vice versa. To do this, you must ensure that the file is in a compatible format. For information on file formats, see the respective programming guides.

Note: A guide to the types of considerations necessary when sharing file data between languages can be found in Chapter 4, "Communicating between COBOL and PL/I," section "File Sharing" on page 67.

Directing Output in ILC Applications

Under LE/VSE, COBOL run-time messages and other related output is directed to the destination specified in the LE/VSE run-time option MSGFILE. User-specified output, such as output from DISPLAY UPON SYSOUT, is directed to the destination specified in the COBOL OUTDD compiler option, which defaults to SYSOUT.

Since the default destination for the LE/VSE MSGFILE is SYSLST, and the default OUTDD filename is treated as SYSLST also, all output from COBOL is by default directed to the same destination and will thus be interspersed. To separate run-time messages and other related output from the user-specified output, the filename specified in the LE/VSE MSGFILE run-time option must be different from the filename specified in the COBOL OUTDD compiler option. See *IBM COBOL for VSE/ESA Programming Guide* for details of using the OUTDD compiler option.

Note: The VS COBOL II compiler under VSE does not support the OUTDD compiler option. When this compiler is used, all output from the DISPLAY statement is sent to SYSLST.

Under LE/VSE, C run-time output such as run-time messages is directed to the destination specified in the LE/VSE run-time option MSGFILE. `stderr` output is also directed to the destination of the MSGFILE option. `stdout` output is by default directed to SYSLST.

Since the default destination for the LE/VSE MSGFILE is also SYSLST, all output from C is by default directed to the same destination and will thus be interspersed. To separate the output, either `stdout` or the LE/VSE MSGFILE must be directed to a file different from the other. For information on redirecting C output, see *LE/VSE C Run-Time Programming Guide*.

Note: In both C and COBOL, when run-time output and user-specified output from one or both languages is directed to the same destination such as SYSLST, each language must manage its own I/O buffers, line counters, etc., for its own user-specified output.

For additional information regarding the LE/VSE MSGFILE run-time option, see *LE/VSE Programming Guide*.

C–COBOL Condition Handling

This section provides two scenarios of condition handling behavior in a C–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 22. If an exception occurs in a COBOL routine, the set of possible actions is as described in "Exception Occurs in COBOL" on page 23.

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 routine, 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 LE/VSE condition handling, see *LE/VSE Programming Guide*LE/VSE Programming Guide

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–COBOL enclave. In LE/VSE ILC, you can issue the language construct to terminate the enclave from a COBOL or C routine.

C

Examples of C language constructs that terminate the enclave are: `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

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

- STOP RUN

  STOP RUN is equivalent to the C `exit()` function. If you code a 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 CEE5ABD

  Calling CEE5ABD causes T_I_U Termination Imminent due to an Unhandled condition) to be signaled. Condition handlers are given a chance to handle the abend. If the abend remains unhandled, normal LE/VSE termination activities occur. For example, the C atexit list is honored and the LE/VSE assembler user exit gains control.

  User-written condition handlers written in COBOL must be compiled with COBOL/VSE.

Exception Occurs in C

This scenario describes the behavior of an application that contains a C and a COBOL routine. Refer to Figure 8 throughout the following discussion.

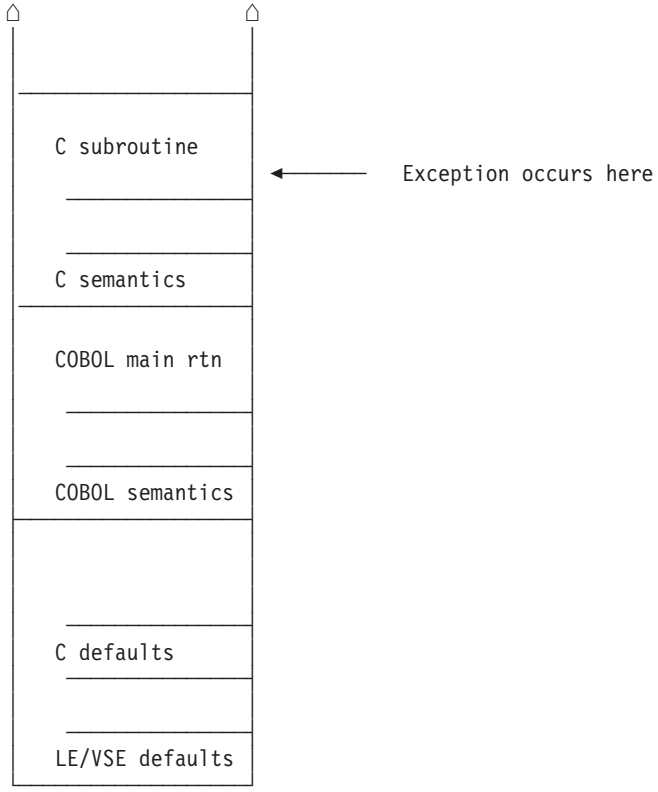


Figure 8. Stack Contents When the Exception Occurs in C

In this scenario, a COBOL main routine invokes a C subroutine. An exception occurs in the C subroutine. The stack contains what is shown in Figure 8.

The actions taken follow the three LE/VSE 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 abend corresponding to the `signal(SIGABND)` or the LE/VSE message 3250 is not ignored. The enclave is terminated.
   - The exception is one of those listed as masked in Table 28 on page 99.
   - You did not specify any action, but the default action for the condition is `SIG_IGN` (see Table 28 on page 99).
   - You are running under CICS and a CICS handler is pending.

   If none of these things are true, 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 routine; see "CEEMRCR and COBOL" on page 25 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 28 on page 99) 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 routine; see "CEEMRCR and COBOL" on page 25 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 LE/VSE default actions take place, as described in Table 27 on page 99.

7. If the condition is of severity 2 or above, LE/VSE 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, LE/VSE terminates the thread.

Exception Occurs in COBOL

This scenario describes the behavior of an application that contains a COBOL and a C routine. Refer to Figure 9 on page 24 throughout the following discussion.

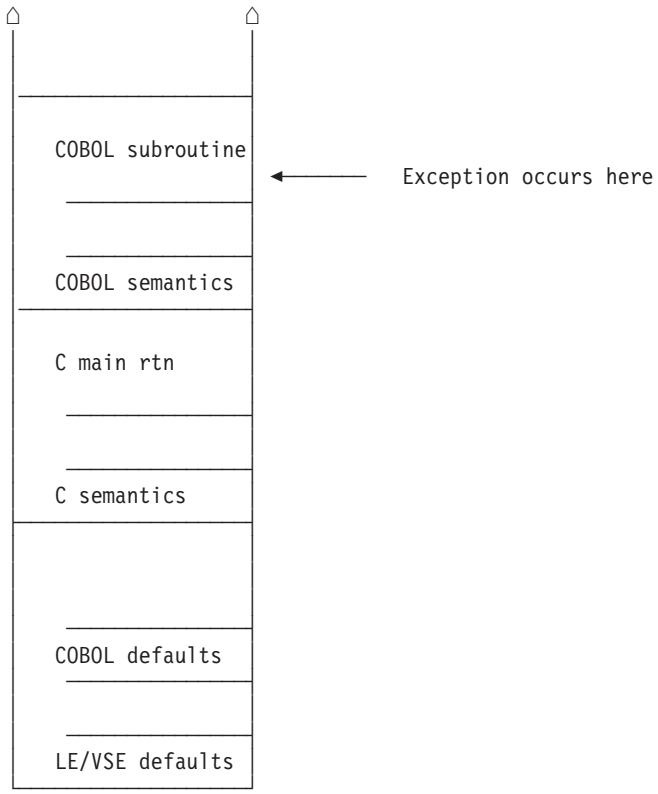


Figure 9. Stack Contents When the Exception Occurs in COBOL

In this scenario, a C main routine invokes a COBOL subroutine. An exception occurs in the COBOL subroutine. The stack contains what is shown in Figure 9.

The actions taken follow the three LE/VSE 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, 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, 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.
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 routine; see "CEEMRCR and COBOL" 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, LE/VSE default actions take place, as described in Table 27 on page 99.
7. If the condition is of severity 2 or above, LE/VSE 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, LE/VSE 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 not one of the following, the routine remains active. A recursion error is raised if you attempt to enter the routine again.

- A VS COBOL II or COBOL/VSE routine compiled with the CMPR2 option
- A VS COBOL II or COBOL/VSE routine compiled with the NOCMPR2 option that does not use nested routines
- A VS COBOL II or COBOL/VSE routine 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

In addition, if the COBOL routine has the INITIAL attribute and contains files, the files are closed. (COBOL supports VSAM and SAM files and these files are closed.)

Sample ILC Application

Figure 10 and Figure 11 on page 27 contain an example of an ILC application. The C routine C1 dynamically calls the COBOL routine CBL1. CBL1 statically calls C routine C2.

EDCCCB

```
 /* Module/File Name: EDCCCB */

 /*****************************************************************/
 /*   Illustration of Interlanguage Communication between C       */
 /*   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] = "Hello World";

  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 CBL1               */
    rtn_ptr (&short_int, &long_int, &floatpt, string);
  fprintf(stderr,"main ENDED\n");
} /* end of main */
```

Figure 10. Dynamic Call from C to COBOL Routine

IGZTILCC

```
CBL LIB,APOST,RMODE(ANY)
CBL NAME
      *Module/File Name:  IGZTILCC

      ****************** 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 ' char-string LOW-VALUE
                   DELIMITED BY SIZE INTO msg-string
      * MAKE A STATIC CALL TO C FUNCTION
            CALL 'C2' USING var1, msg-string.

            DISPLAY 'CBL1 ENDED'.
            GOBACK.
```

Figure 11. Static CALL from COBOL to C Routine

EDCCCB2

```
 /* Module/File Name: EDCCCB2 */
 /* C2 routine example */

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

#pragma linkage(C2,COBOL)
int C2(int*, char*);
int C2(int *num, char* strng)
{
  fprintf(stderr, "num is %d, string is %s\n", *num,strng);
  return(20);
} /* end of main */
```

Figure 12. Statically Called C Routine

Chapter 3. Communicating between C and PL/I

This chapter describes LE/VSE's support for C and PL/I ILC applications. If you are running a C–PL/I ILC application under CICS, you should also consult Chapter 7, "ILC under CICS," on page 93.

General Facts about C–PL/I ILC

- LE/VSE does not support the passing of return codes between C and PL/I routines in an ILC application. See "Return Codes" on page 35 for more information.
- 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.

Preparing for C–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.

LE/VSE ILC Support

Table 14. Supported Languages for LE/VSE ILC

HLL Pair	C	PL/I
C–PL/I	• C/VSE	• PL/I VSE

Migrating C–PL/I ILC Applications

In general, you need to convert (if necessary) and recompile existing C–PL/I ILC applications.

Determining the Main Routine

In LE/VSE, 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 might not be detected by LE/VSE and can cause unpredictable results.

An entry point is defined for each supported HLL. Table 15 on page 30 identifies the desired entry point. The table assumes that your code has been compiled using the LE/VSE-conforming compilers.

Table 15. Determining the Entry Point

HLL	Main Entry Point	Fetched Entry Point
C	CEESTART	CEESTART or routine name if `#pragma linkage(...,FETCHABLE)` is used.
PL/I	CEESTART	CEESTART or routine name if OPTIONS(FETCHABLE) is used.

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 executable phase. If not, `#pragma linkage(...,FETCHABLE)` must be specified in C, or the FETCHABLE option must be specified on the PROCEDURE statement in PL/I. You cannot have more than one entry point in an executable ILC phase with `#pragma linkage(...,FETCHABLE)` or PL/I FETCHABLE option on the PROCEDURE statement. This error is not detected by LE/VSE, but can cause unpredictable results.

Reentrancy Considerations

A PL/I–C application can be constructed to be reentrant. You should compile all PL/I routines in an ILC application using the REENTRANT option of the OPTIONS attribute of the PROCEDURE statement.

Declarations

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 routine 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 linkage(...,PLI)` directive. 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 linkage(...,PLI)` preprocessor directive.

Note: Failure to use the C `#pragma linkage(...,PLI)` preprocessor directive will cause the C compiler to generate parameter lists which are incompatible or in conflict with the recommendations and guidelines found in this book.

