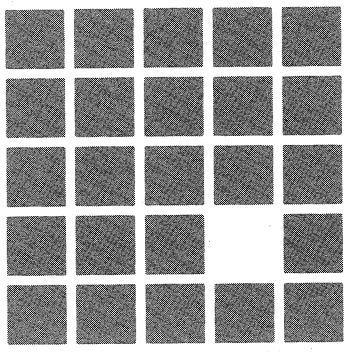


SAS/C® Compiler Interlanguage Communication Feature User's Guide

SAS Institute Inc.

5684

SAS/C® Compiler Interlanguage Communication Feature User's Guide

Release 4.00

SAS Institute Inc.
SAS Circle □ Box 8000
Cary, NC 27512-8000

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. *SAS/C® Compiler Interlanguage Communication Feature User's Guide*, Cary, NC: SAS Institute Inc., 1989. 260 pp.

SAS/C® Compiler Interlanguage Communication Feature User's Guide

ISBN 1-55544-323-0

1st printing, February 1989
2nd printing, September 1989

Note that text corrections may have been made at each printing.

Contents

Examples

Illustrations

Figures

Tables

Credits

Publication Credits

Composition	Gail Freeman, Pat Gervason, Kelly Godfrey, Pam Troutman, Penny Wiard
Graphics	Ginny Matsey
Proofreading	Reid J. Hardin, Susan H. McCoy, Philip R. Shelton, Toni P. Sherrill, Michael H. Smith, Helen F. Weeks, John M. West, Susan E. Willard
Technical Review	Twilah K. Blevins, Oliver Bradley, Anne Corrigan, Mark K. Gass, Jodie M. Gilmore
Writing and Editing	Ingrid Ammondson, Alan Beale, Rick V. Cornell Jr., Jodie M. Gilmore, Tim P. Hunter, Gary R. Meek, Curtis A. Yeo

Software Credits

Many people have contributed to the SAS/C compiler interlanguage communication feature. Principal developers are listed below by product:

Design and Implementation	Alan Beale, Oliver Bradley
ILCLINK Utility	Tim P. Hunter
Pascal Support	Ingrid Ammondson

We owe a great debt to all of our users who have made suggestions for product improvements and to those who have reported bugs.

TESTING AND TECHNICAL SUPPORT:

Good testing and technical support are a vital ingredient of a quality software product. SAS/C ILC would not have been possible without those listed below:

Ingrid Ammondson

Twilah K. Blevins

Karen E. Chacko

Bob Patten Jr.

How to Use This Book

The following sections provide an overview of this book, including its purpose and organization, what you need to know, a reading path for you to follow, some additional documentation you may want to have available, and an explanation of the typographical conventions used in the text.

Purpose

This book documents the Interlanguage Communication (ILC) feature of SAS/C software, Release 4.00. With ILC you can write programs that use C and one or more other high-level languages (such as FORTRAN, COBOL, PL/I, or Pascal) at the same time. Multilanguage programs are highly flexible because you can use each language to do what it does best. For example, C is good for numerical methods, while COBOL is good for text processing. However, such programs are more complex than single-language ones. This book explains how to deal with these complexities.

What You Need to Know

To use this book effectively, you must be familiar with C and at least one other language (usually FORTRAN, COBOL, PL/I, or Pascal). *Familiar* implies that you know how to compile, link, and execute C programs and programs written in the other language, and that you know the various data types and calling conventions (call by reference, call by value, and so on) used by each language. You should also know how your operating system (TSO, CMS, or OS-batch) stores files, how to specify compiler options, and how to issue system commands. This level of experience is sufficient for most readers. If you intend to write your own interfaces to user-supported languages (those languages not supported directly by SAS/C ILC), you need an even greater knowledge of the languages and your operating system.

Organization

This book has two parts. Part 1 gives background information on interlanguage communication and documents the use of ILC with the four standard high-level languages (FORTRAN, COBOL, PL/I, and Pascal). Part 2 discusses using ILC with user-supported languages.

Part 1 Part 1, "Using SAS/C ILC," presents general background information about interlanguage communication and then moves on to the details of using ILC with each of the four supported languages (FORTRAN, COBOL, PL/I, and Pascal). Included in Part 1 are chapters of overall

applicability, such as instructions for linking and debugging interlanguage applications.

Chapters 1–3 give an overview of interlanguage communication. These chapters include a **Terms to Know** section, which lists terms in the chapter that are new or ambiguous.

Chapters 4–7 discuss the details of using ILC with FORTRAN, COBOL, PL/I, and Pascal, respectively.

Chapter 8 discusses in detail linking multilanguage programs with the ILCLINK utility under TSO, CMS, and OS-batch.

Chapter 9 discusses debugging multilanguage programs. It includes a list of common mistakes and explanations of common diagnostic messages.

Chapter 10 covers advanced topics for readers who need to use unusual features of SAS/C software or of other languages in their multilanguage programs.

Chapter 11 is a reference chapter for ILC framework manipulation routines, including synopses, descriptions, and examples.

Chapter 12 addresses the use of packed decimal data in C.

Chapter 13 discusses the use of varying-length string macros in C.

Part 2 In addition to supporting communication with FORTRAN, COBOL, PL/I, and Pascal, the SAS/C ILC feature can be used to communicate with other, less widely used languages. Part 2, "Extending SAS/C ILC," describes how to extend SAS/C ILC to support another language. For example, a site with an Ada compiler can extend ILC so that calls can be made between Ada and C. Each chapter in this part of the book is described below.

Chapter 14 discusses how to use an interface to a user-supported language. In addition to this chapter, users need to refer to the documentation produced by the implementor of the interface to the language in order to use it effectively.

Note that the interface implementor should also read this chapter for a complete understanding of how the interface will be used and of the documentation required.

Chapter 15 discusses on a conceptual level implementing ILC with a user-supported language. This chapter gives the implementor a basic understanding of how SAS/C ILC works before the implementor moves on to Chapter 16, which is more detailed.

Chapter 16 discusses in detail implementing ILC with a user-supported language. Because the material in this chapter is complex, the reader must have an expert understanding of the language and of SAS/C ILC.

Appendices

Appendix 1 lists the ILC run-time messages with their causes and possible resolutions.

Appendix 2 lists the ILCLINK diagnostic messages with their causes and possible resolutions.

Suggested Reading Path

Read Chapters 1 through 3 thoroughly before reading anything else. These chapters provide the necessary background for using the SAS/C ILC feature.

Read Chapters 4 through 7, 10, 12, and 13 as necessary, depending on which language you use and what you want to accomplish with your multilanguage application. If you are using or implementing a user-supported language interface, you should still read at least one of Chapters 4 through 7 to see how ILC works.

Read Chapter 8 before linking your multilanguage program. This chapter discusses the ILCLINK utility in great detail and has many helpful examples for each operating system.

Read Chapter 9 before executing your program. This chapter points out many common ILC programming errors. By reading it *before* you run your program, you can prevent much frustration and anxiety.

Refer to Chapter 11 as necessary when you create and terminate execution frameworks.

Read Chapter 14 if you are using a user-supported language. This chapter will start you in the right direction, although it does not discuss a specific language in detail. Before reading this chapter, you should read Chapters 1 through 3, 8, 9, and at least one of Chapters 4 through 7.

Read Chapters 15 and 16 if you are implementing a user-supported language interface. These chapters show you how to implement and document correctly the interface for your users. Before reading these chapters, you should read Chapters 1 through 3, 8 through 14, and at least one of Chapters 4 through 7.

Other Useful Documentation

As you use the SAS/C ILC feature, it will probably be helpful to have additional documentation at hand. Below is a list of books you may want to have available.

- *Technical Report C-106, Changes and Enhancements to the SAS/C Compiler, Release 4.00*
- *SAS/C Compiler and Library User's Guide*
- *SAS/C Library Reference, Volume 1*
- *SAS/C Library Reference, Volume 2*
- *SAS/C Source Level Debugger User's Guide*
- IBM® manual, *System/370 Principles of Operation* (GA22-7000)
- a language reference guide for each language you use
- a programmer's reference guide for each language you use
- a reference guide for your operating system.

Typographical Conventions

Certain typefaces represent special types of information in the text. The list below demonstrates these.

bold	indicates an important word or concept.
`code`	indicates the item is specific to the C language or a C program. Note that C terms appearing in headings are not in code.
italic type	indicates a term that is being defined or emphasizes an important word in a sentence.

Part 1
Using SAS/C ILC

Chapters

Part 1 presents in Chapters 1 through 3 general information about interlanguage communication. Chapters 4 through 7 describe using SAS/C ILC with the supported languages (FORTRAN, COBOL, PL/I, and Pascal). Chapters 8 through 13 are of general applicability, including discussions of linking and debugging multilanguage programs, advanced topics, reference chapters on framework manipulation routines, and use of packed decimal data and varying-length string macros.

If you plan to use SAS/C ILC with only the supported languages, this part is all you need to read.

1 Introduction to Interlanguage Communication

Introduction

This chapter introduces the four major components of interlanguage communication:

- execution frameworks
- data formats and communication
- language calls, both from another language to C and from C to another language
- linkage of multilanguage applications.

It is important that you grasp these fundamental principles of interlanguage communication before going on to create and run a multilanguage application.

Terms to Know

This section defines terms you should know before reading the chapter. Cross-references are given in italics.

argument
: a variable or expression that is passed from one *routine* to another; also called a parameter.

call by reference
: a method of passing an *argument* between *routines*, where the location of the argument is passed and the called routine may change its value. A copy of the argument is not created. It is also called "pass by reference." Most *high-level languages* other than C use this technique for passing arguments.

call by value
: a method of passing an *argument* between *routines*, where the calling routine creates and passes a temporary copy of the argument. Processing of this copy does not affect the actual value. It is also called "pass by value." C uses this technique to pass arguments to other C functions.

data format
: the physical representation of a *data type*.

data type
: a set of values, together with a set of operations defined on those values; it is usually language specific.

data type conversion macro
: ILC predefined macro that assists you in passing certain types of data, such as arrays, character strings, and bit data, to PL/I, FORTRAN, or Pascal.

environment
: see *execution framework.*

execution framework
: a collection of data and run-time library routines that support the execution of code.

framework
: see *execution framework.*

framework switch
: an action performed by the SAS/C library when one language calls another; the calling language's *execution framework* is made inactive, and the called language's execution framework is made active.

function
: a *routine* written in C to perform a specific task, which may return a value to the function's caller. Functions that do not return a value are *void functions.*

FUNCTION
: a *routine* that returns a value or result to its caller. This term is taken from FORTRAN, but it is used in this book for value-returning *routines* written in any *high-level language.*

high-level language
: a programming language that is independent of a machine-specific instruction set. A single high-level language statement can stand for and execute many machine instructions. C, FORTRAN, COBOL, PL/I, and Pascal are examples of high-level languages; assembler language is not. Note that to be used with SAS/C ILC, a high-level language must have a library and a *framework.*

HLL function pointer
: a pointer to a function that is in a language other than C or assembler.

routine
: a general term for body of code that performs a specific task, written in some programming language. A routine written in C is called a *function* in this book.

run-time library
: a collection of *routines* that can be linked with programs in a particular programming language. These routines perform functions such as memory management, error handling, I/O, and string handling, plus functions unique to a particular language. Each language has its own unique run-time library, and a multilanguage program requires the use of several.

subroutine
: a general term for a *routine* that is called from another routine. This book uses subroutine to describe both C *functions* and non-C *routines*.

SUBROUTINE
: a *routine* that returns no value or result to its caller. This term is taken from FORTRAN, but it is used in this book for *routines* in any *high-level language*.

void function
: a C *function* that does not return a value to its caller.

Advantages of Interlanguage Communication

It is often desirable to combine routines that are written in several programming languages into a single program. Doing so enables you to

- use existing subroutines or subroutine libraries from one language in another language
- write the parts of a program in the language most natural for that application, for instance, writing report generation routines in COBOL
- convert an application gradually from one language to another.

The interlanguage communication support provided by SAS/C software facilitates the development of programs that combine C with other high-level languages. Communication with FORTRAN, COBOL, PL/I, and Pascal is supported. In addition, a site can write its own support for other languages.

Note that this book does not describe communication between C and assembler language. This is much simpler than communication between C and high-level languages, and is described in the *SAS/C Compiler and Library User's Guide*.

Overview

Developing and debugging a multilanguage program is rarely as simple as a program involving only a single language. The goal of the SAS/C interlanguage communication support is to minimize the programming difficulties and reduce the detailed knowledge required to create a robust multilanguage program. In particular, there should be no need to write assembler language interfaces or to understand the internal details of another language, except for its data type representations.

There are two reasons that interlanguage calls require more effort and care in coding than corresponding calls in the same language. Different languages have different *execution frameworks*, and different languages have different conventions for storing and transferring data.

Execution Frameworks

As a general rule, successful execution of code written in a high-level language requires an appropriate execution framework. An execution framework (often called an *environment*) is a collection of data and routines supporting the execution of code. (For instance, memory

allocation tables and error-handling routines are usually components of an execution framework.) The execution framework is normally created by the supporting run-time library when the main program begins execution. The execution framework is unique to each high-level language; therefore, one must be created for each language by some means before code in that language can execute successfully.

For a program written in a high-level language to execute successfully, the execution framework must not only exist; it must also be accessible to the program. This means that, in general, machine registers must be set up to address components of the framework. For example, C code will not execute successfully if register 12 does not address the C Run-time Anchor Block (CRAB). Because each language has its own conventions for framework access, it is usually impossible for more than one framework to be accessible at the same time. Therefore, a call from one language to another must "switch frameworks," that is, make the new language's framework accessible (or *active*) before performing the call and restore the calling framework after the call is complete.

The SAS/C ILC support provides routines to create and delete execution frameworks for C and for the various supported languages. For example, if a main FORTRAN routine is going to call C, it must first call the C library routine `CFMWK` to create the C framework. Similarly, after all calls to C are complete, it must call the `DCFMWK` routine to delete the C framework. (See Chapter 2, "Multilanguage Framework Management," for a detailed discussion of these routines and Chapter 11, "ILC Framework Manipulation Routines," for rigorous descriptions of them.)

Once the necessary frameworks are created, the C library switches frameworks automatically when an interlanguage call is performed. For instance, when the FORTRAN program in the example above calls a C function, the C framework is made active. Similarly, when the C function returns, the FORTRAN framework is automatically restored.

Data Formats and Communication

Interlanguage communication is additionally complicated by the various language conventions for data types and argument passing. Although some types (such as binary integers) are common to all languages, most languages have their own unique data types. In addition, some languages have data types that differ from similarly named types in other languages. For instance, COBOL's COMPUTATIONAL-3 (packed decimal) type and PL/I's AREA type are examples of data types for which there are no C equivalents. The PL/I default integer type FIXED BIN(15) and the C default integer type `int` are an example of similar but differing types. (FIXED BIN(15) corresponds to the C `short`, not the C `int`.) A more complex example is the data type *string*. Both FORTRAN and Pascal have a *string* data type. For FORTRAN, the string length is constant, while for Pascal the length can change during execution. Neither language recognizes the C convention of terminating strings with a 0 (null) character.

Distinct from the problem of data types is the problem of different argument-passing mechanisms. C passes arguments *by value*, which means that the value of each argument is stored in the parameter block passed to the called routine. Most other languages pass *by reference*, which means that the parameter block contains argument addresses rather than argument values. Pascal/VS is unique in that

arguments may be passed either by value or by reference, on an argument-by-argument basis.

When you use the SAS/C ILC support and C is called from another language that uses call by reference, the arguments to the C function must be defined as pointers to data of the appropriate type. More flexibility is available when another language is called from C. Whenever possible, the compiler converts arguments to an appropriate data type and uses an appropriate argument-passing convention. For instance, a C string constant being passed to a routine defined as a FORTRAN FUNCTION will be converted to a FORTRAN format string and will be passed by reference. For arguments where the corresponding data type is not clear, the programmer can use data type conversion macros to define exactly how the argument should be passed. For example, the **_ARRAY** macro can be used to force an array argument to be passed as an array rather than as a pointer. Note that this flexibility and convenience is available only on calls *by* C, not calls *to* C, because the code to call C is generated by another language's compiler, not by the SAS/C compiler.

Calling C from Another Language

When you call one or more C functions from another language, you must do the following:

1. Call the **CFMWK** routine to create the C execution framework. This must be done before any C routines are called. **CFMWK** stores a token value that is later passed to **DCFMWK** to delete the framework.
2. Call any C functions as if they were written in the calling language. All arguments to the C functions must be declared in C as pointers to the appropriate type (assuming call by reference is used). Similarly, if the function is not a void function, the data type returned by C must match the data type expected by the other language. The C functions must be compiled with the **INDep** compiler option.
3. After all C functions have been called, call **DCFMWK** to destroy the C framework, passing it the token stored by the original call to **CFMWK**.

The following example shows a FORTRAN MAIN program that calls C to write out a "Hello, world!" type message:

```
C
C     FORTRAN MAIN routine
C
      PROGRAM MIX
      INTEGER TOKEN,ERR
      CALL CFMWK('FORTRAN.', '.', 0, TOKEN)
      IF (TOKEN.EQ.0) STOP 16
      CALL WRITER('Hello, FORTRAN world!')
      CALL DCFMWK(TOKEN, ERR)
      IF (ERR.NE.0) STOP 8
      STOP
      END
```

```
/* C subroutine */

#include <stdio.h>

void writer(char *message)
{
   char *last;

   /* message ends with exclamation point...find end of message */

   for (last = message; *last != '!'; ++last);
   printf("%.*s\n", last-message+1, message);
   return;
}
```

Calling Another Language from C

When you call one or more routines in another language from C, you must do the following:

1. Call the **mkfmwk** function to create the execution framework for the other language. This must be done before any routines in the other language are called. **mkfmwk** returns a token, which is passed later to **dlfmwk** to delete the framework.
2. Call the routines in the other language. You must ensure that the data types of the C arguments correspond to the appropriate data types in the other language, or you must use data type conversion macros to force compatibility. Also, if the other language returns a value, the data type returned and the type expected by the C language must be compatible. The non-C routines must be declared using a keyword such as **__cobol** or **__pli** to inform the compiler that another language is being called.
3. After all C functions have been called, call **dlfmwk** to destroy the other language's framework, passing it the token returned by the original call to **mkfmwk**.

The following example shows a C **main** program that calls PL/I to write out a "Hello, world!" type message:

```
/* C main function */

#include <ilc.h>

main()
{
   void *token;
   extern __pli void writer();
   if (!(token = mkfmwk("PLI", ""))) exit(16);
   writer("Hello, PL/I world!");
   if (dlfmwk(token)) exit(8);
   exit(0);
}
```

```
/* PL/I subroutine */

WRITER: PROC(MESSAGE);
   DECLARE MESSAGE CHAR(*);
   PUT SKIP EDIT(MESSAGE)(A);
   RETURN;
END WRITER;
```

Linking Multilanguage Programs

Linking multilanguage programs is frequently an error-prone and tedious task because of the differing conventions of various languages. Problems include selection of a correct entry point and correct resolution of subroutines from libraries for several languages.

The SAS/C ILC implementation includes a new linkage utility, ILCLINK, which facilitates linkage of multilanguage programs. ILCLINK is driven by an input file of control statements provided by the user that describe the languages to be used and the required linkage steps. The following example shows a simple control file for linking the FORTRAN-C sample program above under OS:

```
* First language is FORTRAN.
FIRST MIX(FORTRAN)
LANGUAGE FORTRAN,C
* Process with OS linkage editor
PROCESS LKED LIST,XREF
* Use C and FORTRAN subroutine libraries for autocall
AUTOCALL LC370, FORTLIB
INCLUDE MIX,WRITER
NAME MIX(R)
```

2 Multilanguage Framework Management

Introduction

One of the difficulties in writing a program that combines several high-level languages is management of the various execution frameworks. The SAS/C ILC library routines assume most of this burden, with some guidance from the user program. However, understanding the library's techniques for managing frameworks facilitates the development and debugging of a multilanguage application.

This chapter discusses general execution framework considerations for writing multilanguage applications:

- framework components
- framework creation
- framework coexistence
- framework termination, both planned and unexpected
- error handling.

Note that there are many additional considerations for programs composed of more than two languages. These are discussed in Chapter 10, "Advanced Topics."

Terms to Know

This section outlines some terms you should know before you read the rest of the chapter.

main routine
: the first high-level language routine to be executed in a program. In most languages, the main routine is identified at compile time. For instance, in C, a main routine must be named `main`, while in FORTRAN it must begin with a PROGRAM statement.

save area chain
: a linked list of storage areas in which called routines can save their status, such as register contents. When a new

routine is called, a new area is added to the chain, which is removed when the routine returns.

Framework Components

An execution framework is a collection of data and routines that supports the execution of code. This normally includes at least the following components:

- language library routines. In most cases, there are both resident routines, linked with the application, and transient routines, dynamically loaded at run-time or located in a shared region of memory.
- control blocks, such as memory allocation tables, file descriptors, and save area chains. The save area chains are of particular importance because they define the calling sequence (which routines are active and how they were called) for each language. Some languages (such as PL/I, Pascal, and C) have registers dedicated to addressing framework control blocks.
- error handling information. In general, each language uses system-specific macros (such as ESPIE and ABNEXIT) to gain control in error situations. The system exit routines and the data necessary to control them are important and complicated constituents of a framework.
- language-dependent information, providing support for specialized language features. FORTRAN vector save areas, PL/I CONTROLLED variable stacks, and C pseudoregister vectors and coprocessing control blocks are examples of this sort of information.

Framework Creation

The execution framework for a particular language is normally created when a main routine in that language is executed. Depending on the language, the framework may be created by a front-end routine (like PLISTART) or by a library routine (like the FORTRAN VFEIM#), and additional mechanisms may be provided to cater to calls from assembler.

In a multilanguage program, one framework has to be created first, namely, the framework for the first high-level language used. Creating the other frameworks by calling additional main routines is cumbersome at best and not conducive to using subroutine libraries in another language. When you use the SAS/C ILC support, there should be only one main routine, that for the first language. All frameworks except the first are created only as the result of calls to SAS/C library functions that request their creation. Note that the main routine is not required to be in C. It is as easy to create the C framework from COBOL as it is to create the COBOL framework from C. (See Chapter 11, "ILC Framework Manipulation Routines," for reference information on the routines that create and destroy frameworks.)

Creating a Non-C Framework

To create the framework for a non-C language from C, you must call the C library function `mkfmwk`. It is important to note the technique by which `mkfmwk` creates a new framework. `mkfmwk` has no knowledge of the implementation details of other languages. It

generates a framework by calling a SAS/C-supplied main routine in the target language. For instance, when creating the PL/I framework, **mkfmwk** calls a PL/I OPTIONS(MAIN) procedure named L$IPL1M. This technique avoids dependencies on language versions or maintenance levels in the SAS/C library. In addition, source code is supplied for these main routines. If your site has an incompatible compiler (either very old or very new), in many cases you will be able to execute normally by simply recompiling this main routine with your version of the compiler.

Most languages permit a user to specify run-time options when they call the main program. Because **mkfmwk** calls a main routine, it can accept run-time options for the new language from its caller and pass them on to the new framework.

Creating the C Framework

To create the C framework from another language, call the C library routine **CFMWK**. As with **mkfmwk**, **CFMWK** works by calling a C main function contained in the library, this one named L$CICMN. The linkage conventions for **CFMWK** are defined so it can be called easily from any language, without any language dependencies. (In fact, its caller has to supply the language name as an argument so that C will know which language created its framework.)

It is important to note that this technique forces a particular structure on a multilanguage program. One language must be chosen as the "main language," and the only user-supplied main routine must be coded in that language. All other languages will have their frameworks created by calls to the SAS/C library framework creation routines, and no user-coded main routines in these languages may be included.

Framework Coexistence

During the execution of a SAS/C multilanguage program, execution under each framework takes place more or less independently. While C is running, for instance, the FORTRAN framework is quiescent. When C calls a FORTRAN routine, the library deactivates the C framework, activates the FORTRAN framework, and calls the requested routine.

In order to switch frameworks properly, the SAS/C library must get control every time one language calls or returns to another. This is enforced by two different mechanisms. When C calls a routine in another language, the called routine must be declared using a keyword such as **__cobol**. This allows the compiler to generate code that calls a library routine (L$CILCL) to perform the framework switch.

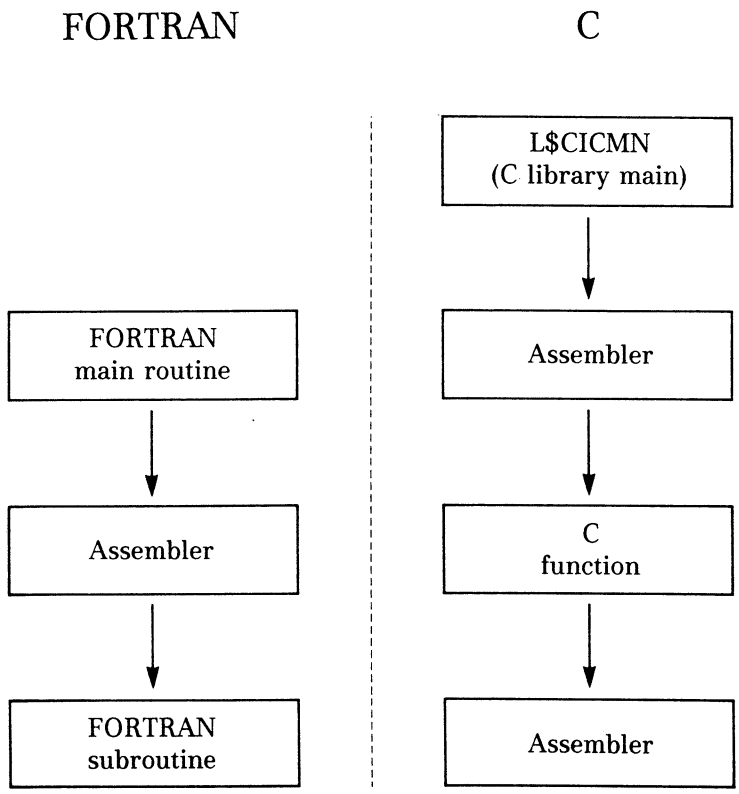
When another language calls a C function, the C function must have been compiled with the **INDep** compiler option. (See the *SAS/C Compiler and Library User's Guide* for more information on the **INDep** option.) When the function is entered, it immediately calls a library routine named L$UPREP, which switches to the C framework before returning to the called function.

Implementation Details

If you are interfacing with assembler or debugging from system dumps, the following information may be useful. Because each framework is maintained separately, each language has its own save area chain (and run-time stack, if applicable). This is necessary

because each language has its own conventions for save area layout and interpretation. A C save area on a PL/I save area chain would be likely to cause PL/I to behave incorrectly, possibly with fatal or very confusing results. When SAS/C interlanguage communication is in use, each language's save area chain appears to contain a mixture of its own save areas and assembler save areas. The assembler save areas are used by the library routines that switch from one framework to the other. The assembler routines in one framework are linked to the non-assembler routines in the other frameworks. For instance, if your calling sequence is FORTRAN->C->FORTRAN, the save area chains for FORTRAN and for C will be as shown in **Figure 2.1.** (Linked save areas are those at the same horizontal level in the figure.)

Figure 2.1 *Multilanguage Save Area Chain Example*



The steps actually performed by the library when it switches frameworks are conceptually quite simple:

1. Point register 13 to the appropriate save area.
2. Modify error handling for the new framework (see **Error Handling** later in this chapter).
3. Load any registers reserved by the new framework. (For example, for C, put the CRAB address in register 12.)
4. Call (or return to) the requested routine.

Efficiency Considerations

Though every effort has been made to streamline the process of switching frameworks, there is necessarily a significant amount of overhead. Structuring your application to avoid interlanguage calls in loops can result in big savings. For instance, you might be able to move a loop to the called language so that you have only one interlanguage call rather than many.

Correctly establishing error handling is one particularly expensive part of a framework switch. There is a tradeoff between maximum reliability and maximum performance which can be controlled by use of the C `=multitask` run-time option. See **Error Handling** later in this chapter for more information on this option and its implications.

Framework Termination

The execution framework for a particular language is normally terminated when the main routine in that language completes execution, or when an explicit termination statement (for example, STOP RUN or HALT) is executed. In general, framework termination includes closing open files, releasing memory, and canceling all error handling.

In a multilanguage program, frameworks for languages other than the first language can be terminated in an orderly fashion by calls to SAS/C library functions. In addition, frameworks can be terminated as a result of unusual events, such as execution of termination statements or run-time errors. Such abrupt framework terminations cause all frameworks to be terminated.

Planned Termination

Frameworks are normally terminated by a call to a SAS/C library function for that purpose. Of course, it is impossible to terminate a framework if it is busy. For instance, you cannot terminate the PL/I framework from a C function called from PL/I because the calling PL/I routine is still using the framework.

To terminate another language's framework from C, call the SAS/C library function `dlfmwk`. The argument to `dlfmwk` is the language token returned by `mkfmwk` to identify the framework when it was created.

When `dlfmwk` is called, it transfers control to the non-C framework identified by the token. The main routine for that language then returns to its caller. This causes normal framework termination to occur and ensures that all normal termination actions take place. Because termination is handled by the other language's library, the C library does not need to get involved or have any special knowledge of the other language's implementation.

Similarly, to terminate the C framework from another language, call the `DCFMWK` routine, passing the token stored by `CFMWK` when the

framework was created. **DCFMWK** switches to the C framework and forces L$CICMN to return to its caller, thereby causing normal C termination to take place.

Unexpected Termination

Framework termination can also occur unexpectedly, due to errors or unusual flows of control. For instance, FORTRAN enforces run-time limits on certain kinds of errors. If the run-time error limit is exceeded, the FORTRAN library simply writes out a final message and terminates the FORTRAN framework. Events such as this make continued execution of other languages very difficult.

The SAS/C interlanguage communication implementation assumes that unexpected termination of a framework is either

- an error
- a deliberate attempt to completely terminate execution (as by execution of a COBOL STOP RUN statement).

The unexpected termination causes all other frameworks to be terminated and forces a return to the caller of the main program. The return code will be the one associated with the first terminated framework. For instance, if FORTRAN caused the termination, the final return code will be the value requested by FORTRAN.

This feature requires that the C library be able to force termination of any framework as necessary. For instance, if the C framework terminates unexpectedly, it must be possible for C to then force the termination of PL/I, whether or not C was the first framework created. As with framework creation, this is achieved by calling a library routine written in the language to be terminated. For instance, to force Pascal to terminate, the library calls the Pascal routine L$IPASQ. L$IPASQ calls the Pascal HALT function to terminate the Pascal framework in the normal way. These quit routines are, like the main routines, supplied in source in the interest of supporting alternate versions of the standard languages.

Whenever the entire program is shut down because a single framework has terminated, the SAS/C library writes a warning message (LSCX281) identifying the framework that terminated. This message is a useful debugging aid because, if termination occurred because of unintended execution of a call to **exit** or a STOP RUN, there may be no other external indication of the language that caused termination.

Error Handling

Error handling is one of the most unpleasant parts of interlanguage communication. In a multilanguage program, several languages may have statements to handle similar errors, and it is important to be sure that an error will be handled by the appropriate language. For instance, division by zero in PL/I should be routed to a PL/I ZERODIVIDE ON-unit, not to a C **SIGFPE** signal handler. Similarly, if you are using debuggers for several languages, the proper debugger needs to get control when an error occurs.

Due to the nature of the system interfaces used to capture errors, it is difficult to guarantee correct routing without substantial overhead. SAS/C ILC supports various options for error handling, depending on the needs of the application.

When full error-handling support is needed, SAS/C ILC attempts to intercept program checks and ABENDs before any other framework. If the error occurred during the execution of non-C code, the error is passed to the other language for processing. For complete reliability in this processing, it is necessary under OS to run each language as a separate task. Under CMS you must reissue the SPIE macro every time a framework switch occurs. To request this form of error supervision, use the C run-time option `=multitask`. This is recommended during program development and testing. This option is specified as a normal run-time option if C is the first language, or via an argument to `CFMWK` if the main program is in some other language.

In many cases, you will not need the pervasive error-handling provided by the `=multitask` option. A faster error-handling implementation is available, tailored to the normal use of the SPIE, STAE, and ABNEXIT macros by FORTRAN, COBOL, and PL/I. This technique avoids much of the overhead but is still adequate for allowing each language to handle in most cases its own errors. This technique is used unless you specify the `=multitask` option.

For some applications, it may be that error handling is only needed in one language. In this case, you can further reduce overhead by suppressing error handling by one or more languages. For instance, you can use the C `=nohtsig` and `=nohcsig` options to suppress C error handling. Of course, when you do this, any errors that do occur in C will be handled by the other language, possibly with unwanted results. You might alternately want to suppress error handling in the non-C language by, for example, using the PL/I NOSTAE and NOSPIE options. When you use this type of error-check suppression and a program check occurs in another language, the program will terminate abnormally. This is because C is in control of the ILC process and recognizes that the program check should not be handled by C.

Be careful when using out-of-block GOTO statements in PL/I or calls to `longjmp` in C for error handling. If one of these techniques causes a block in another language to be terminated, the program will eventually terminate abnormally. The C library attempts to detect this situation and put out a helpful diagnostic, but in some cases the result may simply be an ABEND that happens considerably later than the fatal GOTO.

3 Communication with Other Languages

Introduction

When a program is composed of routines in several languages, the various languages must be able to communicate and process the same data.