Declaration for C Calling PL/I

C Function

```
#pragma linkage (PFUNC, PLI)
double PFUNC( double ); /* function prototype */

int main()
{
  double val,result;

  val=6.2;
  result=PFUNC(val);
  printf("result=%f\n",result);
}
```

PL/I Subroutine

```
%process mar(1,72);
PFUNC: Proc(arg) options(reentrant)
       returns(float binary(53));
  Dcl arg float binary(53);
  Dcl SYSPRINT file;
  Put Skip List ('arg =',arg);
  Return (34.0);
  End;
```

Declaration for PL/I Calling C

PL/I Routine	C Subroutine

`%PROCESS MAR(1,72);`
`PLIPROG:  Proc options(main, reentrant);`
`  Dcl SYSPRINT file;`
`  Dcl cfunc external entry`
`  returns(fixed bin(31);`
`  Dcl arg fixed bin(31);`
`  Dcl a fixed bin(31);`
`  Arg = 10;`
`  A = cfunc(arg);`
`  Put skip list ('A =', A);`
`  End;`
	`#pragma linkage(CFUNC, PLI)`
	`#include <stdio.h>`
	
	`int CFUNC( int parm )`
	`{`
	`  printf("parm = %d\n", parm);`
	`  return (5);`
	`}`

Calling between C and PL/I

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

Types of Calls Permitted

Table 16 describes the types of calls between C and PL/I that LE/VSE allows:

Table 16. Calls Permitted for C and PL/I

ILC Direction	Static Call	Dynamic Call/Fetch
C to PL/I	Yes	Yes
PL/I to C	Yes	Yes[1]

Notes:
[1] C must be non-reentrant or naturally reentrant.

For an understanding of C reentrancy and "writable static", see *IBM C for VSE/ESA User's Guide*

Dynamic Call/Fetch Considerations

Both PL/I and C can specify only one fetchable entry point for an entire phase.

If a phase is introduced as a result of the PL/I FETCH statement or the C `fetch()` function and the phase contains any ILC or the fetching and fetched routines are written in different languages, then the phase cannot be deleted using a corresponding PL/I RELEASE statement or the C `release()` function.

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 PL/I routine is dynamically introduced into the enclave as a result of a fetch, the fetch restrictions described in *IBM PL/I for VSE/ESA Language Reference* apply.

If a PL/I routine is to be dynamically loaded, you must either:

- Specify the routine name as the entry point when you link-edit it, or
- Specify OPTIONS(FETCHABLE) on the PROCEDURE statement and recompile.

The declaration of a PL/I fetched routine within a C routine is shown in Figure 13.

```
typedef int PLIFUNC();
#pragma linkage (PLIFUNC, PLI)
   ⋮
PLIFUNC *fetch_ptr;
fetch_ptr = (PLIFUNC*) fetch("PLIENT");  /* fetch the routine */
fetch_ptr(args);                         /* call PLIENT      */
```

Figure 13. C Fetching a PL/I Routine

PL/I Fetching C

A PL/I routine can fetch a C routine or another PL/I routine that is statically linked to a C routine. 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 14.

```
DCL CENTRY EXTERNAL ENTRY;  /* declare C entry point */
   ⋮
FETCH CENTRY;         /* fetch the routine     */
CALL CENTRY(args);    /* call routine          */
```

Figure 14. 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 will 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 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.

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 on data descriptors, see *IBM PL/I for VSE/ESA Programming Guide*

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 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 `#pragma linkage(...,PLI)` directive.

Receiving Reference Parameters in C

If you do not enclose in parentheses the argument you pass from a PL/I routine 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/I with 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 17 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 17. Supported Data Types between C and PL/I Using C Pointers (by Reference)

C	PL/I
signed short int	REAL FIXED BINARY(15,0)

Table 17. Supported Data Types between C and PL/I Using C Pointers (by Reference) (continued)

C	PL/I
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 Without Using Explicit C Pointers (by Value)

Table 18 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 18. 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 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.
decimal(n,p)	FIXED DECIMAL(n,p)
Note: Data storage alignment must match.	

Strings Passed 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 of *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.

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 C and PL/I aggregate mapping, see *IBM C for VSE/ESA Language Reference* and *IBM PL/I for VSE/ESA Language Reference* respectively.

Return Codes

The passing of return codes between C and PL/I using the PL/I PLIRETC or PLIRETV function is not supported within an ILC application under LE/VSE.

However, it is possible to return data from both a C routine (using the C `return()` statement) and from a PL/I routine (using the PL/I RETURN statement). For example, in the illustration "Fullword Integer" on page 36, the value "ARG" returned by PL/I could instead contain the return code obtained using the PLIRETV function.

Data Equivalents

This section describes how C and PL/I data types correspond to each other.

Equivalent Data Types—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 32, which describes how a C routine can receive parameters that are passed by value and by reference.

Short Integer

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
short int *cpli(short int *);
main()
{
   short int x=5, y;
   y = *cpli(&x);  /* by reference */
   printf("x = %d, y = %d\n", x, y);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG) RETURNS (POINTER);
      Dcl SYSPRINT file;
      DCL ARG REAL FIXED BIN(15,0);
      Put Skip List ('ARG =', ARG);
      ARG = ARG + 1;
      RETURN (ADDR(ARG));
      END;
```

Output:

```
ARG =                          5
x = 6, y = 6
```

Note: Because `short int` is an example of a parameter which must be passed using an explicit C pointer, you cannot code `y = cpli(x)`, passing `x` by value.

Fullword Integer

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
int cpli(int);
main()
{
   int x=5, y;
   y = cpli(x);  /* by value */
   printf("x = %d, y = %d\n", x, y);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG) RETURNS (FIXED BIN(31));
      Dcl SYSPRINT file;
      DCL ARG FIXED BIN(31);
      Put Skip List ('ARG =', ARG);
      ARG = ARG + 1;
      RETURN (ARG);
      END;
```

Output:

```
ARG =                          5
x = 5, y = 6
```

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
int *cpli(int *);
main()
{
   int x=5, y;
   y = *cpli(&x);  /* by reference */
   printf("x = %d, y = %d\n", x, y);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG) RETURNS (POINTER);
      Dcl SYSPRINT file;
      Dcl ADDR builtin;
      DCL ARG FIXED BIN(31);
      Put Skip List ('ARG =', ARG);
      ARG = ARG + 1;
      RETURN (ADDR(ARG));
      END;
```

Output:

```
ARG =                          5
x = 6, y = 6
```

Double-Precision Floating Point

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
double cpli(double);
main()
{
   double x=12.5, y;
   y = cpli(x);  /* by value */
   printf("x = %f, y = %f\n", x, y);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG) RETURNS (FLOAT BINARY(53));
      Dcl SYSPRINT file;
      DCL ARG FLOAT BINARY(53);
      Put Skip List ('ARG =', ARG);
      ARG = ARG + 1;
      RETURN (ARG);
      END;
```

Output:

```
ARG =                   1.250000000000000E+01
x = 12.500000, y = 13.500000
```

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
double *cpli(double *);
main()
{
   double x=12.5, y;
   y = *cpli(&x);  /* by reference */
   printf("x = %f, y = %f\n", x, y);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG) RETURNS (POINTER);
      Dcl SYSPRINT file;
      DCL ARG FLOAT BINARY(53);
      Put Skip List ('ARG =', ARG);
      ARG = ARG + 1;
      RETURN (ADDR(ARG));
      END;
```

Output:

```
ARG =                    1.250000000000000E+01
x = 13.500000, y = 13.500000
```

Extended-Precision Floating Point

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
long double cpli(long double);
main()
{
   long double x=12.1, y;
   y = cpli(x);  /* by value */
   printf("x = %Lf, y = %Lf\n", x, y);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG) RETURNS (FLOAT BIN(109));
      Dcl SYSPRINT file;
      DCL ARG FLOAT BIN(109);
      Put Skip List ('ARG =', ARG);
      ARG = ARG + 1;
      RETURN (ARG);
      END;
```

Output:

```
ARG =                    1.21000000000000000888178419700125E+01
x = 12.100000, y = 13.100000
```

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
long double *cpli(long double *);
main()
{
   long double x=12.5, y;
   y = *cpli(&x);  /* by reference */
   printf("x = %Lf, y = %Lf\n", x, y);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG) RETURNS (POINTER);
      Dcl SYSPRINT file;
      DCL ARG FLOAT BINARY(109);
      Put Skip List ('ARG =', ARG);
      ARG = ARG + 1;
      RETURN (ADDR(ARG));
      END;
```

Output:

```
ARG =                    1.25000000000000000000000000000000E+01
x = 13.500000, y = 13.500000
```

Pointer to an Integer

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
void cpli(int **);
main()
{
   int x=5, *i=&x;
   cpli(&i);
   printf("x = %d\n", x);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG);
      Dcl SYSPRINT file;
      DCL ARG POINTER;
      DCL Z FIXED BIN(31,0) BASED (ARG);
      Put Skip List ('Z =', Z);
      Z = Z + 1;
      END;
```

Output:

```
Z =                                5
x = 6
```

Pointer to an Array

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
void cpli(int (**)[]);
main()
{
   int i, matrix[5];
   int (*temp)[] = &matrix;
   for (i=0; i<5; ++i)
      matrix[i] = i;
   cpli(&temp);
   printf("matrix[3] = %d\n", matrix[3]);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG);
      Dcl SYSPRINT file;
      DCL ARG POINTER;
      DCL Z(5) FIXED BIN(31,0) BASED (ARG);
      Put Skip List ('Z =', Z(4));
      Z(4) = Z(4) + 1;
      END;
```

Output:

```
Z =                                3
matrix[3] = 4
```

Pointer to a Structure

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
struct date
{
   int day;
   int month;
   int year;
} today;
void cpli(struct date **);
main()
{
   struct date *temp = &today;
   today.day = 20;
   cpli(&temp);
   printf("day = %d\n", today.day);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG);
      Dcl SYSPRINT file;
      DCL ARG POINTER;
      DCL 1 TODAY BASED (ARG),
          2 DAY FIXED BIN(31),
          2 MONTH FIXED BIN(31),
          2 YEAR FIXED BIN(31);
      Put Skip List ('DAY =', DAY);
      DAY = DAY + 1;
      END;
```

Output:

```
DAY =                              20
day = 21
```

Fixed Decimal

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
#include <decimal.h>
void cpli(decimal(5,2));
main()
{
   decimal(5,2) x;
   x = 52d;
   cpli(x);
   printf("x = %D(5,2)\n", x);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG);
      Dcl SYSPRINT file;
      DCL ARG FIXED DEC(5,2);
      DCL X FIXED DEC(5,2) EXTERNAL;
      Put Skip List ('ARG =', ARG);
      X = ARG + 1;
      END;
```

Output:

```
ARG =                     52.00
x = 52.00
```

Fixed-Length Character String

Sample C Usage

```
#pragma linkage (cpli,PLI)
#include <stdio.h>
void cpli(char **);
main()
{
   char *string = "Have a nice day!";
   cpli(&string);
}
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
CPLI: PROC(ARG);
      Dcl SYSPRINT file;
      DCL ARG POINTER;
      DCL Z CHAR(16) BASED (ARG);
      Put Skip List (Z);
      END;
```

Output:

```
Have a nice day!
```

Equivalent Data Types—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 32, which describes how an C routine can receive parameters that are passed by value and by reference.

Fullword Integer

Sample PL/I Usage

```
%PROCESS MAR(1,72);
MY_PROG: PROC OPTIONS(MAIN);
    Dcl SYSPRINT file;
    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));
    Put Skip List ('X =', X, ', Y =', Y);
END;
```

C Subroutine

```
#pragma linkage (centry,PLI)
#include <stdio.h>
int centry(int x)
{
   printf("x = %d\n", x);
   return(++x);
}
```

Output:

```
x = 5
X =                                5          , Y =                                6
```

Sample PL/I Usage

```
%PROCESS MAR(1,72);
MY_PROG: PROC OPTIONS(MAIN);
    Dcl SYSPRINT file;
    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);
    Put Skip List ('Y =', Y);
END;
```

C Subroutine

```
#pragma linkage (centry,PLI)
#include <stdio.h>
int centry(int x)
{
   printf("x = %d\n", x);
   return(x+1);
}
```

Output:

```
x = 5
Y =                                6
```

Double-Precision Floating Point

Sample PL/I Usage

```
%PROCESS MAR(1,72);
MY_PROG: PROC OPTIONS(MAIN);
    Dcl SYSPRINT file;
    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));
    Put Skip List ('X =', X, ', Y =', Y);
END;
```

C Subroutine

```
#pragma linkage (centry,PLI)
#include <stdio.h>
double centry(double x)
{
   printf("x = %f\n", x);
   return(++x);
}
```

Output:

```
x = 3.141593
X =                    3.141592649999999E+00  , Y =                    4.141592649999999E+00
```

Sample PL/I Usage	C Subroutine

`%PROCESS MAR(1,72);`
`MY_PROG: PROC OPTIONS(MAIN);`
`  Dcl SYSPRINT file;`
`  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);`
`  Put Skip List ('Y =', Y);`
`END;`
	`#pragma linkage (centry,PLI)`
	`#include <stdio.h>`
	`double centry(double x)`
	`{`
	`  printf("x = %f\n", x);`
	`  return(x+1);`
	`}`

Output:

```
x = 3.141593
Y =                   4.141592649999999E+00
```

Extended-Precision Floating Point

Sample PL/I Usage	C Subroutine

`%PROCESS MAR(1,72);`
`MY_PROG: PROC OPTIONS(MAIN);`
`  Dcl SYSPRINT file;`
`  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));`
`  Put Skip List ('X =', X, ', Y =', Y);`
`END;`
	`#pragma linkage (centry,PLI)`
	`#include <stdio.h>`
	`long double centry(long double x)`
	`{`
	`  printf("x = %Lf\n", x);`
	`  return(++x);`
	`}`

Output:

```
x = 12.500000
X =                    1.25000000000000000000000000000000E+01        , Y =
 1.35000000000000000000000000000000E+01
```

Sample PL/I Usage

```
%PROCESS MAR(1,72);
MY_PROG: PROC OPTIONS(MAIN);
    Dcl SYSPRINT file;
    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);
    Put Skip List ('Y =', Y);
END;
```

C Subroutine

```
#pragma linkage (centry,PLI)
#include <stdio.h>
long double centry(long double x)
{
   printf("x = %Lf\n", x);
   return(x+1);
}
```

Output:

```
x = 12.500000
Y =                    1.350000000000000000000000000000000E+01
```

Pointer to an Integer

Sample PL/I Usage

```
%PROCESS MAR(1,72);
MY_PROG: PROC OPTIONS(MAIN);
    Dcl SYSPRINT file;
    DCL CENTRY EXTERNAL ENTRY
        RETURNS (FIXED BIN(31));
    DCL I FIXED BIN(31);
    DCL Z FIXED BIN(31);
    DCL P POINTER;
    P = ADDR(I);
    I = 5;
    Z=CENTRY(P);
    Put Skip List ('I =', I, ', Z =', Z);
END;
```

C Subroutine

```
#pragma linkage (centry,PLI)
#include <stdio.h>
int centry(int **x)
{
   printf("x = %d\n", **x);
   return(++(**x));
}
```

Output:

```
x = 5
I =                                6          , Z =                                6
```

Pointer to an Array

Sample PL/I Usage

```
%PROCESS MAR(1,72);
MY_PROG: PROC OPTIONS(MAIN);
    Dcl SYSPRINT file;
    DCL CENTRY EXTERNAL ENTRY
        RETURNS (FIXED BIN(31));
    DCL I(5) FIXED BIN(31,0);
    DCL J FIXED BIN(31);
    DCL Z FIXED BIN(31);
    DCL P POINTER;
    P = ADDR(I);
    DO J = 1 TO 5;
         I(J) = J;
    END;
    Z=CENTRY(P);
    Put Skip List ('I(3) =', I(3), ', Z =', Z);
END;
```

C Subroutine

```
#pragma linkage (centry,PLI)
#include <stdio.h>
int centry(int (**x)[])
{
   printf("x[2] = %d\n", (**x)[2]);
   return(++(**x)[2]);
}
```

Output:

```
x[2] = 3
I(3) =                                4          , Z =                                4
```

Pointer to a Structure

Sample PL/I Usage

```
%PROCESS MAR(1,72);
MY_PROG: PROC OPTIONS(MAIN);
    Dcl SYSPRINT file;
    DCL CENTRY EXTERNAL ENTRY
        RETURNS (FIXED BIN(31));
    DCL 1 TODAY,
       2 DAY FIXED BIN(31),
       2 MONTH FIXED BIN(31),
       2 YEAR FIXED BIN(31);
    DCL Z FIXED BIN(31);
    DCL P POINTER;
    P = ADDR(TODAY);
    MONTH = 7;
    Z=CENTRY(P);
    Put Skip List ('MONTH =', MONTH, ', Z =', Z);
END;
```

C Subroutine

```
#pragma linkage (centry,PLI)
#include <stdio.h>
struct date
{
   int day;
   int month;
   int year;
};
int centry(struct date **x)
{
   printf("month = %d\n", (*x)->month);
   return(++(*x)->month);
}
```

Output:

```
month = 7
MONTH =                               8          , Z =                                8
```

Fixed Decimal

Sample PL/I Usage

```
%PROCESS MAR(1,72);
PLIPROG: PROC OPTIONS(MAIN, REENTRANT);
    Dcl SYSPRINT file;
    DCL CFUNC EXTERNAL ENTRY
        RETURNS (FIXED DEC(5,0));
    DCL ARG FIXED DEC(5,0);
    DCL A FIXED DEC(5);
    ARG = 10;
    A = CFUNC(ARG);
    Put Skip List ('ARG =', ARG, ', A =', A);
END;
```

C Subroutine

```
#pragma linkage (CFUNC,PLI)
#include <stdio.h>
#include <decimal.h>
decimal(5,0) CFUNC(decimal(5,0) p)
{
   printf("p = %D(5,0)\n", p);
   return(++p);
}
```

Output:

```
p = 10
ARG =                          11               , A =                          11
```

Fixed-Length Character String

Sample PL/I Usage

```
%PROCESS MAR(1,72);
MY_PROG: PROC OPTIONS(MAIN);
    Dcl SYSPRINT file;
    DCL CENTRY EXTERNAL ENTRY;
    DCL CHARSTRING CHAR(29)
        INIT('Hello C, this is PL/I calling');
    DCL PLI_POINTER PTR;
    PLI_POINTER = ADDR(CHARSTRING);
    CALL CENTRY(ADDR(PLI_POINTER));
END;
```

C Subroutine

```
#pragma linkage (centry,PLI)
#include <stdio.h>
void centry(char ***c_character_string)
{
   printf("%.29s\n", **c_character_string);
}
```

Output:

```
Hello C, this is PL/I calling
```

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 phase. If your application contains PL/I routines and non-reentrant C routines, PL/I's external data maps to C's external data only within a phase. After you cross a phase 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 C RENT compile-time option.

Figure 15 on page 45 illustrates the name scope of external variables in a PL/I–C enclave, if the C routine is non-reentrant. Each of the routines can be a PL/I routine or a C routine. If Routine 3 is a PL/I routine, 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.

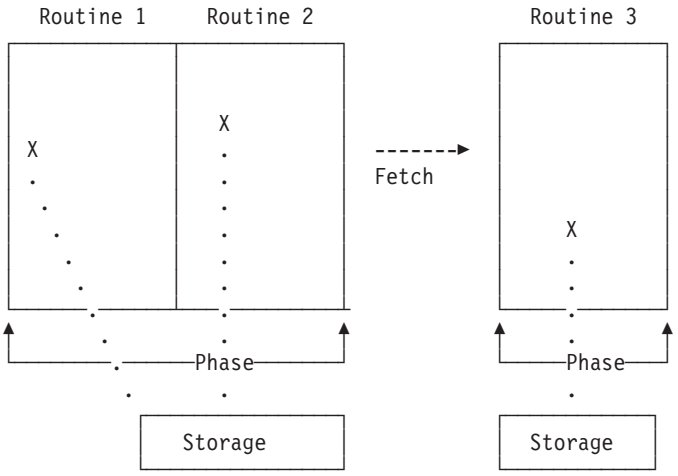


Figure 15. Name Scope of External Variables for PL/I or C Fetch

In Figure 15, external data declared in Routine 1 maps to that declared in Routine 2 in the same phase. If the fetch is made to a C Routine 3 in another phase, the external data does not map because the name scope of external data in C is the phase. 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.

Because the name scopes of PL/I and C are both the phase, you cannot share external data across a phase boundary in a PL/I–C application. Therefore, to avoid confusion use different names for external data in separate phases if possible.

Name Space of External Data

In programming languages, the *name space* is defined as the portion of a phase within which a particular declaration applies or is known. Within the same phase, 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.

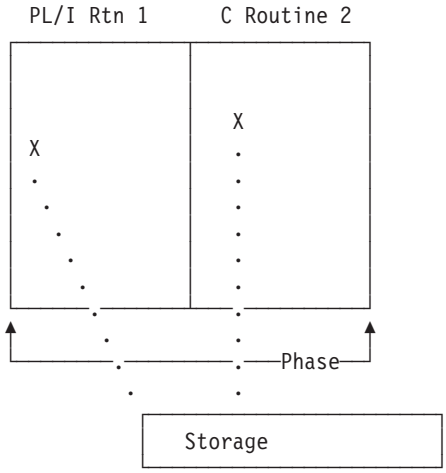


Figure 16. Name Space of External Data in PL/I Static Call to C

Figure 16 illustrates that within the same phase, 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–C application, if no phase boundary is crossed.

How to Use Dynamic Heap Storage Functions

Use the following guidelines when you mix HLL storage constructs and LE/VSE 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 LE/VSE 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

File Sharing

Except for the file specified by the LE/VSE MSGFILE run-time option, LE/VSE provides no support for files that are open in C and PL/I at the same time. You must manage all such files to ensure that no conflicts arise. Performing I/O operations on the same filename, other than the one specified by the LE/VSE MSGFILE run-time option, might cause abnormal termination or data corruption of the file.

LE/VSE provides no support for using C to read from or write to a file created using PL/I, or vice versa. To do this, you must ensure that the file is in a compatible format. For information on file formats, see the respective programming guides.

Note: A guide to the types of considerations necessary when sharing file data between languages can be found in Chapter 4, “Communicating between COBOL and PL/I,” section “File Sharing” on page 67.

Directing Output in ILC Applications

Under LE/VSE, PL/I run-time output such as run-time messages and ON condition SNAP output is directed to the destination specified in the LE/VSE run-time option MSGFILE. User-specified output, such as the output of the PUT SKIP LIST statement, remains directed to the PL/I STREAM PRINT file SYSPRINT.

Since the default destination for the LE/VSE MSGFILE and the default destination for SYSPRINT are both SYSLST, all output from PL/I is by default directed to the same destination and will thus be interspersed. To separate the output, either SYSPRINT or the LE/VSE MSGFILE must be directed to a file different from the

other. Specifying MSGFILE(SYSPRINT) under LE/VSE has no effect unless SYSPRINT has been declared using a different destination to the one used by the LE/VSE MSGFILE.

Under LE/VSE, C run-time output such as run-time messages is directed to the destination specified in the LE/VSE run-time option MSGFILE. `stderr` output is also directed to the destination of the MSGFILE option. `stdout` output is by default directed to SYSLST.

Since the default destination for the LE/VSE MSGFILE is also SYSLST, all output from C is by default directed to the same destination and will thus be interspersed. To separate the output, either `stdout` or the LE/VSE MSGFILE must be directed to a file different from the other. For information on redirecting C output, see *LE/VSE C Run-Time Programming Guide*.

Note: In both C and PL/I, when run-time output and user-specified output from one or both languages is directed to the same destination such as SYSLST, each language must manage its own I/O buffers, line counters, etc., for its own user-specified output.

For additional information regarding the LE/VSE MSGFILE run-time option, see *LE/VSE Programming Guide*.

C—PL/I Condition Handling

This section offers two scenarios of condition handling behavior in a C-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 48. 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 50.

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 LE/VSE 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.

For a detailed description of LE/VSE condition handling, see *LE/VSE Programming Guide*.

Enclave-Terminating 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-PL/I application to be terminated. Those language constructs of interest are listed below.

C

Typical C language constructs that cause the application to terminate are:

- The `abort()`, `raise(SIGTERM)`, `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 *LE/VSE Debugging Guide and Run-Time Messages* 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 17 throughout the following discussion.

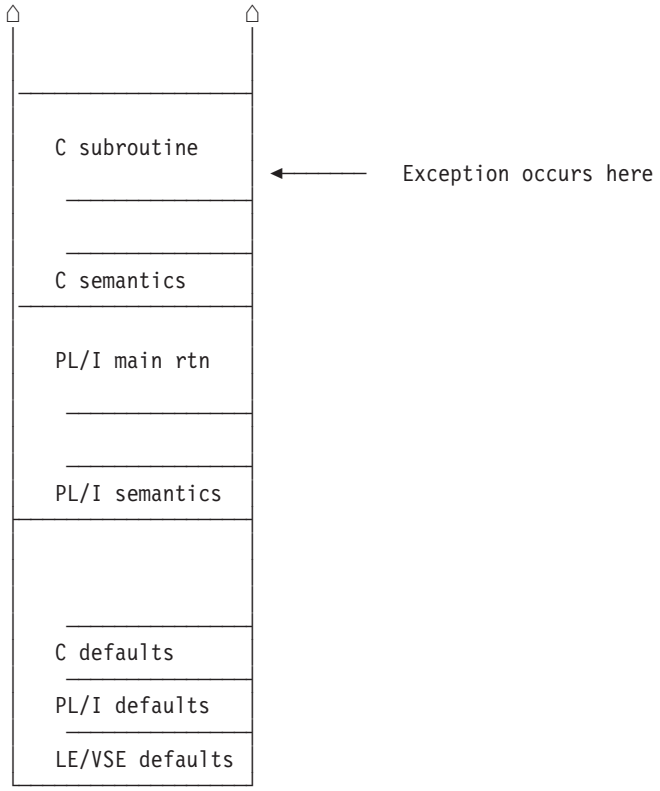


Figure 17. Stack Contents When the Exception Occurs in C

In this scenario, a PL/I main routine invokes a C subroutine. An exception occurs in the C subroutine. The stack contains what is shown in Figure 17.

The actions taken follow the three LE/VSE 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().

  Note: The abend corresponding to the signal(SIGABND) or the LE/VSE message 3250 is not ignored. The enclave is terminated.
- The exception is one of those listed as masked in Table 28 on page 99.
- You did not specify any action, but the default action for the condition is SIG_IGN (see Table 28 on page 99).
- You are running under CICS and a CICS handler is pending.

If none of these things are true, 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 28 on page 99) is to terminate, the condition is percolated.
5. No user-written condition handlers can exist on the PL/I stack frame, because they cannot be registered in PL/I. If an ON-unit has been established for the condition being processed on the PL/I stack frame, it is given control, however. 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 LE/VSE default actions take place, as described in Table 27 on page 99. 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 issues a resume, LE/VSE 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 18 throughout the following discussion.

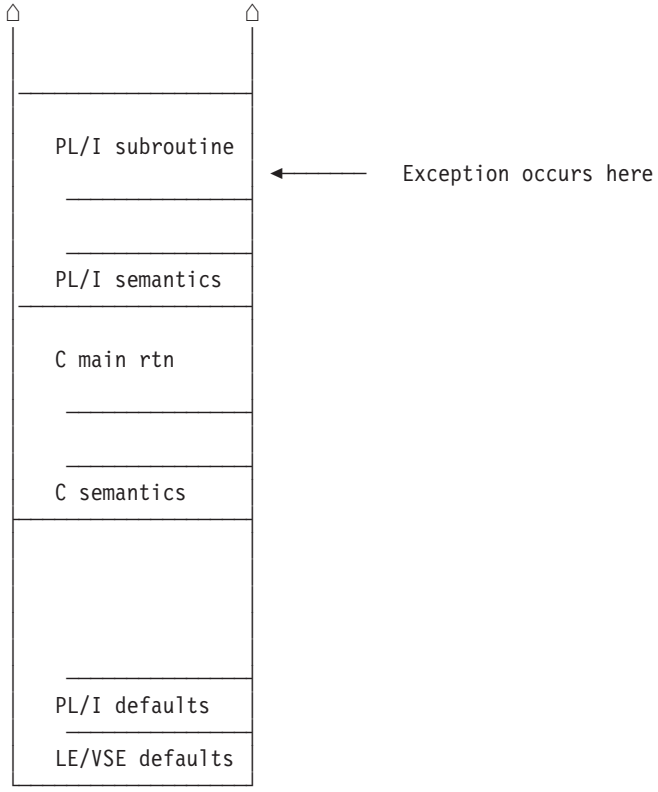


Figure 18. Stack Contents When the Exception Occurs in PL/I

In this example, a C main routine invokes a PL/I subroutine. An exception occurs in the PL/I subroutine. The stack contains what is shown in Figure 18.

The actions taken follow the three LE/VSE 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, described below, takes place.
2. No user-written condition handler can be registered using CEEHDLR on the PL/I stack frame because they cannot be registered in PL/I.

   If an ON-unit that corresponds to the condition being processed is established on the PL/I 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 is established for the condition, so the condition is percolated.

3. 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 PL/I conditions of severity 2 or greater. See the chapter on coding a user-written condition handler in *LE/VSE Programming Guide* for more information.

   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 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.
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 LE/VSE default actions take place, as described in Table 27 on page 99. 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, LE/VSE terminates the thread.

Sample C-PL/I ILC Application

IBMCPL

```
*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 EXAMPLE STARTED');
     PUT SKIP LIST ('*****************************');
     PL1_POINTER = ADDR(CHARSTRING);
     PUT SKIP LIST ('Calling C subroutine');
     J = CECFPLI( ADDR(PL1_SHORT), PL1_INT, PL1_DOUBLE,
        PL1_LONG_DOUBLE, ADDR(PL1_POINTER));
     PUT SKIP LIST ('Returned from C subroutine');
     IF (J ¬= 999) THEN
        PUT SKIP LIST ('Error in return code from C');
     PUT SKIP LIST ('*****************************');
     PUT SKIP LIST ('PL/I CALLING C EXAMPLE ENDED  ');
     PUT SKIP LIST ('*****************************');
  END CEPLI2C;
```

Figure 19. PL/I Main Routine Calling a C Subroutine

EDCCPL

```
 /*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);
 }
```

Figure 20. C Routine Called by PL/I Main Routine

Chapter 4. Communicating between COBOL and PL/I

This chapter describes LE/VSE's support for COBOL and PL/I ILC applications. If you are running a COBOL–PL/I ILC application under CICS, you should also consult Chapter 7, "ILC under CICS," on page 93.

General Facts about COBOL–PL/I ILC

- A COBOL routine cannot be used as a PL/I function; it must be CALLed.
- 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 "How to Use Dynamic Heap Storage Functions" on page 46 for information about how to use PL/I's storage facilities with LE/VSE storage services.
- LE/VSE supports the passing of return codes within an ILC application from COBOL to PL/I but not from PL/I to COBOL. See "Return Codes" on page 60 for more information.

Preparing to Use ILC between COBOL and PL/I

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.

LE/VSE ILC Support

LE/VSE supports ILC between the following combinations of COBOL and PL/I:

Table 19. Supported Languages for LE/VSE ILC Support

HLL Pair	COBOL	PL/I
COBOL–PL/I	• COBOL/VSE Release 1 • VS COBOL II Release 3 or later	• PL/I VSE

Migrating ILC Applications

You need to relink pre-LE/VSE-conforming ILC applications in order to get LE/VSE's ILC support.

You do not need to recompile VS COBOL II routines in your application to run them under LE/VSE. However, you might want to recompile some VS COBOL II routines to obtain LE/VSE's condition handling behavior.

Determining the Main Routine

In LE/VSE, 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 20 identifies the desired entry point. The table assumes that your code has been compiled using the LE/VSE-conforming compilers.

Table 20. Determining the Entry Point

HLL	Main Entry Point	Fetched Entry Point
COBOL	Name of the first object module to get control in the object program	Program name
PL/I	CEESTART	CEESTART or routine name if OPTIONS(FETCHABLE) is used

Declarations

If a PL/I routine invokes a COBOL routine or a COBOL routine invokes a PL/I routine, you must specify entry declarations in the PL/I source code. No special declaration is required within the COBOL routine.

When invoking a COBOL routine 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 routine, you request that the PL/I compiler generate a parameter list for the COBOL routine in the style COBOL accepts.

In a PL/I routine that calls a COBOL routine, 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 routine 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 routine. `OPTIONS(COBOL)` specifies that the entry point can be invoked only by a COBOL routine.

For more information on the COBOL option, see *IBM PL/I for VSE/ESA Language Reference*

In addition to the COBOL option, other options suppress remapping of data aggregates. These are described in “Aggregates” on page 58.

Only data types common to both languages can be passed or received.

Reentrancy

Under LE/VSE, ILC applications in which PL/I calls COBOL are reentrant, assuming that all COBOL and PL/I routines in the application are reentrant.

Calling between PL/I and COBOL

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

Types of Calls Permitted

Table 21 describes the types of calls between COBOL and PL/I that LE/VSE allows:

Table 21. Calls Permitted for COBOL and PL/I

ILC Direction	Static Call	Dynamic Call/Fetch
COBOL to PL/I	Yes	Yes
PL/I to COBOL	Yes	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 phases that contain ILC cannot be released by using the COBOL CANCEL verb. The dynamically loaded phase is instead released by LE/VSE termination processing.

A COBOL routine can dynamically CALL a PL/I routine. Dynamically called PL/I routines must adhere to the restrictions listed in *IBM PL/I for VSE/ESA Language Reference* For specific details about COBOL dynamic CALLs, refer to *IBM COBOL for VSE/ESA Programming Guide*

If a PL/I routine is to be dynamically loaded, you must either:

- Specify the routine name as the entry point when you link-edit it
- Specify PROC OPTIONS(FETCHABLE) and recompile

PL/I Fetching COBOL

PL/I routines can only call COBOL routines that are LE/VSE-conforming. ILC between PL/I and COBOL is supported within the fetched phase.

Passing Data between COBOL and PL/I

The data types supported between COBOL and PL/I are listed below.

Table 22. 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)
COMP-1	REAL FLOAT DECIMAL(6)
COMP-2	REAL FLOAT DECIMAL(16)
PIC S9(n) PACKED-DECIMAL	FIXED DECIMAL(n)
PIC S9(n) 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)	GRAPHIC(n)

Table 22. Supported Data Types between COBOL and PL/I (continued)

COBOL	PL/I
PIC X(n)	CHAR(n)
groups	aggregates
POINTER	POINTER

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 routine 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.

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 *IBM PL/I for VSE/ESA Language Reference* for details about the ALIGNED attribute and *IBM COBOL for VSE/ESA Language Reference* for details about the SYNCHRONIZED clause.

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.

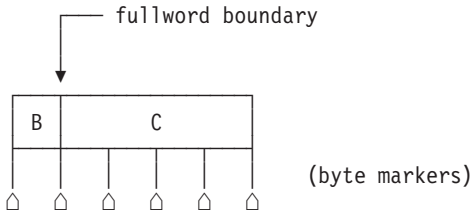
If you specify the SYNCHRONIZED phrase, then BINARY and floating-point data items are aligned on halfword, fullword, or doubleword boundaries, depending on their length. If SYNCHRONIZED is not specified, then all data items are aligned on byte boundary only.

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 *IBM PL/I for VSE/ESA Language Reference*

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);
```

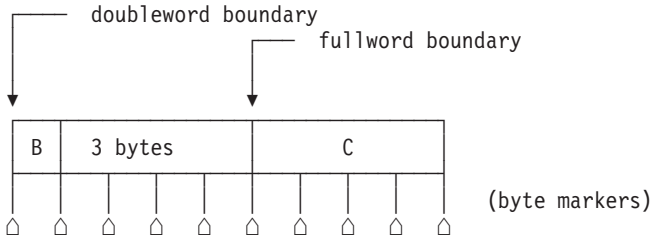
and is held like this:



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.
```

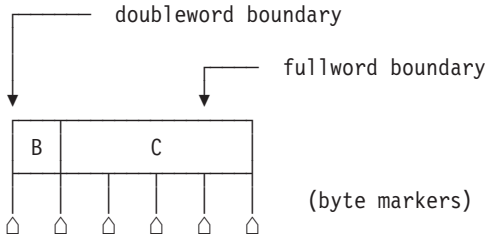
and is held like this:



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:



How to Map 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 below:

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.

ARG*n* Applies to the NOMAP, NOMAPIN, and NOMAPOUT options. It specifies arguments to which these options apply. If ARG*n* is omitted, a specified option applies to all arguments.

Return Codes

The passing of return codes from a COBOL subroutine to PL/I using the RETURN-CODE special register is supported as illustrated in the example "Halfword Integer" on page 63.

However, any return code set in a PL/I subroutine (using the PLIRETC function or the RETURN statement) is ignored when control is returned to a calling COBOL routine. To return any information from a called PL/I routine to a calling COBOL routine, a suitably declared/defined argument passed by reference can be used. For example, to match the implicit declaration of the COBOL :RETURN-CODE special register, use a fullword integer argument. In the illustration "Fullword Integer" on page 61, the argument "X" could be used to pass back the return code.

Data Equivalents

This section describes how PL/I and COBOL data types correspond to each other.

Equivalent Data Types—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.

Halfword Integer

Sample COBOL Usage

```
CBL RMODE(ANY),APOST
       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.
           DISPLAY X
           GOBACK.
       END PROGRAM CSFB15.
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
PTFB15: Proc(X) Options(Cobol);
  Dcl SYSPRINT file;
  Dcl X Fixed Binary(15,0);
  Put Skip List('X =', X);
  X = X + 1;
End;
```

Output:

```
X =                           16
0017
```

Fullword Integer

Sample COBOL Usage

```
CBL RMODE(ANY),APOST
       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.
           DISPLAY X
           GOBACK.
       END PROGRAM CSFB31.
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
PTFB31: Proc (X) Options(COBOL);
  Dcl SYSPRINT file;
  Dcl X Fixed Binary(31);
  Put Skip List('X =',  X);
  X = X + 1;
End;
```

Output:

```
X =                                5
000000006
```

Single-Precision Floating Point

Sample COBOL Usage