Sharing data effectively between languages requires an understanding of each language's data types and formats. The first section of this chapter presents an overview of

- data types and formats
- how data types and formats differ from one language to another
- how these differences affect data sharing.

There are three techniques by which data in one language can be made available to another: via subroutine arguments or function return values, via external variables, or via shared files. All three techniques can be used in the same program. The second section of this chapter discusses each of these techniques.

The last two sections of this chapter discuss the ILC compiler options and multilanguage application linkage considerations, respectively. This chapter gives you a strong base on which to build when you move on to creating a specific multilanguage application.

Terms to Know

This section outlines some terms you should know before reading the rest of the chapter. Cross-references are in italics.

argument promotion
: the conversion of a routine argument from one type to another, performed by the C compiler during processing of a call; for example, `character` to `int`, `float` to `double`, or `array` to `pointer`.

character literal
: a constant that represents a single character, such as the C constant `'x'`. Most other languages do not distinguish between character literals and *string literals*.

external symbol
: the name of an area of storage defined to the linker or loader so that it can be accessed by name from several different object modules.

external variable
: a variable that is declared in such a way that it can be accessed by more than one routine or compilation.

formatted I/O
: a form of I/O in which data are transmitted in printable characters rather than in internal format. Sometimes called text I/O or stream I/O.

function prototype
: in C, a declaration of a function that indicates the data types of its arguments, as well as the data type returned.

IBM 370 standard linkage
: the linkage technique used by assembler language routines, most high-level languages, and many software packages. Registers 1, 13, 14, and 15 are used to address a parameter list, a register save area, the calling routine's return address, and the called routine's first instruction, respectively.

PL/I descriptor
: a control block defining the format of a complicated argument, such as an array or a structure. PL/I normally passes arguments with descriptors indirectly by passing the descriptor address.

string literal
: a constant that represents a sequence of characters, such as `"Hello"` in C or `'Hello'` in FORTRAN or Pascal. See also *character literal*.

unformatted I/O
: a form of I/O in which data are transmitted in internal format. Sometimes called binary I/O or record I/O.

Data Types and Data Formats

Each high-level language defines its own data types. (A *data type* designates a set of values, together with a set of operations defined for such values.) Some data types are common to most languages, while others are unique. For instance, virtually all languages support a type like the C `int`, while the C `union` type and the Pascal SET type are unique to a single language.

Each data type of a language has an associated *data format* (physical representation), which defines how data are stored and interpreted in

memory. The data format is an important aspect of a data type. Note that types with different formats have to be considered different types. For instance, the C type `short int` and the PL/I type FIXED DECIMAL(3) are both types whose values are small integers that occupy 2 bytes of memory. But, the value 0x012c has the value 300 when interpreted as a `short int` and the value 12 when interpreted as a FIXED DECIMAL(3).

Because languages do not recognize the data types of other languages, you need to use variables with the same data format to share data between languages. For instance, as shown by the previous example, if a C `short int` is written to a binary file and read back into a PL/I FIXED DECIMAL(3), the data are interpreted differently. However, if the input variable is of type FIXED BINARY(15), no change in value occurs, because the format of a FIXED BINARY(15) is the same as the format for a `short int`.

Later chapters in this book (Chapter 4, "Communication with FORTRAN," Chapter 5, "Communication with COBOL," and so on) include, for each language supported by SAS/C ILC, a table showing equivalent data types.

For more information on common 370 data formats, see the IBM manual *System/370 Principles of Operation* (GA22-7000).

Common Data Types

Numeric types

The 370 architecture supports three numeric data formats: binary, packed decimal, and floating-point. Furthermore, these formats support several different sizes. For instance, floating-point data can be stored as 4, 8, or 16 bytes, depending on the required precision. The default size for a particular format varies from language to language. For example, the usual size for floating-point numbers is 8 in C and Pascal but 4 in FORTRAN and PL/I. When sharing numeric data between languages, it is necessary that all languages use both the same format and the same size.

The C language does not have a data type that uses packed decimal format. However, SAS/C ILC provides the `pdval` and `pdset` macros, which convert between packed decimal and floating-point data formats. These macros are described in Chapter 12, "Using Packed Decimal Data in C."

COBOL and PL/I support a PICTURE data type, which is frequently used as numeric. Such data are stored in character form and therefore should be processed in C as character rather than as numeric.

Boolean and bit types

Many languages have a boolean type that is used to store truth values, such as the results of comparisons. The FORTRAN LOGICAL, PL/I BIT(1), and Pascal BOOLEAN are examples of such types. The C language does not have an explicitly boolean type; truth values are regarded simply as integers. Depending on the size of another language's boolean objects, truth values can generally be treated in C as either `char` or `int`.

Some languages also have one or more bit data types, with primitive operations like "logical and," "logical or," and "logical not." PL/I BIT(n) ALIGNED is an example. The Pascal SET type is another type implemented as a bit string, with the union and intersection operators implemented via logical or and logical and. If the size of a bit string matches one of the C `unsigned` types, it can be easily processed in C

using standard C operators. For other sizes, it is best to consider such variables to be arrays of `char` and process them one character at a time.

Bit types that are not necessarily aligned to a byte boundary, such as PL/I's BIT(n) UNALIGNED, are very difficult to process in C. Values of this type should be converted to another type for use in C.

Character and string types

One of the ways in which C differs from most other programming languages is in its treatment of characters and character strings. In most languages, a single character is regarded simply as a character string of length 1. In C, a single character is regarded as a very small integer, and the character literal `'a'` and the string literal `"a"` represent distinct data types.

Although C does not have a character string type, it does have a character string data format. Character strings in C are represented as a sequence of characters, terminated by a null (zero) character. Because no other common language uses this format, sharing string data between C and other languages must be performed carefully.

The other commonly used 370 languages have two different string formats: a fixed-length format and a varying-length format. The fixed-length format is just a sequence of characters of a known length without any special terminator. This format is supported by FORTRAN, COBOL, PL/I, and Pascal. A C type with the same format is

```
char [n]
```

Another type with the same format, which may be more convenient in some cases, is

```
struct {char text [n];}
```

The varying-length string format is used by PL/I and Pascal. A string in this format consists of a 2-byte integer that contains the current length of the string, followed by a fixed-length text area of some size. The current length of the string specifies how many characters of text are significant. (Any remaining characters are ignored.) A varying-length string can be modified to be any length up to the size of the text area. Although C has no operators for processing varying-length strings, it does have this suitable data type

```
struct {short len;
        char text [n];}
```

SAS/C ILC provides macros for defining and converting varying-length strings for the convenience of programs that share string data with PL/I or Pascal. See Chapter 13, "C Varying-Length String Macros," for more information on these facilities. There is also a `VString` compiler option that causes the compiler to put length prefixes at the front of all C strings, permitting string literals to be passed conveniently as arguments to PL/I or Pascal. (See **C Compiler Options for ILC Programs** later in this chapter.)

Array types

All languages have one or more array types that are used to store a number of elements of a single type. Array types are selected by an index or subscript. Arrays can have more than one dimension. (This is treated in some languages, such as C, as defining an array of arrays rather than a multidimensional array.) As a rule, an array is stored in memory as a contiguous set of elements.

In order to share an array between C and another language, it is generally necessary only that the types in the two languages specify the same number of elements and that the element types have the same format.

Despite arrays' basic simplicity, there are some language differences to be aware of when sharing arrays. Note that the first element of a C array is addressed as element 0, while the first element of an array in most other languages is addressed as element 1. Some languages allow the first element index to be defined uniquely for each array.

An additional problem that is specific to FORTRAN is that FORTRAN stores multidimensional arrays in "column major order," while other languages, such as C, use "row major order." This difference, plus the difference in the first element number, means that if the array A is shared between C and FORTRAN, then the C `A [m] [n]` would be accessed as A(n+1,m+1) in FORTRAN.

Finally, note that C is unique in that it treats arrays and pointers synonymously. Even though a C pointer to an `int` object can usually be treated as an array of `int`, a Pascal pointer to INTEGER cannot be treated as an array of INTEGER. When you share data with other languages, you must be careful to process shared arrays only as arrays and shared pointers only as pointers.

Structure types

Most languages support a structure, or "record," type that consists of one or more named elements. Each element can be of a different type, frequently some other structure type. The data in a structure are generally stored simply as a list of elements placed consecutively in memory. Note that, even though FORTRAN does not have a structure data type, named COMMON blocks can be treated as structures for many purposes.

In general, a structure in C and a structure in another language have the same format if corresponding elements have the same format. However, different languages use different rules for alignment of structure elements. For instance, consider the PL/I structure defined by

```
DECLARE 1 S,
   2 CH CHAR(1),
   2 I FIXED BIN(31);
```

and the C structure defined by

```
struct {
   char ch;
   int i;
}s;
```

These structures do not have the same format because PL/I stores the character CH in the byte immediately preceding the integer I, while C puts the character `ch` in the first byte of the word before `i`, followed by 3 padding bytes. The most general way to remedy this problem is to add additional padding fields to the structure to force the formats to be the same. For instance, a CHAR(3) padding element can be placed after CH in the PL/I declaration to force the structure to be formatted like the C structure.

Other techniques that may be helpful include the use of compiler options (such as the SAS/C `BYtealign` option) or language keywords (such as the COBOL SYNCHRONIZED clause or the SAS/C `__noalignmem` qualifier) that modify or simplify the way structure elements are aligned.

Pointer types

Pointer types, in those languages that support them, are stored in a standard format as 4-byte memory addresses. However, the way pointers are used varies from language to language. C and Pascal feature "strongly typed pointers," in which pointers to one type of data are distinguished from pointers to other types. In C, a pointer to `char` and a pointer to `int` are different types with the same data format. PL/I and COBOL, on the other hand, use generic pointers. In PL/I, all POINTER variables are considered to have the same type, and the same POINTER variable can be used to access variables of various types.

Objects addressed via shared pointers are shared effectively themselves. For this reason, the types addressed via a shared pointer should have the same format. For example, a PL/I POINTER that is used to address a BASED FIXED BINARY(31) variable should not be treated in C as a pointer to `double`. If a generic pointer type is required in C, the standard pointer type `void *` should be used.

Esoteric types

Most languages support data types that are unique to that language. Sometimes, you are able to share such data with another language using a structure in the other language that matches the format of the original type. For instance, a FORTRAN COMPLEX number can be easily processed in C using the type:

```
struct {float re,im;}
```

On the other hand, a PL/I LABEL variable cannot possibly be used productively in C. A good rule of thumb is that you should not try to share data of any type that has no meaning independent of the language being used.

An important application of this rule is to file I/O. Each language supports its own I/O, and many languages support FILE variables. However, each language has its own rules for how I/O is performed. (For instance, FORTRAN files can be BACKSPACEd, while PL/I files cannot.) It is impossible to share FILE variables between languages. Even if, by chance, such variables have the same size, the meaning and interpretation of the data are completely language dependent.

Data Sharing

Argument Passing

Each language has its own conventions for how arguments are passed from one routine to another. When you use more than one language in a program, you need to be aware of the conventions associated with each language you use. When you use SAS/C ILC and call a C function from another language, you must define the C function to have arguments that match the conventions of the calling language.

When you use SAS/C ILC to call another language from C, the C compiler and library transform the C argument list into one in the format required by the called language. Even though, in this case, you do not have to create the list in the correct format yourself, it is useful to have an understanding of the other language's conventions.

The languages supported by SAS/C ILC all use a form of *IBM 370 standard linkage*. On entry to a called function, register 1 always addresses an argument list, register 15 always addresses the called routine's first instruction, and register 14 contains the caller's return address.

Note that Chapters 4 through 7 contain more language-specific information on the conventions for particular languages.

C calling conventions (call by value)

C, unlike most other high-level languages, uses a call-by-value convention for argument passing. This means that the argument list addressed by register 1 for a C function call contains the argument values stored consecutively. The following complications should be noted:

- An array is always converted to a pointer to the first array element before the function call.
- `short` and `char` values are promoted to `int` before they are stored in the argument list. Thus, they always occupy a fullword of storage in the argument list.
- `double` values are always stored on a doubleword boundary in the argument list. This may leave a 4-byte gap after the previous argument. `float` values are always promoted to `double` before they are stored in the argument list.
- Structure and union arguments can be of any size, but they always occupy an integral number of fullwords in the argument list. If the size is less than 4 bytes, they are passed right-aligned in a fullword. If the size is greater than 4 bytes, they are passed left-aligned in as many fullwords as necessary. If the argument size is a multiple of 8 bytes, the argument is aligned on a doubleword boundary, possibly leaving a gap.

Note that no explicit indicator is set to indicate the last argument.

Call by reference

FORTRAN, COBOL, and PL/I all use call-by-reference conventions for passing arguments. (There are additional considerations for PL/I, as discussed in **PL/I descriptors** later in this chapter.)

When call by reference is used, register 1 addresses a list of argument addresses stored in consecutive fullwords. The last argument in the list has the X'80000000' bit set to indicate that it is the final argument. Note that this bit has no effect when the argument

is used to address data because the hardware ignores this bit in a memory address.

A call-by-reference argument list is easy to process in C because it has the same format as a call-by-value argument list in which all arguments are pointers. To illustrate, assume that CSUB is a C function in the following FORTRAN statements:

```
INTEGER I
REAL*8 X
CALL CSUB(I, 5, X)
```

If CSUB is defined in C by

```
void csub(int *i, int *n, double *x)
```

the argument list passed by FORTRAN is identical to that expected by C.

Pascal calling conventions

Pascal/VS allows the programmer to control how individual arguments are passed. Pascal uses the terms "pass by value," "pass by VAR," and "pass by CONST" to describe the available techniques. Both "pass by VAR" and "pass by CONST" use call by reference, as described above. That is, the address, not the value, of an argument passed by VAR or CONST is stored in the list addressed by register 1. (However, Pascal does not set the end-of-list bit in the last argument.)

"Pass by value," which is the Pascal default, is more complicated:

- For record, array, and string arguments, the address, not the value, of the argument is passed.
- For other argument types, the value of the argument is passed. However, the alignment rules are somewhat different than the C alignment rules, and the C promotions do not occur. For instance, if two CHAR variables and an INTEGER are passed by value, the argument list will contain the two characters, a 2-byte gap, and the INTEGER. The corresponding C argument list will contain three fullword integers. See the *Pascal/VS Programmer's Guide* for more detailed information on the implementation of Pascal pass by value.

When you call C from Pascal, the use of pass by VAR or pass by CONST is recommended. If all arguments are passed by VAR or CONST, you can simply declare each C argument to be a pointer to an appropriate type, as shown in **Call by reference**. Avoid using call by value for values that are smaller than 4 bytes or for SHORTREAL values because the differences from C argument passing conventions make this type of data difficult or impossible to access in C.

PL/I descriptors

PL/I supports forms of argument passing that require control information to be passed in addition to the argument address. For instance, PL/I permits a string argument to be declared as CHAR(*), which indicates that the string length is unknown and may vary from one call to the next. PL/I supports such arguments by passing descriptor addresses rather than data addresses. Descriptors contain the actual data address as well as other information, such as string lengths and array dimensions, that may not be known to the called

routine. Descriptor formats are documented in the IBM manuals *PL/I Optimizing Compiler Execution Logic* and *OS PL/I Version 2 Problem Determination.*

If information from the descriptors is useful, the descriptors can be processed in C as structures, but they are of little use for most applications. For this reason, when you pass arguments from PL/I to C that require descriptors, you should declare the C function in PL/I as OPTIONS(ASM). This causes PL/I to pass all arguments by reference without any descriptors.

Passing arguments from C to other languages

To declare in C a function written in another language, use one of the keywords `__fortran`, `__cobol`, `__pli`, or `__pascal`. Note that these keywords all begin with a double underscore.

When you call a function declared in C to be in another language, the arguments specified can be modified by the compiler and library to conform to the argument-passing conventions of the other language. Each argument can be specified in one of three ways: as a nonpointer value, as a pointer value, or by using a data type conversion macro. These three forms of argument are processed as follows:

- If possible, a nonpointer argument is transformed to match the conventions of the called language. For instance, a `short int` passed to Pascal is passed by value and stored in a halfword of the argument list. Similarly, if the argument to a PL/I routine is a C structure with the format of a PL/I CHAR VARYING string, a PL/I string descriptor is created, and the descriptor address is passed to PL/I.
- A pointer argument is stored in the argument list directly and passed unchanged to the called routine. The compiler and library assume that the pointer is the address of a call-by-reference argument. Note that by casting data of some other type to `void *` you can store a fullword of arbitrary data in the argument list with the assurance that the data will not be modified.

  An exception to the rule of passing pointer arguments unchanged is made when passing a string literal. In this case, the argument is modified so that it will be treated as a string, rather than as an array or pointer, in the target language. (For instance, when calling FORTRAN, an additional control argument is created to pass the string length.) When you communicate with PL/I or Pascal, string literals are passed as fixed-length strings unless the C compiler option `VString` is used to pass them as varying-length strings.

  If you want to pass a string literal to another language using direct call by reference, rather than as a string, you should cast it to `void *`.
- In some cases, there may be another way to interpret a C object in another language besides the above defaults. For instance, a C `char` array might correspond in PL/I to a POINTER, an array of CHAR(1), or a CHAR(n) string. In such cases, the default interpretation may not be correct. In the `char` array example, by default the array is converted to a `char *` value according to C rules, and the pointer is stored in the argument list.

  SAS/C ILC provides data type conversion macros for use in cases like this so that the programmer can specify the type of the argument in the other language. In the C `char` array example

above, the programmer would use the `_ARRAY` macro to pass the array as an array or the `_STRING` macro to pass it as a string. Note that the data type conversion macro names all begin with a single underscore, and the remainder of the name is always uppercase.

The macros that can be used with each language are described in detail in the chapter on communication with that language.

Return Value Handling

Most languages, with the exception of COBOL, support subroutines that return a value to their caller. In contrast to argument passing, in which there is much similarity between languages, return value handling is specific to each language. Some languages (C and FORTRAN) return values in registers, while others (PL/I and Pascal) use an extra argument to address the return value. There is no agreement on which registers to use or the location of the return value in the argument list.

When you use SAS/C ILC, the C library passes return values from one language to another so that each language's conventions are satisfied. However, it is sometimes necessary to restrict the types of data that can be returned, due to limitations of one of the languages. For instance, C does not support functions that return arrays. For this reason, a routine declared in Pascal to return an array (which is permitted by Pascal) cannot be implemented in C.

See the communication chapter for each language for complete information on the permitted return value types and for any applicable restrictions.

External Data Sharing

In addition to data sharing using arguments and return values, data can be shared between languages via external variables. This may be more convenient than sharing by argument passing if large amounts of data are to be shared.

An *external variable* is a variable that is defined (in a language-specific way) so that it can be referenced by name from more than one compilation. Usually, an external variable is defined to the linker or loader as an *external symbol*, which causes all uses of the variable to be resolved to a single location in the load module. This implementation permits several languages to use the same external variable, provided that they all use the same name. Some languages, such as PL/I and C, provide other implementations of external variables as an alternative to the external symbol implementation. Such variables are *private* externals and cannot be shared with other languages.

Note that some languages, such as C and Pascal, distinguish between declarations and definitions of external variables. A definition of an external variable defines the storage associated with the external symbol, while a declaration merely references that storage. Languages that do not make this distinction always define the storage for an external variable. An external variable shared between languages must be defined in only one of the languages.

- C `extern` variables can be shared with another language if the `NORENT` compiler option is used. The `RENT/RENTExt` implementation of non-`const externs` is as pseudoregisters, which are not accessible from other languages. It is usually best to define the variable in the other language when sharing a C `extern` with another language.

- FORTRAN named COMMON blocks can be shared with C. (They are accessed in C as `extern` structures.) A COMMON block is always defined by FORTRAN, so a C definition must not also appear. Dynamic COMMON blocks cannot be shared.
- COBOL does not support external variables.
- PL/I STATIC EXTERNAL variables can be shared with C. A STATIC EXTERNAL variable is always defined by PL/I, so a C definition must not also appear. PL/I CONTROLLED EXTERNAL variables have a special format and cannot be shared with C.
- Pascal permits external variables to be defined using a DEF declaration or to be referenced using a REF declaration. Both kinds of variables can be shared with C. A C definition must be provided if DEF is not used or omitted if DEF is used. Pascal also supports global variables using a VAR declaration within a SEGMENT. These variables cannot be shared with C.

When external variables are shared between languages, as with all other forms of sharing, the data formats in the various languages must agree.

File Sharing

Data written by one language can usually be read by another language, provided that both languages support the same data formats. However, you must understand the I/O conventions of the languages involved. Most languages support two different kinds of I/O: formatted I/O (sometimes called stream I/O or text I/O) and unformatted I/O (sometimes called record I/O or binary I/O). The use of formatted I/O usually enhances file sharing because, as a rule, the data are stored as printable characters, and no knowledge of internal data formats is needed. Further, the internal formats of the input and output variables need not be the same.

When unformatted I/O is used, data are generally written or read in internal format. For this reason, the data format of each output variable must be the same as the format of each input variable. Note that because unformatted I/O avoids the overhead of formatting, it is usually more efficient than formatted I/O.

Note: You should never attempt to process the same file simultaneously from more than one language. For instance, if a C function is writing a file that must be read by a called COBOL routine, you must close the file in C before opening it in COBOL. This is required because different languages use different file processing techniques and because the 370 operating systems provide very little support for simultaneous file accesses.

Also note that FILE variables are implemented uniquely by each language. It is not possible to use a PL/I FILE variable in a C function, or a C FILE pointer in PL/I.

C Programming Considerations

This section summarizes a number of C language extensions and compiler options that are useful in multilanguage programs.

Declaring Routines in Other Languages

A routine in another language called from SAS/C ILC must be declared to be in another language, using one of the keywords `__fortran`, `__cobol`, `__pli`, `__pascal`, or `__foreign`. (The `__foreign` keyword specifies that the declared function is in a user-

supported language. See Chapters 14 through 16 for more information.) Note that each of these keywords begins with a double underscore. Other language routine names can be specified in either upper- or lowercase because the compiler translates the names of external functions to uppercase during compilation.

Note that other language routine names must obey the naming conventions of both languages and must generate the same external symbol in both languages. These are some results of this requirement:

- The name of a routine in another language cannot begin with a dollar sign ($). It can contain a dollar sign (in a position other than the first) only if the C compiler option `DOllars` is used.
- The name of a routine in another language cannot contain the at sign (@).
- The name of a routine in another language must replace each underscore (_) with a pound sign (#) if the routine name in C includes an underscore.
- Different languages use different rules for truncation of routine names that are too long. For instance, the C compiler truncates the name `VeryLongName` to `VERYLONG`, while PL/I truncates the same name to VERYAME.

Examples of declarations of routines in other languages are

```
extern __fortran void DGEMUL();
extern __fortran double dexp(); /* a FORTRAN math library routine */
extern __cobol void withhold();
extern __pli char *bufaddr();
```

Note that you must not specify a prototype for a routine in another language. A prototype causes unnecessary conversions to take place and interferes with the operation of the data type conversion macros such as `__STRING` and `__ARRAY`.

The @ (Call-by-Reference) Operator

The `@` operator, which is described in more detail in the *SAS/C Compiler and Library User's Guide*, is frequently used when calling a routine in another language. If `x` is an expression such that `&x` is a valid C expression, then `@x` and `&x` have the same meaning. (That is, both denote the address of `x`.) If, however, `x` is a non-lvalue expression, such as `a+b`, then `@x` is the address of a temporary copy of `x`. For instance, in the case of `@(a+b)`, where `a` and `b` have type `int`, the compiler generates code to compute the value of `a+b`, store the value in a temporary integer, and then load the address of the temporary.

An `@` expression can be used only as a function call argument.

Because all nonpointer arguments to a routine in another language (other than Pascal) are passed by reference automatically, the primary use for the `@` operator in a multilanguage program is to pass a pointer value by reference. For instance, to pass a PL/I procedure a POINTER whose value is `&z`, you must code

```
plifunc(@&z);
```

If you code `plifunc(&z)` instead, `&z` will be stored directly in the argument list, causing the PL/I POINTER argument to contain the value rather than the address of `z`.

The @ operator is also useful when calling Pascal to force pass by reference rather than pass by value.

Note that the compiler option **AT** must be specified if you use the @ operator.

ILC Argument Promotions

When you call a routine in another language from C, argument promotions are performed differently than for a call to a normal C function in order to preserve information important to the other language. Although arrays are converted to pointers when passed to another language, **char** and **short** expressions are not promoted to **int**, and **float** arguments are not promoted to **double**. If promotion is desired, you can use a cast to force the promotion.

Another violation of strict C language rules occurs when a string literal is passed to another language. The argument value stored by the compiler for such an argument is of the form required by the called language, rather than simply a pointer to the first character of the string. This means that two calls to a Pascal routine of the forms

```
pascfun("abc")
```

and

```
pascfun((void *)"abc")
```

will have different effects. (Which call is correct depends on the way that pascfun is defined in Pascal: the former is correct for a STRING argument, and the latter is correct for a pointer argument passed by value.)

You should also be aware of the possibly surprising result of a C language rule that is honored for a call to another language. The type of a character literal, such as **'a'**, is **int**, not **char**. This is a consequence of the fact that a character literal such as **'abcd'** may contain more than one character. For this reason, an interlanguage call such as

```
cobfun('a')
```

passes an integer rather than a character argument. To pass a character, you must code

```
cobfun((char) 'a')
```

Other Language Function Pointers

You can declare function pointers as well as functions to be in another language. For instance,

```
__fortran double (*eqn)();
```

defines **eqn** to be a pointer to a routine written in FORTRAN, which returns a **double**. A function pointer in any language other than C or assembler will hereafter be called an *HLL function pointer.* HLL function pointers have a size of 4 bytes, and the data they address are in a format known only to the language of the function. (For example, a **__pli** function pointer addresses a PL/I ENTRY variable.)

The format used by Pascal/VS for FUNCTION and PROCEDURE arguments requires 28 bytes. Because the compiler treats all function pointers as having a size of 4 bytes, additional work is necessary for a C function that has a Pascal routine as an argument. See Chapter 7, "Communication with Pascal," for details.

You cannot use a language keyword such as **__fortran** in any declaration where its application is ambiguous. For instance, the following declaration is wrong:

```
__fortran int (*(*weird)())();  /* erroneous declaration */
```

It is unclear which is in FORTRAN: the function addressed by **weird**, or the function addressed by the value returned by **weird**. If it is necessary to define functions of this sort, you can use a **typedef** to disambiguate. For example,

```
typedef __fortran int (*fortfun)();
fortfun weird();
```

declares **weird** to be a (C) function that returns a pointer to a FORTRAN routine that returns an integer. (Note that, even with the help of **typedefs**, types of this sort are very confusing and difficult to understand, and you should avoid them if possible.)

An HLL function pointer can be given a value in only one of two ways: it can be passed as an argument from the other language or it can be copied by assignment from another function pointer of the same type.

Because the format of an HLL function pointer is unknown, the following code is erroneous and cannot be compiled:

```
__fortran double myfunc();
__fortran double (*somefunc)();
somefunc = &myfunc;  /* ERROR: conversion not supported */
```

The only uses that can be made of a foreign function pointer are to assign it to another similar function pointer, to call it, or to pass it as an argument to a function in C, assembler, or the language of the function pointer. (Thus, you cannot pass a **__pli** function pointer to a FORTRAN routine.)

Function Pointer Arguments

When you call C from another language, you can pass the name of a routine in that language (or a PL/I ENTRY variable), provided that the corresponding C argument is declared as an HLL function pointer of the appropriate type.

Similarly, when you call another language from C, you can pass a function or function address provided that it is declared appropriately in the other language (EXTERNAL in FORTRAN, ENTRY in PL/I, and FUNCTION or PROCEDURE in Pascal). The passed function can be in C, assembler, or the called language and must be declared correctly in the calling routine. (For a function in the called language, use the appropriate language keyword, such as **__pli**. For a function in assembler, use the **__asm** keyword.) If the function is in C or assembler, the value stored in the argument list is converted by the C

library to the format of the called language. See the communication chapters later in this book for details for a particular language.

__alignmem and __noalignmem

The __alignmem and __noalignmem keywords can be used when defining a structure tag to specify how fields should be aligned. These keywords take precedence over the alignment requested by the BYtealign or NOBYtealign options. These keywords can be useful for sharing structures with another language that does not use C rules for structure mapping. For instance, consider a COBOL record that is defined in the linkage section as

```
01 SHARED-DATA.
   05 NAME PICTURE X(17).
   05 ZIP PICTURE 9(5) COMPUTATIONAL.
   05 STATE PICTURE X(15).
```

An equivalent C structure type could be defined by

```
__noalignmem struct shared {
   char name [17];
   int zip;
   char state [15];
};
```

If __noalignmem were not specified, 3 bytes of padding would be inserted in the C structure between the name and zip fields, while the COBOL structure would have no such padding. (Alternately, the SYNCHRONIZED keyword could be used in the COBOL record definition to force alignment of ZIP in COBOL.)

See SAS Technical Report C-106, *Changes and Enhancements to the SAS/C Compiler, Release 4.00*, for more information on __alignmem and __noalignmem.

C Compiler Options for ILC Programs

This section briefly describes compiler options that can be useful for functions that call or are called by another language. For more information, see the *SAS/C Compiler and Library User's Guide* and SAS Technical Report C-106.

AT

The AT option is required if you use the a operator to pass function arguments by reference.

BYtealign

The BYtealign option can be useful for functions that communicate with another language to cause structures to be mapped without padding between fields. This feature is particularly valuable for functions that communicate with COBOL because by default COBOL never aligns record items.

INDep

The INDep option specifies that the compiled functions can be called with the C framework inaccessible. This option is required for any C function that can be called from another language. It is not required

for a C function that calls another language and is not recommended unless required.

The **INDep** option causes the code for each function to call a library routine named L$UPREP on entry. Use of the ILC feature precludes use of the **INDep** option for other purposes, such as those described in Chapter 12, "Using the INDep Option for Interlanguage Communication," and Chapter 13, "Executing SAS/C Programs without the Run Time Library," in the *SAS/C Compiler and Library User's Guide*. For a multilanguage program, a correct version of L$UPREP is selected by the ILCLINK utility and will be different from the L$UPREP for which source is distributed. You cannot replace the L$UPREP selected by ILCLINK with a modified version of your own.

NORENT

When the **NORENT** option is used, **extern** C variables are defined and referenced as external symbols. This permits them to be shared with other languages that also use this technique. For example, a C **extern int** references the same storage as a PL/I STATIC EXTERNAL FIXED BIN(31) variable with the same name. As the option name implies, use of the **NORENT** option means that the resulting program will not be reentrant.

If you use the **RENT** or **RENTExt** option, **extern** data are normally stored in pseudoregisters, which cannot be shared with other languages. You may still be able to share **extern** data with another language if it is declared **const**. See SAS Technical Report C-106 for details.

VString

The **VString** compiler option changes the way in which string literals are passed to PL/I and Pascal. If the **VString** option is not specified, a string literal is passed to PL/I or Pascal as a fixed-length string (PL/I CHAR(*) or Pascal CONST PACKED ARRAY OF CHAR). If the **VString** option is specified, a string literal is passed to PL/I or Pascal as a varying-length string (PL/I CHAR(*) VARYING or Pascal CONST STRING).

The **VString** option increases storage requirements slightly for a C function because it causes each character string literal to be preceded by a 2-byte length prefix. This prefix has no effect on any C statements or operations other than calls to PL/I or Pascal. In particular, the address of the string is still considered to be the address of its first character, not the address of the prefix.

If your application requires that string literals be passed to some routines as fixed-length and some as varying-length, you should specify the **VString** option and use the **__STRING** data type conversion macro when a fixed-length argument is required, as in this example:

```
/* Example presumes use of VString compiler option */
__pli void subv(), subf();
subv("This is a varying-length string.");
subf(__STRING("This is a fixed-length string.," 0));
```

Linking Considerations

Creating an executable module from components in several languages can be very difficult for the following reasons:

- Documentation that is accurate in a single language context may be misleading when several languages are combined. For instance, a load module can have only one entry point, so all but one language's documented entry point must be ignored.
- Restrictions of one language may affect the others. For instance, because Pascal/VS does not produce object code that can execute in 31-bit addressing mode, load modules containing both the SAS/C language and Pascal/VS must be linked with AMODE=24, even though the restriction does not apply to the SAS/C language by itself.
- Linking in several steps may be necessary to reconcile conflicts between languages. For example, when you combine C and PL/I, it is generally necessary to run the CLINK utility using the PREM option even if reentrancy is not required, to prevent the C language's use of pseudoregisters from interfering with PL/I.
- The SAS/C ILC implementation puts unique demands on the linking process. For instance, a program in which C is called by another language requires a version of the L$UPREP run-time function determined by which language or languages will be calling C.

To ameliorate these problems, SAS/C ILC provides the ILCLINK utility, which is a driver program for creation of multilanguage modules. The goal of ILCLINK is to enable the user to describe the program to be linked using relatively simple control statements. ILCLINK invokes whatever other utilities are required, as directed by its control statements. Information about the requirements of other languages and of SAS/C ILC is built into ILCLINK so that the user does not need to be aware of them.

ILCLINK control statements enable the user to specify the following:

- the languages in which the program is written
- the first language to get control and the name of the main routine
- the names of the link utilities to be run (for example, CLINK, the OS linkage editor, the CMS LOAD and GENMOD commands) and any necessary options
- the names of any required autocall (GLOBAL) libraries
- control statements for the link utilities, such as CLINK or linkage editor INCLUDE statements
- one or more TSO or CMS commands to be executed by ILCLINK (this is optional). These commands can be used, for instance, to allocate DD statements referenced by linkage editor control statements.