```
CBL RMODE(ANY),APOST
       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.
           DISPLAY X
           GOBACK.
       END PROGRAM CSFTD6.
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
PTFTD6: Proc ( X ) Options(COBOL);
  Dcl SYSPRINT file;
  Dcl X Float Decimal(6);
  Put Skip List('X =', X);
  X = X + 1;
End;
```

Output:

```
X =                     1.60000E+01
 .17000000E 02
```

Double-Precision Floating Point

Sample COBOL Usage

```
CBL RMODE(ANY),APOST
       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.
           DISPLAY X
           GOBACK.
       END PROGRAM CSFTD16.
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
PTFTD16: Proc (X) Options(COBOL);
  Dcl SYSPRINT file;
  Dcl X Float Decimal(16);
  Put Skip List('X =', X);
  X = X + 1;
End;
```

Output:

```
X =                      0.000000000000000E+00
 .10000000000000000E 01
```

Fixed-Length Character String

Sample COBOL Usage

```
CBL RMODE(ANY),APOST
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CTFSTR.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 STR    PIC X(80).
       LINKAGE SECTION.
       PROCEDURE DIVISION.
           MOVE 'Test COBOL-PL/I message.' TO STR.
           CALL 'PSFSTR' USING STR.
           GOBACK.
       END PROGRAM CTFSTR.
```

PL/I Subroutine

```
%PROCESS MAR(1,72);
PSFSTR: Proc (X) Options(COBOL);
  Dcl SYSPRINT file;
  Dcl X Char(80);
  Put Skip List('X =', X);
End;
```

Output:

```
X =                    Test COBOL-PL/I message.
```

Equivalent Data Types—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.

Halfword Integer

Sample PL/I Usage

```
%PROCESS MAR(1,72);
PSFB15: Proc Options(Main);
  Dcl SYSPRINT file;
  Dcl PLIRETV builtin;
  Dcl X Fixed Binary(15,0);
  Dcl CTFB15 external entry
             Options(COBOL retcode);
      X=1;
      Call CTFB15( X );
      Put Skip List ('X =', X, ', RC = ', PLIRETV);
End;
```

COBOL Subroutine

```
CBL RMODE(ANY)
       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.
           COMPUTE X = X + 1.
     * Set return code 4
           MOVE 4 TO RETURN-CODE.
           GOBACK.
       END PROGRAM CTFB15.
```

Output:

```
0001
X =                              2              , RC =                              4
```

Note: As well as showing the passing of halfword integer data between PL/I and COBOL, this example illustrates how the return code is passed from COBOL to PL/I.

Fullword Integer

Sample PL/I Usage

```
%PROCESS MAR(1,72);
PSFB31: Proc Options(Main);
  Dcl SYSPRINT file;
  Dcl X Fixed Binary(31);
  Dcl CTFB31 external entry
             Options(COBOL);
      X=1;
      Call CTFB31( X );
      Put Skip List ('X =', X);
End;
```

COBOL Subroutine

```
CBL RMODE(ANY)
       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.
           COMPUTE X = X + 1.
           GOBACK.
       END PROGRAM CTFB31.
```

Output:

```
000000001
X =                                2
```

Single-Precision Floating Point

Sample PL/I Usage

```
%PROCESS MAR(1,72);
PSFTD6: Proc Options(Main);
  Dcl SYSPRINT file;
  Dcl X Float Decimal(6);
  Dcl CTFTD6 external entry
             Options(COBOL);
      X=1;
      Call CTFTD6( X );
      Put Skip List ('X =', X);
End;
```

COBOL Subroutine

```
CBL RMODE(ANY)
       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.
           COMPUTE X = X + 1.
           GOBACK.
       END PROGRAM CTFTD6.
```

Output:

```
 .10000000E 01
X =                      2.00000E+00
```

Double-Precision Floating Point

Sample PL/I Usage	COBOL Subroutine

`%PROCESS MAR(1,72);`
`PSFTD16: Proc Options(Main);`
`  Dcl SYSPRINT file;`
`  Dcl X Float Decimal(16);`
`  Dcl CTFTD16 external entry`
`  Options(COBOL);`
`  X=1;`
`  Call CTFTD16( X );`
`  Put Skip List ('X =', X);`
`End;`
	`CBL RMODE(ANY)`
	`  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.`
	`  COMPUTE X = X + 1.`
	`  GOBACK.`
	`  END PROGRAM CTFTD16.`

Output:

```
 .10000000000000000E 01
X =                      2.000000000000000E+00
```

Fixed-Length Character String

Sample PL/I Usage	COBOL Subroutine

`%PROCESS MAR(1,72);`
`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;`
	`CBL RMODE(ANY)`
	`  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.`

Output:

```
Test PL/I-COBOL message.
```

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 23. 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 phase. Figure 21 and Figure 22 on page 66 illustrate these differences.

Because the name scope for COBOL and PL/I is different, 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 phase or across different phases. 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 phase a different name, you can change the language mix in the application later, and your application will still behave as you expect it to.

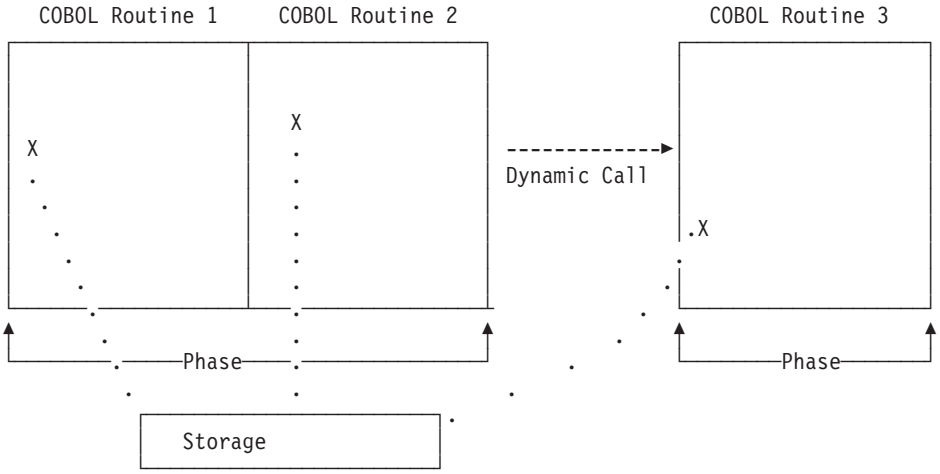


Figure 21. Name Scope of External Variables for COBOL Dynamic CALL

In Figure 21, COBOL routines 1, 2, and 3 comprise a COBOL run unit (an LE/VSE enclave). EXTERNAL data declared in COBOL routine 1 maps to that declared in COBOL routine 2 in the same phase. When a dynamic call to COBOL routine 3 in another phase is made, the EXTERNAL data still maps, because the name scope of EXTERNAL data in COBOL is the enclave.

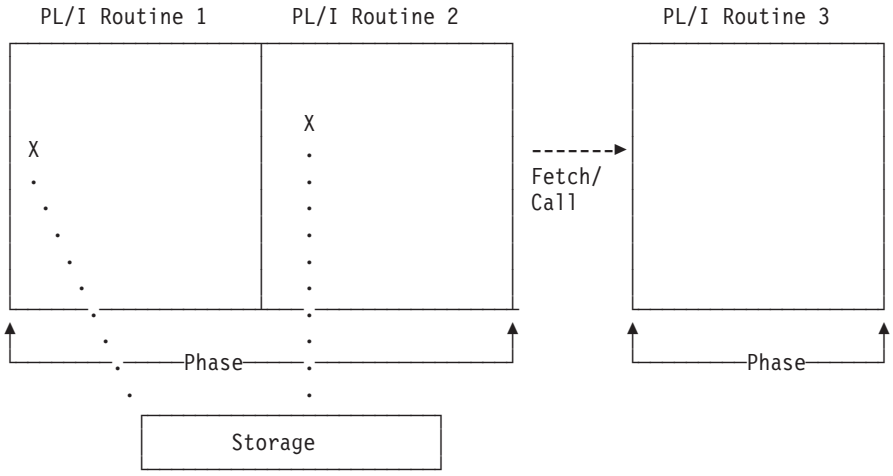


Figure 22. Name Scope of External Variables for PL/I Fetch

The name scope of external data in PL/I is the phase. In Figure 22, external data declared in PL/I routine 1 maps to that declared in PL/I routine 2 in the same phase. The fetched PL/I Routine 3 in another phase cannot share external data with Routine 1 or Routine 2.

Name Space of External Data

In programming languages, the *name space* is defined as the portion of a phase within which a particular declaration applies or is known. Figure 23 and Figure 24 on page 67 illustrates that, like the name scope, the name space of external data differs between PL/I and COBOL.

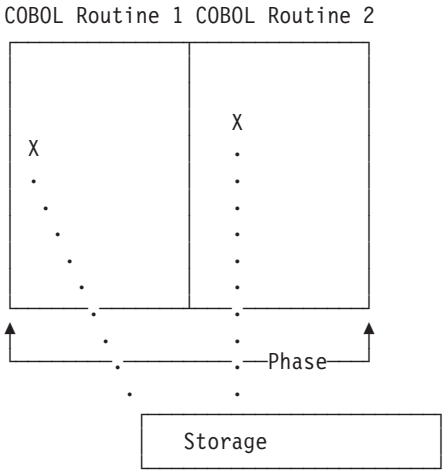


Figure 23. Name Space of External Data for COBOL Static CALL to COBOL

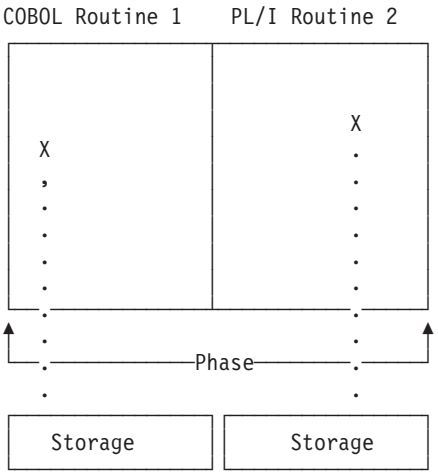


Figure 24. Name Space of External Data in COBOL Static Call to PL/I

Figure 23 and Figure 24 illustrate that within the same phase, the name space of COBOL routines is the same. However, the name spaces of a COBOL routine and a PL/I routine within the same phase 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.

File Sharing

Except for the file specified by the LE/VSE MSGFILE run-time option, LE/VSE provides no support for files that are open in COBOL and PL/I at the same time. You must manage all such files to ensure that no conflicts arise. Performing I/O operations on the same filename, other than the one specified by the LE/VSE MSGFILE run-time option, might cause abnormal termination or data corruption of the file.

You can use PL/I to read from or write to a file that was previously created using COBOL, and vice versa, by specifying the COBOL option of the ENVIRONMENT attribute on the PL/I file declaration. However, if structures are used in a file, mapping can be different, as described in "Aggregates" on page 58. When structures are in a file and you do not 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 PL/I 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.

For more information on the COBOL option, see *IBM PL/I for VSE/ESA Programming Guide*

Directing Output from ILC Applications to MSGFILE

Under LE/VSE, COBOL run-time messages and other related output is directed to the destination specified in the LE/VSE run-time option MSGFILE. User-specified output, such as output from DISPLAY UPON SYSOUT, is directed to the destination specified in the COBOL OUTDD compiler option, which defaults to SYSOUT.

Since the default destination for the LE/VSE MSGFILE is SYSLST, and the default OUTDD filename SYSOUT is treated as SYSLST also, all output from COBOL is by default directed to the same destination and will thus be interspersed. To separate run-time messages and other related output from the user-specified output, the filename specified in the LE/VSE MSGFILE run-time option must be different from the filename specified in the COBOL OUTDD compiler option. See *IBM COBOL for VSE/ESA Programming Guide* for details of using the OUTDD compiler option.

Note: The VS COBOL II compiler under VSE does not support the OUTDD compiler option. When this compiler is used, all output from the DISPLAY statement is sent to SYSLST.

Under LE/VSE, PL/I run-time output such as run-time messages and ON condition SNAP output is directed to the destination specified in the LE/VSE run-time option MSGFILE. User-specified output, such as the output of the PUT SKIP LIST statement, remains directed to the PL/I STREAM PRINT file SYSPRINT.

Since the default destination for the LE/VSE MSGFILE and the default destination for SYSPRINT are both SYSLST, all output from PL/I is by default directed to the same destination and will thus be interspersed. To separate the output, either SYSPRINT or the LE/VSE MSGFILE must be directed to a file different from the other. Specifying MSGFILE(SYSPRINT) under LE/VSE has no effect unless SYSPRINT has been declared using a different destination to the one used by the LE/VSE MSGFILE.

Note: In both COBOL and PL/I, when run-time output and user-specified output from one or both languages is directed to the same destination such as SYSLST, each language must manage its own I/O buffers, line counters, etc., for its own user-specified output.

For additional information regarding the LE/VSE MSGFILE run-time option, see *LE/VSE Programming Guide*.

COBOL—PL/I Condition Handling

This section offers two scenarios of condition handling behavior in a COBOL–PL/I 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 70. 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 72.

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 LE/VSE 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 routine, 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.

For a detailed description of LE/VSE condition handling, see *LE/VSE Programming Guide*.

Enclave-Terminating Language Constructs

Enclaves might be terminated due to 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–PL/I enclave. The language construct that terminates the enclave can be issued from either a COBOL or PL/I routine.

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 CEE5ABD

  Calling CEE5ABD causes T_I_U (Termination Imminent due to an Unhandled condition) to be signaled. Condition handlers are given a chance to handle the abend. If the abend remains unhandled, normal LE/VSE termination activities occur. For example, the LE/VSE assembler user exit gains control.

  User-written condition handlers written in COBOL must be compiled with COBOL/VSE.

PL/I

The PL/I language constructs that cause the enclave 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.
- 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 *LE/VSE Debugging Guide and Run-Time Messages* 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 25 throughout the following discussion.

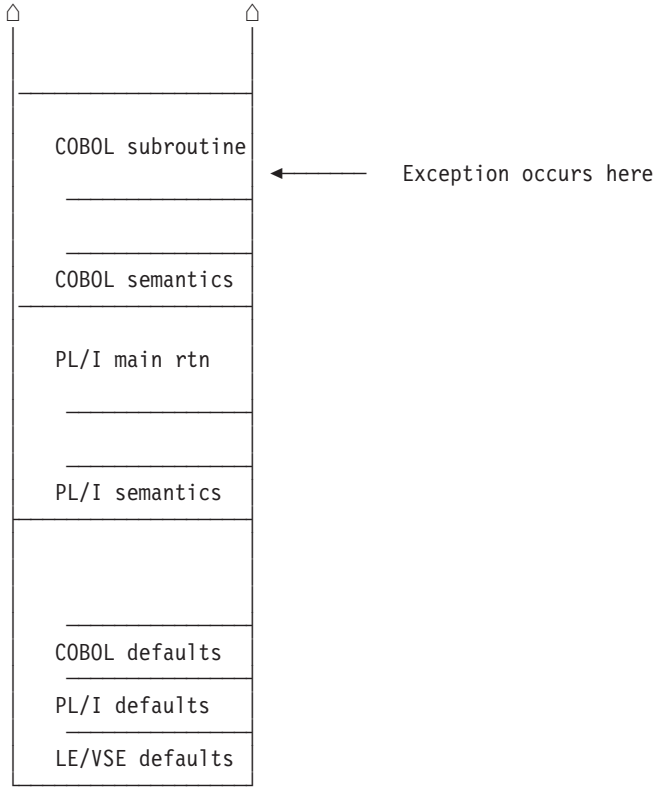


Figure 25. Stack Contents When the Exception Occurs in COBOL

In this scenario, a PL/I main routine invokes a COBOL subroutine. An exception occurs in the COBOL subroutine. The stack contains what is shown in Figure 25.

The actions taken follow the three LE/VSE 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 *LE/VSE Programming Guide*) and moving the resume cursor in an application that contains a COBOL routine (see "GOTO Out-of-Block and Move Resume Cursor" on page 73).

In this example, no user-written condition handler is registered for the condition, so the condition is percolated.

3. No user-written condition handlers can be registered on the PL/I stack frame, because they cannot be written in PL/I.

   If a PL/I ON-unit on the PL/I stack frame corresponds to the condition being processed, however, it is given control. If it issues a GOTO out-of-block, the condition handling step ends. Processing continues at the label of the GOTO. Be careful when issuing a GOTO out-of-block in an application that contains a COBOL routine; see "GOTO Out-of-Block and Move Resume Cursor" on page 73 for more information.

   If the ON-unit ends, the PL/I normal return action occurs.

   In this example, there is not an ON-unit established for the condition, so the condition is percolated.

4. 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 LE/VSE default actions take place.

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 LE/VSE default actions take place, as described in Table 27 on page 99. 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 are reached.
   - If no condition handler moves the resume cursor and issues a resume, LE/VSE 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 26 throughout the following discussion.

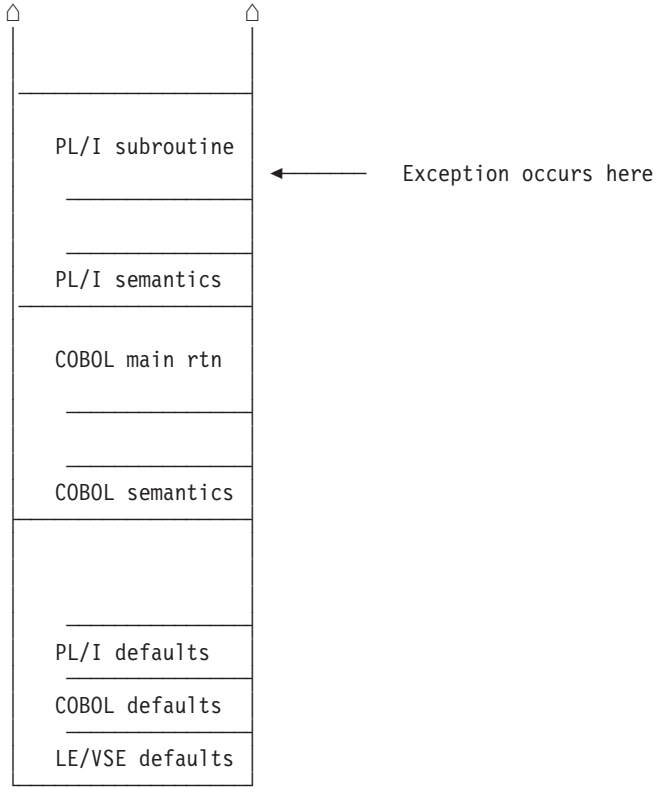


Figure 26. Stack Contents When the Exception Occurs in PL/I

In this scenario, a COBOL main routine invokes a PL/I subroutine. An exception occurs in the PL/I subroutine. The stack contains what is shown in Figure 26.

The actions taken follow the three LE/VSE 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, described below, takes place.
2. No user-written condition handlers can be registered using CEEHDLR on the PL/I stack frame, because they cannot be registered in PL/I.
3. If an ON-unit that corresponds to the condition being processed is established on the PL/I stack frame, however, it is given control. If it issues a GOTO out-of-block, the condition handling step ends. Processing continues at the label of the GOTO. You must be careful when issuing a GOTO out-of-block in an application that contains a COBOL routine; see "GOTO Out-of-Block and Move Resume Cursor" on page 73 for more information.

   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/VSE.) 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 routine. 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. 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 LE/VSE default actions take place, as described in Table 27 on page 99. 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 issues a resume, LE/VSE 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 routine, the COBOL routine can be re-entered via another call path.

If the terminated routine is **not** one of the following, the routine is not deactivated. A recursion error is raised if you attempt to enter the routine again.

- A VS COBOL II or COBOL/VSE routine compiled with the CMPR2 option
- A VS COBOL II or COBOL/VSE routine compiled with the NOCMPR2 option that does not use nested routines
- A VS COBOL II or COBOL/VSE routine 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

In addition, if the COBOL routine has the INITIAL attribute and contains files, the files are closed. (COBOL supports VSAM and SAM files and these files are closed.)

Sample PL/I–COBOL Application

IBMPCB

```
*PROCESS MACRO;
 PL1CBL: PROC OPTIONS(MAIN);
 /*Module/File Name: IBMPCB

 /*********************************************************************/
 /* FUNCTION   :  Interlanguage communication call to a COBOL         */
 /*               program.                                            */
 /*                                                                   */
 /* This example illustrates an interlanguage call from               */
 /* a PL/I main program to a COBOL subprogram.                        */
 /* 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);
```

Figure 27. PL/I Routine Calling COBOL Subroutine (Part 1 of 2)

```
   /*********************************************************/
   /* 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 subprogram example ended');

END PL1CBL;
```

Figure 27. PL/I Routine Calling COBOL Subroutine (Part 2 of 2)

IGZTPCB

```
CBL LIB,APOST,NODYNAM
      *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
```

Figure 28. COBOL Routine Called by a PL/I Main (Part 1 of 2)

```
*****************************************************************
*      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.
```

Figure 28. COBOL Routine Called by a PL/I Main (Part 2 of 2)

Chapter 5. 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 7, "ILC under CICS," on page 93.

Supported Data Types

Table 24 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 24. Data Types Common to All Supported HLLs

C	COBOL	PL/I
signed long int	PIC S9(9) USAGE IS BINARY	Real Fixed Bin(31,0)
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–COBOL
C and COBOL static external variables do not map to each other.

C–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.

COBOL–PL/I
COBOL and PL/I static external variables do not map to each other.

Condition Handling

This section describes what happens during LE/VSE condition handling and enclave termination.

For a detailed description of LE/VSE condition handling, see *LE/VSE Programming Guide*.

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 29 throughout the following discussion.

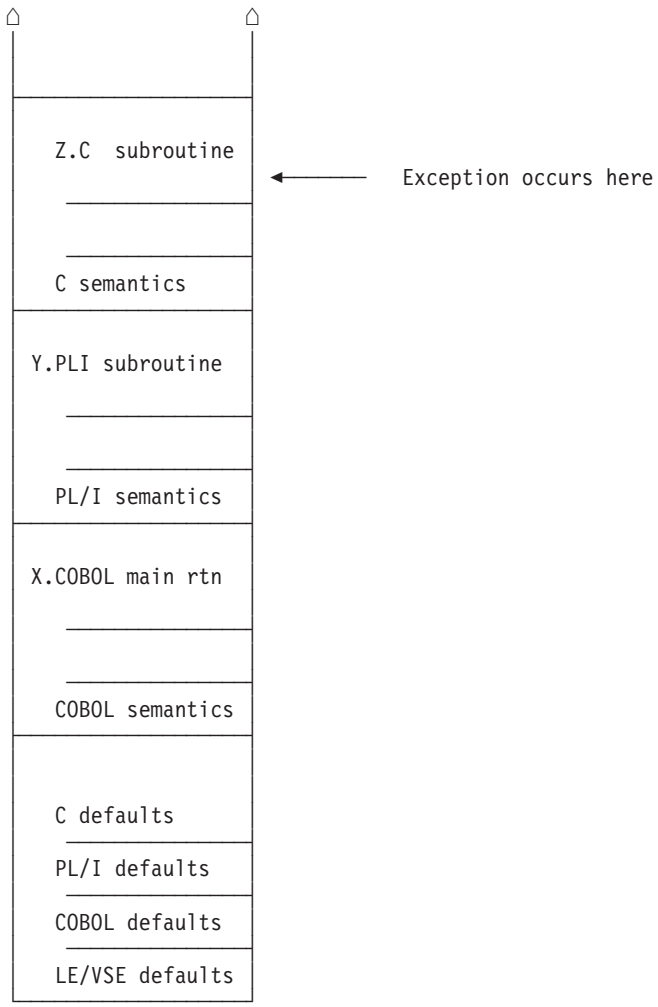


Figure 29. Stack Contents When the Exception Occurs in C

In this example, X.COBOL invokes Y.PLI, which invokes Z.C. An exception occurs in Z.C. The stack contains what is shown in Figure 29. 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 LE/VSE 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 LE/VSE message 3250 and the `signal(SIGABND)` is not ignored. The enclave is terminated.

- The exception is one of those listed as masked in Table 28 on page 99.
- You do not specify any action, but the default action for the condition is `SIG_IGN` (see Table 28 on page 99).
- You are running under CICS and a CICS handler is pending.

If none of these things are true, 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 28 on page 99) 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/VSE.) 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 LE/VSE default actions take place, as described in Table 27 on page 99. 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, LE/VSE terminates the thread.

User handlers that you register using CEEHDLR must be written in the same language you are using to do the registration.

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/I STOP 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.
- An LE/VSE-initiated abend.
- A user-requested abend (call to CEE5ABD).

  You can call CEE5ABD 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 *LE/VSE Programming Reference* for more information about the CEE5ABD callable service.

  If you call CEE5ABD and request an abend with clean-up, T_I_U is signaled. Condition handlers are given a chance to handle the abend. If the abend remains unhandled, normal LE/VSE termination activities occur. For example, the C `atexit` list is honored if a C routine is present on the stack, and the LE/VSE assembler user exit gains control.

Sample N-Way ILC Application

IBMNWAY

```
*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 that calls a COBOL program.       */
  /*                                                         */
  /* Our example illustrates a 3-way interlanguage call from */
  /* a PL/I main program to a C routine 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 the 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 30. PL/I Main Routine of ILC Application

EDCNWAY

```
 /*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>
#pragma linkage (NWAY2C,PLI)
#pragma linkage (NWAY2CB,COBOL)
void NWAY2CB(char *);
  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 lower case 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 31. C Routine Called by PL/I in a 3-Way ILC Application

IGZTNWAY

```
 CBL APOST
*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 32. COBOL Routine Called by C in a 3-Way ILC Application

Chapter 6. Communicating between Assembler and HLLs

This chapter describes LE/VSE's support for assembler– ILC applications. Throughout this chapter, an LE/VSE-conforming assembler routine refers to an assembler routine coded using the CEEENTRY and associated macros, and assembled using High Level Assembler (HLASM).