The following is a sample input file of ILCLINK control statements to build a CMS module using both C and PL/I under CMS. Numbers preceding a control statement are references to notes, not part of the control statements.

```
1. FIRST *(PLI2)
2. LANGUAGE PLI2,C
3. PROCESS CLINK (PREM
4. AUTOCALL LC370
5. INCLUDE PH2C
6. PROCESS LOAD PH2P (NODUP
7. AUTOCALL PLILIB
8. PROCESS GENMOD PH2
```

The control statements have the following meanings:

1. The FIRST statement indicates that the main program is in PL/I. No procedure name is specified, which causes ILCLINK to determine that the correct entry point is PLISTART.
2. The program uses code generated from the SAS/C language and PL/I Version 2.
3. The first program that should be run is CLINK. The CLINK PREM option is specified to prevent C pseudoregister definitions from interfering with PL/I.
4. The SAS/C run-time library is defined as an autocall library for CLINK.
5. The CLINK INCLUDE statement causes CLINK to include the object module PH2C TEXT in its output.
6. The next program that should be run is the CMS LOAD command. The object module PH2P TEXT should be processed at this time. The CMS option NODUP is used to suppress "duplicate CSECT" warning messages, which are usually generated for multilanguage programs.
7. The PL/I run-time library is defined as an autocall library for LOAD.
8. A MODULE file named PH2 is to be generated from the output of the LOAD command.

A complete description of ILCLINK is in Chapter 8, "Linking Multilanguage Programs with the ILCLINK Utility."

4 Communication with FORTRAN

Introduction

This chapter provides the technical details for using SAS/C ILC with FORTRAN. The topics covered are

- the versions of FORTRAN supported by SAS/C ILC
- execution framework considerations
- FORTRAN data types and their corresponding types in C
- passing data from FORTRAN to C and vice versa
- returning data to FORTRAN from C and vice versa
- error handling considerations
- data sharing considerations
- linking considerations
- hints on passing multidimensional arrays between FORTRAN and C.

Many sections include examples of correct and incorrect calls to each language. These examples, plus the accompanying discussion, provide the necessary background to write FORTRAN-C applications. Before reading this chapter, you should be familiar with the material in Chapters 1 through 3.

Versions Supported

SAS/C ILC directly supports communication with programs compiled with VS FORTRAN Version 1 Release 4 or VS FORTRAN Version 2. At the time the program is linked, you must specify to ILCLINK which of the two levels of FORTRAN is in use.

Routines compiled with earlier IBM FORTRAN compilers, such as the G1 or H compilers, should also work correctly if the FORTRAN main and termination routines (L$IFORM and L$IFO1Q) are recompiled with the corresponding compiler. (Source for these routines can be found in SASC.SOURCE under OS or LSU MACLIB under CMS.) If you recompile these routines and replace the corresponding object files, you should specify the language as FORTRAN (not FORTRAN2) to ILCLINK.

Framework Considerations

A restriction of IBM FORTRAN implementations makes it impossible to create, delete, and re-create the FORTRAN framework successfully in the same program. (See the *VS FORTRAN Programming Guide*.) The easiest way to bypass this restriction is to call `mkfmwk` in the C `main` function and terminate the FORTRAN framework only immediately before returning from `main`. If there are calls to `mkfmwk` to create the FORTRAN framework elsewhere in the C program, these calls will have no harmful effects. (However, each extra call to `mkfmwk` must be matched by a corresponding `dlfmwk` call before program termination.)

The FORTRAN main program must not be called MAIN because this name is reserved by the C library. For FORTRAN 66 programs, you should use the FORTRAN compiler option NAME to assign some other name. For FORTRAN 77 programs, you should use a PROGRAM statement in the main routine to assign another name.

FORTRAN Data Types

Table 4.1 lists the common FORTRAN data types and their C equivalents. Consult this table for general information about corresponding data types and to determine how to declare variables shared between the two languages.

Table 4.1
FORTRAN-C Corresponding Data Types

FORTRAN Type	C Type
INTEGER*4	`int, long`
INTEGER*2	`short`
REAL*4	`float`
REAL*8	`double`
LOGICAL*4	`unsigned int, unsigned long`
LOGICAL*1	`char`
COMPLEX*8	`struct {float re,im;}`
COMPLEX*16	`struct {double re,im;}`
CHARACTER*1	`char`
CHARACTER*n	`char [n],` `struct {char text [n];}`
array	`type [n]`

Data types omitted from **Table 4.1** have no close equivalent in the other language.

See Chapter 3, "Communication with Other Languages," for general information on data formats and data sharing.

Passing Data to C from FORTRAN

When you write a C function that can be called from FORTRAN, you must do the following:

1. Compile the C function with the `INDep` option, which is always required for C functions called from another high-level language.
2. Declare each argument to be a pointer to data of the C type corresponding to the type of the data being passed from FORTRAN. (Arguments must be declared as pointers because FORTRAN uses call by reference.)

Table 4.2 shows the type that should be declared for the corresponding C parameter for each FORTRAN argument type. For example, the C argument corresponding to a FORTRAN REAL*4 value should be declared as `float *`. A Yes in the Special Considerations column indicates that additional information on passing values of this type is available in the sections following the table.

Table 4.2
Argument Types for Calls from FORTRAN to C

FORTRAN Type	C Type	Special Considerations
INTEGER*4	`int *, long *`	
INTEGER*2	`short *`	
REAL*4	`float *`	
REAL*8	`double *`	
LOGICAL*4	`unsigned int *,` `unsigned long *`	
LOGICAL*1	`char *`	
COMPLEX*8	`struct {` `  float re,im;}*`	
COMPLEX*16	`struct {` `  double re,im;}*`	
CHARACTER*1	`char *`	

CHARACTER*n	`char [n],`
	`char *,`
	`struct {`
	`  char text [n];}*`
		Yes

EXTERNAL	`__fortran (*)()`	
array	`type [n],` `type *`	Yes

CHARACTER Arguments

The length of a string passed from FORTRAN is not available to C. Therefore, you may want to pass the string length as an additional argument. Another alternative is to end each string argument with an unusual character (such as ~) and search for this terminator in the called C function.

Array Arguments

An array passed from FORTRAN can be declared in the called C routine as either an array or a pointer. The C array element type or pointed-to type should be a type with the same format as the FORTRAN element type. For instance, an INTEGER array passed from FORTRAN can be defined in C to have either the type `int []` or `int *`.

FORTRAN and C address arrays differently. In FORTRAN, the first element of an array is normally element 1, while in C the first element is element 0. For multidimensional arrays, FORTRAN and C use a different order of indexing. The following FORTRAN and C declarations define arrays with the same memory layout:

```
DIMENSION Z(10,5)
float z [5][10]
```

As with string lengths, FORTRAN does not provide information on the dimensions of array arguments. These must be passed as additional arguments if needed by C.

There is no C language construct corresponding to the FORTRAN "dummy array" concept. See **Hints,** later in this chapter, for an example of C code that circumvents this deficiency.

Returning Data to FORTRAN from C

If you call a C function from FORTRAN as a FUNCTION and expect a value to be returned, then the C function must be defined as returning a value of a type with the same format as the return type expected by FORTRAN. If you call a C function as a SUBROUTINE and use a FORTRAN CALL statement, then the C function must be defined as returning `void`.

There is no C equivalent to the FORTRAN RETURN n statement.

Note the following special considerations when returning data to FORTRAN from C.

Returning COMPLEX

You cannot write a C function that returns a COMPLEX value. You should instead pass a COMPLEX argument whose value can be modified by the C function.

Returning CHARACTER

A C function that returns a CHARACTER*n value to FORTRAN should be defined as returning

```
struct {char text [n];}
```

This is true even if n is 1, because a C function that returns `char` is treated by the library as returning a LOGICAL*1 value.

Examples: Calling C from FORTRAN

Assume the following C function headers:

```
void csub(int *ip, float *xp, char *s)

typedef struct {
   char text [20];
} str20;

str20 cfun(short *shp, double *dp, float a [20])
```

The following calls from FORTRAN to C are correct:

```
      CHARACTER*20 CFUN
      INTEGER*4 I
      INTEGER*2 ISH
      REAL*4 X, A(20)
      CHARACTER*32 S
      CHARACTER*20 STR
C
      CALL CSUB(I, X, S)
      CALL CSUB(14, 3.72, 'Hello, world!')
      STR = CFUN(ISH, .314159D1, A)
```

The following calls are incorrect and will probably cause errors during execution for the reasons explained in the comments that follow the calls:

```
        CALL CSUB(12.0, X, S)
C     ERROR: The first argument should be INTEGER*4 rather than a
C            REAL*4 constant.
        STR = CFUN(ISH, 3.14159, A)
C     ERROR: The second argument should be REAL*8 instead of a REAL*
C            constant.
        CALL CFUN(ISH, 3.14159D0, A)
C     ERROR: CFUN returns a value, so it must be called as a
C            FUNCTION not a SUBROUTINE.
```

Passing Data to FORTRAN from C

To call a FORTRAN routine from C, you must do the following:

1. Declare the FORTRAN routine in C using the `__fortran` keyword to inform the compiler that the routine is written in FORTRAN.
2. Make sure that each argument is passed correctly to FORTRAN. (Consult **Table 4.3.** A Yes in the Special Considerations column indicates that additional information is available in the sections

following the table.) There are three different ways an argument can be passed:

- For argument types without a Special Considerations entry in the table and for others as described below, the argument can be passed directly and will be converted correctly by the compiler.
- Any argument except a character string argument can be passed using a pointer to an appropriate type of data. The argument can be a pointer variable or an address expression using the `&` or `@` operator. In this case, the compiler makes no attempt at conversion and simply stores the pointer in the FORTRAN parameter list.
- Some arguments should be passed using data type conversion macros to convert the argument to the required FORTRAN data type. For instance, the `_STRING` macro can be used to specify that a C character pointer corresponds to a CHARACTER*n argument rather than a LOGICAL*1. **Table 4.3** shows situations in which a macro can be used with a particular type.

Table 4.3
Argument Types for Calls from C to FORTRAN

FORTRAN Type	C Type	Special Considerations	Macro
INTEGER*4	`int, long`		
INTEGER*2	`short`		
REAL*4	`float`		
REAL*8	`double`		
LOGICAL*4	`unsigned int, unsigned long`		
LOGICAL*1	`char`		
COMPLEX*8	`struct { float re,im;}`		
COMPLEX*16	`struct { double re,im;}`		
CHARACTER*1	`char`	Yes	
CHARACTER*n	`char [n], char *, struct { char text [n];}`	Yes	`_STRING`
EXTERNAL	`(*)(), __fortran (*)()`	Yes	
array	`type [n], type *`	Yes	
array of CHARACTER*n	`char [m][n], struct { char text [n];}[m]`	Yes	`_STRING`

char Arguments

The C language definition specifies that a character constant, such as `'a'`, has type `int` rather than type `char`. For this reason, an argument that is a character constant is passed correctly to FORTRAN, assuming the FORTRAN argument is INTEGER. To pass

the value as a LOGICAL*1, use a cast (that is, `(char)'a'`). To pass the value as a CHARACTER*1, write it as a string literal rather than as a character literal (that is, as `"a"`).

String Arguments

Arguments that are declared in FORTRAN as CHARACTER*n should be passed in one of the following ways:

- using a string literal, such as `"abc"`
- using a structure of type `struct{char text [n];}`
- using the `_STRING` macro, as described below.

The `_STRING` macro, defined in the header file `<ilc.h>`, is used to inform the compiler that a `char *` or `char []` argument is to be passed to FORTRAN as a CHARACTER*n string rather than as a pointer. The argument should be specified as

```
_STRING(str, len)
```

where `str` is the pointer and `len` is the string length. `len` may be specified as 0 if the string length should be computed using the `strlen` function. A `len` of 0 is appropriate only if the FORTRAN argument is CHARACTER*(*). (A more complete description of `_STRING` is in **Data Type Conversion Macros** later in this chapter.)

To illustrate the above, suppose that FPRINT is a FORTRAN subroutine with a single CHARACTER*(*) argument. In this case, the following C calls are all correct:

```
struct {char text [9];} str1 = {"First way"};
char *str2 = "Second way";
void __fortran FPRINT();

FPRINT(str1);
FPRINT(_STRING(str2, 6));   /* only "Second" is passed */
FPRINT(_STRING(str2, 0));   /* entire string is passed */
FPRINT("Third way");
```

You should not pass a C `char *` or `char []` value directly for a FORTRAN CHARACTER*n or CHARACTER*(*) argument because no length information will be passed. This may cause the called FORTRAN routine to generate a diagnostic or produce incorrect results. For this reason, the following call to FPRINT is incorrect:

```
FPRINT(str2);  /* ERROR: no string length is passed */
```

Array Arguments

See the comments in **Passing Data from C to FORTRAN** earlier in this chapter for a discussion of the differences between FORTRAN arrays and C arrays. Although they are subject to these differences, C arrays can be passed directly to FORTRAN. The `_ARRAY` macro (as described in Chapter 6, "Communication with PL/I") can also be used but is unnecessary.

For a FORTRAN argument defined as an array of CHARACTER*n, the C argument should be a two-dimensional array of characters or a one-dimensional array of string structures. For an argument defined as an array of CHARACTER*(*), the C argument must be passed using the `_STRING` macro, and a length of 0 cannot be specified.

Function Pointers Either a `__fortran` function pointer or a standard C function pointer can be passed to a FORTRAN EXTERNAL argument. However, a C function pointer must not address a function in another load module.

Returning Data to C from FORTRAN

If you are calling a FORTRAN FUNCTION from C, it must be declared in C as returning a value of a type with the same format as the returned FORTRAN type. If you are calling a FORTRAN SUBROUTINE from C, it must be declared in C as returning `void`.

You cannot use the FORTRAN RETURN n statement in a FORTRAN routine called from C.

Note the following special considerations when returning data to C from FORTRAN.

Returning COMPLEX You cannot call FORTRAN routines returning COMPLEX from C. Such routines should be modified to store the return value into a COMPLEX argument.

Returning CHARACTER*n A FORTRAN routine that returns a CHARACTER*n value to C should be defined as returning a

```
struct {char text [n];}
```

This is true even if n is 1, because a FORTRAN routine returning `char` is assumed by the library to return a LOGICAL*1 value.

Data Type Conversion Macros

This section describes the use of the `_STRING` data type conversion macro with FORTRAN.

_STRING Pass String Argument to FORTRAN

SYNOPSIS

```
#include <ilc.h>

_STRING(char *str, unsigned len);
```

DESCRIPTION

The `_STRING` macro is used to pass a `char *` or `char []` value to a FORTRAN CHARACTER*n or CHARACTER*(*) argument. The `str` argument should be a pointer to the first byte of the string to be passed. (It can be a string literal.) The `len` argument should be the string length to be passed to FORTRAN. If `len` is specified as 0, the string length is determined at execution time by invoking `strlen`.

`_STRING` can also be used to pass an array of strings to a FORTRAN CHARACTER*(*) array argument. The C array of strings must be a two-dimensional array of `char` or an array of string structures

```
struct {char text [n];}
```

It cannot be an array of `char *`. In the case of a string array, the value of `n` cannot be specified as 0.

EXAMPLE

See **Examples: Calling FORTRAN from C** below.

Examples: Calling FORTRAN from C

Assume the following FORTRAN FUNCTION and SUBROUTINE definitions:

```
SUBROUTINE FSUB(I, X, S)
INTEGER I
REAL X
CHARACTER *(*) S

REAL*8 FFUN(IS, DP, A)
INTEGER*2 IS
REAL*8 DP
REAL A(20)
```

The following calls from C to FORTRAN are correct:

```
__fortran void fsub();
__fortran double ffun();

int i;
short sh;
float x;
char *s;
float a [20], *ap;
double result, *dp;

fsub(i, x, _STRING(s, 15));
fsub(17, 2.7128F, "Hello, world!");
result = ffun(sh, 54.4, a);
ap = malloc(20 * sizeof(float));
result = ffun((short) 25, dp, ap);
```

The following calls are incorrect and will probably cause errors during execution for the reasons explained in the comments that follow the calls:

```
fsub(i, x, s);
     /* The third argument requires a string length */
result = ffun(25, 54.4, a);
     /* The first argument should be short instead of int */
```

Error Handling Considerations

Under OS, creation of the FORTRAN framework requires that a FT06F001 DD card be present for FORTRAN error message output. Any attempt to create the FORTRAN framework will fail if no DD card is present, possibly without a message being generated. (This problem does not occur in CMS.)

After some number of run-time errors, the FORTRAN library will terminate the FORTRAN framework. (See the *VS FORTRAN Programming Guide* for more information.) When both FORTRAN and C are in use and the FORTRAN framework is terminated due to errors (or due to use of the STOP statement), the C framework is terminated as well. The C library generates a message in this case to explain the reason for termination.

External Data Sharing Considerations

When you use the SAS/C **NORENT** compiler option, you can share data between FORTRAN and C via COMMON blocks as well as via subroutine arguments. (Note that there is no reason not to use the **NORENT** option when your program contains FORTRAN because FORTRAN object code is never reentrant.) To share data in this way, you should declare an **extern** C structure whose name is the same as the FORTRAN COMMON block. Each element of the structure should be defined to be an equivalent data type to the corresponding FORTRAN variable in the COMMON block. The COMMON must not

be a dynamic COMMON. For example, the following FORTRAN COMMON block and C structure allow access to the same data:

```
COMMON /MYDATA/ I,J,A(40),CH
CHARACTER*20 CH

extern struct {
   int i,j;
   float a [40];
   char ch [20];
} mydata;
```

Note that the COMMON block must be declared but not defined in C because the FORTRAN declarations constitute the definition.

Note that some versions of FORTRAN only accept COMMON blocks whose names are six characters or fewer. Even for versions of FORTRAN without this restriction, shared names are limited to eight characters by the linkage editor's restrictions on external symbols.

Also, be aware that the SAS/C compiler changes the underscore (_) character in external names to a pound sign (#). For this reason, an external variable shared between C and FORTRAN should not have an underscore in its name.

Linking Considerations

Both the FORTRAN and C libraries contain a function named "exit." With VS FORTRAN Version 2, the FORTRAN EXIT routine is always included in the load module, which can lead to problems if a C function calls the C `exit` because it will access the FORTRAN version. C functions that call `exit` and that can be used with FORTRAN should include `<stdlib.h>`, which renames `exit` to `_C_exit` (avoiding conflict with the FORTRAN library). The `<fortmath.h>` header file (discussed below) also contains this redefinition of `exit`.

There can also be naming conflicts between FORTRAN and C math routines. For programs that want to use mathematical functions in both languages, it is possible to call either the C math functions or the FORTRAN math functions from C. In either case, you should include the header file `<fortmath.h>`. The C math functions are then called as normal; for example, the call `exp(4.7)` calls the C `exp` function. (The header file renames `exp` to `_exp` to avoid the naming conflict.) On the other hand, to call the FORTRAN DEXP function, you should use the function name `F_DEXP`. This generates a correct interlanguage call to the FORTRAN library function. Note that one reason to call the FORTRAN math library functions from C is to ensure that the same algorithm is always used for the same function, regardless of whether the call is from FORTRAN or from C.

Hints

Multidimensional Arrays

One convenient FORTRAN feature that is not present in C is the ability to have multidimensional arrays in which the array dimensions are not known at compile time. This is especially handy for writing or

using functions and subroutines that perform matrix manipulations. This section gives an example of how to achieve the same effect in C.

Here is a simple fragment of FORTRAN code using this feature:

```
      REAL*4 FUNCTION ITEM(A,M,N,I,J)
C     Return item I,J of a M by N matrix.
      DIMENSION A(M,N)
      ITEM = A(I,J)
      RETURN
      END
```

Because C requires constant array bounds for all but the last dimension, this code cannot be duplicated directly in C. However, this effect can be achieved by treating the two-dimensional FORTRAN array as a one-dimensional C array and using the C preprocessor to perform the subscript calculations. Treating the two-dimensional FORTRAN array as a one-dimensional C array works because elements are contiguous in each representation and because only the address of the first element of the array is passed from FORTRAN to C.

This leads to the following C code:

```
float item(float a[ ], int m, int n, int i, int j)
{
#define asub(s1,s2) a[((s2)-1)*m+((s1)-1) ]
      return asub(i,j);
   }
```

Note that the **asub** macro compensates for the C language's use of zero-based rather than FORTRAN's one-based array indexing.

This technique of using a one-dimensional C array to "stand in" for a multidimensional FORTRAN array also can be used conveniently for array arguments passed from C to FORTRAN, because C arrays are always passed as pointers to element zero.

5 Communication with COBOL

Introduction

This chapter provides the technical details for using SAS/C ILC with COBOL. The topics covered are

- the versions of COBOL supported by SAS/C ILC
- execution framework considerations
- COBOL data types and their corresponding types in C
- passing data from COBOL to C and vice versa
- returning data to COBOL from C and vice versa
- restrictions on using the SAS/C COBOL interface.

Many sections include examples of correct and incorrect calls to each language. These examples, plus the accompanying discussion, provide the necessary background to write COBOL-C applications. Before reading this chapter, you should be familiar with the material in Chapters 1 through 3.

Versions Supported

SAS/C ILC supports communication with programs compiled with OS/VS COBOL (Version 1) or VS COBOL II (Version 2). At the time the program is linked, you must specify to ILCLINK which of the two levels of COBOL is in use.

Framework Considerations

A restriction of VS COBOL II causes COBOL run-time options to be ignored when `mkfmwk` is called to create the COBOL framework. If you need nondefault COBOL run-time options for your application, you should write the main program in COBOL and use `CFMWK` to create the C framework, passing any necessary C run-time options at this time.

COBOL Data Types

Table 5.1 lists the common COBOL data types and their C equivalents. Consult this table for general information about corresponding data types.

Table 5.1
COBOL-C Corresponding Data Types

COBOL Type	C Type
COMP PIC S9(9)	`int, long`
COMP PIC S9(4)	`short`
COMP PIC 9(9)	`unsigned int, unsigned long`
COMP PIC 9(4)	`unsigned short`
COMP-1	`float`
COMP-2	`double`
COMP-3	no equivalent
PIC X	`char`
PIC X(n),	`char [n],`
PIC DISPLAY	`struct {char text [n];}`
POINTER	`type *, void *`
01	`struct`
01 ...	`type [n]`
mm ... OCCURS n TIMES	

Data types omitted from **Table 5.1** have no close equivalent in the other language.

See Chapter 3, "Communication with Other Languages," for general information on data formats and data sharing.

Passing Data to C from COBOL

When you write a C function that can be called from COBOL, you must do the following:

1. Compile the C function with the `INDep` option, which is always required for C functions called from another high-level language.
2. Declare each argument to be a pointer to data of the C type corresponding to the type of data being passed from COBOL. (Arguments must be declared as pointers because COBOL uses call by reference.)

   Table 5.2 shows the type that should be declared for the corresponding C parameter for each COBOL argument type. For example, the C argument corresponding to a COBOL COMP PIC S9(4) value should be declared as `short *`. A Yes in the Special

Considerations column indicates that additional information on passing values of this type is available in the sections following the table.

Table 5.2
Argument Types for Calls from COBOL to C

COBOL Type	C Type	Special Considerations
COMP PIC S9(9)	`int *, long *`	
COMP PIC S9(4)	`short *`	
COMP PIC 9(9)	`unsigned int *, unsigned long *`	
COMP PIC 9(4)	`unsigned short *`	
COMP-1	`float *`	
COMP-2	`double *`	
COMP-3	`char (*) [n]`	Yes
PIC X	`char *`	Yes
PIC X(n),	`char [n],`	Yes
PIC DISPLAY	`char *, struct { char text [n];}*`	
POINTER	`type **, void **`	
01	`struct *`	Yes
01 ... mm ... OCCURS n TIMES	`type [n]`	

PIC X(n) Arguments

The length of a string passed from COBOL is not available to C. Therefore, you may want to pass the string length as an additional argument. Another alternative is to end each string argument with an unusual character (such as ~) and search for this terminator in the called C function.

COMP-3 Arguments

The `pdval` and `pdset` macros can be used to access and modify COMP-3 (packed decimal) data passed from COBOL. See Chapter 12, "Using Packed Decimal Data in C," for information on the use of these macros.

Table Arguments

A table passed from COBOL can be declared either as an array or a pointer in the called C function. Note that COBOL table declarations may have subordinate level items that are redundant in C and can be omitted. For instance, consider the following COBOL table:

```
01  C-ARRAY.
    05  C-ITEM OCCURS 10 TIMES COMP PIC 9(9).
```

The corresponding C argument can be declared in C as `struct {unsigned item [10]}`, `unsigned [10]`, or `unsigned *`.

As is the case with string lengths, COBOL does not pass information on the dimensions of tables. These must be passed as additional arguments if needed by C.

Record Arguments

When a COBOL record is passed to C, the C argument should be a pointer to an *equivalent structure*. An equivalent structure is a structure in which each C member matches the corresponding COBOL item based on the correspondences defined in **Table 5.1.**

Note that COBOL does not align items in a record unless the SYNCHRONIZED clause is used. You can use the C compiler option `BYtealign` or the keyword `__noalignmem` to suppress alignment in C structures corresponding to COBOL records.

Returning Data to COBOL from C

Because COBOL does not provide a way for programs called by COBOL to return a value, any C function called from COBOL should be declared as returning `void`. Because COBOL always uses call by reference, a C function can modify the areas addressed by its arguments to pass data back to its caller.

Examples: Calling C from COBOL

Assume the following C function header:

```
struct cobrec {
    int field1;
    char field2; }

void csub(int *itemsptr,
    char comment [20],
    struct cobrec *cobrecptr,
    unsigned short table [10])
```

The following call from COBOL to C is correct:

```
WORKING-STORAGE SECTION.
77  ITEMS    PIC S9(9) COMP.
77  AMOUNT   PIC S9(9) DISPLAY.
77  COMMENT PIC X(20).
01  SINGLE.
    05  FIELD-1  PIC S9(9) COMP SYNCHRONIZED.
    05  FIELD-2  PIC X.
01  SEVERAL.
    05  ELEMENT  PIC 9(4) COMP OCCURS 10 TIMES.

CALL "CSUB" USING ITEMS COMMENT SINGLE SEVERAL.
```

The following calls are incorrect and will probably cause errors during execution for reasons explained in the comments that follow the call:

```
CALL "CSUB" USING AMOUNT COMMENT SINGLE SEVERAL.
* ERROR: AMOUNT is USAGE DISPLAY, not USAGE COMPUTATIONAL.
CALL "CSUB" USING ELEMENT (3) COMMENT SINGLE SEVERAL.
* ERROR: ELEMENT has PIC 9(4), and so does not match a C int.
```

Passing Data to COBOL from C

To call a COBOL routine from C, you must do the following:

1. Declare the COBOL routine in C using the `__cobol` keyword, which informs the compiler that the routine is written in COBOL.
2. Make sure that each argument is passed correctly to COBOL by consulting **Table 5.3.** A Yes in the Special Considerations column indicates that additional information is available in the sections following the table. There are two ways an argument can be passed:
   - For argument types without a Special Considerations entry in the table and for others as described below, the argument can be passed directly and will be converted automatically by the compiler.
   - Any argument can be passed using a pointer to an appropriate type of data. This could be a pointer variable or an address expression using the `&` or `@` operator. In this case, the compiler makes no attempt at conversion and simply stores the pointer in the COBOL parameter list.

Note that when calling COBOL, you do not need to use data type conversion macros such as `_STRING`.

Table 5.3
Argument Types for Calls from C to COBOL

COBOL Type	C Type	Special Considerations
COMP PIC S9(9)	`int, long`	
COMP PIC S9(4)	`short`	
COMP PIC 9(9)	`unsigned int, unsigned long`	
COMP PIC 9(4)	`unsigned short`	
COMP-1	`float`	
COMP-2	`double`	
COMP-3	`char [n]`	Yes

(*continued*)

Table 5.3 (continued)

COBOL Type	C Type	Special Considerations
PIC X	`char`	Yes
PIC X(n), PIC DISPLAY	`char [n],` `char *,` `struct { char text [n];}`	Yes
POINTER	`type *, void *`	Yes
01	`struct`	Yes
01 ... mm ... OCCURS n TIMES	`type [n]`	

Packed Decimal Arguments

You can create packed decimal data in C using the `pdval` macro, as described in Chapter 12, "Using Packed Decimal Data in C." Such data should be stored in a `char` array whose size is `(m+1)/2`, where `m` is the number of digits declared in COBOL.

char Arguments

The C language definition specifies that a character constant, such as `'a'`, has type `int` rather than type `char`. For this reason, an argument that is a character constant will be passed to COBOL assuming the COBOL argument is COMP PIC S9(9). To pass the value as a PIC X, use a cast (that is, `(char)'a'`) or write it as a string literal rather than as a character literal (that is, as `"a"`).

String Arguments

String literals and character arrays are passed correctly to COBOL PIC X(n) arguments without any special effort on your part. Note that literals should contain at least as many characters as specified by the COBOL picture. The `_STRING` macro (as described in Chapter 4, "Communication with FORTRAN," and Chapter 6, "Communication with PL/I") can also be used but is unnecessary.

Pointer Arguments

If you want to pass a C pointer value to a VS COBOL II POINTER item, you must use the `&` or the `@` operator. For instance, if `recp` is a pointer to a C structure, you must write `cobfun(&recp)`, not `cobfun(recp)`. (The latter stores the value of `recp`, not the address, in the COBOL parameter list.) Similarly, if `rec` is a C structure, in order to pass the address of `rec` to a COBOL POINTER, you should write `cobfun(@&rec)`, not `cobfun(&rec)`.

Structure Arguments

When a C structure is passed to COBOL, the COBOL argument should be an *equivalent record*. An equivalent record is a record in which each COBOL item matches the corresponding C field based on the correspondences defined in **Table 5.1**.

Note that, by default, COBOL does not align items in a record unless the SYNCHRONIZED clause is used. You can use the C compiler option `BYtealign` or the keyword `__noalignmem` to suppress alignment in C structures corresponding to COBOL records.

Returning Data to C from COBOL

Because COBOL does not provide any way for a program to return a value to its caller other than by altering a parameter, COBOL programs called from C should always be declared in C as returning `void`.

Note that the COBOL RETURN-CODE special register cannot be accessed or modified from C.

Examples: Calling COBOL from C

Assume the following COBOL linkage section:

```
PROGRAM ID.  COBSUB.

LINKAGE SECTION.
77  C-ITEMS   PIC S9(9) COMP.
77  C-MESSAGE PIC X(20).
77  REC-ADDR POINTER.
01  C-RECORD.
    05  FIELD-1  PIC S9(9) COMP.
    05  FIELD-2  PIC X.

PROCEDURE DIVISION USING C-ITEMS C-MESSAGE REC-ADDR C-RECORD.
```

The following call to COBOL from C is correct:

```
__cobol void cobsub();
int itemcnt;
char comment [20];
__noalignmem struct cobrec {
   int field1;
   char field2;
};
struct cobrec c_record1, c_record2, *recordptr;

cobsub(12, comment, @&c_record1, c_record2);
```

The following call is incorrect and will probably cause errors during execution for reasons explained in the comment following the call:

```
cobsub(itemcnt, "A wrong comment", recordptr, c_record2);
      /* ERROR: The string literal is the wrong length (fewer than
                20 characters).  Also, &recordptr, not recordptr,
                should be passed. */
```

Restrictions

The following restrictions apply when using SAS/C ILC with COBOL:

- A C function cannot be called dynamically from COBOL. For this reason, you must use the COBOL compiler option NODYNAM to compile any COBOL program that calls C.
- COBOL requires that all compilations of a run unit be compiled either with the RESIDENT option or the NORESIDENT option. The SAS/C COBOL interface includes several COBOL modules that were compiled with the RESIDENT option. Therefore, you must compile all COBOL routines in a COBOL-C mixture using this option.

  Alternately, you can recompile the SAS/C interface routines (L$ICB1M and L$ICB1Q for OS/VS COBOL, L$ICB2M and L$ICB2Q for VS COBOL II) with the NORESIDENT option. Note that the ENDJOB option is required when you compile L$ICB1M.

6 Communication with PL/I

Introduction

This chapter provides the technical details for using SAS/C ILC with PL/I. The topics covered are

- the versions of PL/I supported by SAS/C ILC
- PL/I data types and their corresponding types in C
- passing data from PL/I to C and vice versa
- returning data to PL/I from C and vice versa
- data type conversion macros
- error handling considerations
- data sharing considerations
- linking considerations
- restrictions on using the SAS/C PL/I interface
- hints on debugging and calling value-returning C functions from PL/I.

Many sections include examples of correct and incorrect calls to each language. These examples, plus the accompanying discussion, provide the necessary background to write PL/I-C applications. Before reading this chapter, you should be familiar with the material in Chapters 1 through 3.

Versions Supported

SAS/C ILC directly supports communication with programs compiled with the PL/I Optimizing Compiler (Version 1, Release 5), the PL/I Checkout Compiler (Version 1, Release 3), and OS PL/I (Version 2). At the time the program is linked, you must specify to ILCLINK which version of PL/I is in use. If the CMPAT(V1) option of OS PL/I is in use, you must specify the language as PLI, not PLI2.

The PL/I support is not compatible with the level F PL/I compiler. If you recompile the PL/I main and termination routines (L$IPL1M and L$IPLIQ), it should be possible to communicate with earlier releases of the PL/I Optimizing Compiler. (Source for these routines can be found in SASC.SOURCE under OS or LSU MACLIB under CMS.)

PL/I Data Types

Table 6.1 lists the common PL/I data types and their C equivalents. You should consult this table for general information about corresponding data types and to determine how to declare variables shared between the two languages.

Table 6.1
PL/I-C Corresponding Data Types

PL/I Type	C Type
FIXED BINARY(15)	`short`
FIXED BINARY(31)	`int, long`
FIXED DECIMAL	no equivalent
FLOAT DECIMAL(6)	`float`
FLOAT DECIMAL(16)	`double`
CHAR(1)	`char`
CHAR(n),	`char [n],`
PICTURE	`struct {char text [n];}`

CHAR(n) VARYING	`struct {`
	`  short len;`
	`  char text [n];}`

BIT(n)	no equivalent
POINTER	`type *, void *`
array	`type []`
structure	`struct`

Data types omitted from the table have no close equivalent in the other language.

See Chapter 3, "Communication with Other Languages," for general information on data formats and data sharing.

Passing Data to C from PL/I

When you write a C function that can be called from PL/I, you must do the following:

1. Compile the C function with the `INDep` option, which is always required for C functions called from another high-level language.
2. Declare the C function in PL/I as OPTIONS(ASM, INTER). This means that the C function must be declared in C as returning `void`.
3. Declare each argument in C to be a pointer to data of a C type with the same format as the data being passed from PL/I. (Arguments must be declared as pointers because PL/I uses call by reference.)

Table 6.2 shows the type that should be declared for the corresponding C parameter for each PL/I argument type. For example, the C argument corresponding to a PL/I FIXED BINARY(15) variable should be declared as `short *`. A Yes in the Special Considerations column indicates that there are special considerations described in the sections following the table for passing this type of data.

Table 6.2
Argument Types for Calls from PL/I to C

PL/I Type	C Type	Special Considerations
FIXED BINARY(15)	`short *`	
FIXED BINARY(31)	`int *, long *`	
FIXED DECIMAL	`char (*) [n]`	Yes
FLOAT DECIMAL(6)	`float *`	
FLOAT DECIMAL(16)	`double *`	
CHAR(1)	`char *`	Yes
CHAR(n),	`char [n],`	Yes
PICTURE	`char *,`	
	`struct {`	
	`  char text [n];}*`	
CHAR(n) VARYING	`struct {`	Yes
	`  short len;`	
	`  char text [n];}*`	
BIT(n)	`char *`	Yes
POINTER	`type **, void **`	Yes
ENTRY	`__pli (*)()`	Yes
array	`type []`	Yes
structure	`struct *`	Yes

FIXED DECIMAL Arguments

The `pdval` and `pdset` macros can be used to access and modify FIXED DECIMAL data passed from PL/I. See Chapter 12, "Using Packed Decimal Data in C," for information on the use of these macros.

CHAR(n) Arguments

C arguments representing PL/I CHAR(n) variables can be declared as `char []`, `char *`, or `struct {char text[n];}`.

The size of a CHAR(n) string passed from PL/I is not available to C. Therefore, you may want to pass the string length as an additional argument or pass the data as a CHAR(n) VARYING value. Another alternative is to end each string argument with an unusual character (such as ~) and search for this terminator in the called C function.

CHAR(n) VARYING Arguments

A C argument corresponding to a PL/I CHAR(n) VARYING string should be declared as

```
struct {short len; char text [n];}
```

The value of the `len` field is the current string length, and the characters of `text` after the current length are undefined. Information is not available about the maximum string length. If the called C function requires this information, you should pass it as an additional argument.

BIT(n) Arguments

A PL/I BIT(n) bit string can be processed most easily in C as a `char [(n+7)/8]` array. The first bit of the string must be on a byte boundary, which can be forced by use of the PL/I ALIGNED keyword. The size of the bit string is not available unless it is passed as an additional argument.

Array Arguments

An array passed from PL/I can be declared either as an array or a pointer in the called C function. The type of the array element or pointed-to object should have the same data format as the PL/I element type. For instance, a FLOAT(6) array passed from PL/I can be defined in C as either `float []` or `float *`.

The first element of a PL/I array should always be addressed in C using subscript 0. (By default, PL/I arrays start with element 1, but this can be changed on an individual array basis.)

Structure Arguments

You can pass a structure from PL/I to C if the C argument is declared to be a pointer to an equivalent structure. An equivalent structure is a structure in which each C member matches the corresponding PL/I element, using the correspondences defined by **Table 6.1**. Note that PL/I uses a different algorithm for aligning structure fields than C; therefore, you may have to introduce filler items to force the same mapping. (See "Structure Mapping" in the *OS and DOS PL/I Language Reference Manual* for a complete description of the PL/I alignment rules.)

Returning Data to PL/I from C

C functions must be declared in PL/I as OPTIONS(ASM) to prevent the generation of PL/I descriptors for arguments. PL/I allows OPTIONS(ASM) routines to be called only through the CALL statement, not through functional notation. For this reason, C functions called from PL/I should always be declared as returning `void`. If data are to be returned from C, they should be returned by storing into the areas addressed by the function's arguments, rather than via the `return` statement.

If all function arguments and return values are limited to a small set of types, it is possible to bypass the OPTIONS(ASM) requirement and call a C function that returns a value from PL/I. See **Calling C Functions That Return a Value** later in this chapter for information on this.

Examples: Calling C from PL/I

Assume the following C function headers:

```
void csub1(short *isp, float *xp, char *str)

typedef struct {
   short len;
   char text [20];
} vstr20;

struct plidata {
   int number;
   vstr20 name;
};

void csub2(struct plidata *datap, vstr20 *vstrp)

void csub3(struct plidata **ptrp)
```

The following calls from PL/I to C are correct:

```
DECLARE (CSUB1, CSUB2, CSUB3) ENTRY OPTIONS(ASM,INTER);
DECLARE IS FIXED BIN(15);
DECLARE X FLOAT(6);
DECLARE STR CHAR(25);
DECLARE LOC POINTER;
DECLARE 1 DATA BASED(LOC),
          2 NUMBER FIXED BIN(31),
          2 NAME CHAR(20) VARYING;
DECLARE MESSAGE CHAR(20) VARYING INITIAL('Goodbye, world!');
CALL CSUB1(IS, X, 'Hello, world!');
CALL CSUB1(IS, 3.14159E0, STR);
CALL CSUB2(DATA, MESSAGE);
CALL CSUB3(LOC);
```

The following calls are incorrect and will probably cause errors during execution for the reasons explained in the comments that follow the call:

```
CALL CSUB1(17, X, 'Never mind');
     /* ERROR: The first argument should be FIXED BINARY(15)
               instead of a FIXED DECIMAL(2) constant */
CALL CSUB1(IS, 3.14159, STR);
     /* ERROR: The second argument should be FLOAT(6)
               instead of a FIXED DECIMAL(6,5) constant */
```

```
CALL CSUB2(DATA, 'Greetings!');
     /* ERROR: The second argument should be CHAR VARYING
               instead of a fixed length CHAR constant */
```

To avoid problems such as these, you can give full PL/I declarations for C routines, as in the following example:

```
DECLARE CSUB1 ENTRY(FIXED BINARY(15), FLOAT(6), CHAR(*))
              OPTIONS(ASM,INTER);
```

This declaration causes the arguments to be converted appropriately whenever possible. (For instance, both of the erroneous calls to CSUB1 in this example would be corrected by the presence of this declaration.)

Passing Data to PL/I from C

To call a PL/I routine from C, you must do the following:

1. Declare the PL/I routine in C using the `__pli` keyword to inform the compiler that the routine is written in PL/I.
2. Make sure each argument is passed correctly to PL/I by consulting **Table 6.3.** In **Table 6.3**, a Yes in the Special Considerations column indicates that additional information is available in the sections following the table. There are three ways an argument can be passed:
   - For argument types without a Special Considerations entry in the table and for others as described below, the argument can be passed directly and will be converted correctly by the compiler.
   - Any argument can be passed using a pointer to an appropriate type of data. This can be a pointer variable or an address expression using the `&` or `@` operator. In this case, the compiler makes no attempt at conversion and simply stores the pointer in the PL/I parameter list.

     Note: If the PL/I argument type is not scalar (arithmetic or pointer), PL/I expects to receive a pointer to a descriptor rather than to data. If you pass aggregate arguments directly using pointers, it is your responsibility to build any necessary descriptors.
   - Some arguments should be passed using macros to convert the argument to the required PL/I data type. For instance, the `__ARRAY` macro can be used to specify that an integer pointer corresponds to an array argument rather than to a FIXED BIN(31) scalar. Macros that can be used with particular types are shown in **Table 6.3.**

Table 6.3 *Argument Types for Calls from C to PL/I*

PL/I Type	C Type	Special Considerations	Macro
FIXED BINARY(15)	`short`		
FIXED BINARY(31)	`int, long`		
FIXED DECIMAL	`char [n]`	Yes	
FLOAT DECIMAL(6)	`float`		
FLOAT DECIMAL(16)	`double`		
CHAR(1)	`char`	Yes	
CHAR(n),	`char [n],`	Yes	`__STRING`

PICTURE	`char *,`
	`struct {`
	`  char text [n];}`		

CHAR(n) VARYING	`struct {`
	`  short len;`
	`  char text [n];}`
		Yes	

BIT(n)	`char *`	Yes	`__BIT`
POINTER	`type *, void *`	Yes	
ENTRY	`(*)(),` `__pli (*)()`	Yes	

array	`type []`	Yes	`__ARRAY,`
			`__ARRAY2,`
			`__ARRAY3`

array of CHAR(n)	`char [][n],`
	`struct {`
	`  char text [n];}[]`
		Yes	`__STRARRAY`

array of CHAR(n) VARYING	`struct {`
	`  short len;`
	`  char text [n];}[]`
		Yes	`__STRARRAY`

structure	Not supported	Yes	

char Arguments The C language definition specifies that a character constant, such as `'a'`, has type `int` rather than `char`. For this reason, an argument that is a character constant is passed correctly to PL/I, assuming the PL/I argument is a FIXED BIN(31). To pass the value as a CHAR(1), use a cast, such as `(char)'a'`, or write the value as a string literal rather than as a character literal, such as `"a"`.

String Arguments

When a C string literal is passed to PL/I, the compiler assumes that the corresponding PL/I argument is declared to be CHAR(*). In many cases, the argument is actually a CHAR(*) VARYING. If you use the SAS/C compiler option `VString`, all string literals are generated with a PL/I VARYING string prefix and are passed to PL/I as CHAR(*) VARYING. If string literals must sometimes be passed as CHAR(*) and sometimes as CHAR(*) VARYING, you should specify the `VString` compiler option and use the `__STRING` macro when the corresponding PL/I argument is a CHAR(*).

Fixed-length string arguments

Arguments that are declared in PL/I as CHAR(n) or CHAR(*) should be passed in one of the following ways:

- using a string literal, such as `"abc"` (only if the `VString` option is not used)
- using a structure of type `struct {char text [n];}`
- using the `__STRING` macro, as described in **Data Type Conversion Macros** later in this chapter.

Varying-length string arguments

Arguments that are declared in PL/I as CHAR(n) VARYING or CHAR(*) VARYING should be passed in one of the following ways:

- using a string literal, such as `"abc"` (only if the `VString` C compiler option is used)
- using a structure of type

```
struct {short len; char text [n];}
```

To assist in sharing VARYING strings between C and PL/I, the header file `<vstring.h>` declares some useful macros for manipulation of these structures. See Chapter 13, "C Varying-Length String Macros," for details.

Bit Arguments

C data that should be processed as BIT(n) in PL/I should be passed to PL/I using the `__BIT` macro, which is described in **Data Type Conversion Macros** later in this chapter.

Pointer Arguments

To pass a C pointer value to a PL/I POINTER argument, you must use the `&` or `@` operator. For instance, if `dblp` is a pointer to a `double`, you must write `plifun(&dblp)`, not `plifun(dblp)`. (The latter stores the value of `dblp`, not the address, in the PL/I parameter list.) Similarly, if `dbl` is a C `double`, to pass the address of `dbl` to a PL/I POINTER, you must write `plifun(@&dbl)`, not `plifun(&dbl)`.

Array Arguments

To pass a C array to a PL/I array argument, you must use a macro. The `__ARRAY`, `__ARRAY2`, and `__ARRAY3` macros are provided to pass one-, two-, and three-dimensional arrays, respectively. The `__STRARRAY` macro must be used to pass an array of strings.

Note that even though C arrays have lower bound 0, they are always passed to PL/I with lower bound 1. (For this reason, array arguments passed from C to PL/I should always be declared in PL/I with lower bound 1 or *.) This makes it easier to use existing PL/I routines that process arrays, because they are more likely to access

data starting with element 1. However, remember that the PL/I subscript and the C subscript to access the same element differ by 1.

Structure Arguments

Because of the complexity of PL/I structure descriptors, passing a structure from C to PL/I is not supported. The most reasonable way to pass structure data from C to PL/I is to declare a PL/I POINTER argument, declare the structure as BASED on the pointer, and then pass the address of the C structure as described in **Pointer Arguments** earlier in this chapter.

Function Pointer Arguments

Either a `__pli` function pointer or a standard C function pointer can be passed to a PL/I ENTRY argument.

Returning Data to C from PL/I

A PL/I function called from C must be declared in C as returning a value of an appropriate type. If you are calling a PL/I SUBROUTINE (rather than a FUNCTION, which returns a value) from C, it must be declared in C as returning `void`. The following sections describe special considerations for returning CHAR(n), CHAR(n) VARYING, and BIT(n) arguments.

Returning CHAR(n) or CHAR(n) VARYING

A PL/I function that returns CHAR(n) or CHAR(n) VARYING must be declared in C as returning the appropriate structure type.

The appropriate structure type for CHAR(n) is

```
struct {char text [n];}
```

For CHAR(n) VARYING, it is

```
struct {short len;
        char text [n];}
```

Returning BIT(n)

You cannot call a PL/I function returning a BIT string from C. Modify such functions to store the return value using a BIT argument, and use the `_BIT` macro to pass the area in which the result should be stored.

Data Type Conversion Macros

This section describes the use of data type conversion macros such as `_ARRAY`, `_BIT`, and `_STRING` with PL/I.

_ARRAY, _ARRAY2, _ARRAY3 — Pass Array Argument to PL/I

SYNOPSIS

```
#include <ilc.h>

_ARRAY(type *addr, unsigned dim);

_ARRAY2(type *addr, unsigned dim1, unsigned dim2);

_ARRAY3(type *addr, unsigned dim1, unsigned dim2,
        unsigned dim3);
```

DESCRIPTION

The **_ARRAY**, **_ARRAY2**, and **_ARRAY3** macros are used to pass array arguments to PL/I with appropriate descriptors. **_ARRAY** builds a descriptor for a one-dimensional array, **_ARRAY2** for a two-dimensional array, and **_ARRAY3** for a three-dimensional array. The array should be an array of arithmetic or pointer type. String arrays should be passed using the **_STRARRAY** macro, described later in this chapter.

The first argument to these macros must be one of the following: a pointer to the appropriate type, an array of the appropriate type, or a pointer to an array of the appropriate type. For instance, if the PL/I array is declared as (*,*) FLOAT(16), the first argument can be defined in C as having the type **double *, double [n], double [m][n]**, or **double (*) [n]**. The number of dimensions in C is not required to be the same as the number of dimensions in PL/I. This facilitates the treatment of PL/I multidimensional arrays as one-dimensional in C, which may be necessary if the array bounds are not known at compile time.

The **dim** arguments specify the upper bounds for each dimension. The lower bound of each dimension is passed to PL/I as 1. For instance, the call **ARRAY2(x,5,17)** passes **x** to PL/I as an array whose PL/I dimensionality would be (1:5, 1:17).

EXAMPLE

See **Examples: Calling PL/I from C** later in this chapter.

_BIT Pass Bit Argument to PL/I

SYNOPSIS

```
#include <ilc.h>

_BIT(void *area, unsigned size);
```

DESCRIPTION

The `_BIT` macro is used to pass a C storage area to a PL/I BIT(n) or BIT(*) argument. The `str` argument should be a pointer to the first byte of the area to be passed. (It can be a string literal.) The `len` argument should be the bit string length to be passed to PL/I. (That is, a value of 17 indicates 17 bits, not 17 bytes.)

CAUTION

`_BIT` should be used only for fixed-length BIT strings.

EXAMPLE

Call a PL/I subroutine whose argument is a 32-bit string from C. The PL/I subroutine is defined as follows:

```
BITSUB: PROC(BSTR);
   DECLARE BSTR BIT(32) ALIGNED;
```

Two calls, one passing a 4-byte area and one passing a constant string of bits, are shown below:

```
__pli void bitsub();
unsigned int bitwork;
bitsub(_BIT(&bitwork, 32));
bitsub(_BIT("\x92\x49\x24\x92",32));
```

_STRARRAY Pass String Array Argument to PL/I

SYNOPSIS

```
#include <ilc.h>

_STRARRAY(struct *addr, unsigned dim);
```

DESCRIPTION

The **_STRARRAY** macro is used to pass a string array argument to PL/I with an appropriate descriptor. The PL/I array can be an array of either fixed-length or varying-length strings.

The **addr** argument must be a pointer to a C structure or an array of C structures. The structure must be defined as

```
struct {char text [n];}
```

for a fixed-length string array or as

```
struct {short len; char text [n];}
```

for a varying-length string array. (n is the maximum length of each array element.)

The **dim** argument specifies the upper bounds for the array. The lower bound is passed to PL/I as 1. For instance, the call **_STRARRAY(x,100)** passes **x** to PL/I as an array whose PL/I declaration would be (1:100) CHAR(n) or (1:100) CHAR(n) VARYING.

EXAMPLE

Call a PL/I routine that reads data into an array of varying-length strings from C. The PL/I routine is defined as follows:

```
BLDTABL: PROC(TABLE);
   DCL TABLE(*) CHAR(40) VARYING;
```

The array is declared in C as an array of varying string structures:

```
__pli void bldtabl();
typedef struct {
   short len;
   char text [40];
} vstr40;
static vstr40 table [500];

bldtabl(_STRARRAY(table, 500));
```

_STRING Pass String Argument to PL/I

SYNOPSIS

```
#include <ilc.h>

_STRING(char *str, unsigned len);
```

DESCRIPTION

The **_STRING** macro is used to pass a **char *** or **char[]** value to a PL/I CHAR(n) or CHAR(*) argument. The **str** argument should be a pointer to the first byte of the string to be passed. (It can be a string literal.) The **len** argument should be the string length to be passed to PL/I. If **len** is specified as 0, the string length is determined at execution time by invoking **strlen**.

CAUTIONS

_STRING should be used for PL/I fixed-length arguments. For a varying-length argument, the corresponding C argument should be a structure, and no macro should be used.

EXAMPLE

See **Examples: Calling PL/I from C** later in this chapter.

Examples: Calling PL/I from C

Assume the following PL/I PROCEDURE definitions:

```
PSUB: PROC(IS, X, STR) /* no value returned */;
   DECLARE IS FIXED BIN(15);
   DECLARE X FLOAT(6);
   DECLARE STR CHAR(*);

PFUND: PROC(VEC, MAT) RETURNS(FLOAT(16));
   DECLARE VEC (*) FLOAT(16);
   DECLARE MAT (*,*) FLOAT(16);

PFUNS: PROC(LOC) RETURNS(CHAR(20) VARYING);
   DECLARE LOC POINTER;
   DECLARE 1 DATA BASED(LOC),
      2 NUMBER FIXED BIN(31),
      2 NAME CHAR(20) VARYING;
```

The following calls from C to PL/I are correct:

```
typedef struct {
   short len;
   char text [20];
} vstr20;

__pli void psub();
__pli double pfund();
__pli vstr20 pfuns();
short sh;
float x;
char *str;
double vec [100];
unsigned nvec;
double *mat;
vstr20 vmsg;

struct {
   char text [13];
} message = {"Hello, world!" };

struct plidata {
   int number;
   vstr20 name;
};
struct plidata namelist [100], *datap;

psub(sh, x, _STRING(str, 15));
psub((short) nvec, 54.4F, message);
mat = calloc(nvec * nvec, sizeof(double));
printf("pfund result is %g\n",
   pfund(_ARRAY(vec, nvec), _ARRAY2(mat, nvec, nvec)));
vmsg = pfuns(@(datap+30));
```

The following calls are incorrect and will probably cause errors during execution, for the reasons explained in the comments that follow the call:

```
psub(sh, x, str);
     /* ERROR: The third argument requires a string length */
x = pfund(vec, mat);
     /* ERROR: The _ARRAY macros must be used to pass arrays
               to PL/I */
vmsg = pfuns(&namelist [30]);
     /* ERROR: Pointers must be explicitly passed by reference */
```

Error Handling Considerations

After most kinds of execution error, the PL/I library terminates the PL/I framework unless the PL/I program provides an ERROR ON unit to recover. When both C and PL/I are in use and the PL/I framework is terminated due to errors (or due to use of the STOP or EXIT statement), the C framework is terminated as well. The C library generates a message in this case to explain the reason for termination.

External Data Sharing Considerations

When you use the **NORENT** compiler option, you can share data between PL/I and C using STATIC EXTERNAL variables in PL/I. To share data in this way, declare an **extern** C structure whose name is the same as the PL/I structure, with each element of the structure defined to have the same data format as the corresponding PL/I element. (Note that you may have to introduce padding elements into the structure definitions to resolve differences between the ways that C and PL/I map structures.) For example, the following C and PL/I structures allow access to the same data:

```
DECLARE 1 SHARED STATIC EXTERNAL,
          2 NEXT POINTER,
          2 CUST(20),
            3 NUMBER FIXED BIN(31),
            3 NAME CHAR(20);

extern struct shared_t {
   struct shared_t *next;
   struct {
      int number;
      char name [20];
   } cust [20];
} shared;
```

Note that a shared variable must be declared but not defined in C because the PL/I declarations constitute the definition.

Also note that the structure name should be seven characters or less because PL/I does not allow eight-character external names. In addition, be aware that the SAS/C compiler translates the underscore (_) character in internal names to a pound sign (#). For this reason, a shared C external variable with an underscore in its name must be declared in PL/I with a pound sign in place of the underscore.

Linking Considerations

Pseudoregister Removal

If you compile with the C `RENT` or `RENTExt` option, the C code should always be processed by CLINK before linking it with the PL/I routines, and the CLINK PREM option should be used to remove the C pseudoregisters. If CLINK is not used, both languages share the same pseudoregister definitions. If there are only a few small C external items, this causes no problems, but, usually, C has over 4K of external data. This prevents successful creation of the PL/I framework because PL/I does not support a total pseudoregister length greater than 4K.

Restrictions

Be aware of the following PL/I restrictions when you mix PL/I and C:

- You cannot use PL/I multitasking.
- Most PL/I programs use the FILE SYSPRINT, which, under OS, is normally associated with the DDname SYSPRINT. Under OS-batch, the C file `stdout` also is normally directed to the SYSPRINT DDname. Such simultaneous use may lead to garbled or lost output. You can avoid such conflicts by using output redirection in C to change the definition of `stdout` or by reopening SYSPRINT in PL/I using the TITLE option. Note that PL/I sends library diagnostics to SYSPRINT, so this can be a problem even for PL/I code that does not reference SYSPRINT directly.
- You can use PL/I interlanguage communication to FORTRAN or COBOL in combination with SAS/C communication between PL/I and C. However, in this case, you cannot communicate with the same language using both PL/I and C. For instance, a program structure in which C calls a PL/I routine that calls a COBOL routine that calls C is not supported.
- You cannot handle attention interrupts in both C and PL/I. You can either define a `SIGINT` handler in your C code or compile your PL/I with the INTERRUPT option and define an ATTENTION ON-unit, but the effects of doing both are unpredictable. Note that joint handling of program checks (C `SIGFPE`, `SIGSEGV`, and `SIGILL`, and PL/I OVERFLOW, UNDERFLOW, and ZERODIVIDE) is supported. Each language handles only those program checks that occur in its own code.
- You should be careful about using GOTO in ON-units in PL/I. If a GOTO terminates a C routine, this is not immediately detected and results in a confusing ABEND later in execution. Similarly, avoid the use in C of `longjmp` where its use may terminate a called PL/I routine.

Hints

Debugging

You can debug with both the SAS/C source-level debugger and the PL/I Checkout Compiler or the OS PL/I PLITEST debugger if you are unsure which language is responsible for an error. However, you should be aware of the following special considerations:

- In order to use the PLITEST debugger in full-screen mode, PLITEST must be invoked from an ISPF panel. The full-screen debugging mode requires ISPF information that is not available when the PL/I framework is created by **mkfmwk**. For this reason, you can use PLITEST in line mode only when C is the first language. If you want to use PLITEST in full-screen mode, you must define the main program of your application to be in PL/I and create the C framework using a call to **CFMWK**.
- If you are using the Checkout Compiler or if some of your PL/I code was compiled with the INTERRUPT option, both the PL/I debugger and the C debugger will attempt to handle attention interrupts. If the attention key is hit, it is unpredictable whether the interrupt will be processed by the currently active language. It is best to avoid the use of the INTERRUPT option in PL/I if you will be using the C debugger.
- When you use the PLITEST debugger in full-screen mode, it executes a CLIST named AQAINM3A to begin full-screen execution. This CLIST executes a CONTROL NOMSG CLIST statement before transferring control to the program to be debugged. The CONTROL NOMSG causes some messages from the C debugger to be suppressed. To avoid this problem, insert a "CONTROL MSG" command in the CLIST before the ISPEXEC SELECT command that starts up the program. This should not have any adverse effects on the program being debugged or on the PL/I debugger.
- By default, the PL/I Checkout Compiler allocates all available storage for its own use, which probably will not leave enough memory free for the C program or debugger to execute. You should use the Checkout Compiler SIZE option to leave at least 768K free for use of the C debugger.
- When you use the Checkout Compiler with SAS/C ILC, you must always specify the Checkout Compiler option COMPATIBLE so that PL/I pointers are represented as 4-byte rather than 16-byte values.

Calling C Functions That Return a Value

The restriction against calling C functions of a type other than **void** from PL/I is due to the PL/I requirement that OPTIONS(ASM) routines be called as SUBROUTINEs rather than as FUNCTIONs. In certain cases, OPTIONS(ASM) can be omitted from the PL/I declaration of a C function, and the C function can be called as a value-returning function. The requirements for this to be successful are as follows:

- The data type returned by the C function must be an arithmetic or pointer type. It cannot be a string type or any type for which PL/I requires a descriptor.
- All arguments to the C function must have arithmetic or pointer type.

- Alternately, other kinds of PL/I data, such as strings or arrays, can be passed to the C function. However, in this case, the C arguments must be declared as pointers to the PL/I descriptors, and the descriptors must be processed by the called C program. PL/I descriptor formats are discussed in the IBM manuals *PL/I Optimizing Compiler Execution Logic* and *OS PL/I Version 2 Problem Determination.*

7 Communication with Pascal

Introduction

This chapter provides the technical details for using SAS/C ILC with Pascal. The topics covered are

- the versions of Pascal supported by SAS/C ILC
- Pascal data types and their corresponding types in C
- passing data from Pascal to C and vice versa
- returning data to Pascal from C and vice versa
- data type conversion macros
- external data sharing considerations
- restrictions on using the SAS/C Pascal interface
- hints on declaring Pascal routines and processing data in C.

Many sections include examples of correct and incorrect calls to each language. These examples, plus the accompanying discussion, provide the necessary background to write Pascal-C applications. Before reading this chapter, you should be familiar with the material in Chapters 1 through 3.

Versions Supported

SAS/C ILC supports communication with programs compiled with Pascal/VS Release 2.2. Routines compiled with earlier versions of Release 2 should also work correctly.

Routines compiled with other versions of Pascal may work properly if the L$IPASM and L$IPASQ routines provided in source are recompiled with the proper version of the Pascal compiler, for instance, VS Pascal.

Pascal Data Types

Table 7.1 lists the common Pascal data types and their C equivalents. You should consult this table for general information about corresponding data types and to determine how to declare variables shared between the two languages.

Table 7.1
Pascal-C Corresponding Data Types

Pascal Type	C Type
CHAR	`char`
PACKED −128..127	`signed char`
PACKED 0..255	`char`
BOOLEAN	`char`
INTEGER	`int, long`
PACKED −32768..32767	`short`
PACKED 0..65535	`unsigned short`
REAL	`double`
SHORTREAL	`float`
ARRAY [1..n] OF type	`type [n]`

ARRAY [1..n] OF CHAR	`char [n],`
	`char *,`
	`struct {`
	`  char text [n];}`

STRING(n)	`struct {`
	`  short len;`
	`  char text [n];}`

enumerated scalar	
<=256 values	`char`
>256 values	`unsigned short`
SET OF type	no equivalent
@var	`type *, void *`
RECORD	`struct`

Data types omitted from the table have no close equivalent in the other language. See Chapter 3, "Communication with Other Languages," for general information on data formats and data sharing.

PACKED Arguments

The Pascal keyword PACKED is used to specify that an aggregate data item is to be stored in the minimum number of bytes possible, without regard to the normal boundary requirements for the individual data elements. The Pascal alignment rules for unPACKED data are equivalent to those for C.

For example, an unPACKED RECORD is aligned on the boundary required by the largest data element within the RECORD, and each element is aligned as shown in the *Pascal/VS Programmer's Guide*. In a PACKED RECORD, the elements are aligned on byte boundaries, as is the RECORD itself.

To share PACKED aggregate data, you must be sure that the corresponding C aggregate is properly aligned. You may be able to use the C compiler option `BYtealign` to align data elements within structures on byte boundaries. In addition, you can use the `__alignmem` and `__noalignmem` keywords when defining a structure tag to define how structure members should be aligned. These keywords are described in detail in Chapter 3.

Note that in some cases alignment differences have no effect, such as when passing a PACKED ARRAY OF REAL. Whether or not an ARRAY passed to C is PACKED has no effect, unless you are passing a PACKED ARRAY OF RECORDS. In this case, you need to ensure proper alignment in C as described in the previous paragraph. There is no C equivalent for a PACKED ARRAY OF unPACKED RECORDs.

Passing Data to C from Pascal

When you write a C function that will be called from Pascal, you must do the following:

1. Compile the C function with the `INDep` option, which is always required for a C function called from another high-level language.
2. Declare the C function to Pascal as EXTERNAL.
3. Validate each argument as follows:
   - Determine the C data type that corresponds to the Pascal data type using **Table 7.2.**
   - Determine whether you are using pass by reference or pass by value. Arguments declared as VAR or CONST in Pascal are passed by reference. If you are using pass by reference for an argument, declare it in C as a pointer to the appropriate C data type. It is recommended that you use pass by reference to avoid the argument list alignment problems that result from using pass by value. The C alignment rules are described in detail in Chapter 3.
   - Use pass by value for arguments that are not declared as VAR or CONST to Pascal. If you are using pass by value for an argument, declare it to be of the appropriate C data type. Note that pass by value is not supported for some data types. This is shown in **Table 7.2** by an N/A entry. For some types, such as RECORDs, pass by value is not supported by Pascal. For other types, such as SHORTREAL, pass by value to C is

impossible because Pascal's conventions for argument alignment are different from the C language's.

- Check that all elements in aggregate data items have the same alignment in both languages.

Table 7.2 shows the C argument declarations that should be coded for the Pascal data types when using either pass by value or pass by reference. For example, a Pascal REAL variable would be declared in C as `double` if passed by value or `double *` if passed by reference. A Yes in the Special Considerations column indicates that additional information on passing values of this type is available in the sections following the table.

Table 7.2 *Argument Types for Calls from Pascal to C*

Pascal Type	C Type		Special Considerations
	Pass by Value	Pass by Reference	
CHAR	N/A	`char *`	
PACKED −128..127	N/A	`signed char *`	
PACKED 0..255	N/A	`char *`	
BOOLEAN	N/A	`char *`	
INTEGER	`int, long`	`int *, long *`	
PACKED −32768..32767	N/A	`short *`	
PACKED 0..65535	N/A	`unsigned short *`	
REAL	`double`	`double *`	
SHORTREAL	N/A	`float *`	
ARRAY [1..n] OF type	N/A	`type [n],` `type *`	Yes

ARRAY [1..n] OF CHAR	N/A	`char [n],`
		`char *,`
		`struct {`
		`  char text [n];}*`	

STRING(n)	N/A	`struct {`
		`  short len;`
		`  char text [n];}*`
			Yes

(*continued*)

Table 7.2 *(continued)*

Pascal Type	C Type: Pass by Value	C Type: Pass by Reference	Special Considerations
SET OF type	see below	`char *`	Yes
@ var	`type *, void *`	`type **, void **`	
RECORD	N/A	`struct *`	Yes
FUNCTION/ PROCEDURE	`struct { __pascal type (*f)(); void *display [6]; } pascal_fp;`	N/A	Yes

In the following discussions, aggregate data items are assumed to be unPACKED. If you are working with PACKED data, refer to **PACKED Arguments** earlier in this chapter.

ARRAY Arguments

Pascal does not support pass by value for ARRAY arguments. An ARRAY passed from Pascal can be declared in the called C function as either an array or a pointer. The C array element type or pointed-to object should have the same data format as the type of the Pascal element type. For example, an ARRAY [1..10] OF REAL passed from Pascal could be defined in C as either `double [10]` or `double *`.

The first element of an ARRAY passed from Pascal should be addressed in C using subscript 0.