Note: For further information on using assembler under LE/VSE, refer to the:

- *LE/VSE Programming Guide*.
- IBM-provided samples that you can find at the *z/VSE Home Page* whose URL is given in "Where to Find More Information" on page xv.

Calling Assembler from an HLL

LE/VSE-conforming assembler can be dynamically called/fetched from any LE/VSE-conforming HLL. In addition, LE/VSE-conforming assembler can dynamically load another routine by using the LE/VSE CEELOAD macro.

Using CEELOAD imposes restrictions on further dynamic loading or dynamic calls or fetches, you *cannot*:

- Dynamically load a routine with CEELOAD that has already been dynamically loaded by CEELOAD or been fetched or dynamically called.
- Fetch or dynamically call a routine that has already been dynamically loaded by CEELOAD.
- CEELOAD any C routine(s) that have been prelinked. Instead, you should use the CEEFETCH/CEERELES macros.

Results are unpredictable if these rules are violated.

The syntax of CEELOAD is described in *LE/VSE Programming Guide*.

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.

C

For assembler to call C, or vice versa, you should include the following statement in the C code:

```
#pragma linkage(...,OS)
```

PL/I

For assembler to call PL/I, OPTIONS(BYVALUE) should be specified on the declaration of the PL/I procedure. For PL/I to call assembler, OPTIONS(ASSEMBLER) should be specified on the declaration of the external assembler routine.

Cancelling or Releasing Assembler

An assembler routine must be released using the same language that called/fetched it. COBOL, C, and PL/I can only CANCEL or release the assembler routine if there is no ILC with other HLLs in the target phases.

Restrictions on the COBOL CANCEL Statement

Under certain configurations, using a CANCEL statement in an application that consists of multiple phases with a mixture of COBOL/VSE or VS COBOL II and assembler (LE/VSE-conforming or non-LE/VSE-conforming) routines is not supported and can result in program checks. The following describes one such configuration.

An application might consist of these routines:

- COBOL RTNA linked with assembler routine ASMI (further references to this instance of ASMI are in the form ASMI(A))
- COBOL RTNX linked with assembler routine ASMI (further references to this instance of ASMI are in the form ASMI(X))
- Assembler routine ASM
- COBOL routines RTNB, RTNC, RTND, RTNE (these routines are all contained in the RTNB phase)

All of the COBOL routines are compiled with NODYNAM and RES. All CALLs in the COBOL routines are static, except where RTNA calls RTNX; this is a dynamic CALL.

The purpose of ASMI is to check if ASM is loaded, and if it is, branch to it. If ASM is not loaded, ASMI loads it and branches to it.

The purpose of ASM is to check if RTNB is loaded, and if it is, branch to it. If RTNB is not loaded, ASM loads it and branches to it.

The routine flow is as follows:

1. RTNA => ASMI(A) => ASM => RTNB => RTND => RTNE
2. Return to RTNA by reversing the route (all COBOL routines use GOBACK)
3. RTNA => RTNX => ASMI(X) => ASM => RTNB => RTNC => RTNE
4. Return to RTNA by reversing the route
5. RTNA then issues a CANCEL for RTNX
6. RTNA => ASMI(A) => ASM => RTNB => RTND => RTNE

Because storage is released for RTNB, RTNC, and RTNE in the scope of the CANCEL statement in step 5, a program check results in step 6.

If you plan to use a configuration such as this, avoid the use of CANCEL or use COBOL programming that uses dynamic calls instead of assembler language programming, such as the combination of ASMI and ASM that uses CDLOAD/BALR.

LE/VSE-Conforming Assembler Invoking an HLL Main Routine

When a C or PL/I main routine is called from LE/VSE-conforming assembler, the actions in Table 25 on page 89 take place.

Note: Unlike C and PL/I, COBOL has no mechanism to statically declare a routine "main"; rather, a main routine is determined dynamically when a COBOL routine is the first routine in an enclave. Therefore, it makes sense to discuss calling a COBOL main only in the context of creating a new enclave in which a COBOL routine is the first to run. Only parts of the following tables apply to a COBOL main.

Table 25. What Occurs When LE/VSE-Conforming Assembler Invokes an HLL Main

Type of Assembler Invocation	LE/VSE Is Up
CEELOAD macro	Symbolic feedback code CEE393 is signaled. CEELOAD cannot load a main routine.
EXEC CICS LINK and EXEC CICS XCTL	Nested enclave is created. The COBOL routine could be a main in this case.
EXEC CICS LOAD and BALR	This is not supported.
CDLOAD and BALR	Symbolic feedback code CEE393 is signaled. You cannot load and BALR a main routine under LE/VSE. However, this is supported in COBOL because the COBOL routine would be a subroutine, not a main.

Note: See *LE/VSE Programming Guide* for information on nested enclaves.

Non-LE/VSE-Conforming Assembler Invoking an HLL Main Routine

When a C, COBOL, or PL/I main routine is called from a non-LE/VSE-conforming assembler routine, the actions in Table 26 take place.

Table 26. What Occurs When Non-LE/VSE-Conforming Assembler Invokes an HLL Main

Type of Assembler Invocation	LE/VSE Is Not Up	LE/VSE Is Up
EXEC CICS LINK and EXEC CICS XCTL	Initial enclave is created.	Nested enclave is created. The COBOL routine could be a main in this case.
EXEC CICS LOAD and BALR	This is not supported.	This is not supported.
CDLOAD and BALR	Initial enclave is created.	Symbolic feedback code CEE393 is signaled. You cannot load and BALR a main routine under LE/VSE. However, this is supported in COBOL because the COBOL routine would be a subroutine, not a main.

Note: The supported method for using assembler programs with other LE/VSE-conforming HLL programs is to use the *IBM-provided LE/VSE assembler macros*. For further information, refer to the *LE/VSE Programming Guide*.

Assembler Main Calling HLL Subroutines for Better Performance

To improve performance of a C, COBOL, or PL/I routine called repeatedly from assembler, use an LE/VSE-conforming assembler routine in a preinitialized environment, because the LE/VSE environment is maintained across calls. If the assembler routine is not LE/VSE-conforming, the LE/VSE environment is initialized and terminated at every call. (See *LE/VSE Programming Guide* for information on creating and 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 CEEFETCH or CEELOAD macros). The assembler routine is a main routine and the called HLL routine is a subroutine.

For example, see Figure 33 on page 91, which demonstrates an LE/VSE-conforming assembler routine statically calling a COBOL routine.

CEEILCOB

```
*/* Module/File Name: CEEILCOB */

* ====================================================================
*   Bring up the LE/VSE environment
* ====================================================================
CEEILCOB CEEENTRY PPA=MAINPPA
         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
         DUMP  RC=(2)                  LIGHTS OUT!*
*   Terminate the LE/VSE 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   CEEILCOB
```

Figure 33. LE/VSE-Conforming Assembler Routine Calling COBOL Routine

IGZTASM

```
*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 34. COBOL Routine Called from LE/VSE-Conforming Assembler

Chapter 7. ILC under CICS

In general, LE/VSE 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 an LE/VSE-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.

Language Pairs Supported in ILC under CICS

To understand what support LE/VSE offers your ILC application, see the description for the specific language pair below, and the applicable ILC chapter. If your ILC application involves multiple HLLs, see Chapter 5, "Communicating between Multiple HLLs," on page 79.

C and COBOL

LE/VSE supports ILC between routines written in C and COBOL under CICS as follows:

- C routines can statically call COBOL/VSE routines.
- COBOL/VSE routines can statically call C routines.
- C routines can `fetch()` COBOL/VSE routines.
- COBOL/VSE routines can dynamically CALL prelinked C fetchable subroutines.
- Dynamically-called C and COBOL routines that are statically linked together must be:
  - prelinked together
  - reentrant
- C routines calling COBOL routines must pass the EIB and COMMAREA as the first two parameters if the called program has been translated.
- There is **no** support for ILC calls to or from routines written in pre-LE/VSE-conforming versions of C or COBOL.

All components of your C–COBOL ILC application must be reentrant.

For more information on ILC between C and COBOL, see Chapter 2, "Communicating between C and COBOL," on page 3.

C and PL/I

LE/VSE supports ILC between routines written in C and PL/I VSE under CICS as follows:

- C routines can statically call PL/I routines.
- PL/I routines can statically call C routines.
- C routines can `fetch()` PL/I routines that have OPTIONS(FETCHABLE) specified.
- PL/I routines can FETCH only those C routines that have not been run through the CICS translator. A PL/I routine cannot dynamically call a 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 C from PL/I, the static read/write pointer is not swapped.

- 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-LE/VSE-conforming versions of C or PL/I.

All components of your C–PL/I ILC application must be reentrant.

For more information on ILC between PL/I and C, see Chapter 3, "Communicating between C and PL/I," on page 29.

COBOL and PL/I

LE/VSE supports ILC between routines compiled with COBOL/VSE and PL/I VSE under CICS as follows:

- COBOL/VSE routines can statically call PL/I routines.
- PL/I routines can statically call COBOL/VSE routines.
- COBOL/VSE routines can dynamically CALL PL/I routines that have OPTIONS(FETCHABLE) specified.
- PL/I routines can FETCH COBOL/VSE routines.
- PL/I routines calling COBOL/VSE routines must pass the EIB and COMMAREA as the first two parameters if the called program contains any EXEC CICS commands.

For more information on ILC between COBOL and PL/I, see Chapter 4, "Communicating between COBOL and PL/I," on page 55.

If there is any ILC within a run unit under CICS, each compile unit must be compiled with an LE/VSE-conforming compiler.

There is no support for CICS for ILC calls to or from routines written in previous versions of COBOL or PL/I.

Assembler

LE/VSE-conforming assembler main routines are supported under CICS providing they:

- Do not use the HANDLE LABEL option of any appropriate EXEC CICS commands.
- Use the NOEPILOG and NOPROLOG translator options.

When calling a High Level Language (HLL) routine that will use CICS services from an assembler main routine, you must ensure that the appropriate CICS control blocks (EIB, DFHCOMMAREA) are passed as parameters.

To initialize initial stack storage, LE/VSE-conforming assembler main programs under CICS should use the run-time option STORAGE=(00,NONE,CLEAR,nnk) either as an application-specific setting, or as a CICS-wide installation default setting.

For information on the use of CICS commands in an assembler language, refer to the CICS documentation.

COBOL Considerations

Static calls are allowed in VS COBOL II and COBOL/VSE **to** but not **from** routines written in non-LE/VSE-conforming assembler.

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.

For information on the use of CICS commands in an assembler language subroutine, refer to the CICS documentation.

Link-Editing ILC Applications under CICS

You must link ILC applications with the CICS stub, DFHELII, in order to get ILC support under LE/VSE.

ILC applications in which C is one of the participating languages must be link-edited AMODE(31).

Any dynamically-called COBOL/C routines that are statically linked together must be prelinked together before linkediting.

You cannot relink an old C object program (for example, you are running an old batch application under CICS).

CICS ILC Application

The following examples illustrate how you can use ILC under CICS. A COBOL main routine, 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 routine

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 LE/VSE service CEEMOUT to write the message to the CESE transient data queue.

IGZTCICS

```
 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 35. COBOL CICS Main Program That Calls C and PL/I Subroutines

IBMCICS

```
/*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 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 36. PL/I Routine Called by COBOL CICS Main Program

EDCCICS

```
 /*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.         */
 /*                                                                  */
 /*********************************************************************/
#pragma linkage(CUCICS,COBOL)
#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 37. C Routine Called by COBOL CICS Main Program

Appendix. Condition Handling Responses

Table 27 and Table 28 list condition handling responses, as referenced in the condition handling sections of the pairwise chapters.

Table 27. LE/VSE Default Responses to Unhandled Conditions. LE/VSE'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 (*LE/VSE 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 28 contains default C language error handling semantics.

Table 28. C Conditions 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)
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)

Table 28. C Conditions and Default System Actions (continued)

C Condition	Origin	Default Action
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 CEE5SPM callable service.