STRING Arguments

Pascal does not support pass by value for the STRING(n) type. A C argument corresponding to a Pascal STRING(n) should be declared as

```
struct {
   short len;
   char text [n];}
```

The value of the `len` field is the current string length. The characters of `text` after the current length are undefined. If the called C function requires information about the maximum string length, you will need to pass this as an additional argument.

When you declare a C function in Pascal as having a CONST STRING argument, the argument is passed as if it were declared STRING(65535). The corresponding C argument should be declared as

```
struct {
   short len;
   char text [65535];}*
```

A C function cannot be declared in Pascal as having a VAR STRING argument unless a maximum length is specified explicitly. That is, you can use VAR STRING(n), but not VAR STRING.

SET Arguments

Pass by reference is the best way to pass a Pascal SET argument. The argument can then be declared in C as `char *`.

If you must pass a SET argument by value, you need to determine how your argument is stored in Pascal. The Pascal rules for SET formats are complicated and differ based on whether or not your argument is PACKED. For example, given the following Pascal declarations

```
A : PACKED SET OF 0..31;
B : SET OF 0..31;
```

variable A will occupy 32 bits (4 bytes) and variable B will occupy 256 bits (32 bytes). For A, the variable should be declared in C as a `char [4]` array. For B, the variable should be declared in C as a `char [32]` array. Consult the *Pascal/VS Programmer's Guide* to determine the storage requirements for your argument in order to code the corresponding C variable. See **Interpreting Set Data in C** later in this chapter for an example of accessing a Pascal SET in C.

RECORD Arguments

Because a Pascal RECORD is always passed by reference, you should declare the argument to be a pointer to an equivalent C structure. Each member of the C structure must have a data format that matches the corresponding Pascal field as defined in **Table 7.1.** Even though Pascal and C have equivalent boundary alignment rules, it is recommended that you verify that the RECORD maps correctly onto the C structure. If your RECORD is PACKED, refer to **PACKED Arguments** earlier in the chapter.

FUNCTION/ PROCEDURE Arguments

Pascal FUNCTION/PROCEDURE arguments are 28 bytes long and consist of a 4-byte area containing the function address followed by 24 bytes of control information. Function pointers passed from Pascal must be declared in C as

```
struct {
   __pascal type (*f)();
   void *display [6];
   } pascal_fp;
```

To call the passed FUNCTION/PROCEDURE in C, you code

```
(*pascal_fp.f)()
```

Note that a `__pascal` function pointer can be used only when it is an element of a structure, as shown above, because the display information is an integral part of the function pointer. For instance, both of the following assignments are erroneous:

```
__pascal int (*wrong)();
__pascal int func;

wrong = &func;          /* will be rejected by the compiler */
```

```
wrong = pascal_fp.f;    /* any use of wrong after this statement */
                        /* will have unpredictable results */
```

Returning Data to Pascal from C

You can call a C function from Pascal as a FUNCTION and expect a return value, as long as there is a data type equivalence listed in **Table 7.1** and C supports returning the data type. The C function should be declared as returning the equivalent type expected by Pascal.

Although C does not support returning arrays, if Pascal expects a (PACKED) ARRAY OF CHAR return value, you can return a fixed-length string structure from C as shown in **Table 7.1.** Similarly, if Pascal expects a STRING(n) return value, you can return the varying-length string structure shown in **Table 7.1.**

If you are calling a C function as a PROCEDURE, the C function must be defined as returning `void`.

Examples: Calling C from Pascal

Assume the following C declarations:

```
float cfun1 (double arrvar [8], unsigned short size,
      int *count, double **ptr1, double *ptr2)

   typedef struct {
      short len;
      char text [20];
   } vstr20;

   typedef struct {
      char text [20];
    } fxstr20;

void csub2 (char *str, vstr20 *name, char *state)

fxstr20 cfun3()
```

and the following Pascal declarations:

```
CONST
   ARRSIZE = 8;
   SETSIZE = 4;
   VSIZE = 20;

TYPE
   RARRAY = ARRAY [1..ARRSIZE] OF REAL;
   RPTR = @ REAL;
   CARRAY = PACKED ARRAY [1..ARRSIZE] OF CHAR;
   STATE = (TX, AL, NY, CA);
   STSET = SET OF STATE;
   USHORT = PACKED 0..65535;
   VSTR = STRING(VSIZE);
```

```
VAR
   COUNT : INTEGER;
   SIZE : PACKED 0..65535;
   RARR : RARRAY;
   CARR : CARRAY;
   SHREAL : SHORTREAL;
   PTR1 : RPTR;
   PTR2 : RPTR;
   BIGST : STSET;
```

The following Pascal declarations of and calls to C functions are correct:

```
FUNCTION CFUN1(VAR RARR : RARRAY;
               CONST SIZE : USHORT;
                     COUNT : INTEGER;
                 VAR PTR1 : RPTR;
                     PTR2 : RPTR) : SHORTREAL;

PROCEDURE CSUB2(VAR CARR : CARRAY;
                    NAME : VSTR;
                VAR BIGST : STSET);

FUNCTION CFUN3 : CARRAY;

   SHREAL := CSUB1(RVAR, SIZE, COUNT, PTR1, PTR2);
   CSUB2(CARR, NAME, BIGST);
   CARR := CSUB3;
```

Because Pascal is very strongly typed, a call to a C function from Pascal will be correct if the C function is declared correctly. The following function declarations are incorrect and will likely cause execution errors if the functions are called, for the reasons explained in the comments that follow the call:

```
FUNCTION CSUB1(VAR RARR : RARRAY;
                   SIZE : USHORT;
                  COUNT : INTEGER;
               VAR PTR1 : RPTR;
                   PTR2 : RPTR) : SHORTREAL;
      (* ERROR: The SIZE argument should be declared VAR or CONST
                since the C function expects pass by reference. *)

PROCEDURE CSUB2(VAR CARR : STRING(8);
                    NAME : VSTR;
                VAR BIGST : STSET);
      (* ERROR: The CARR argument cannot be declared STRING(n)
                since the C function expects a character pointer
                rather than a string structure. *)
```

Passing Data to Pascal from C

To call a Pascal routine from C, you must do the following:

1. Declare the Pascal routine in C using the `__pascal` keyword to indicate the routine is written in Pascal.
2. Ensure that each argument is passed correctly to Pascal by using **Table 7.3.** For example, the C argument corresponding to a Pascal PACKED 0..65535 variable should be declared as `unsigned short`. In **Table 7.3**, a Yes in the Special Considerations column indicates that additional information is available in the sections following the table. Below are details on passing arguments to Pascal:

   - The compiler uses pass by value by default for calls to routines declared with the `__pascal` keyword. If you want to use pass by value and the argument type does not have an entry in the Special Considerations column, you can pass the argument directly.

     Note that Pascal does not support pass by value for the data types RECORD, ARRAY, and STRING(n). For the corresponding C types, the compiler always uses pass by reference.
   - If you want to pass an argument by reference and there are no special considerations for the data type, you can preface the argument with the ampersand (`&`) or at (`@`) operator, or pass a pointer to the appropriate data type.
   - Some arguments should be passed using macros to convert the argument to the required Pascal data type. SET arguments should be passed by value using the `_SET` macro. The `_STRING` macro can be used to pass a character array or string literal as a (PACKED) ARRAY OF CHAR. In **Table 7.3**, the Macro column indicates if there is a macro available or required to pass an argument of the type.

Table 7.3 *Argument Types for Calls from C to Pascal*

Pascal Type	C Type	Special Considerations	Macro
CHAR	`char`	Yes	
PACKED −128..127	`signed char`		
PACKED 0..255	`char`		
BOOLEAN	`char`		
INTEGER	`int, long`		
PACKED −32768..32767	`short`		
PACKED 0..65535	`unsigned short`		

(*continued*)

Table 7.3 *(continued)*

Pascal Type	C Type	Special Considerations	Macro
REAL	`double`		
SHORTREAL	`float`		
ARRAY [1..n]	`type [n],`		
OF type	`type *`		
ARRAY [1..n]	`char [n],`	Yes	`_STRING`
OF CHAR	`char *,`		
	`struct {`		
	`char text [n];}`		
STRING(n)	`struct {`	Yes	
	`short len;`		
	`char text [n];}`		
SET OF	see below	Yes	`_SET`
type			
@var	`type *, void *`	Yes	
RECORD	`struct`	Yes	
FUNCTION/	`__pascal type (*)(),`	Yes	
PROCEDURE	`type (*)()`		

Aggregate data items are assumed to be unPACKED. If you are working with PACKED data, refer to **PACKED Arguments** earlier in this chapter.

Character Literal Arguments

The C language definition specifies that a character constant, such as `'a'`, has type `int` rather than `char`. A character constant argument is passed correctly to Pascal, assuming the Pascal declaration is INTEGER. To pass the argument as a Pascal CHAR, use a cast such as `(char) 'a'`.

ARRAY Arguments

Pascal does not support pass by value for ARRAYs. An array passed to Pascal can be declared either as an array or a pointer in the calling C function. The type of the array element or pointed-to object should have the same data format as the Pascal element type. For instance, either a C `float *` or `float [n]` could be passed to a Pascal routine that expects an ARRAY [n] OF SHORTREAL argument.

Remember that arrays begin with element 0 in C.

STRING Arguments

Pascal does not support pass by value for STRING(n) arguments. To pass a STRING(n) argument, you can use a variable-length string structure as shown in **Table 7.3.** Because arguments of type CONST STRING are treated by Pascal as STRING(65535), you can also use a variable-length string structure for an argument declared in Pascal as CONST STRING.

To treat all string literal arguments as Pascal CONST STRING, use the `VString` compiler option as described in Chapter 3. If you need to pass a string literal of fixed length (such as to a Pascal (PACKED) ARRAY of CHAR), you can use the `_STRING` macro.

If you do not use the `VString` compiler option, string literals are passed as (PACKED) ARRAY [n] OF CHAR.

SAS/C ILC does not support passing Pascal VAR STRING arguments.

RECORD Arguments

Pascal does not support pass by value for the RECORD data type. To pass a C structure that will be processed as a Pascal RECORD, you should define corresponding elements of the structure/RECORD to have equivalent data types. The structure will be passed by reference.

In addition, while alignment rules for unPACKED records correspond between the languages, you should check the alignment of the individual elements. You can use the C compiler option `BYtealign` or the keyword `__noalignmem` to suppress alignment in a C structure. This may be useful when working with PACKED records.

SET Arguments

To pass a C storage area as a Pascal SET, you must first determine the physical size of the SET in order to define a corresponding object in C. For instance, a Pascal SET that occupies 32 bits could be defined as an `unsigned int` or `char [4]` in C. Consult the *Pascal/VS Programmer's Guide* for information on this topic.

If you want to use pass by reference, pass the address of the variable using the ampersand (`&`) or at (`@`) operator, or the array name if the C variable is an array. If you want to use pass by value, use the `_SET` macro described in **Data Type Conversions Macros.** See **Interpreting Set Data in C** for an example of accessing a Pascal SET in C.

Pointer Arguments

If you use pass by reference, use the ampersand (`&`) or at (`@`) operator to pass a C pointer to a Pascal pointer (`@type`) argument. If you use pass by value, you can pass the pointer. For example, given the C declaration

```
int *intptr;
```

the following statement passes the *address* of `intptr` to Pascal:

```
pasfunc (&intptr)
```

and this statement passes the *value* of `intptr`:

```
pasfunc (intptr)
```

The former is appropriate for a VAR or CONST pointer; the latter is appropriate for a pass-by-value pointer.

FUNCTION/ PROCEDURE Arguments

When you a call a Pascal routine with a FUNCTION/PROCEDURE argument, the corresponding C argument can be a C function name or function pointer, a Pascal function name, or a `__pascal` function pointer previously passed to C from Pascal. Here are some examples of correct calls to a Pascal PROCEDURE with a PROCEDURE argument:

```
PROCEDURE PASFUN(PROCEDURE F);

void csub(pascal_fp)
   struct {
      __pascal type (*f)();
      void *display [6];
      } pascal_fp;
{
   __pascal void pasfun();
   __pascal void passub();
   void cfun();
   void (*cfunptr)();

   pasfun(&cfun);
   pasfun(cfunptr)
   pasfun(passub);
   pasfun(pascal_fp.f);
}
```

Returning Data to C from Pascal

A Pascal PROCEDURE must be declared in C as returning `void`. A Pascal FUNCTION must be declared in C as returning a value of the C type corresponding to its Pascal return type. For example, a Pascal FUNCTION declared as

```
FUNCTION X(Y: REAL) : REAL;
```

must be declared in C as

```
__pascal double x();
```

The following special considerations apply:

- A Pascal FUNCTION that returns a STRING(n) value must be declared in C as returning a value of type

```
struct {
   short len;
   char text [n];
   }
```

- A Pascal FUNCTION that returns a fixed-length string ((PACKED) ARRAY [n] OF CHAR) must be declared in C as returning a value of type

```
struct {
   char text [n];
   }
```

- A Pascal FUNCTION that returns a SET OF type variable must be modified to store the return value using a SET argument. The SET should be passed by reference.

Data Type Conversion Macros

This section describes the use of two data type conversion macros, **_SET** and **_STRING**, with Pascal.

_SET Pass SET Argument to Pascal

SYNOPSIS

```
#include <ilc.h>

_SET(void *area, unsigned size);
```

DESCRIPTION

The **_SET** macro is used to pass a C storage area by value to a Pascal SET argument. The **str** argument should be a pointer to the first byte of the area to be passed or a string literal. The **len** argument should be the number of bits to be passed to Pascal. (That is, a value of 17 indicates 17 bits, not 17 bytes.) There is 1 bit allocated for each possible element of the SET. A bit value of 1 indicates that the element is in the SET. Depending upon whether or not the SET is PACKED and the type of the SET elements, there may be additional bits allocated that can never be set.

CAUTION

The method for determining the mapping of a Pascal SET is complex. Refer to the *Pascal/VS Programmer's Guide* for information on this topic.

EXAMPLE

Call a Pascal procedure that takes a halfword bit string and a 256-byte string as pass-by-value arguments. The Pascal PROCEDURE is defined as follows:

```
TYPE SET1 = PACKED SET OF 0..9;  (* 10 bits in C *)
TYPE SET2 = SET OF 0..9;         (* 256 bits in C *)

PROCEDURE SETSUB(S1 : SET1;
                 S2 : SET2);
```

The following sample C code is used to call SETSUB:

```
__pascal void setsub();
short s1 = 0;
char s2 [32];

/* set bits 0-3 in PACKED SET */
s1 |= 0xf000;

memset(s2, 0x00, 32);
/* set bits 0 and 8 in unPACKED SET */
s2 [0]= s2 [1]= 0x80;

setsub(_SET(&s1, 10), _SET(s2, 256));
```

Note the use of **&s1** rather than **s1** above. The first argument to **_SET** is the address of the set, not the contents of the set.

_STRING Pass Fixed-Length String Argument to Pascal

SYNOPSIS

```
#include <ilc.h>

_STRING(char *str, unsigned len);
```

DESCRIPTION

The `_STRING` macro is used to pass a `char *` or `char []` value to a Pascal (PACKED) ARRAY OF CHAR [`len`]. The `str` argument should be a pointer to the first byte of the string to be passed. (It can be a string literal.) The `len` argument should be the string length to be passed to Pascal. If `len` is specified as 0, the string length is determined at execution time by invoking `strlen`. The string is passed by reference.

CAUTIONS

`_STRING` should be used only for (PACKED) ARRAY OF CHAR arguments, which is not the normal Pascal/VS type for string processing. For an argument that is processed as a varying-length string in Pascal (CONST or VAR STRING(n)), the corresponding C argument should be a structure or string literal, and no macro should be used.

The C compiler option `VString` should be used to pass string literals to Pascal as STRING(n) arguments.

EXAMPLE

See **Examples: Calling Pascal from C**, below.

Examples: Calling Pascal from C

Assume the following Pascal declarations:

```
CONST
   VECSIZE = 20;
   HEADSIZE = 16;
   VSIZE = 20;

TYPE
   CSHORT = PACKED -32768..32767;
   VECTOR = ARRAY [1..VECSIZE] OF REAL;
   MATRIX = ARRAY [1..VECSIZE, 1..VECSIZE] OF REAL;
   VSTR = STRING(20);
   HEADING = PACKED ARRAY [1..HEADSIZE] OF CHAR;
   DATA = record
      NUMBER: INTEGER;
      NAME:   VSTR;
      end;
   DATAPTR = @ DATA;
```

```
PROCEDURE PASUB(     ISH: CSHORT;
                VAR RSH: SHORTREAL;
              CONST MSG: STRING);
    EXTERNAL;

FUNCTION PAFUND(VAR VEC: VECTOR;
                VAR MAT: MATRIX;
                    HEAD: HEADING) : REAL;
    EXTERNAL;

FUNCTION PAFUNS( DPTR: DATAPTR ) : VSTR;
    EXTERNAL;
```

The following calls from C to Pascal are correct if compiled using the `VString` compiler option:

```
#include <ilc.h>

typedef struct {
   short len;
   char text(|20|);
   } vstr20;

__pascal void pasub();
__pascal double pafund();
__pascal vstr20 pafuns();
short sh;
float fl;
double v [20], m [20][20];
double val;
vstr20 vmsg;
struct pasrec {
   int number;
   vstr20 name;
} first_record;
struct pasrec *last_record;

pasub(sh, &fl, "string of any length");
     /* This example requires the VString compiler option */
val = pafund(v, m, _STRING("length 16 string", 16));
vmsg = pafuns(&first_record);
```

The following calls are incorrect and will probably cause errors during execution for the reasons explained in the comments that follow the call:

```
pasub((short) 0x0240, 11.32f, first_record.name);
     /* ERROR: The second argument must be passed by reference, not
               by value */
```

```
val = pafund(v, m, "length 16 string");
     /* ERROR: The third argument must either be a structure or a
               _STRING call.  This example would be correct if
               the VString compiler option was not used. */

vmsg = pafuns(&last_record);
     /* ERROR: The third argument must be passed by value, not
               by reference */
```

External Data Sharing Considerations

When you use the SAS/C **NORENT** compiler option, you can share data between Pascal and C using REF/DEF variables in Pascal and **extern** variables in C. Because both C and Pascal allow external variables to be either defined or declared, you can define the data in either language and declare it in the other. For example, if you use a DEF variable in Pascal, then you should declare but not define the variable in C.

If aggregate data are shared, you must be sure that elements are properly aligned, as described previously.

Because of 370 linkage editor limitations, external variable names must be eight characters or less. In addition, when you are naming Pascal routines, be aware that the C compiler translates underscores (_) in external names to pound signs (#).

Note that global automatic variables of the main Pascal routine cannot be shared with C functions.

Restrictions

You should be aware of the following restrictions when you mix C and Pascal:

- You can use the Pascal interlanguage communication to FORTRAN, COBOL, or PL/I when you use SAS/C ILC. However, in this case, you cannot communicate with the same language from both Pascal and C. For example, if you have a program where Pascal calls COBOL using the Pascal interlanguage communication support, your C code cannot call COBOL. Nor, in this case, can COBOL call C.
- Joint handling of program checks is supported. Each language handles only those program checks that occur in its own code. For example, you can code a **SIGFPE** handler in C and an ONERROR PROCEDURE for fixed-point divide by zero exception in Pascal. Each routine receives control when appropriate.
- When you use the Pascal debugger, you must use the SAS/C **=multitask** run-time option, as described in **Error Handling** in Chapter 2, "Multilanguage Framework Management." This is because the Pascal debugger uses an unusual style of error processing that cannot be handled as efficiently by the SAS/C library as the styles implemented by other languages.

Hints

Declaring a Called Pascal Routine

If Pascal is not the first language called, no Pascal routine can be declared as MAIN or REENTRANT, or as a PROGRAM. This is because the SAS/C library routine responsible for establishing the Pascal framework (L$IPASM) is a Pascal PROGRAM. The EXTERNAL directive should be used for Pascal routines called from C.

Interpreting SET Data in C

A Pascal SET is represented as a series of bits. There is 1 bit for each possible element in the set, which is set to 1 if the element is in the set. Depending upon whether or not the SET is PACKED and the type of the SET elements, there may be additional bits allocated that can never be set.

The following example is a simple Pascal program that initializes and passes a SET of integers to a C function that prints each element of the set:

```
PROGRAM PASSET;

   CONST
      SETSIZE = 32;

   TYPE
      STYPE = SET OF 0..SETSIZE-1;

   PROCEDURE PRSET(VAR PSET : STYPE;
                       MAXELEM : INTEGER);
      EXTERNAL;


   PROCEDURE CFMWK(CONST LANG : STRING;
                   CONST RTOPT : STRING;
                   CONST DMOPT : INTEGER;
                     VAR FMWK : INTEGER);
      EXTERNAL;

   PROCEDURE DCFMWK(VAR FMWK : INTEGER;
                     VAR ERRFLG : INTEGER);
      EXTERNAL;
```

```
VAR
   PSET : STYPE;
   FMWK : INTEGER;
   ERRFLG : INTEGER;

BEGIN

   PSET := [1,3,5,7,9];
   CFMWK('PASCAL.', '.', 0, FMWK);
   PRSET(PSET,SETSIZE);
   DCFMWK(FMWK, ERRFLG);
   IF ERRFLG <> 0 THEN
      RETCODE(8);
   HALT;
END.
```

Note that the set is passed by reference, while the set size (number of bits) is passed by value.

Here is a C function to print the set. The `inset` macro can be used to test whether or not a value is in any set represented as `char *` or `char [n]` whose lowest possible element value is 0.

```
#include <stdio.h>

#define inset(set, i) ((set[i/8] << (i % 8)) & 0x80)

void prset(char *pset, int maxelem)
{
   int i;

   for (i = 0; i <= maxelem; ++i) {
      if (inset(pset,i)) printf("%d ", i);
         putchar('\n');
   }
}
```

8 Linking Multilanguage Programs with the ILCLINK Utility

Introduction

The interlanguage communication feature defines a number of C library support routines that must be linked with each program. The number and names of these routines vary depending on the languages used to create the program and how the program will be executed. By automating the linking process, ILCLINK, a multilanguage linking utility, can simplify greatly the task of producing a multilanguage module.

ILCLINK reads an input file that contains a description of a multilanguage program and a sequence of commands that specify how the program should be linked. As the commands are executed, ILCLINK selects the appropriate support routines and ensures that they are included in the final executable module.

ILCLINK does not inspect or modify the program object modules or load modules. Its purpose is to invoke other utilities, such as CLINK or the linkage editor, with the desired options and input files. Because ILCLINK needs only to interpret its control statements in order to do this, using ILCLINK adds only a very small amount of overhead to the linking process. In return, ILCLINK ensures that the resulting load module includes the support routines necessary for interlanguage communication between C and other high-level languages. The ILCLINK listing file provides a detailed record of the input and utilities that were used to create the load module.

The sections that follow explain ILCLINK in detail. The topics covered include

- the input file and output files
- ILCLINK options
- language codes for directly supported and user-supported languages
- ILCLINK processes
- control statement format and usage
- usage considerations
- use of ILCLINK under TSO, CMS, and OS-batch
- examples of linking multilanguage programs with ILCLINK
- interlanguage support routines
- default data set allocations under TSO.

This chapter assumes some knowledge of the utilities that are used to link programs under MVS and CMS, such as the linkage editor and the CMS LOAD and GENMOD commands. **References** at the end of this chapter lists further documentation for these utilities.

Input File

Input to ILCLINK is a sequence of control statements contained in an input file. The input file can have either variable-length or fixed-length records. The records can be up to 255 characters long.

Output Files

By default, ILCLINK creates at least three output files. Other output files can be created by other commands or utilities invoked by ILCLINK.

Terminal Output ILCLINK displays all of its output, including diagnostic messages and a listing of the input statements, to the terminal under TSO and CMS, and to a data set under OS-batch. This file is called the *terminal file.* Creation of a terminal file can be suppressed by the NOTERM option.

Listing A copy of the terminal output is also written to a *listing file.* Creation of a listing file may be suppressed by the NOLIST option.

Utility During its execution, ILCLINK may create a temporary utility file that contains input to CLINK or the linkage editor.

ILCLINK Options

ILCLINK accepts the options shown in **Table 8.1.** Default options are shown in **boldface.** The minimum abbreviation is shown in uppercase.

Table 8.1
ILCLINK General Options

Option	Description
Warn	Display warning messages (that is, diagnostics associated with return code 4) in the terminal and listing output files.
NOWarn	Do not display warning messages in the terminal and listing output files.
Term	Create the terminal output file.
NOTerm	Do not create the terminal output file.
LIst	Create the listing file.
NOLIst	Do not create the listing file.
Upper	Display all messages in uppercase.
NOUpper	Display messages in mixed case.
Echo	Show all operating system commands issued by ILCLINK in the terminal and listing output files.
NOEcho	Do not show operating system commands in the terminal and listing output files.

Language Codes

The FIRST and LANGUAGE control statements identify the programming languages used in the program. These statements accept a shortened form of the language name. In turn, ILCLINK associates these names with a three-character code that is used to select the appropriate ILC support routines.

For example, the three-character code for OS/VS COBOL is CB1. If a LANGUAGE control statement includes the name COBOL, then ILCLINK causes the support routine L$ICB1C to be included. Similarly, the Pascal/VS (code PAS) version of this routine is L$IPASC.

Table 8.2 shows the codes for the languages supported directly by ILCLINK.

Table 8.2
Language Codes

Language	Name Used in LANGUAGE and FIRST Statements	Code
SAS/C	C	C$$
PL/I Optimizing Compiler	PLI	PL1
OS PL/I (Version 2)	PLI2	PL2
OS/VS COBOL	COBOL	CB1
VS COBOL II (Version 2)	COBOL2	CB2
VS FORTRAN Version 1	FORTRAN	FO1
VS FORTRAN Version 2	FORTRAN2	FO2
Pascal/VS	PASCAL	PAS

In addition to the codes shown above, ILCLINK creates codes for user-supported languages by using the first three characters of an unrecognized name (that is, a name not in **Table 8.2**) as the code. For example, if a LANGUAGE control statement includes the name INTERNAL, then ILCLINK uses INT as the code for the language. If the name is less than three characters long, ILCLINK adds dollar signs ($) to the name to make a three-character code. For example, the code for language Z would be Z$$.

ILCLINK Processes

The processing performed by ILCLINK is generally controlled by groups of control statements known as *processes*. This group generally includes

1. a PROCESS control statement that specifies the utility to be invoked
2. optional AUTOCALL control statements that identify the secondary input libraries (object code libraries under OS or TEXT libraries under CMS) that are to be made available to the utility
3. optional control statement input for the utility.

Acceptable input statements vary depending on the utility invoked. The meanings of the AUTOCALL library names and how these names are used depend upon the associated utility as well as the operating system under which ILCLINK is running. The following sections explain the input statements and AUTOCALL libraries in detail.

PROCESS CLINK and PROCESS LINK Control Statements

CLINK and the linkage editor are invoked by the PROCESS CLINK and PROCESS LINK control statements, respectively. Input statements for CLINK and the linkage editor can be placed in the ILCLINK input file following the PROCESS control statement and any AUTOCALL control statements. These input statements are collected into a temporary file and passed to the utility as its SYSIN file. The input statements are translated to uppercase by ILCLINK but not otherwise modified.

For example, consider the following sequence of statements:

```
process link list,xref
autocall lc370,fortlib,mylib
   include ccode(main1,sub1)
   include fortcode(sub2,sub3)
   alias project
   name prog1(r)
```

The PROCESS control statement invokes the linkage editor with the options LIST and XREF. The AUTOCALL statement makes the libraries LC370, FORTLIB, and MYLIB available as SYSLIB input. ILCLINK collects the four subsequent statements into a temporary file that contains the following records:

```
INCLUDE CCODE(MAIN1,SUB1)
INCLUDE FORTCODE(SUB2,SUB3)
ALIAS PROJECT
NAME PROG1(R)
```

and invokes the linkage editor using this file as SYSIN input.

PROCESS LOAD Control Statement

The PROCESS LOAD control statement invokes the CMS LOAD command. For this process, the input statements are actually CMS INCLUDE commands.

For example,

```
process load main1 (clear rld
autocall lc370,fortlib,mylib
   include sub1
   include sub2 sub3
```

causes ILCLINK to issue the following sequence of CMS commands:

1. GLOBAL TXTLIB LC370 FORTLIB MYLIB
2. LOAD MAIN1 (CLEAR RLD
3. INCLUDE SUB1
4. INCLUDE SUB2 SUB3

Note: ILCLINK may add additional operands to the commands shown above, depending on the circumstances. See **PROCESS LOAD** later in this chapter.

PROCESS GENMOD Control Statement

The PROCESS GENMOD control statement invokes the CMS GENMOD command. The GENMOD process takes no additional input.

AUTOCALL Libraries

The AUTOCALL libraries named in an AUTOCALL control statement are used to provide secondary input to CLINK, the linkage editor, and the LOAD command. Other libraries can be provided via control language or operating system commands prior to invocation of ILCLINK.

TXTLIBs as AUTOCALL libraries under CMS

The names used in an AUTOCALL statement are the filenames of TEXT libraries, or TXTLIBs. During initialization, ILCLINK determines the names of the current GLOBALed TXTLIBs. Before ILCLINK invokes CLINK, the LKED command, or the LOAD command, it issues a GLOBAL TXTLIB command that contains the names of the previously GLOBALed TXTLIBs and the TXTLIBs named in the AUTOCALL statements. After the process terminates, ILCLINK issues another GLOBAL TXTLIB command to "reset" the GLOBALed TXTLIBs back to the previous state. For example, suppose LC370 TXTLIB and MYLIB TXTLIB are GLOBALed before ILCLINK is invoked. For the following process,

```
process load prog1
autocall fortlib
.
. (optional INCLUDE control statements)
.
```

ILCLINK issues the commands

```
GLOBAL TXTLIB LC370 MYLIB FORTLIB
LOAD PROG1 ...
.
. (INCLUDE commands)
.
GLOBAL TXTLIB LC370 MYLIB
```

Partitioned data sets as AUTOCALL libraries

The names used in an AUTOCALL statement are DDnames that have been allocated to object module or load module libraries.

For CLINK, ILCLINK dynamically concatenates these DDnames to SYSLIB. If SYSLIB has not been allocated, ILCLINK creates a temporary data set and allocates it to SYSLIB.

For the linkage editor, ILCLINK concatenates the DDnames to SYSLIB. If SYSLIB has not been allocated, ILCLINK concatenates the second and subsequent DDnames to the first and passes the first DDname to the linkage editor as the SYSLIB DDname.

When the utility terminates, ILCLINK deconcatenates the DDnames. ILCLINK does not change the original SYSLIB concatentation. If the SYSLIB DDname is allocated when ILCLINK is invoked, then it is returned to its original state when ILCLINK terminates.

Return Codes

ILCLINK terminates if the return code from a utility indicates that the output of the utility cannot be used as input to a subsequent process. These are the maximum allowable return codes for each utility:

- CLINK: 4
- linkage editor: 8
- last LOAD or INCLUDE command: 4
- GENMOD command: 0.

Control Statements

This section summarizes the control statements accepted by ILCLINK. Each statement is described in terms of

- the action it performs
- where it can be placed in the input file
- how it is used
- an example of its use.

Other control statements, such as linkage editor control statements or operating system commands, can also appear in the ILCLINK input file.

General Format

ILCLINK control statements are similar to linkage editor control statements. Each statement is identified by its *operation*, such as LANGUAGE or AUTOCALL. The name of the operation must appear in or after column 2 of the statement and must be followed by one or more blanks.

Control statements can be entered in either upper- or lowercase. Except for the SYSTEM statement, ILCLINK control statements cannot be continued on subsequent lines.

Format Conventions

The statement descriptions use the following typographic conventions to indicate how the statements are coded:

- The operation names are shown in **boldface.**
- Brackets [] indicate optional operands.
- Other punctuation (such as commas and parentheses) must be entered as shown.
- An ellipsis (...) indicates that the preceding operand can be repeated as necessary.
- Italic type indicates fields to be supplied by the user.

In general, each field (operation, operands, and comment) must be separated by at least one blank. Options in a comma-separated list should not have embedded blanks.

Summary of Statement Order

In general, an ILCLINK input file contains a FIRST statement, one or more LANGUAGE statements, and one or more processes.

A *process* consists of a PROCESS statement followed by one or more optional AUTOCALL statements, which are followed by control statements or commands that are to be supplied as input to the utility invoked by the PROCESS statement. A process is terminated by another PROCESS statement, a SYSTEM statement, or end of file.

The rules governing statement sequence are

- A comment can appear anywhere.
- SYSTEM statements can appear anywhere.
- LANGUAGE and FIRST statements must appear before any PROCESS statements.
- AUTOCALL statements must appear immediately following the PROCESS statement with which they are associated.

Comment Statement

Comment statements are used to document the actions specified by other control statements. Comments are displayed in the terminal and listing file but have no other effect. They can appear anywhere in the input file. The format of the comment statement is

* *comment*

Unlike other ILCLINK control statements, the * that indicates a comment must appear in column 1 of the statement. If a comment is placed among PROCESS input statements, it is not added to the process input file.

FIRST Statement

The FIRST statement specifies the name and language of the load module entry point. Either the name or the language can be allowed to default.

The format of the FIRST statement is

FIRST [*entrypoint*][(*language*)] *comment*

There can be only one FIRST statement in the input file. If used, it must appear before the first PROCESS statement. If no FIRST statement is used, ILCLINK assumes that the FIRST language is C and the entry point name is MAIN.

language is one of the language names listed in **Table 8.2** or the name of a user-supported language.

If used, *entrypoint* must be a one- to eight-character symbolic name or an asterisk (*). An asterisk indicates that ILCLINK is to use the default entry point for the specified language. **Table 8.3** illustrates the default entry points that ILCLINK uses.

Table 8.3
Default Entry Points

	Default Entry Point	
Language Code	Under CMS	Under OS
C	MAIN	MAIN
PLI	DMSIBM	PLISTART
PLI2	PLISTART	PLISTART
COBOL	(none)	(none)
COBOL2	(none)	(none)
FORTRAN	(none)	(none)
FORTRAN2	(none)	(none)
PASCAL	PASCALVS	PASCALVS

The entry point name can also be specified in a linkage editor ENTRY control statement or by the RESET option of a CMS LOAD or INCLUDE command. If the name occurs in more than one location, all occurrences must agree.

If neither an entry point name nor an asterisk is specified in a FIRST statement, ILCLINK makes no assumptions about the name of the entry point.

Examples using the FIRST statement

These examples illustrate the use of the FIRST statement to specify entry points and languages for main routines:

1. The following statement indicates that the name of the entry point is STARTUP, which is written in FORTRAN:

```
first startup(fortran)
```

2. The statement below indicates that ILCLINK should use PASCALVS as the entry point name. The main routine is written in Pascal.

```
first *(pascal)
```

3. This statement indicates that MAIN is the entry point name and that the main routine is written in C:

```
first *
```

4. This statement indicates that the main routine is written in PL/I Version 2 and that ILCLINK should make no assumptions about the name of the entry point:

```
first (pli2)
```

5. The following statement indicates that the entry point is MAIN and that the main routine is written in C. This statement provides the equivalent of the default values used by ILCLINK if no FIRST statement is used.

```
first main(c)
```

LANGUAGE Statement

The LANGUAGE statement specifies the names of the languages used in the load module.

The format of the LANGUAGE statement is

LANGUAGE *language1*[,*language2*...] *comment*

where *language* is one of the language names listed in **Table 8.3** or the name of a user-supported language.

There can be any number of LANGUAGE statements in the input file. All LANGUAGE statements must appear before the first PROCESS statement. C can be specified in a LANGUAGE statement, but it is always assumed. The language named in a FIRST statement

does not need to be named in a LANGUAGE statement, although it is not an error to do so.

If no LANGUAGE statements are used, ILCLINK assumes that the only language is C.

The statement below specifies that PL/I and FORTRAN, in addition to C, were used to produce the object code for the load module:

```
language pli,fortran
```

PROCESS Statement

The format of the **PROCESS** statement is

PROCESS *utility*[(*synonym*)] *operands*

where

utility
: is a process name, either CLINK, LINK, LKED, LOAD, or GENMOD. Each of these utilities is discussed in detail in the sections following this one.

synonym
: is the name of the utility to be invoked for the process. In general, the *synonym* field defaults to the name of the process. However, it can be used when the utility is to be invoked by an alias or synonym. For example, the following statement invokes the linkage editor via the alias name IEWLF128:

```
process link(iewlf128)
```

If used, the synonym name must be enclosed by parentheses and immediately follow the process name, with no intervening blanks.

operands
: are parameters and options for the utility. These can vary according to the operating system under which ILCLINK is running. The possible operands for each process are described below.

PROCESS CLINK

PROCESS CLINK invokes the CLINK object code preprocessor. There can be only one PROCESS CLINK statement in the input file. If used, the CLINK process should precede any LINK, LKED, or LOAD processes.

The format of the PROCESS CLINK statement is

PROCESS CLINK[(*synonym*)] [*option*[,*option*...]] *comment*

Under CMS, the PROCESS CLINK statement can also have the format

PROCESS CLINK[(*synonym*)] [*redirection*] [(*option* [*option*...] [) [*comment*]]]

The default *synonym* for the CLINK process is CLINK.
Option is a CLINK option. Any of the following options can be used:

PREM	NOPREM
TERM	NOTERM
AUTO	NOAUTO
LIST	NOLIST
WARN	NOWARN

Note: The AUTO option is only accepted by CLINK under CMS.
For example,

```
process clink noterm,warn
```

invokes CLINK with the NOTERM and WARN options.

Under CMS, the options can be preceded with a left parenthesis and followed by an optional right parenthesis, as in this example:

```
process clink (noterm noauto
```

Also under CMS, a redirection parameter, used to redirect CLINK's output to a nonterminal file, can precede the CLINK options. For example,

```
process clink >clink.map.a (noauto
```

redirects CLINK's output to CLINK MAP A1.

Input to the CLINK process can be any statement accepted by CLINK. For example,

```
process clink
   autocall lc370,proglib
   include lib1(p1,p2,p3)
   include lib2(p4,p5,p6)
   insert name1
```

causes ILCLINK to invoke CLINK with an input file containing the following statements:

```
INCLUDE LIB1(P1,P2,P3)
INCLUDE LIB2(P4,P5,P6
INSERT NAME1
.
. (INCLUDE ILC object code statements)
.
```

As shown above, ILCLINK adds the appropriate CLINK INCLUDE control statements for ILC object code to the CLINK input file automatically. These statements are added prior to the first NAME control statement in the CLINK input file or at the end of the input file if no NAME statement is used.

A return code greater than 4 from CLINK causes ILCLINK to terminate.

PROCESS LINK and PROCESS LKED

PROCESS LINK invokes the linkage editor. LKED and LINK can be used interchangeably. The LINK process should follow the CLINK process, if a CLINK process is used.

The format of the PROCESS LINK statement is

PROCESS LINK[(*synonym*)] [*option*[,*option*...]] *comment*

Under CMS, the PROCESS LINK statement can also have the format

PROCESS LINK[(*synonym*)] [(*option* [*option*...][) [*comment*]]]

where *option* is a linkage editor option. This example shows some options under OS:

```
process link xref,call,let
```

This example shows the same options under CMS:

```
process lked (xref call let
```

The default *synonym* for the LINK process is LKED (under CMS) or IEWL (under TSO and OS-batch).

Input to the LINK process can be any statement accepted by the linkage editor. For example,

```
process link
   autocall lc370,proglib
   include lib1(p1,p2,p3)
   include lib2(p4,p5,p6
   name prog(r)
```

causes ILCLINK to invoke the linkage editor with a SYSIN file containing the following statements:

```
INCLUDE LIB1(P1,P2,P3)
INCLUDE LIB2(P4,P5,P6
.
. (INCLUDE ILC object code statements)
.
NAME PROG(R)
```

If no CLINK process has been used and this is the first LINK process, ILCLINK adds the appropriate INCLUDE linkage editor control statements for ILC object code to the SYSIN file automatically. These statements are added prior to the first NAME control statement in the SYSIN file or at the end of the SYSIN file if no NAME statement is found.

Otherwise, if a CLINK process has been used and this is the first LINK process, ILCLINK adds the CLINK output object code to the linkage editor SYSLIN input automatically, via an INCLUDE control statement in the SYSIN file created by ILCLINK.

If the FIRST statement specifies an entry point name and no linkage editor ENTRY control statement appears in the process input, then ILCLINK adds an ENTRY statement with the specified entry point

name. As with INCLUDE statements, the ENTRY statement is added prior to the first NAME control statement or at the end of the SYSIN file if no NAME statement is found.

A return code greater than 8 from the linkage editor causes ILCLINK to terminate.

Under CMS, do not use PROCESS LINK and PROCESS LOAD in the same ILCLINK input file. Doing so causes unpredictable results.

PROCESS LOAD

The PROCESS LOAD statement invokes the CMS LOAD command. This process can only be used under CMS.

The format of the PROCESS LOAD statement is

PROCESS LOAD[(*synonym*)] [*fname* [*fname2...*]] [(*option* [*option...*] [) [*comment*]]]

The default *synonym* is LOAD. *fname* is the name of a TEXT file. *option* is a LOAD command option.

Input to the LOAD process are optional CMS INCLUDE commands. For example,

```
process load progmain (clear map noauto
autocall lc370,mylib
   include sub1 sub2 (nodup
   include sub3 sub4
```

causes ILCLINK to issue a GLOBAL TXTLIB command to GLOBAL LC370 TXTLIB and MYLIB TXTLIB. ILCLINK then issues the LOAD command followed by two INCLUDE commands.

If the FIRST statement specifies an entry point name and no RESET option has been used, then ILCLINK adds a RESET option with the entry point name to the last LOAD/INCLUDE statement it issues.

For the first LOAD process in an input file, if a CLINK process has been used, ILCLINK adds CLINK370 to the end of the list of TEXT filenames in the LOAD command automatically. If no CLINK process has been used, ILCLINK issues the appropriate INCLUDE commands automatically for ILC object code following all of the INCLUDE commands in the input file.

For example, suppose that a CLINK process has been used and the FIRST statement specified an entry point name of STARTUP. For the statements

```
process load progmain (clear map noauto
autocall lc370,mylib
   include sub1 sub2 (nodup
   include sub3 sub4
```

ILCLINK issues these commands:

```
GLOBAL TXTLIB LC370 MYLIB
LOAD PROGMAIN CLINK370 (CLEAR MAP NOAUTO
INCLUDE SUB1 SUB2 (NODUP
INCLUDE SUB3 SUB4 (RESET STARTUP
.
. (INCLUDE commands for ILC object code)
.
```

A return code greater than 4 from the last LOAD or INCLUDE command causes ILCLINK to terminate.

Do not use PROCESS LINK and PROCESS LOAD in the same ILCLINK input file. Doing so causes unpredictable results.

PROCESS GENMOD

The PROCESS GENMOD statement invokes the CMS GENMOD command. This process can be used only under CMS.

The format of the PROCESS GENMOD statement is

PROCESS GENMOD[(*synonym*)] *operands*

where *operands* are the parameters and options for a GENMOD command.

If the FROM option for the GENMOD command has not been used in the command, then ILCLINK adds a FROM option automatically that specifies the name of the first loaded CSECT. The first loaded CSECT is the first CSECT that appears in the LOAD MAP created by the LOAD command.

For example, if the name of the first CSECT is MAIN@, then the statement

```
process genmod myprog (nomap
```

causes ILCLINK to issue the following command:

```
GENMOD MYPROG (NOMAP FROM MAIN@
```

A GENMOD process is complete on the PROCESS GENMOD statement. No AUTOCALL statements can follow a PROCESS GENMOD statement. No other input is expected.

If no LOAD process is used, the GENMOD process is ignored.

A return code greater than 0 from the GENMOD command causes ILCLINK to terminate.

AUTOCALL Statement

The AUTOCALL statement specifies secondary input libraries (SYSLIB data sets or GLOBALed TEXT libraries) for the CLINK, LINK, LKED, and LOAD processes.

The format of the AUTOCALL statement is

AUTOCALL *library1*[,*library2...*] *comment*

There can be any number of AUTOCALL statements following a PROCESS statement. If more than one is used, no other statements (except comment statements) can appear between AUTOCALL statements.

The libraries specified in an AUTOCALL statement are made available to the process only for the duration of the process. The libraries are added to the list of secondary input libraries before the process executes. When the process terminates, the libraries are removed from the list of secondary input libraries.

Under TSO and OS-batch, the following statement causes the DDnames LC370 and MYLIB to be concatenated to SYSLIB. Under CMS, LC370 TXTLIB and MYLIB TXTLIB are added to the list of GLOBALed TXTLIBs.

```
autocall lc370,mylib
```

SYSTEM Statement

The SYSTEM statement specifies a string that is to be issued as a command to the operating system.

The format of the SYSTEM statement is

SYSTEM *command*

SYSTEM commands are issued via the C **system** function. The first nonblank character following the word SYSTEM is considered to be the start of the command. The command string is not translated to uppercase before being passed to **system**.

Unlike other ILCLINK statements, SYSTEM statements can be continued on subsequent lines. To continue a SYSTEM statement, add a plus sign (+) at the end of the line. All characters up to, but not including, the plus sign are considered to be part of the command. Leading blanks on continuation lines are not included in the command.

When ILCLINK is used under TSO, the *tso:* prefix is added to the command before it is issued.

Examples using the SYSTEM statement

These examples illustrate the use of the SYSTEM statement under TSO and CMS:

1. Under TSO, the statement

   ```
   system ALLOC FI(LIBTWO) DA(LIBTWO.OBJ) SHR REUSE
   ```

   causes the string

   ```
   "tso:ALLOC FI(LIBTWO) DA(LIBTWO.OBJ) SHR REUSE"
   ```

   to be issued as an operating system command via the **system** function.
2. Under CMS, the statement

   ```
   system filedef libtwo disk os txtlib c +
          dsn sascuser libtwo obj (perm
   ```

   causes the string

   ```
   "filedef libtwo disk os txtlib c dsn sascuser libtwo obj (perr
   ```

   to be issued as an operating system command via the **system** function.

   The SYSTEM statement causes message LSCI020 to be issued, which displays the return code from the command. The return code from commands issued via the SYSTEM statement is otherwise ignored.

The SYSTEM statement is not honored under OS-batch. An attempt to use a SYSTEM statement in this environment causes warning message LSCI040 to be issued.

Usage Notes

General Considerations

1. The return code from ILCLINK is the maximum return code from all individual control statements. ILCLINK terminates if the return code from any individual statement exceeds 8.
2. Because the CLINK process always writes to the same output file, use only one CLINK process per invocation of ILCLINK.
3. If you use a CLINK process, it must precede all LINK processes. Failure to do so causes unpredictable results.
4. The ILCLINK input file can specify multiple LINK, LOAD, or GENMOD processes. (In fact, this technique can be necessary under CMS if more than one TEXT library must be used as SYSLIB input to the linkage editor.) ILCLINK includes the ILC support routines in the CLINK process or the first LINK process.
5. Although the object modules generated by most compilers can be preprocessed by CLINK, it is more efficient to omit these modules from the CLINK process. CLINK the C object modules separately, and then link the non-C object modules and the CLINK output with a LINK or LOAD process.
6. In a program that contains object modules produced by either the PL/I Optimizing Compiler or by the OS PL/I Version 2 compiler, do not preprocess the PL/I object modules with CLINK. Both the C object modules and the PL/I object modules can contain pseudoregisters, and it is important that the pseudoregisters for each language be handled separately. In practice, the simplest way to do this is to force CLINK to "remove" the C pseudoregisters before linking the C object modules with the PL/I object modules, as follows:

   a. If the C object modules were created using the **RENT** or **RENTEXT** compiler option, preprocess the object modules with CLINK, using the PREM option.
   b. Link the PL/I object modules with the CLINK output object module.

   The PL/I message

   ```
   IBM005I TOO MANY FILES AND CONTROLLED VARIABLES
   ```

   indicates the need to run CLINK to remove C pseudoregisters.
7. ILCLINK does not add the C library to the AUTOCALL list automatically. Access this library by either explicitly naming it in an AUTOCALL control statement, adding it to the SYSLIB concatenation, or naming it in a GLOBAL TXTLIB command.
8. The **CFMWK** function is usually included automatically in the load module, but you may need to explicitly include it if it is called unnecessarily. An example of an unnecessary call to **CFMWK** is a C function that calls a PL/I routine that calls **CFMWK**. (Although the call to **CFMWK** is superfluous, it is not an error for the PL/I routine to call **CFMWK** in this situation.) **CFMWK** can be explicitly included by naming it in a linkage editor INCLUDE control statement or in a CMS LOAD or INCLUDE command.

9. ILCLINK always adds the libraries named in an AUTOCALL control statement to the autocall libraries (if any) that were available prior to its invocation. These libraries can be allocated to SYSLIB via an ALLOCATE command (under TSO), named in a GLOBAL TXTLIB command (under CMS), or named in a SYSLIB DD statement (under OS-batch). Libraries specified in one of these ways precede all other libraries and are available to all processes.
10. ILCLINK does not check for the presence of a linkage editor or LOAD command control statements in object or load modules. For example, suppose that an object module contains a linkage editor ENTRY control statement. If the name specified in the ENTRY statement conflicts with the name specified in the FIRST statement, ILCLINK does not detect the conflict. Usually, the linkage editor or LOAD command issues an error message for this situation.

CMS Considerations

1. The LOAD command may issue numerous messages of the form

```
DMSLIO202W Duplicate identifier identifier
```

   when loading multilanguage programs. These messages do not indicate a problem and can be suppressed by using the NODUP option on the PROCESS LOAD control statement.
2. OS/VS COBOL and the PL/I Optimizing Compiler generate an ENTRY control statement that does not have a blank in column 1. CLINK and the LOAD command accept this statement, but the LKED command does not. You must use the compiler option OSDECK to create an object module that contains an ENTRY control statement beginning in column 2.
3. Generally, all compiler-generated object modules use an ENTRY control statement or some other mechanism to force the correct entry point for that language to be chosen by the LKED or LOAD command. In a multilanguage program, you must use the RESET option of the LOAD command or an ENTRY control statement to specify the correct symbolic name for the entry point. If you specify an entry point name in a FIRST control statement, ILCLINK ensures that the correct entry point is selected.
4. The GENMOD command may not start at the lowest address when saving the module. This most often occurs when an external reference has the same name as the entry point name. Use the FROM option for the GENMOD command to specify the symbolic name of the lowest address in the program. (Examine the LOAD MAP file produced by the LOAD command to determine the name of the first CSECT.) If the PROCESS GENMOD control statement does not contain a FROM option, ILCLINK adds a FROM option using the first name in the LOAD MAP.
5. CLINK may produce the message

```
LSCL603 Warning: multiple occurrences of PLISTART
                 routine in library.
```

   when processing C and PL/I object modules. This message does not indicate a problem.
6. ILCLINK invokes CLINK directly. It does not invoke the CLINK EXEC.

7. If more than one TEXT library has been concatenated to SYSLIB, the LKED command ignores all but the first library. If more than one TXTLIB is named in an AUTOCALL statement for a LINK process, ILCLINK generates a FILEDEF for each TXTLIB, using the CONCAT option, but issues the following diagnostic:

```
LSCI030 Warning: Concatenated SYSLIB FILEDEFs are not
                 supported by the LKED command.
```

8. ILCLINK does not issue a GLOBAL TXTLIB command for more than eight TXTLIBs when running under Releases 3 and 4 of CMS. When running under later releases of CMS, ILCLINK does not issue a GLOBAL TXTLIB command for more than 256 TXTLIBs.

TSO and OS-Batch Considerations

1. The ILC support routines are available in both object module and load module format. These data sets are named SASC.ILCOBJ and SASC.ILCSUB, respectively. The object module format is used as input to CLINK, and the load module format is used as input to the linkage editor. The ILCLINK CLIST command and the ILCLINK cataloged procedure allocate these data sets to ILCLIB (object modules) and ILCSLIB (load modules). The ILC support routines are included from ILCLIB if a CLINK process is used. If there is no CLINK process, the ILC support routines are included from ILCSLIB.
2. ILCLINK adds autocall libraries to the SYSLIB concatenation in the order they are named in the AUTOCALL control statements. To prevent errors caused by differing data set block sizes (SYNAD EXIT errors in the linkage editor, for example), ensure that the data set with the largest block size is the first data set in the concatenation. BLKSIZE=3200 is the largest block size accepted by the linkage editor.
3. Do not use the same DDname in both an AUTOCALL control statement and a linkage editor INCLUDE control statement. This usually causes ILCLINK to issue message LSCI042 and terminate with a return code of 8.
4. If the ILCLINK CLIST is executed in batch, the SYSTERM DDname is allocated to SYSOUT instead of the terminal.
5. Under TSO, ILCLINK avoids the use of the SYS prefix for DDnames by using TSC as the first three characters of the preassigned DDnames. For example, ILCLINK uses TSCPRINT instead of SYSPRINT as the DDname for printed output from CLINK and the linkage editor.
6. Under TSO, printed output from CLINK is written to the CLNKLIST data set. Printed output from the linkage editor is written to the LINKLIST data set. Override these default destinations by allocating the DDname TSCPRINT to another data set either before invoking the ILCLINK CLIST command

(which causes both CLINK and linkage editor printed output to be written to that data set) or by using a SYSTEM control statement to allocate TSCPRINT to your preferred data set before the PROCESS CLINK and PROCESS LINK control statements.

Running ILCLINK under TSO

The ILCLINK CLIST

The format of the ILCLINK CLIST command is

ILCLINK *dsname* [(*options*]

where

dsname
: is the name of a data set containing ILCLINK control statements. The data set can be a sequential data set or a member of a partitioned data set. If the data set belongs to another user, the fully qualified name of the data set must be specified and enclosed in apostrophes. If the data set name is not fully qualified, the ILCLINK CLIST adds the user's prefix and a final qualifier of DATA, if necessary.

options
: are any ILCLINK options. (Refer to **ILCLINK Options**, earlier in this chapter, for a list of ILCLINK options.)

For example,

```
ILCLINK ilc(prog1) noterm echo
```

invokes ILCLINK, using ILC.DATA(PROG1) as the name of the input data set. The options used are NOTERM and ECHO.

ILCLINK CLIST data set options

The ILCLINK CLIST accepts several options that can be used to specify the names of the output data sets. Each option takes a data set name as its value.

In general, if the data set belongs to another user, the fully qualified name of the data set must be specified, and preceded and followed by three apostrophes. For example,

```
LOAD('''SASCUSER.ILC.LOAD(PROG1)''')
```

If the data set name is not fully qualified, the ILCLINK CLIST prefixes the data set name with the user's prefix and adds a final qualifier determined by the option. For example,

```
CLKOBJ(ILC)
```

causes ILCLINK to use *prefix*.ILC.OBJ as the name of the CLINK output data set.

```
CLKOBJ(ILC.CLINK)
```

causes ILCLINK to use *prefix*.ILC.CLINK.OBJ as the name of the CLINK output data set.

The data set options are listed below. The minimum abbreviation for each option is shown in uppercase.

CLKobj (dsname)
specifies the name of the CLINK output data set. The data set can be a sequential data set or a member of a partitioned data set. The final qualifier is assumed to be OBJ. If not explicitly specified, the CLIST supplies this qualifer automatically.

LOad (dsname)
specifies the name of the output load module. The data set name must be a member of a partitioned data set. If the name is not fully qualified and a final qualifier of LOAD is not specified, and then the CLIST assigns a final qualifier of LOAD.

If LOAD is not used and the name of the input data set is not fully qualified, the CLIST derives a load module data set name by replacing the final DATA of the input data set name with LOAD. Similarly, if the input data set is a member of a partitioned data set, the CLIST assigns the member name of the input data set as the member name of the load module data set. If no member name can be determined, the member name is selected by the linkage editor.

For example, if the CLIST is called as follows:

```
ILCLINK ilc(prog1)
```

then the name used for the load module data set is ILC.LOAD(PROG1).

PRint (dsname)
specifies the name of the ILCLINK listing file. The data set must be a sequential data set. The final qualifier is assumed to be ILCLIST. If not explicitly specified, the CLIST supplies this qualifier automatically.

PRINT(*) specifies that the listing file is to be displayed on the terminal. This option implies NOTERM and is effectively the same as using NOLIST and TERM.

NOPRint
indicates that a listing file is not to be created. This is the default. NOPRINT is the same as NOLIST.

ILCLINK concatenates the data sets named in AUTOCALL control statements to SYSLIB. For PROCESS CLINK, if the SYSLIB DDname has not been allocated, ILCLINK creates a temporary data set and allocates the SYSLIB DDname to it. This temporary data set is created as if the following ALLOCATE command has been issued:

```
ALLOCATE NEW TRACK SPACE(1,0) DIR(1) +
         RECFM(F B) BLKSIZE(3200) LRECL(80)
```

Running ILCLINK under CMS

The ILCLINK EXEC

The format of the ILCLINK EXEC is

ILCLINK *filename* [*options*]

where

filename
: is the filename of the input file. The filetype of the input file must be ILCDATA. The input file may be on any disk.

options
: are any ILCLINK options.

For example,

```
ILCLINK prog1 (nowarn echo
```

invokes ILCLINK, using PROG1 ILCLINK * as the fileid of the input data set. The options used are NOWARN and ECHO.

ILCLINK EXEC loads ILCLINK as a Nucleus Extension so that other commands can execute in the user area. When ILCLINK terminates, the EXEC drops the Nucleus Extension.

Listing File

The listing file produced by ILCLINK is written to the A disk, using the same filename as the input file and a filetype of ILCLOG.

Utility Files

The temporary utility file used by ILCLINK is created on the A disk, using a filetype of TEXT. The filename of the file is the same as the filename of the input file, prefixed with a dollar sign ($).

Running ILCLINK under OS-Batch

The ILCLINK Cataloged Procedure

The ILCLINK cataloged procedure contains the JCL shown below:

```
//ILCLINK  PROC
//LKED     EXEC PGM=ILCL,REGION=1536K
//STEPLIB  DD DSN=sasc.load,DISP=SHR C COMPILER LIBRARY
//         DD DSN=sasc.linklib,DISP=SHR C RUNTIME LIBRARY
//*
//*********************************************************************
//* DATA SETS NEEDED BY ILCLINK                                       ***
//*********************************************************************
//*
//ILCPRINT DD SYSOUT=*,DCB=(RECFM=VBA,LRECL=137,BLKSIZE=3200)
//ILCUT1   DD DSN=&&ILCUT1,UNIT=SYSDA,SPACE=(800,(5,5)),
//            DCB=BLKSIZE=800
//*
//*********************************************************************
//* DATA SETS NEEDED BY CLINK AND LINKAGE-EDITOR                      ***
//*********************************************************************
//*
//ILCLIB   DD DSN=sasc.ilcobj,DISP=SHR
//ILCSLIB  DD DSN=sasc.ilcsub,DISP=SHR
```

```
//SYSPRINT DD SYSOUT=*,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=*
//SYSLIN   DD UNIT=SYSDA,DSN=&&CLKOUT,SPACE=(3200,(20,20)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(3200,(100,20))
//SYSLMOD  DD DSN=&&LOADMOD(MAIN),DISP=(,PASS),UNIT=SYSDA,
//            SPACE=(1024,(50,20,1))
```

Note that the names of the data sets shown in *italics* may be changed by the installation. Ask your SAS Software Representative for C compiler products for more information.

The following example shows typical JCL needed to run ILCLINK. The PARM.LKED value passes options to ILCLINK. Separate the option names with blanks, not commas. In the example, the options used are NOWARN and ECHO.

```
//ILCLINK   JOB jobcard information
//STEP1     EXEC  ILCLINK,PARM.LKED='NOWARN ECHO'
//LKED.SYSLMOD DD DISP=OLD,DSN=your.load.library
//LKED.lib DD DISP=your.autocall.library
//LKED.ILCIN  DD *
      .
      . (ILCLINK control statements)
      .
/*
//
```

DD statements needed to run ILCLINK

The number of data sets used by ILCLINK varies according to the number of different AUTOCALL libraries used and the processes run. Eight data sets have preassigned DDnames. These data sets are described below.

The ILCPRINT DD statement is required if the LIST option is used (it is the default). This statement describes the listing data set that contains ILCLINK's printed output, including a listing of the input file and any diagnostic messages. The default specification for this data set is SYSOUT=*.

The ILCUT1 DD statement is always required. This statement describes the temporary utility data set used by ILCLINK to contain input to CLINK or the linkage editor. Space must be allocated for this data set, but the DCB requirements are supplied by ILCLINK.

The ILCLIB DD statement is required if PROCESS CLINK is used. This statement describes the data set that contains ILC interface routines in object code format.

The ILCSLIB DD statement is required if PROCESS CLINK is not used. This statement describes the data set that contains ILC interface routines in load module format.

The SYSTERM DD statement is required if the TERM option is used (it is the default) or if any process produces SYSTERM output. This statement describes the data set used to contain ILCLINK's terminal file and any SYSTERM output from CLINK or the linkage editor. The default specification for this file is SYSOUT=*.

The SYSLIN DD statement is required if PROCESS CLINK is used. This statement describes the data set used to contain the CLINK

output object code, which is subsequently used as input to the linkage editor. The data set can be sequential or partitioned. The BLKSIZE of this data set must be no less than 3200.

The SYSUT1 DD statement is always required. This statement describes the intermediate data set used by the linkage editor. This data set must be a sequential data set. Space must be allocated for this data set, but the DCB requirements are supplied by the linkage editor.

The SYSLMOD DD statement is always required. This statement describes the output load module library for the linkage editor. The data set must be a partitioned data set. For more information about this DD statement, refer to the appropriate documentation for the linkage editor. (See **References** later in this chapter.)

Additional DD statements

Each DDname specified in an ILCLINK AUTOCALL control statement or a linkage editor INCLUDE or LIBRARY control statement must be described with a DD statement. The data sets described by these statements must conform to the expectations of the associated utility, either CLINK or the linkage editor.

Temporary SYSLIB

For PROCESS CLINK, if the SYSLIB DD statement has not been used, ILCLINK creates a temporary data set and allocates the SYSLIB DDname to it. This temporary data set is issued as if the following DD statement has been used:

```
//SYSLIB   DD  DISP=(NEW,DELETE),SPACE=(3200,(1,0,1)),
//             DCB=(RECFM=FB,BLKSIZE=3200,LRECL=80)
```

Examples Using ILCLINK Control Statements

In all of the following examples, the ILCLINK control statements are shown in uppercase. Other statements are shown in mixed case. This convention is used for readability only, because ILCLINK accepts input in both upper- and lowercase.

Data set names or fileids used for compiler-related files that are shown in lowercase are examples. These names do not necessarily match the names used for the data sets or files that should actually be used. Typically, the names used for these files are chosen by each individual installation.

Example 8.1 shows a simple use of ILCLINK under TSO; **Example 8.2** gives the equivalent statements under CMS. **Examples 8.3**, **8.4**, and **8.5** show how to allocate the necessary DDnames under TSO, CMS, and OS-batch, respectively.

Example 8.1 shows a sample input file that describes a program that was created using both C and OS/VS COBOL. The initial function is the C `main` function.

Example 8.1
Sample OS and TSO ILCLINK Program

```
FIRST MAIN(C)  ❶
LANGUAGE COBOL  ❷
*
PROCESS LINK MAP,LIST,TERM,RMODE=24,AMODE=24  ❸
AUTOCALL LC370,COBLIB  ❹
include cobj(ch2,ch2c)  ❺
include cblobj(ch2cob)
name ch2(r)
```

Statement Notes for Example 8.1

1. The FIRST control statement identifies the name of the entry point (MAIN) and the language in which it is written (C). This entry statement specifies the same entry point information as ILCLINK would have used if a FIRST control statement had not appeared in the input file.
2. The LANGUAGE control statement indicates that COBOL was used to create some of the object modules. C is implied, both by its use in the FIRST statement and by default. (ILCLINK always assumes that C was used to create some or all of the object modules.)
3. The PROCESS LINK control statement invokes the linkage editor. The MAP, LIST, TERM, RMODE=24, and AMODE=24 options are used.
4. The AUTOCALL statement gives the DDnames of two autocall libraries, LC370 and COBLIB. For this example, LC370 is allocated to the C resident library and COBLIB to the COBOL library using the equivalent of the following ALLOCATE commands:

   ```
   ALLOCATE FILE(LC370) DA('sasc.obj') SHR
   ALLOCATE FILE(COBLIB) DA('sys1.coblib') SHR
   ```

   before ILCLINK is invoked.
5. These three statements are passed to the linkage editor, along with INCLUDE control statements added by ILCLINK for the ILC interface routines. Because a CLINK process is not used, ILCLINK includes these routines in load module format, using the DDname ILCSLIB. After these statements are added, the SYSIN file created by ILCLINK is

   ```
   INCLUDE ILCSLIB(L$ICB1X,L$ICB1C,L$ICB1Q,L$ICB1L)
   INCLUDE ILCSLIB(L$ICB1F,L$ICB1F)
   INCLUDE ILCSLIB(L$IMIXD)
   INCLUDE ILCSLIB(L$ICB1P)
   INCLUDE COBJ(CH2,CH2C)
   INCLUDE CBLOBJ(CH2COB)
   ENTRY MAIN
   NAME CH2(R)
   ```

   Note that ILCLINK adds a linkage editor ENTRY control statement that names the entry point specified by the FIRST control statement.

The COBJ and CBLOBJ DDnames are allocated, either via a TSO ALLOCATE command or a DD statement, prior to invoking ILCLINK.

Example 8.2 is the same as **Example 8.1**, except that the PROCESS LOAD and PROCESS GENMOD control statements are used to create a CMS MODULE file.

Example 8.2
Sample CMS ILCLINK Program

```
FIRST MAIN(C)  ❶
LANGUAGE COBOL  ❷
*
PROCESS LOAD CH2 CH2C CH2COB (NODUP  ❸
AUTOCALL LC370,COBLIBVS
*
PROCESS GENMOD CH2  ❹
```

Statement Notes for Example 8.2

1. The FIRST control statement identifies the name of the entry point (MAIN) and the language in which it is written (C). This entry statement specifies the same entry point information as ILCLINK would have used if a FIRST control statement had not appeared in the input file.
2. The LANGUAGE control statement indicates that COBOL was used to create some of the object modules. C is implied, both by its use in the FIRST statement and by default. (ILCLINK always assumes that C was used to create some or all of the object modules.)
3. The PROCESS LOAD and AUTOCALL control statements cause the following CMS commands to be issued (assuming that no TXTLIBs had been GLOBALed prior to invoking ILCLINK):

   ```
   GLOBAL TXTLIB LC370 COBLIB
   LOAD CH2 CH2C CH2COB (NODUP
   ```

   ILCLINK issues the following INCLUDE commands after the LOAD command completes:

   ```
   INCLUDE L$ICB1X L$ICB1C L$ICB1Q L$ICB1L
   INCLUDE L$ICB1M L$ICB1F
   INCLUDE L$IMIXD
   INCLUDE L$ICB1P (RESET MAIN
   ```

   These commands cause the ILC interface routines to be loaded. Note that ILCLINK adds a RESET option to the last INCLUDE command to specify the entry point named in the FIRST statement.
4. The PROCESS GENMOD control statement invokes the GENMOD command to create a module named CH2 MODULE A1.