Language Environment Glossary

A

abend. Abnormal end of application.

addressing mode. An attribute that refers to the address length that a routine is prepared to handle upon entry. Addresses may be 24 or 31 bits long.

aggregate. A structured collection of data items that form a single data type. Contrast with *scalar*.

AMODE. Addressing mode.

application. A collection of one or more routines cooperating to achieve particular objectives.

argument. An expression used at the point of a call to specify a data item or aggregate to be passed to the called routine.

array. An aggregate that consists of data objects, each of which may be uniquely referenced by subscripting.

array element. A data item in an array.

assembler. see *High Level Assembler*.

atexit list. A list of actions specified in the C `atexit()` function that occur at normal program termination.

B

by content. See *pass by content*.

by reference. See *pass by reference*.

by value. See *pass by value*.

C

C language. A high-level language used to develop software applications in compact, efficient code that can be run on different types of computers with minimal change.

callable services. A set of services that can be invoked by an LE/VSE-conforming high level language using the conventional LE/VSE-defined call interface, and usable by all programs sharing the LE/VSE conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

callee. Receiver of a call.

caller. A routine that calls another routine.

CICS. Customer Information Control System.

CICS run unit. Consists of a statically and/or dynamically bound set of one or more phases which can be loaded by a CICS loader. A CICS run unit is equivalent to an LE/VSE *enclave*.

CICS translator. A routine that accepts as input an application containing EXEC CICS commands and produces as output an equivalent application in which each CICS command has been translated into the language of the source.

COBOL. COmmon Business-Oriented Language. A high level language, based on English, that is primarily used for business applications.

COBOL run unit. A COBOL-specific term that defines the scope of language semantics. Equivalent to an LE/VSE *enclave*.

command line. The command used to invoke an application program, and the associated program arguments and LE/VSE run-time options. This can be the job control EXEC statement and the associated PARM parameter, or the parameter string passed to the C `system()` function.

COMMAREA. A communication area made available to applications running under CICS.

compilation unit. An independently compilable sequence of HLL statements. Each HLL product has different rules for what makes up a compilation unit. Synonym for *program unit*.

condition. An exception that has been enabled, or recognized, by LE/VSE and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

condition handler. A user-written condition handler or language-specific condition handler (such as a PL/I ON-unit) invoked by the LE/VSE *condition manager* to respond to conditions.

condition handling. In LE/VSE, the diagnosis, reporting, and/or tolerating of errors that occur in the run-time environment.

condition manager. Manages conditions in the common execution environment by invoking various user-written and language-specific *condition handlers*.

condition step. The step of the LE/VSE condition handling model that follows the enablement step. In the condition step, user-written condition handlers and PL/I ON-units are first given a chance to handle a condition. See also *enablement step* and *termination imminent step*.

condition token. In LE/VSE, a data type consisting of 96 bits (12 bytes). The condition token contains structured fields that indicate various aspects of a condition including the severity, the associated message number, and information that is specific to a given instance of the condition.

constructed reentrancy. The attribute of applications that contain external data and require additional processing to make them reentrant. Contrast with *natural reentrancy*.

D

data type. The properties and internal representation that characterize data.

default. A value that is used when no alternative is specified.

disabled/enabled. See *enabled/disabled*.

double-precision. Pertaining to the use of two computer words to represent a number in accordance with the required precision. See also *precision* and *single-precision*.

dynamic call. A call that results in the resolution of the called routine at run time. Contrast with *static call*.

E

enabled/disabled. A condition is enabled when its occurrence will result in the execution of condition handlers or in the performance of a standard system action to handle the condition as defined by LE/VSE.

A condition is disabled when its occurrence will apparently be ignored by the condition manager.

enablement. The determination by a language at run time that an exception should be processed as a condition. This is the capability to intercept an exception and to determine whether it should be ignored or not; unrecognized exceptions are always defined to be enabled. Normally, enablement is used to supplement the hardware for capabilities that it does not have and for language enforcement of the language's semantics. An example of supplementing the hardware is the specialized handling of floating-point overflow exceptions based on language specifications (on some machines this can be achieved through masking the exception).

enablement step. The first step of the LE/VSE condition handling model. In the enablement step it is determined whether an exception is to be *enabled* and processed as a condition. See also *condition step* and *termination imminent step*.

enclave. In LE/VSE, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

entry point. In assembler language, the address or label of the first instruction that is executed when a routine is entered for execution.

environment. A set of services and data available to a program during execution. In LE/VSE, environment is normally a reference to the run-time environment of HLLs at the enclave level.

EXEC interface block (EIB). In CICS, a control block containing information useful in the execution of an application, such as a transaction identifier and a time and a date when the transaction is started.

exception. The original event such as a hardware signal, software detected event, or user-signaled event which is a potential condition. This action may or may not include an alteration in a program's normal flow. See also *condition*.

execution time. Synonym for *run time*.

execution environment. Synonym for *run-time environment*.

external data. Data that persists over the lifetime of an enclave and maintains last-used values whenever a routine within the enclave is reentered. Within an enclave consisting of a single phase, it is equivalent to COBOL external data.

F

feedback code (fc). A condition token value. If you specify *fc* in a call to a callable service, a condition token indicating whether the service completed successfully is returned to the calling routine.

fetchable main. A IBM PL/I for VSE/ESA routine specified with PROC OPTIONS(MAIN).

The routine performing the FETCH and CALL must be compiled with the IBM PL/I for VSE/ESA compiler.

fixed decimal. See *packed decimal format*.

function. A routine that is invoked by coding its name in an expression. The routine passes a result back to the invoker through the routine name.

H

handle cursor. Points to the first condition handler within the stack frame that is to be invoked when a condition occurs. As condition handling progresses, the handle cursor moves to earlier handlers within the stack frame, or to the first handler in the calling stack frame.

header file. A file that contains system-defined control information that precedes user data.

high level language (HLL). A programming language above the level of assembler language and below that of program generators and query languages.

HLL. High level language.

I

ILC. Interlanguage communication.

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

interlanguage communication (ILC). The ability of routines written in different programming languages to communicate. ILC support allows the application writer to readily build applications from component routines written in a variety of languages.

L

Language Environment. A set of architectural constructs and interfaces that provides a common run-time environment and run-time services to applications compiled by Language Environment-conforming compilers.

Language Environment for z/VSE. An IBM software product that is the implementation of Language Environment on the VSE platform.

LE/VSE. Short form of Language Environment for z/VSE.

LE/VSE-conforming. Adhering to LE/VSE's common interface.

linkage editor. A program that resolves cross-references between separately assembled object modules and then assigns final addresses to create a single relocatable phase. The linkage editor then stores the phase in a program library in main storage.

link-edit. To create a loadable computer program by means of a linkage editor.

M

main program. The first routine in an enclave to gain control from the invoker.

N

n-way ILC application. An ILC application that includes a C routine, COBOL program, and PL/I routine.

name scope. The portion of an application within which a particular declaration of external data applies or is known.

name space. The portion of a phase within which a particular declaration of external data applies or is known.

natural reentrancy. The attribute of applications that contain no static external data and do not require additional processing to make them reentrant. Contrast with *constructed reentrancy*.

nested program. In COBOL, a program that is directly contained within another program.

non-LE/VSE conforming. Any HLL program that does not adhere to LE/VSE's common interface. For example, VS COBOL II, DOS/VS COBOL, and DOS/VS PL/I are all non-LE/VSE conforming HLLs. Synonym for *pre-LE/VSE conforming*.

O

object module. A portion of an object program suitable as input to a linkage editor. Synonym for *object deck*.

ON-unit. The specified action to be taken upon detection of the condition named in the containing ON statement.

P

packed decimal format. A format in which each byte in a field except the rightmost byte represents two numeric digits. The rightmost byte contains one digit and the sign. For example, the decimal value +123 is represented as 0001 0010 0011 1100.

parameter. Data items that are received by a routine.

pass by content. A COBOL argument passing style synonymous with passing an argument by value directly. In this style, R1 contains a pointer to a copy of the argument.

pass by reference. In programming languages, one of the basic argument passing semantics. The address of the object is passed. Any changes made by the callee to

the argument value will be reflected in the calling routine at the time the change is made.

pass by value. In programming languages, one of the basic argument passing semantics. The value of the object is passed. Any changes made by the callee to the argument value will not be reflected in the calling routine.

phase. An application or routine in a form suitable for execution. The application or routine has been compiled and link-edited; that is, address constants have been resolved.

PL/I. A general purpose scientific/business high level language. It is a high-powered procedure-oriented language especially well suited for solving complex scientific problems or running lengthy and complicated business transactions and record-keeping applications.

pointer. A data element that indicates the location of another data element.

precision. A measure of the ability to distinguish between nearly equal values. See also *single-precision* and *double-precision*.

pre-LE/VSE conforming. Any HLL program that does not adhere to LE/VSE's common interface. For example, VS COBOL II, DOS/VS COBOL, and DOS/VS PL/I are all pre-LE/VSE conforming HLLs. Synonym for *non-LE/VSE conforming*.

procedure. A named block of code that can be invoked, usually via a call. In LE/VSE, the term *routine* is used as generic for a procedure or a function.

R

reentrant. The attribute of a routine or application that allows more than one user to share a single copy of a phase.

resume. To begin execution in an application at the point immediately after which a condition occurred. A resume occurs when the condition manager determines that a condition has been handled and normal application execution should continue.

resume cursor. Designates the point in the application where a condition occurred when it is first reported to the condition manager. The resume cursor also designates the point where execution resumes after a condition is handled, usually at the instruction in the application immediately following the point at which the error occurred. The resume cursor can be moved with the CEEMRCR callable service.

routine. In this book, used as a general term to describe a named external routine, written in any of the languages discussed, and with or without internal (contained) subroutines.

run time. Any instant at which a program is being executed. Synonym for *execution time*.

run-time environment. A set of resources that are used to support the execution of a program. Synonym for *execution environment*.

run unit. One or more object programs that are executed together. In LE/VSE, a run unit is the equivalent of an *enclave*.

S

scalar. A quantity characterized by a single value. Contrast with *aggregate*.

scope. 1. A term used to describe the effective range of the enablement of a condition and/or the establishment of a user-generated routine to handle a condition. Scope can be both statically and dynamically defined. 2. The portion of an application within which the definition of a variable remains unchanged.

single-precision. Pertaining to the use of one computer word to represent a number in accordance with the required precision. Needed for proper alignment. See also *precision* and *double-precision*.

static call. A call that results in the resolution of the called program statically at link-edit time. Contrast with *dynamic call*.

subroutine. In general, any routine within an application called by another routine.

T

thread. The basic run-time path within the LE/VSE program management model. It is dispatched by the system with its own instruction counter and registers. The thread is where actual code resides.

termination imminent step. The final step of the 3-step LE/VSE condition handling model. In the termination imminent step, user-written condition handlers and PL/I ON-units are given one last chance to handle a condition or perform cleanup before the thread is terminated. See also *condition step* and *enablement step*.

U

unpacked decimal format. A format for representing numbers in which the digit is contained in bits 4 through 7 and the sign is contained in bits 0 through 3 of the rightmost byte. Bits 0 through 3 of all other bytes contain 1s (hex F). For example, the decimal value of +123 is represented as 1111 0001 1111 0010 1111 0011. Synonym for *zoned decimal format*.

user-written condition handler. A routine established by the CEEHDLR callable service to handle a condition or conditions when they occur in the common run-time environment. A queue of user-written condition handlers established by CEEHDLR may be associated with each stack frame in which they are established.

V

void function. The C representation of a procedure invocation. A void function is a function that does not return a value.

VSE (Virtual Storage Extended). A system that consists of a basic operating system (VSE/Advanced Functions) and IBM-supplied programs required to meet the data processing needs of a user.

W

writable static. In C, writable static may be any of the following:

- Program variables with the `extern` storage class
- Program variables with the `static` storage class
- Writable strings

The LE/VSE term for writable static is *external data*.

Z

zoned decimal format. Synonym for *unpacked decimal format*.

Index

T

V

Readers' Comments — We'd Like to Hear from You

IBM Language Environment for z/VSE
Writing Interlanguage Communication Applications
Version 1 Release 4 Modification Level 6

Publication No. SC33-6686-02

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:

- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: FAX (Germany): 07031+16-3456
  FAX (Other Countries): (+49)+7031-16-3456
- Send your comments via e-mail to: s390id@de.ibm.com

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	E-mail address

Readers' Comments — We'd Like to Hear from You
SC33-6686-02

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Entwicklung GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

Fold and Tape | **Please do not staple** | Fold and Tape

SC33-6686-02

Cut or Fold
Along Line

File Number: S370/S390-40
Program Number: 5686-CF8

SC33-6686-02

Spine information:

IBM Language Environment for z/VSE

LE/VSE V1R4.6 Writing ILC Applications

Version 1
Release 4 Modification
Level 6

SC33-6686-02