Example 8.3 uses the SYSTEM control statement to allocate the required DDnames under TSO. The VS FORTRAN Version 1 compiler was used to create some of the object modules in the program. The entry point is written in FORTRAN.

Example 8.3
Allocating DDnames for ILCLINK under TSO

```
FIRST F1(FORTRAN)  ❶
LANGUAGE C,FORTRAN  ❷
*
SYSTEM ALLOC DD(COBJ) DA('SASCUSER.ILC.OBJ') SHR  ❸
SYSTEM ALLOC DD(LC370) DA('sasc.obj') SHR
*
PROCESS CLINK  ❹
AUTOCALL LC370
include cobj(f1c)
*
SYSTEM ALLOC DD(FORTOBJ) DA('SASCUSER.ILC.OBJ') SHR  ❺
SYSTEM ALLOC DD(FORTLIB) DA('sys1.vfortlib') SHR
*
PROCESS LINK MAP,LIST,TERM,RMODE=24,AMODE=24  ❻
AUTOCALL FORTLIB
include fortobj(f1,f1f3,f1f4)
name f1(r)
*
SYSTEM FREE DD(COBJ,LC370,FORTOBJ,FORTLIB)  ❼
```

Statement Notes for Example 8.3

1. The FIRST control statement specifies that the name of the entry point is F1 and that it is written in FORTRAN.
2. The LANGUAGE control statement identifies the two languages used to create the program. Note that this statement is superfluous, because FORTRAN is named in the FIRST statement and C is always assumed.
3. These SYSTEM statements cause the TSO ALLOCATE command to be invoked. The LC370 DDname is allocated to the C resident library, SASC.OBJ.
4. These three statements invoke CLINK. SASCUSER.ILC.OBJ(F1C) is the only C object module.
5. As in the previous SYSTEM statements, these SYSTEM statements invoke the ALLOCATE command.
6. These statements invoke the linkage editor using the options MAP, LIST, TERM, RMODE=24, and AMODE=24. FORTLIB is allocated to an autocall library. As before, ILCLINK adds INCLUDE control statements for the appropriate ILC interface routines and an ENTRY control statement to identify the entry point name. The SYSIN input to the linkage editor is

```
INCLUDE ILCLIB(L$CICMN)
INCLUDE ILCLIB(L$IFO1X,L$IFO1C,L$IFO1Q,L$IFO1L)
INCLUDE ILCLIB(L$IMIXD)
INCLUDE ILCLIB(L$IFO1P)
INCLUDE SYSLIN    (CLINK OUTPUT)
INCLUDE FORTOBJ(F1,F1F3,F1F4)
ENTRY F1
NAME F1(R)
```

7. This SYSTEM control statement invokes the FREE command to free the DDnames that are allocated.

Example 8.4 uses the SYSTEM control statement to issue FILEDEFs for the required DDnames under CMS. The VS FORTRAN Version 1 compiler was used to create some of the object modules in the program. The entry point is written in FORTRAN. (This example is equivalent to **Example 8.3** for TSO users.)

Example 8.4
Allocating DDnames for ILCLINK under CMS

```
FIRST F1(FORTRAN)  ❶
LANGUAGE C,FORTRAN  ❷
*
PROCESS CLINK  ❸
AUTOCALL LC370
include f1c
*
SYSTEM FILEDEF FORTOBJ DISK MYILC TXTLIB *  ❹
SYSTEM FILEDEF FORTLIB DISK vsfort txtlib *
SYSTEM FILEDEF SYSLMOD DISK MYILC LOADLIB A1 (RECFM U BLKSIZE 6144
*
PROCESS LKED (MAP LIST TERM  ❺
AUTOCALL FORTLIB
include fortobj(f1,f1f3,f1f4)  ❻
name f1(r)
*
SYSTEM FILEDEF FORTOBJ CLEAR  ❼
```

Statement Notes for Example 8.4

1. The FIRST control statement specifies that the name of the entry point is F1 and that it is written in FORTRAN.
2. The LANGUAGE control statement identifies the two languages used to create the program. Note that this statement is superfluous because FORTRAN is named in the FIRST statement and C is always assumed.
3. These statements invoke CLINK. F1C TEXT is the only C object module.
4. These SYSTEM control statements issue the FILEDEFs needed by the LKED command.
5. These statements cause ILCLINK to issue the following FILEDEF command for autocall input to the LKED command:

   ```
   FILEDEF SYSLIB DISK fortlib txtlib *
   ```

   and invoke the following LKED command:

   ```
   LKED $xxxxxxx (MAP LIST TERM
   ```

   where *$xxxxxxx* is the name of the temporary SYSIN file created by ILCLINK.
6. These statements form part of the SYSIN input to the LKED command. After ILCLINK adds INCLUDE statements for ILC support routines and CLINK output and adds an ENTRY

statement, the LKED SYSIN file contains these statements:

```
INCLUDE L$CICMN
INCLUDE L$IFO1X,L$IFO1C,L$IFO1Q,L$IFO1L
INCLUDE L$IMIXD
INCLUDE L$IFO1P
INCLUDE CLINK370  (CLINK OUTPUT)
INCLUDE FORTOBJ(F1,F1F3,F1F4)
ENTRY F1
NAME F1(R)
```

7. This SYSTEM control statement clears the FORTOBJ FILEDEF issued in statement 6. The FORTLIB and SYSLMOD FILEDEFs are cleared by the LKED command.

Example 8.5 is the same as **Example 8.3**, except that it shows the JCL required to execute ILCLINK under OS-batch. The required DDnames are allocated using DD statements.

The VS FORTRAN Version 1 compiler was used to create some of the object modules in the program. The entry point is written in FORTRAN.

Example 8.5
Allocating DDnames for ILCLINK under OS-Batch

```
//EXAMPLE    JOB  jobcard information
//STEPONE    EXEC ILCLINK
//LKED.SYSLMOD DD DISP=SHR,DSN=SASCUSER.ILC.LOAD
//LKED.COBJ    DD DISP=SHR,DSN=SASCUSER.ILC.OBJ
//LKED.LC370   DD DISP=SHR,DSN=sasc.obj
//LKED.FORTOBJ DD DISP=SHR,DSN=SASCUSER.ILC.OBJ
//LKED.FORTLIB DD DISP=SHR,DSN=sys1.vfortlib
//LKED.ILCIN   DD *
/*
FIRST F1(FORTRAN)  ❶
LANGUAGE C,FORTRAN  ❷
*
PROCESS CLINK  ❸
AUTOCALL LC370
include cobj(f1c)
*
PROCESS LINK MAP,LIST,TERM,RMODE=24,AMODE=24  ❹
AUTOCALL FORTLIB
include fortobj(f1,f1f3,f1f4)
name f1(r)
//
```

Statement Notes for Example 8.5

1. The FIRST control statement specifies that the name of the entry point is F1 and that it is written in FORTRAN.
2. The LANGUAGE control statement identifies the two languages used to create the program. Note that this statement is superfluous because FORTRAN is named in the FIRST statement and C is always assumed.
3. These three statements invoke CLINK. SASCUSER.ILC.OBJ(F1C) is the only C object module.

4. These statements invoke the linkage editor using the options MAP, LIST, TERM, RMODE=24, and AMODE=24. FORTLIB is allocated to an autocall library. ILCLINK adds INCLUDE control statements for the appropriate ILC interface routines and an ENTRY control statement to identify the entry point name. The SYSIN input to the linkage editor is

```
INCLUDE ILCLIB(L$CICMN)
INCLUDE ILCLIB(L$IFO1X,L$IFO1C,L$IFO1Q,L$IFO1L)
INCLUDE ILCLIB(L$IMIXD)
INCLUDE ILCLIB(L$IFO1P)
INCLUDE SYSLIN    (CLINK OUTPUT)
INCLUDE FORTOBJ(F1,F1F3,F1F4)
ENTRY F1
NAME F1(R)
```

Interlanguage Communication Support Routines

This section explains the internal rules that ILCLINK uses to select the interlanguage communication support routines. In general, it is not necessary to be familiar with these rules. However, this description is provided for programmers who are developing advanced multilanguage applications.

The major service peformed by ILCLINK is the inclusion of a number of support routines for the interlanguage communication feature. For a given program, the support routines that are needed vary according to

- the number and combination of languages used to create the program
- the language used for the entry point of the program.

ILCLINK uses the following rules to determine which service routines are needed. (The characters *xxx* in the routine names should be replaced by an appropriate language code. See **Table 8.2** for the language codes used by ILCLINK.)

- If C is not used for the entry point, include L$CICMN.
- For each language used in the program other than C, include L$I*xxx*X, L$I*xxx*C, L$I*xxx*Q, and L$I*xxx*L.
- For each language used in the program other than C and the language used for the entry point, include L$I*xxx*F and L$I*xxx*M.
- If any language other than C is used, include L$IMIXD.
- If exactly two languages are used to create the program, include L$I*xxx*P.
- If more than two languages are used to create the program, include L$IMIXP.

Examples Using ILC Support Routines

The following examples illustrate the rules above:

1. SAS/C is the only language in the program. No ILC support routines are included.

2. SAS/C and VS FORTRAN Version 1 were used to create the program. The entry point is in C. In this case, the necessary routines are

```
L$IFO1X, L$IFO1C, L$IFO1Q, L$IFO1L (rule 2)
L$IMIXD (rule 4)
L$IFO1P (rule 5)
```

3. SAS/C and VS FORTRAN Version 1 were used to create the program. The entry point is in FORTRAN. In this case, the necessary routines are

```
L$CICMN (rule 1)
L$IFO1X, L$IFO1C, L$IFO1Q, L$IFO1L (rule 2)
L$IMIXD (rule 4)
L$IFO1P (rule 5)
```

4. SAS/C, the OS PL/I Optimizing Compiler, and VS COBOL II were used to create the program. The entry point is in COBOL. In this case, the necessary routines are

```
L$CICMN (rule 1)
L$IPL1X, L$IPL1C, L$IPL1Q, L$IPL1L (rule 2)
L$IPL1F, L$IPL1M (rule 3)
L$ICB2X, L$ICB2C, L$IPL1Q, L$ICB2L (rule 2)
L$IMIXD (rule 4)
L$IMIXP (rule 5)
```

Under CMS, each routine is in a separate TEXT file. The filename of each TEXT file matches the routine name. Under OS, the object code versions of the routines are members of the SASC.ILCOBJ data set, and the load module versions are members of the SASC.ILCSUB data set. For more information about these data sets, refer to your SAS Software Representative for C compiler products.

Default Data Set Allocations under TSO

Table 8.4 shows the default data set sizes and DCB parameters used by ILCLINK when creating new data sets under TSO.

Table 8.4 *Default Data Set Size Values*

	Block		Space Allocation		DCB Parameters		
DDname	Type	Size	Primary	Secondary	RECFM	LRECL	BLKSIZE
SYSPRINT	Track	—	10	5	FBA	121	1210
SYSLIN	Block	3200	20	20	FB	80	3200
SYSUT1	Block	1024	200	50	—	—	—
SYSLMOD	Block	1024	50	20	—	—	—

Note that the default size values can be changed by the site. For more information, ask your SAS Software Representative for C compiler products.

Where no defaults are specified, the DCB parameters are determined by CLINK or the linkage editor.

References

Information about the utilities and commands discussed in this chapter can be found in the following publications:

SAS Institute Inc., *SAS/C Compiler and Library User's Guide*. Cary, NC: SAS Institute Inc., 1988.

International Business Machines Corporation, *MVS/Extended Architecture Linkage Editor and Loader User's Guide*, IBM Publication GC26-4011.

International Business Machines Corporation, *OS/VS Linkage Editor and Loader*, IBM Publication GC26-3813.

International Business Machines Corporation, *Virtual Machine/System Product CMS Command Reference*, IBM Publication SC19-6209.

International Business Machines Corporation, *Virtual Machine/System Product CMS for System Programming*, IBM Publication SC24-5286.

9 Debugging Multilanguage Programs

Introduction

Debugging a multilanguage application can be more difficult than debugging a single-language program because errors made in one language may affect the behavior of other languages and because many errors that are easy to make have unpredictable consequences. This chapter offers some debugging hints for multilanguage programs, focusing on symptoms and solutions that are unique to interlanguage communication. General debugging techniques are given first, followed by specific tips for each language supported by SAS/C ILC. Additional debugging information, such as a detailed explanation of each ILC run-time message, can be found in Appendix 1, "ILC Library Diagnostic Messages."

Note that errors in a multilanguage program might not have anything to do with interlanguage communication. For instance, I/O problems, such as errors reading a data file, are solved the same way whether the program is in one language or several. Having multiple languages in a program adds complexity, but it is not the only area in which mistakes can be made.

Also note that even though this chapter is organized according to problem symptoms, you should read all of it. Errors are discussed in association with their usual effect, but, because of the complexity of ILC, the same error does not always produce the same outcome. For instance, an error that normally causes an 0C4 or 0C5 ABEND by storing at a random location might sometimes instead cause incorrect output.

ILC User ABENDs

User ABENDs with codes 1233, 1234, and 1235 are issued by the SAS/C library when certain severe error conditions are detected. Note that in addition to the causes listed, any of these ABENDs can occur if

the program overlays library storage, though this would be unusual. The SAS/C debugger command **STORAGE** or the run-time option **=storage** can be used to detect many library storage overlays.

ABEND 1233 A 1233 ABEND indicates an error in the SAS/C library, unless storage has been overlaid. If your program generates this ABEND, you should note any error messages generated before the ABEND, save the dump if one was generated, and call the Technical Support Department at SAS Institute.

ABEND 1234 A 1234 ABEND indicates that you have made an interlanguage call to a language whose framework has not been created. The following are the most likely reasons for this ABEND:

- You failed to call **mkfmwk** before calling a non-C routine from C, or failed to call **CFMWK** before calling a C function from some other language.
- You made an interlanguage call after deleting the framework of the called language with **dlfmwk** or **DCFMWK**.
- A previous call to **mkfmwk** or **CFMWK** failed, and your program neglected to check for an error. In this case, if the **quiet** function was not used, you should have received a message from the SAS/C library (or from the other language's library) explaining why the framework could not be created.

ABEND 1235 A 1235 ABEND indicates a library-detected error other than a call to a language whose framework has not yet been created. Before the ABEND, the C library generates a diagnostic message describing the error. Possible causes for this ABEND include the following:

- A **longjmp** or out-of-block GOTO was used to terminate a routine in a language other than the one that issued the jump or GOTO.
- A routine in another language was called from C and was not declared with an appropriate keyword (such as **__fortran**).
- A call to another language was made while the program was terminating (possibly from an **atexit** routine).
- The program attempted to call another language from more than one coprocess.
- No memory was available to process the argument list in a call to a routine in another language.

Other ABENDs

Finding the Point of ABEND When a multilanguage program ABENDs with an ABEND code other than those described above, the first necessary debugging task is to determine the language in which the ABEND occurred. Note that you may get tracebacks or diagnostic messages from all active languages, and that the language that sends the first message is not necessarily the one that caused the ABEND.

When an ABEND occurs in a language other than C, the SAS/C library writes the message LSCX051, giving the name of the language that ABENDed. If you do not receive this message, the ABEND occurred while the C framework was active. (If you are running C under TSO, this message may not be sent to the terminal unless your

TSO profile specifies WTPMSG. Use of this profile attribute is recommended to ensure that you receive all ILC messages.)

Occasionally, when an ABEND occurs in another language, the SAS/C library is unable to allow the ABEND to proceed without either changing the point of ABEND as observed by the other language, or changing the ABEND code. Since changing the point of ABEND could make the other language's diagnostics useless, the library instead changes the original ABEND code to 0C6, giving an odd address within one byte of the actual point of ABEND. When this takes place, the library issues message LSCX052 ("ABEND *xxx* reinstated as 0C6") to inform you of the original ABEND code. See Appendix 1, "ILC Library Diagnostic Messages," for more information on this message.

When you get one or more tracebacks for a multilanguage program ABEND, note that each language's traceback includes only routines written in that language, plus SAS/C library interface routines. In a C traceback, these routines and their purposes are labeled so that if you also have a traceback in the other language, you can put the two together to obtain a complete picture of the calling sequence at the time of ABEND. See the discussion of framework switching in Chapter 2, "Multilanguage Framework Management," for further information on calling sequences and save area chaining.

After you have successfully found the point of ABEND, you can investigate further by using normal debugging techniques for the language in which the ABEND occurred.

Common Pitfalls

ABENDs in multilanguage programs are frequently the result of a few common errors. If you do the following, you can avoid many of these errors:

- Compile C functions called from non-C with `INDep`.
- Make sure argument types agree.
- Check function return types.
- Make sure the module is linked with the correct entry point.
- Include only one main routine.

The sections that follow discuss in more detail the effects of not following these steps.

Failure to compile with INDep

When another language calls a C function that was not compiled with `INDep`, the called function is unable to locate the C framework. This results in an immediate ABEND, usually an 0C1 or 0C4 ABEND. Because the called function is unable to locate the C framework, the ABEND appears to be an ABEND in the calling language, not in C. Further, the ABENDing language may not be able to produce useful information about the location of the failure.

You should always check for this particular mistake when you have an 0C*x* ABEND in another language where the other language is unable to determine the point of ABEND.

Incompatible argument types

Failure to define compatibly the data types of arguments to a subroutine with the types expected can lead to almost any kind of ABEND. For instance, an integer passed as a pointer can cause an 0C4 ABEND, and an integer passed as packed decimal can cause an

0C7 ABEND. Consult the tables on data type compatibility in the appropriate non-C language chapter to ensure that all arguments are declared and passed properly. Also, make sure that each argument is passed correctly by value or by reference, whichever is expected by the called language. For example, a scalar argument declared as VAR in Pascal must be explicitly passed by reference from C.

Failure to use a required data type conversion macro such as `_STRING` when calling PL/I from C can also cause random ABENDs because PL/I may interpret an actual data address as a descriptor address. Similarly, failure to declare a C function in PL/I as OPTIONS(ASM) may cause errors because C may interpret a pointer to a descriptor as a pointer to data instead.

One way to check for argument mismatch problems of this sort is to print out the value of each argument on entry to a troublesome subroutine, using a debugger or print statements in the code. An argument that prints erroneously or that causes an ABEND during printing can be investigated in more detail. If you debug by adding I/O statements to the code, it is best to print each argument with a separate statement and to separate each output line from the previous one by several blank lines. This makes it less likely that successfully printed lines will be in a buffer (and therefore not written) when an ABEND occurs.

Function return type mismatches

A subtle cause of errors is a call from one language to another where the declared types of the return value are not compatible. This occurs most frequently with calls to a C function from a non-C routine, where the C function does not return a value. Such functions must be declared as returning `void` in C. When a C function that returns no value is called from C, the code works correctly even when the function is not declared `void`. However, when a function that returns no value is called from another language, a random return value is stored if the function is not properly declared `void`. Even worse, if the calling language is PL/I or Pascal, the return value is stored in a random location because the caller has not passed the address of a return area. This may cause an immediate ABEND or may overlay other data, causing unpredictable results at a later time.

This sort of error can also occur for calls from C to another language, such as a PL/I routine that returns a FIXED BINARY result but that is declared in C to return `void`.

Incorrect entry points

If a multilanguage load module has an incorrect entry point due to an incorrect ILCLINK FIRST statement, the program almost certainly will ABEND shortly after starting execution. It may ABEND before any language framework has been successfully established, which means that no traceback or useful diagnostics are produced. Frequently, the ABEND resulting from this error is an 0C7.

An ABEND that produces neither diagnostics nor traceback from the language of the main routine but produces them for some other language is very likely to be an entry point error.

For best results, you should always explicitly specify a FIRST control statement in your ILCLINK input file. If you do not have a FIRST statement, ILCLINK assumes that the program entry point is in C. If the entry point is in some other language, the error will

probably not be detected at link time. That is, an executable load module will be produced with no unusual ILCLINK, linkage editor, or loader messages, even though the load module does not run successfully.

Using more than one main routine

If you create a multilanguage program with more than one main routine, the results are unpredictable and may depend on the languages in use and the exact method by which the program is linked. Some possible results of this error are as follows:

- A random 0C*x* type of ABEND occurs, either when a framework is created or later during an interlanguage call.
- The framework for a language other than the first fails to be initialized successfully.
- Two frameworks are created for the same language. This can produce quite puzzling results. For instance, C external variables or FORTRAN dynamic COMMONs might seem to have two sets of values.
- Even though `mkfmwk` indicates success, C is unable to find the created framework later, resulting in a user ABEND 1234.
- One of the secondary main routines is executed during a call to `mkfmwk`, with unpredictable data in the argument list.

If your application has more than one main routine, you must change all but the first one executed so that the rest are no longer main routines. For instance, rename a C `main` function or change a FORTRAN PROGRAM statement to a SUBROUTINE statement.

Incorrect Results

When a multilanguage program produces incorrect results, you should check for some of the same pitfalls as for an ABEND. Arguments or return values declared incorrectly frequently produce incorrect results rather than an ABEND. You can investigate such problems in exactly the same way as you would an ABEND, by printing out argument values before and after a call, to determine whether all are being passed correctly.

Incorrect results can also occur as the result of storage overlays. If the value of a variable changes when the variable is not shared, or if no references have been made to the variable, you should suspect a storage overlay. Consider using the SAS/C debugger `MONITOR` command to investigate storage overlays. Note that the `MONITOR` command will identify storage overlays that occur in a language other than C, although, of course, the offending source line cannot be identified. Also, `MONITOR` can be used to monitor a storage area belonging to another language, provided you know its address.

Incorrect File Output

If you are using the same file in more than one language and the file output is garbled or lost, make sure the file is never open in two languages at the same time. Except for files allocated to the terminal, the 370 operating systems do not support output file sharing. This includes the case where a single program shares a file with itself by

accessing the file through several paths, such as from both C and PL/I.

If you must share a file between languages, be sure to close the file in one language before opening it in another. Because of the overhead of repeated opens and closes, you may prefer to use different output files from different languages and combine them later using a utility or another program.

Miscellaneous Tips

The following sections may help you write more correct multilanguage programs and debug them more easily. The list below offers some general tips; the sections that follow are specific to one language (such as FORTRAN or PL/I).

- Use the C run-time option `=multitask` during program development and debugging to lessen interference between frameworks. Note that when you use the Pascal/VS debugger, you must use the `=multitask` option.
- Do not use the C `quiet` function to suppress C diagnostics.
- Use the `=btrace` run-time option to generate a traceback with any C diagnostic message.
- Use the `STORAGE` debugger command to check for storage overlays.
- When running under TSO, make sure your profile specifies WTPMSG.
- Avoid the use of complicated signal handling and non-linear flows of control (such as `longjmp`), if possible.
- Make shared variables of a simple type, if possible. You should prefer sharing arrays to sharing structures, and function pointers should be shared only if necessary. (This advice follows from the fact that complicated types such as structures and function pointers are highly language-dependent in implementation and might not be easily shared. For instance, sharing a structure with PL/I may require you to build your own structure descriptors. Also, the differing structure alignment rules between languages can cause problems.)
- If you are sharing external variables, be sure to use the C compiler option `NORENT` or declare the variables `const` in C.
- Remember that strings passed to C from another language are rarely null-terminated.
- Remember that C character literals (such as `'C'`) are passed to other languages as fullword integers rather than as strings.
- Always write a message out before calling `exit` or otherwise terminating the program, at least during development. This makes it easier to locate unexpected-termination errors.

FORTRAN Tips

- Remember to declare arguments to C functions explicitly. If no declaration is present, FORTRAN chooses a data type using normal FORTRAN rules, which may not be appropriate. For instance, if you neglect to declare a variable named TOKEN, which is intended to be the C framework token stored by `CFMWK`, it is assumed to be a REAL*4 variable rather than the correct INTEGER*4.
- Constants have data types in FORTRAN. When you pass a constant to C, be sure it has the correct type. Pass 1 for an `int *`

argument, 1.0 for a `float *` argument, and 1.0D0 for a `double *` argument.

- Always use the `_STRING` data type conversion macro when passing any string argument other than a literal or string structure to FORTRAN.
- Remember that FORTRAN and C use different conventions for accessing array elements.
- If you use the C `exit` function, include the header file `<stdlib.h>` or `<fortmath.h>` because, otherwise, the FORTRAN EXIT function will be called instead, with unpredictable results.

COBOL Tips

- Distinguish carefully between COBOL DISPLAY items and COBOL COMPUTATIONAL items. The former should be processed in C as an array of `char`, the latter as an approriate variety of numeric data, for instance, `double` for COMP-2.
- Use the C `__noalignmem` modifier or the COBOL SYNCHRONIZED keyword to ensure identical mapping of records in COBOL and C.
- Be sure to use a set of COBOL compiler options compatible with those used by the SAS/C library (as described in Chapter 5, "Communication with COBOL"). Using other options may cause ABEND and/or incorrect results that are very difficult to diagnose.

PL/I Tips

- Remember that almost all C functions should be declared in PL/I as OPTIONS(ASM, INTER). (See Chapter 6, "Communication with PL/I," for details and exceptions.) This includes the library functions `CFMWK`, `DCFMWK`, `QCFMWK`, and `ACFMWK`.
- Remember to declare arguments to C functions explicitly. If no declaration is present, PL/I chooses a data type using normal PL/I rules, which may not be appropriate. For instance, if you neglect to declare a variable named TOKEN, which is intended to be the C framework token stored by `CFMWK`, it is assumed to be a FLOAT(6) variable rather than the correct FIXED BIN(31).
- Note that a PL/I FIXED BIN variable is, by default, FIXED BIN(15), which corresponds to a C `short int`, not to an `int`.
- Constants have data types in PL/I. When you pass a constant to C, be sure it has the correct type. Pass 1.0E0 for a `float *` argument and 1.0000000E0 for a `double *` argument. Note that 1, 1.0, and 1.00000000 are all FIXED DECIMAL constants and should not be passed to a C function unless the C function expects a packed decimal value of the same precision and scaling.

  Similar surprises can occur when you pass expressions from PL/I to C. For instance, if I is a FIXED BIN(15) variable, the type of I/2 is FIXED BIN(31,16), a type for which there is no C equivalent.

  In general, when you call C from PL/I, you should either

  - pass only variables, never constants or expressions
  - declare the types of all arguments for each C function, in which case the PL/I compiler will convert constants and expressions to the correct type.

- Be sure to note whether string variables shared with PL/I are fixed-length or varying-length, as the corresponding C data types are different.

- When you pass a C pointer to PL/I, you must use the `&` or `@` operator to pass the pointer's address. If the pointer is passed directly, the pointer's value is interpreted by PL/I as the pointer address.
- Under OS, PL/I and C both use the DDname SYSPRINT for standard output. If SYSPRINT is not a terminal file, this can lead to lost output or ABENDs in either language. You can avoid this problem by opening SYSPRINT in PL/I using the TITLE option, or by using the `_stdonm` external variable to specify a different `stdout` filename to C.
- Be very careful with PL/I ON-units to avoid executing a GOTO statement that terminates a called C function.
- Always use the CLINK preprocessor when you link a program containing both PL/I and C. This keeps the pseudoregisters for the two languages separate and prevents a PL/I "too many pseudoregisters" error.
- If you use the PL/I Checkout Compiler with C, use the Checkout Compiler SIZE option to leave enough memory free for the use of the C library and debugger.

Pascal Tips

- Because Pascal can use either call by value or call by reference, be sure that C arguments correspond to the technique being used. For instance, the C argument corresponding to a Pascal REAL should be declared `double` if the argument is passed by value, or `double *` if passed by CONST or VAR. (Note, however, that some kinds of data, such as strings and records, are passed by reference even if the program requests pass by value.)
- Note whether string variables shared with Pascal are fixed-length (ARRAY OF CHAR) or varying-length (STRING(n) or CONST STRING), because the corresponding C data types are different.
- When you pass a C pointer to Pascal and the Pascal routine expects pass by CONST or pass by VAR, you must use the `&` or `@` operator to pass the pointer's address. If the pointer is passed directly, the pointer's value is interpreted by Pascal as the pointer address.
- The use of the C compiler option `VString` is strongly recommended for programs that communicate with Pascal.
- Make sure that structures are aligned identically in both languages. If you use PACKED RECORDs in Pascal, use the `__noalignmem` keyword or the SAS/C compiler option `BYtealign` to suppress alignment of the corresponding C structure.
- In OS batch processing, Pascal uses the DDname SYSPRINT for diagnostic output, while C may use it for standard output. This can lead to lost output or ABENDs in either language. You can avoid this problem by using the ERRFILE run-time option in Pascal to cause Pascal to use another DDname, or by using the `_stdonm` external variable to specify a different `stdout` filename to C.

10 Advanced Topics

Introduction

This chapter discusses the use of unusual features of SAS/C software, or of other languages, in a multilanguage program. Most multilanguage programs do not require the use of these features. Topics discussed include

- dynamic loading
- addressing mode considerations
- reentrancy
- signal/condition handling
- coprocesses
- using more than two languages
- running several multilanguage programs simultaneously.

Dynamic Loading in a Multilanguage Program

Dynamic loading is a feature supported by three languages, SAS/C, VS COBOL, and PL/I. Because each language implements dynamic loading differently, you must be careful to use dynamic loading in a manner appropriate both to the language doing the load and to the language in which the loaded module is written.

As a general rule, limit all use of dynamic loading in a multilanguage program to one of the following situations:

- A routine dynamically loads a load module that is written entirely in the same language as the loading routine. For instance, a PL/I routine (possibly called from C) uses the FETCH statement to load a load module that contains only PL/I code.
- A routine written in language X loads a load module containing routines in several languages, and the language of the entry point is language X. The frameworks for the other languages in the load module have not yet been created. They will be created by a call to the appropriate SAS/C library routine in the loaded module. Such frameworks must be destroyed before the load module is unloaded.

For instance, a C function calls **loadm** to load a load module containing both C and COBOL code. The load module entry point is **dynamn**, written in C. The loaded module calls **mkfmwk** to create the COBOL framework and calls **dlfmwk** to delete the COBOL framework before the load module is unloaded.

Other uses of dynamic loading are unlikely to work successfully, due to the different implementations of the various languages. For instance, it seems plausible to develop a program structure in which C calls a PL/I routine that uses the FETCH statement to load a load module containing C functions. However, the C functions will be unable to execute successfully because the PL/I FETCH statement will not create a C PRV (pseudoregister vector) for the C module. This in turn causes C library failures and incorrect access to **extern** variables if the **RENT** compiler option had been in use.

Also note that calls to **mkfmwk** for the same language (or calls to **CFMWK**) in several load modules lead to unpredictable errors.

Dynamic Loading with FORTRAN

One profitable use for dynamic loading in a multilanguage situation is to circumvent the IBM restriction that prevents the FORTRAN framework from being created more than once. This restriction applies on a load module basis. That is, when the FORTRAN framework is created, the load module that created it is modified so that a later attempt to create the framework will fail. Because the failure is caused by modification of the load module that created the framework, if this load module is unloaded and reloaded, the framework can be created again.

Note that when you use this technique, all the FORTRAN routines, as well as the calls to **mkfmwk** and **dlfmwk**, must be in this same dynamically loaded module.

MVS/XA Addressing Mode Considerations

When a new framework is created by **mkfmwk** or **CFMWK**, the new framework begins execution in the same addressing mode as the current addressing mode of the calling program. Because C does not support changing addressing mode during execution, if C communicates with any language that does not support 31-bit addressing (such as Pascal/VS), the C code must also execute with AMODE=24.

PL/I and COBOL, unlike C, allow some parts of a program to execute in 24-bit addressing mode, and some in 31-bit addressing mode. Use of interlanguage communication with C does not affect this ability, but all the C code must still execute in the addressing mode established by the first C load module. If C executes with AMODE=24, you must be sure that all data passed from another language to C resides below the 16-megabyte line because, otherwise, C is unable to access it.

Reentrancy

Multilanguage programs can be reentrant if both languages support reentrancy and if any restrictions on reentrant usage are observed. Note that in a reentrant program you cannot share data via external variables unless the data is never modified, because shared external variables are stored within the load module.

Even though FORTRAN supports partially reentrant programs by providing a tool to separate a FORTRAN program into reentrant and non-reentrant portions, this does not allow you to create reentrant FORTRAN-C mixtures. When you mix FORTRAN and another language and use the separation tool, all non-FORTRAN code is placed in the non-reentrant module, even if some or all of this code is reentrant.

Multilanguage Signal/Condition Handling

Most languages provide facilities that allow the program to gracefully handle computational errors, such as division by zero. Sometimes, as with the C `signal` facility, the program can also handle asynchronous events, such as use of the terminal attention key. Error handling in a multilanguage program is discussed in detail in **Error Handling** in Chapter 2, "Multilanguage Framework Management."

When you use the C `signal` function to trap an asynchronous event, such as terminal attention or an IUCV interrupt, the signal handler can only be called when the C framework is active. If a routine in another language is running when the interrupt occurs, the interrupt is kept pending until C is reactivated by a call from or return by the other language. Programs that require a quick response to such signals should avoid heavy use of routines in other languages.

Implementation of asynchronous event handling in other languages should be used with caution. Because such features can be implemented in diverse ways, it is impossible to generalize about the handling of an interrupt that occurs while a C function is running. The interrupt may be left pending until the other language resumes, or it may be handled immediately. In the latter case, whether or not the results will be incorrect because the C code was interrupted depends on the implementation.

When a PL/I ATTENTION ON-unit is used to handle terminal attention interrupts, the interrupt remains pending until the PL/I framework is active.

You should avoid using features in more than one language for handling the same asynchronous event. For instance, you should not define a `SIGINT` handler in C and an ATTENTION ON-unit in PL/I. When an interrupt occurs, it is unpredictable which language will process the interrupt and when the processing will take place.

Coprocesses in a Multilanguage Program

A SAS/C program that uses ILC can, like a normal C program, be composed of several coprocesses. However, only a single coprocess can create or destroy frameworks, or communicate with other languages. This requirement is imposed because other languages do not support the non-hierarchical transfers of control permitted by the use of coprocesses. When the C framework is created by another

language, through use of **CFMWK**, this limits all use of interlanguage communication to the main coprocess.

Use of interlanguage communication in a coprocessing program has one significant difference from normal use. In a normal multilanguage application, if a framework created by C terminates unexpectedly, C execution is terminated as well. If a framework created by a C coprocess other than the main coprocess terminates, only the creating coprocess is terminated, that is, all other coprocesses continue execution. This feature provides a way for a C program to continue to execute in the unusual cases where this is desirable, after unexpected termination of another framework.

Using More Than Two Languages

Even though most multilanguage applications use only two languages, it is possible to use more than two if necessary. This complicates the implementation and forces some additional restrictions, but conceptually it is not much more difficult than using two languages. Some of the important considerations are as follows:

- When more than two languages are used, the language frameworks must be terminated in the opposite order of their creation. This is required because the system calls used for error handling are stacked by the operating system, and termination of frameworks in the wrong order results in an incorrect error-handling environment. Of course, out-of-order termination is unavoidable if a framework terminates unexpectedly. In this case, after all frameworks have terminated, all error handling should have been successfully cancelled, although there is a period of time in which the environment is unstable. Unpleasant failures in this sort of situation cannot be completely avoided, but in this case the library puts out a worst-case message, warning of the possibility of failure.
- When only a single non-C language is active and a C function is called from the other language, the **INDep** interface routine L$UPREP can access a language-dependent location (such as the user word of the PL/I TCA) to locate the C framework. When C is called from another language and more than two languages are active, L$UPREP cannot immediately determine which language it was called by and, therefore, cannot access the other language's control blocks. In this case, L$UPREP must issue an expensive system call to locate the C framework. This makes language transitions more expensive when three languages are in use than when only two are in use.
- Some other languages, such as PL/I, support their own forms of interlanguage communication. There is no reason that you cannot use another language's interlanguage communication and C interlanguage communication in the same application. However, you must avoid direct communication with the same language using SAS/C ILC and another ILC implementation. For instance, if C calls PL/I, and PL/I calls FORTRAN, then C may not call or be called by FORTRAN. This is because the existence of PL/I interlanguage communication is not visible to C, and C will treat a call to C from FORTRAN as a call from PL/I, which will prove very unpleasant.

Running Several Multilanguage Programs Simultaneously

In general, there is no problem running more than one multilanguage program at a time, as long as the other languages allow it. For instance, one multilanguage program could invoke another using the C `system` function, or two multilanguage programs could run under ISPF in split-screen mode. When these programs use more than two languages, however, the program must make additional calls to the C library to allow it to successfully manage the various frameworks.

As noted above, with a three-language application, the C library uses operating system facilities to keep track of the various frameworks and which ones are active. Under OS, each task is managed independently, so there are no special requirements if the programs run under different TCBs. However, under CMS or MVS, when only a single TCB is in use, the following situation can arise.

Suppose a three-language program is active, and that it causes another three-language program to be activated. (For instance, it may issue SVC 202 under CMS or use the ISPF SELECT service under MVS.) When the new program asks the operating system for the location of the current C framework, the framework for the previous application may be found, rather than the one for the new application. This will probably cause memory overlays of the previous framework.

To allow this problem to be bypassed, the C library provides the routines **QCFMWK** and **ACFMWK**. **QCFMWK** quiesces the C framework, that is, it notifies the operating system that a C framework is becoming inactive. Similarly, **ACFMWK** activates a C framework by informing the operating system that the framework is resuming execution. A three-language program that might allow another three-language program to receive control must call **QCFMWK** before permitting a change of this sort, and call **ACFMWK** afterwards. (These routines are described in detail in Chapter 11, "ILC Framework Manipulation Routines.")

Note that such changes can be indirect results of program action. Under CMS, an ISPF application that invokes the ISPF DISPLAY service and has not inhibited split screen mode allows a switch to another application running on the other half of the screen. Such programs need to call **QCFMWK** before the ISPF call, and **ACFMWK** afterwards.

Note that use of these routines is only necessary when more than two languages are in use. (However, their use causes no harm in the two-language case.)

11 ILC Framework Manipulation Routines

Introduction

This chapter describes the execution framework manipulation routines provided with the SAS/C ILC feature. Each routine is listed with a synopsis, description, return value discussion, cautions, implementation process, portability considerations, and an example.

The four standard routines are described first; the last two routines are for use with advanced applications that use three or more languages.

If you are using or implementing a user-supported language (a language other than FORTRAN, COBOL, PL/I, or Pascal), you should consult Chapters 14 through 16 for additional information on the use of these routines.

mkfmwk Create the Framework for a Non-C Language

SYNOPSIS

```
#include <ilc.h>

void *mkfmwk(char *lang, char *options);
```

DESCRIPTION

The **mkfmwk** function is called from C to create the framework for another high-level language. The first argument to **mkfmwk** is a null-terminated string giving the generic name of the language whose framework is to be created (one of "FORTRAN", "COBOL", "PLI", or "PASCAL"). The language name may be specified in either upper- or lowercase.

The **options** argument to **mkfmwk** is a null-terminated string containing run-time options for the other language. The format of the string depends on the conventions used by the other language; it will not be translated to uppercase or modified in any other way by the library.

You can call **mkfmwk** even if the framework for the other language might have already been created. (This situation could arise when several modules are combined, each of which calls routines in another language.) However, **dlfmwk** must be called the same number of times as **mkfmwk** is called before the framework can be deleted.

RETURN VALUE

Normally, **mkfmwk** returns a "token" identifying the framework. This token must be passed to **dlfmwk** to delete the new framework. If an error occurs that prevents the framework from being created, a NULL pointer is returned.

See Chapter 14, "Using ILC with a User-Supported Language," for additional information on the use of language tokens in a user-language context.

CAUTIONS

Run-time options for VS COBOL II will be ignored, due to limitations of this version of COBOL.

IMPLEMENTATION

mkfmwk calls a SAS/C library-supplied main routine in the other language to cause the framework to be created. (In PL/I, for example, this routine is declared as an OPTIONS(MAIN) procedure.) For this reason, you cannot include your own main routine in a language whose framework is to be created by **mkfmwk**.

PORTABILITY

mkfmwk is not portable.

mkfmwk **Create the Framework for a Non-C Language**
(continued)

EXAMPLE

```
#include <ilc.h>

   /* Create the PL/I framework, requesting the
      use of the PL/I Version 2 debugger */

   void *pli_token;

   pli_token = mkfmwk("PLI", "TEST(ALL,*,;)");
   if (!pli_token) abort();
      .
      .     /* call PL/I procedures */
      .
   dlfmwk(pli_token);
```

SEE ALSO

dlfmwk, CFMWK, DCFMWK

dlfmwk Delete the Framework for a Non-C Language

SYNOPSIS

```
#include <ilc.h>

int dlfmwk(void *token);
```

DESCRIPTION

The **dlfmwk** function is called from C to delete a non-C framework created by **mkfmwk**. The argument is the token returned by **mkfmwk** when the framework was created. If any routines in the language have been called and have not yet returned, the framework will not be deleted, and an error code is returned to the caller.

RETURN VALUE

The return value from **dlfmwk** is 0 if the argument token was valid and it was possible to delete the framework. Otherwise, a nonzero value is returned.

CAUTIONS

If **mkfmwk** has been called several times for a language, you must call **dlfmwk** once with each returned token before the framework is actually deleted.

Note that due to limitations of IBM FORTRAN implementations, it is not possible to create the FORTRAN framework a second time after it has been deleted. A method for working around this problem using SAS/C dynamic loading is described in Chapter 10, "Advanced Topics."

PORTABILITY

dlfmwk is not portable.

EXAMPLE

Use of **dlfmwk** is illustrated in the example for **mkfmwk**.

SEE ALSO

mkfmwk, CFMWK, DCFMWK

CFMWK Create the C Framework

SYNOPSIS

```
void CFMWK(char *language, char *options, int ILC_flags, int token);

/* CFMWK is never called from C                                    */
/* Below are the data types corresponding to each language for  */
/* each argument:                                                  */
/* language:                    CHARACTER in FORTRAN           */
/*                              PICTURE X(n) in COBOL          */
/*                              CHAR(*) in PL/I                */
/*                              STRING in Pascal               */
/* options:                     same data type as language     */
/* ILC_flags:                   INTEGER in FORTRAN             */
/*                              PICTURE 9(9) COMPUTATIONAL     */
/*                                 in COBOL                    */
/*                              FIXED BINARY(31) in PL/I       */
/*                              INTEGER in Pascal              */
/* token:                       same data type as ILC_flags    */
```

DESCRIPTION

The **CFMWK** routine is called from a language other than C to create the C framework. **CFMWK** must be called using standard linkage, with register 1 addressing a standard call-by-reference parameter list.

The **language** argument is a fixed-length character string specifying the generic name of the language calling **CFMWK**, terminated with a period. Valid values are "FORTRAN.", "COBOL.", "PLI.", and "PASCAL.". If the **language** argument is not correct, the results are unpredictable.

The **language** argument may also be passed as a varying-length character string with the string length contained in the first two bytes. This is the format of Pascal strings and PL/I CHAR VARYING variables. The string length must be less than 256 in this case, and the terminating period must still be present.

The **options** argument is a string containing C run-time options, followed by a period. If no run-time options are needed, a string containing only a period should be passed. The options should each be preceded by an equal sign and separated by spaces, exactly as if they were specified on a C command line. Any tokens in the string that are not recognized as C run-time options are ignored. Note that run-time options that affect the operation of interlanguage communication, as described below, must be specified using the **ILC_flags** argument rather than the **options** argument.

The **ILC_flags** argument is used to specify C run-time options that affect the operation of interlanguage communication. They are passed separately from the normal C run-time options because they must be processed before the C framework can be created. The argument value is treated as a bit string, with each bit representing a particular option.

CFMWK **Create the C Framework**
(continued)

The bits are defined as follows:

```
4 -- =nohtsig (suppress library ABEND handling)
2 -- =nohcsig (suppress library program check handling)
1 -- =multitask (use multitasking framework control)
```

For instance, an `ILC_flags` value of 3 requests that library program check handling be suppressed and multitasking be used, but that library ABEND handling should not be suppressed. If the corresponding run-time options are specified in the `options` argument, it is not considered an error, but the options may have little or no effect. See Chapter 2, "Multilanguage Framework Management," for more information on the effects of these options.

The `token` argument specifies a fullword integer variable. `CFMWK` stores a "token" representing the C framework in this variable. (This token can later be passed to `DCFMWK` to request that the C framework be deleted.) If the C framework cannot be created, a token of 0 is stored. Note that call by reference must be used for the call, and a FORTRAN or PL/I dummy argument must not be created, in order for the return token to be stored correctly.

You can call `CFMWK` even if the C framework has previously been created. (This situation could arise when several modules are combined, each of which calls routines in C.) However, `DCFMWK` must be called the same number of times as `CFMWK` is called before the C framework can be deleted.

RETURN VALUE

`CFMWK` does not have a return value because it may be called from languages such as COBOL that do not support return values.

CAUTIONS

In PL/I, `CFMWK` must be declared as OPTIONS(ASM).

In Pascal, all arguments to `CFMWK` must be declared as VAR or CONST. The `token` argument must be declared VAR because it is modified by `CFMWK`.

IMPLEMENTATION

`CFMWK` creates the C framework calling L$CICMN, a main function included in the C library. For this reason, it is not possible for a program that uses `CFMWK` to create the C framework also to have a user `main` function in C.

PORTABILITY

`CFMWK` is not portable.

EXAMPLES

The following examples show, for each supported language, calls to `CFMWK` and `DCFMWK` to create and destroy the C framework. Unimportant code (such as COBOL IDENTIFICATION DIVISION statements) has been omitted.

CFMWK **Create the C Framework**
(continued)

FORTRAN

```
      PROGRAM CEXAM
      INTEGER*4 TOKEN, ERR
C
C     CREATE C FRAMEWORK AND USE THE C DEBUGGER
C
      CALL CFMWK('FORTRAN.', '=DEBUG.', 0, TOKEN)
      IF (TOKEN.EQ.0) STOP 16
      CALL DCFMWK(TOKEN, ERR)
      IF (ERR.NE.0) STOP 8
      STOP
      END
```

COBOL

```
WORKING-STORAGE SECTION.
77  C-TOKEN PIC 9(9) COMP.
77  C-OPTIONS PIC X(7) VALUE "=DEBUG.".
77  ILC-OPTIONS PIC 9(9) COMP VALUE 0.
77  COBOL-NAME PIC X(6) VALUE "COBOL.".
77  DCFMWK-ERR-FLAG PIC 9(9) COMP.

PROCEDURE DIVISION.

*
*    CREATE THE C FRAMEWORK AND USE THE C DEBUGGER
*
     CALL "CFMWK" USING COBOL-NAME C-OPTIONS ILC-OPTIONS C-TOKEN.
     IF C-TOKEN EQUAL 0
        MOVE 16 TO RETURN-CODE
        STOP RUN.
     CALL "DCFMWK" USING C-TOKEN DCFMWK-ERR-FLAG.
     IF DCFMWK-ERR-FLAG NOT EQUAL 0
        MOVE 8 TO RETURN-CODE.
     STOP RUN.
```

PL/I

```
DECLARE (C_TOKEN, ILC_OPTS, ERR) FIXED BINARY(31);
DECLARE (CFMWK, DCFMWK) ENTRY OPTIONS(ASM,INTER);

/* CREATE THE C FRAMEWORK AND USE THE C DEBUGGER */

ILC_OPTS = 0;
CALL CFMWK('PLI.', '=DEBUG.', ILC_OPTS, C_TOKEN);
IF C_TOKEN = 0 THEN DO;
   CALL PLIRETC(16);
   STOP;
END;
CALL DCFMWK(C_TOKEN, ERR);
```

CFMWK **Create the C Framework**
(continued)

```
IF ERR ¬= 0 THEN CALL PLIRETC(8);
STOP;
```

Pascal

```
procedure CFMWK(const LANG : STRING;
                const COPTS : STRING;
                const ILCOPTS : INTEGER;
                var TOKEN : INTEGER);
   EXTERNAL;
procedure DCFMWK(const TOKEN : INTEGER;
                 var ERRFLAG : INTEGER);
   EXTERNAL;

var
   TOKEN : INTEGER;
   ERRFLAG : INTEGER;

begin

   (* Create the C framework, use     *)
   (* multitasking framework control. *)

   CFMWK('PASCAL.', '.', 1, TOKEN);
   if TOKEN = 0 then begin
      RETCODE(16);
      HALT
   end;
   DCFMWK(TOKEN, ERRFLAG);
   if ERRFLAG <> 0 then
      RETCODE(8);
   HALT;
end.
```

SEE ALSO

DCFMWK, mkfmwk, dlfmwk

DCFMWK Delete the C Framework

SYNOPSIS

```
void DCFMWK(int token, int err);

/* DCFMWK is never called from C                                    */
/* Below are the corresponding types for each language for each */
/* argument:                                                        */
/* token:                          INTEGER in FORTRAN               */
/*                                 PICTURE 9(9) COMPUTATIONAL       */
/*                                    IN COBOL                      */
/*                                 FIXED BINARY(31) in PL/I         */
/*                                 INTEGER in Pascal                */
/* err:                            same data type as token          */
```

DESCRIPTION

The **DCFMWK** routine is called from a language other than C to delete the C framework created by **CFMWK**. **DCFMWK** must be called using standard linkage, with register 1 addressing a standard call-by-reference parameter list. If any routines in C have been called and have not yet returned, the framework will not be deleted, and an error code is stored.

The **token** argument is the token for the language that was stored by **CFMWK** when the framework was created.

The **err** argument is an integer variable in which a value can be stored that indicates the success or failure of **DCFMWK**. A value of 0 indicates success, while any other value indicates failure.

If **CFMWK** has been called several times, you must call **DCFMWK** once with each returned token before the C framework will actually be deleted.

RETURN VALUE

DCFMWK does not have a return value because it may be called from languages such as COBOL that do not support return values.

CAUTIONS

In PL/I, **DCFMWK** must be declared as OPTIONS(ASM).

In Pascal, the **token** argument to **DCFMWK** must be declared as CONST or VAR, and the **err** argument must be declared as VAR.

PORTABILITY

DCFMWK is not portable.

EXAMPLE

Use of **DCFMWK** is illustrated in the examples for **CFMWK**.

SEE ALSO

CFMWK, mkfmwk, dlfmwk

QCFMWK Quiesce the C Framework

SYNOPSIS

```
#include <ilc.h>          /* only when called from C */

void QCFMWK(int token, int err);

/* Below are data types and considerations for each argument:   */
/*    token:                    must be 0 in C                  */
/*                              INTEGER in FORTRAN              */
/*                              PICTURE 9(9) COMPUTATIONAL      */
/*                                 in COBOL                     */
/*                              FIXED BINARY(31) in PL/I        */
/*                              INTEGER in Pascal               */
/*    err:                      pass &err in C                  */
/*                              same data type as token in      */
                                   other languages               */
```

DESCRIPTION

The **QCFMWK** routine may be called either from C or from another language to inform the library that the active C framework should be quiesced because the program is about to take some action that could cause creation or activation of another C framework. **QCFMWK** must be called using standard linkage, with register 1 addressing a standard call-by-reference parameter list.

The **token** argument is an integer token representing the active C framework. When **QCFMWK** is called from C, this argument must be zero and must be passed by value. When **QCFMWK** is called from a non-C language, the argument must be the token returned by **CFMWK** when the C framework was created. In this case, the argument must be passed by reference.

The **err** argument is an integer variable in which a value can be stored that indicates the success or failure of **QCFMWK**. For a call from C, the variable's address must be passed. For a call from a non-C language, the argument must be passed by reference. A value of 0 stored in **err** indicates success, while any other value indicates failure.

RETURN VALUE

QCFMWK does not have a return value because it may be called from languages such as COBOL that do not support return values.

CAUTIONS

In Pascal, the **token** argument to **QCFMWK** must be declared as CONST or VAR, and the **err** argument must be declared as VAR.

After a call to **QCFMWK**, no calls to another language using SAS/C ILC may be performed until **ACFMWK** is called to reactivate the C framework.

PORTABILITY

QCFMWK is not portable.

QCFMWK **Quiesce the C Framework**
(continued)

USAGE NOTES

QCFMWK is normally used only in programs composed of three or more languages, which may allow another multilanguage program to begin or resume execution other than by use of SAS/C library facilities. For instance, use of ISPF display services, or of SVC 6 or SVC 202 to call another program, could permit another C framework to become active. Although use of **QCFMWK** in programs using fewer than three languages is permitted, there is little reason for it.

EXAMPLES

Two examples are shown below. The first example shows a C function that uses **_cms202** to invoke the EXEC command, which might call another C program.

```
#include <ilc.h>
#include <svc.h>
#include <lcstring.h>

int fmwkerr;

struct {
   char cmd [8];
   char name [8];
   int fence [2];
} plist = {
   "EXEC    ",
   "        ",
   { -1, -1 }
};

char *execname;
int execret;

QCFMWK(0, &fmwkerr);
if (fmwkerr) {
   printf("Unable to quiesce the C framework\n");
   exit(16);
}
/* copy name of EXEC to PLIST */
memcpyp(plist.name, execname, 8, strlen(execname), ' ');
_ldregs(R1,&plist);
_cms202();           /* invoke the EXEC */
execret = _stregs(R15);

ACFMWK(0, &fmwkerr);
if (fmwkerr) {
   printf("Unable to activate the C framework\n");
   exit(16);
}
```

QCFMWK Quiesce the C Framework
(continued)

The second example shows a PL/I routine that creates a C framework. Later, it calls an ISPF service that might invoke another C program.

```
DECLARE (C_TOKEN, ILC_OPTS, ERR) FIXED BINARY(31);
DECLARE (CFMWK, DCFMWK, QCFMWK, ACFMWK) ENTRY OPTIONS(ASM,INTER);
DECLARE ISPLINK ENTRY OPTIONS(ASM, INTER, RETCODE);
DECLARE SELECT_LEN FIXED BINARY(31);
DECLARE SELECT_OPTS CHAR(*);

ILC_OPTS = 0;
CALL CFMWK('PLI.', '.', ILC_OPTS, C_TOKEN);
IF C_TOKEN = 0 THEN DO;
   PUT SKIP EDIT('Unable to create the C framework')(A);
   CALL PLIRETC(16);
   STOP;
END;

/* additional processing */

CALL QCFMWK(C_TOKEN, ERR);
IF ERR ¬= 0 THEN DO;
   PUT SKIP EDIT('Unable to quiesce the C framework')(A);
   CALL PLIRETC(16);
   STOP;
END;
SELECT_OPTS = 'PANEL(ILCPNL)';
SELECT_LEN = LENGTH(SELECT_OPTS);
CALL ISPLINK('SELECT ', SELECT_LEN, SELECT_OPTS);
CALL ACFMWK(C_TOKEN, ERR);
IF ERR ¬= 0 THEN DO;
   PUT SKIP EDIT('Unable to activate the C framework')(A);
   CALL PLIRETC(16);
   STOP;
END;
```

SEE ALSO

CFMWK, ACFMWK

ACFMWK Activate the C Framework

SYNOPSIS

```
#include <ilc.h>          /* only when called from C */

void ACFMWK(int token, int err);

/* Below are data types and considerations for each argument: */
/*    token:                    must be 0 in C               */
/*                              INTEGER in FORTRAN           */
/*                              PICTURE 9(9) COMPUTATIONAL   */
/*                                 in COBOL                  */
/*                              FIXED BINARY(31) in PL/I     */
/*                              INTEGER in Pascal            */
/*    err:                      pass &err in C               */
/*                              same data type as token in   */
                                   other languages           */
```

DESCRIPTION

The **ACFMWK** routine may be called either from C or from another language to inform the library that the C framework should be reactivated, after it was quiesced by a call to **QCFMWK**. **ACFMWK** must be called using standard linkage, with register 1 addressing a standard call-by-reference parameter list.

The **token** argument is an integer token representing the active C framework. When **ACFMWK** is called from C, this argument must be zero and be passed by value. When **ACFMWK** is called from a non-C language, the argument must be the token returned by **CFMWK** when the C framework was created. In this case, the argument must be passed by reference.

The **err** argument is an integer variable in which a value can be stored that indicates the success or failure of **ACFMWK**. For a call from C, the variable's address must be passed. For a call from a non-C language, the argument must be passed by reference. A value of 0 stored in **err** indicates success, while any other value indicates failure.

RETURN VALUE

QCFMWK does not have a return value because it may be called from languages such as COBOL that do not support return values.

CAUTIONS

In Pascal, the **token** argument to **ACFMWK** must be declared as CONST or VAR, and the **err** argument must be declared as VAR.

PORTABILITY

ACFMWK is not portable.

ACFMWK **Activate the C Framework**
(continued)

EXAMPLES

Use of **ACFMWK** is illustrated in the examples for **QCFMWK.**

SEE ALSO

CFMWK, ACFMWK

12 Using Packed Decimal Data in C

Introduction

A data format used by COBOL and PL/I that is not directly supported by C is packed decimal. This is the format of COMP-3 data in COBOL, or FIXED DECIMAL data in PL/I. One way of handling such data in C is through use of the SAS/C in-line machine code interface, which allows you to issue packed decimal instructions directly. However, this is cumbersome and requires knowledge of the packed decimal hardware facilities. As an alternative, the header file `<packed.h>` defines two macros, `pdval` and `pdset`, for converting packed decimal data to and from `double`, which can be easily processed in C. This chapter describes these two macros.

Even though packed decimal data is frequently treated by the program as fractional data, it is treated by the hardware as integral. That is, a COBOL item with PICTURE 999V99, or a PL/I FIXED DECIMAL(5,2) variable, might be assigned the value 123.45, but the associated storage locations would actually contain 12345. Ordinarily, when packed decimal is converted to C `double`, there is no loss of accuracy because the format of `double` is wide enough to hold the largest possible packed decimal integer.

It is possible to "scale" a packed decimal value when converting to `double`, for instance, converting the hardware 12345 in the example above to a `double` 123.45. You should be cautious when doing this because 123.45 cannot be expressed as an exact binary fraction and, therefore, is subject to roundoff errors. Calculations performed using scaled values may be inaccurate in cases where no error would have occurred without the scaling.

The `pdset` and `pdval` macros assume that the packed decimal data is contained in a `char[n]` array. When you pass a COMP-3 COBOL item or a FIXED DECIMAL PL/I variable to C, the corresponding C argument should be declared `char(*)[n]`, that is, as a pointer to a `char[n]` array. If the item is a structure element, it should be declared an array, not a pointer to an array. If the source item contains a maximum of d digits, the value of `n` in the declaration should be $(d+1)/2$. (Each digit occupies half a byte, with an extra half byte for a sign.)

Note that `pdval` and `pdset` may be used in any C program; they are not limited to multilanguage applications.

pdset Assign Double Value to Packed Decimal

SYNOPSIS

```
#include <packed.h>

void pdset(char (*target) [], double source,
           unsigned scale, double round);
```

DESCRIPTION

The **pdset** macro is used to convert a double value to packed decimal and store the result in a character array. The **target** argument is a pointer to a character array that the packed decimal result is to be stored in. The maximum **target** size is 8 bytes (15 digits). This argument must be defined as a pointer to a **char** array, not as a **char *** pointer or a **char** array. The **source** argument is the double value that is to be converted. The **scale** argument specifies a scale factor. The converted **source** is multiplied by **pow(10.0, scale)** during processing. The **scale** value must be between 0 and 15. The **round** value specifies a number that is to be added to the scaled **source** before conversion to packed decimal. After the **round** amount is added, any fractional portion is discarded.

If the **source** value is the result of computations with non-integral data, a **round** value of zero is not recommended, as it may cause the effect of a very small inaccuracy to be considerably magnified. For instance, a computed value of 1.1699998 will be stored as 116 (rather than 117), with a **scale** value of 2 and a **round** of 0.0.

If the converted value is too large to store in the **target** field, an "all nines" result of the appropriate number of digits and sign is stored. For example, if the value should be negative and have six digits, but was too large, the result would be −999999.

RETURN VALUE

None.

PORTABILITY

pdset is not portable.

EXAMPLE

Take three percent of a packed decimal value in a structure passed from COBOL, storing the result in another structure item. Even though the input data is defined in COBOL as having two decimal places, it is processed by C without scaling, to avoid roundoff error.

```
#include <packed.h>

   struct pddata {
      char income [6];
      char outgo [6];
```

pdset **Assign Double Value to Packed Decimal**
(continued)

```
        /* Expected COBOL data declarations:
           INCOME PIC 9(9)V99 COMP-3.
           OUTGO  PIC 9(9)V99 COMP-3.   */
    };

void percent3(struct pddata *data)
    {
        double cents;

        cents = pdval(&data->income, 0);
        cents * = 0.03;   /* compute 3 percent */
        /* store in record after rounding */
        pdset(&data->outgo, cents, 0, 0.5);
        return;
    }
```

SEE ALSO

pdval

pdval Convert Packed Decimal to Double

SYNOPSIS

```
#include <packed.h>

double pdval(char (*source) [], unsigned scale);
```

DESCRIPTION

The **pdval** macro is called to convert a packed decimal value (stored in a character array) to **double**. The **source** argument should be the address of the array containing the packed decimal data. The maximum source size is 8 bytes (15 digits). Note that this argument must be a pointer to a **char** array, not a **char *** pointer or a **char** array. The **scale** argument specifies a scale factor. The converted **source** is multiplied by **pow(10.0, -scale)** during processing. The **scale** value must be between 0 and 15.

RETURN VALUE

The return value is the value contained in the **source** argument, appropriately scaled. If the **scale** value is invalid, the constant **HUGE_VAL** is returned.

ERRORS

If the **source** array does not contain valid packed decimal data, an 0C7 ABEND results.

PORTABILITY

pdval is not portable.

EXAMPLE

Print the value of a packed decimal value passed from COBOL.

```
#include <packed.h>

void printamt(char (*amount) [6])
      /* Expected COBOL data declaration:
         AMOUNT PIC 9(9)V99 COMP-3.   */
   {
      double dollars;

      dollars = pdval(amount, 2);  /* convert to dollars & cents */
      printf("Amount is $ % 12.2f\n", dollars);
      return;
   }
```

SEE ALSO

pdset

13 C Varying-Length String Macros

Introduction

C functions that are called from PL/I or Pascal may frequently be required to process character strings that are preceded by a `short` length field. The header file `<vstring.h>` defines a number of useful macros for programs that need to process this kind of data.

As a debugging aid, the `printf` format `%V` is also provided. This format behaves the same as the `%s` format, but it expects the corresponding argument to be a pointer to a PL/I or Pascal format varying-length character string.

Varying-Length String Macro Descriptions

The macros defined by `<vstring.h>` are as follows:

`VSTRING(max)` generates a structure type designator for a varying-length character string whose maximum length is `max`.

`vstrlen(vstr)` returns the current length of the varying-length character string `vstr`.

`vstrmax(vstr)` returns the maximum length of the varying-length character string `vstr`.

`vstrinit(cons)` generates an initializer for a varying-length character string, where `cons` is a string literal that is to provide the initial value.

`vstrcpy(vstr, str)`
copies a null-terminated string `str` to a varying-length string `vstr`. No check is made to avoid copying too much data. It returns the value of its first argument.

`strvcpy(str, vstr)`
copies a variable-length string `vstr` to a C character array `str` and adds a terminating null character. No check is made to avoid copying too much data. It returns the value of its first argument.

Examples Using Varying-Length String Macros

Some examples of varying-length string macros in a function that might be called from PL/I are given below:

```
#include <vstring.h>
#include <string.h>

typedef VSTRING(20) vstr20;
typedef VSTRING(40) vstr40;
extern char suffix [21];

void example(vstr20 *instr, vstr40 *outstr)
   {
      char cbuf [41];

      printf("String length = %d, data = %V\n",
             vstrlen(*instr), instr);
      strvcpy(cbuf, *instr);   /* concatenate instr */
      strcat(cbuf, suffix);    /* and suffix        */
      vstrcpy(*outstr, cbuf);  /* into outstr       */
      return;
   }
```

Part 2
Extending SAS/C ILC

Chapters

This part describes how to use the SAS/C ILC feature to communicate with less widely used languages such as Ada or SNOBOL. SAS/C ILC does not support these languages directly; however, it permits support for such languages to be added by a knowledgable user.

Note that this facility has two audiences. One audience is the *user* of an interface to a user-supported language. This audience need only read Chapter 14. The knowledge required to use the interface to a user-supported language is considerably less than the knowledge required to implement it. Unless stated otherwise in your interface documentation (supplied by the implementor), knowledge of the internals of C, or of the target language, should not be required.

The other audience is the *implementor* of the interface to a user-supported language. Before attempting to extend ILC in this fashion, you must have an expert knowledge of your language and its implementation. You must also be familiar with SAS/C and assembler language.

It is very important that both users and implementors be well-versed in the material in Part 1 of this book before reading Part 2.

14 Using ILC with a User-Supported Language

Introduction

This chapter provides a general set of directions for using a user-supported interface to a non-standard language. It must be augmented by the documentation supplied by the implementor of the interface to your language. Implementors will find this chapter helpful in determining what to include in this documentation.

Before reading this chapter you should be familiar with C, your language, and the documentation produced by the implementor of the interface. Knowing one of the standard languages (FORTRAN, COBOL, PL/I, or Pascal) is also helpful. This allows you to use examples in that language as models for your language, at least in any areas where the two languages are similar. Be sure to read Chapters 1 through 3, 8, 9, and at least one of Chapters 4 through 7 before reading this chapter.

Because SAS/C ILC puts no restrictions on the syntax or semantics of a user-supported language, it is impossible to give useful examples in this chapter. Refer to your interface documentation for such examples.

Language Names

Each user-supported language has two names, assigned by the interface implementor, a *framework name* and an *ILCLINK name*. The framework name is the name specified by the first argument in calls to the framework routines **mkfmwk** and **CFMWK**. Usually, this name is the standard language name, for example, "ADA" or "MODULA2." (A name longer than eight characters must be abbreviated.)

The ILCLINK name is used in ILCLINK control statements. This ILCLINK name may differ from the framework name for one of two reasons. First, only the first three characters of this name are significant, so the name might have to be abbreviated to assure uniqueness. For instance, "LISP" and "LISA" might have to be contracted to "LSP" and "LSA," respectively. Second, if different versions of the same language have different linking requirements, there might be more than one ILCLINK name for the same framework name. For instance, if two versions of Ada are in use, they might be assigned ILCLINK names of "AD1" and "AD2," even though the framework name for both is "ADA." Your ILC interface documentation should include the exact names assigned to your language.

Creating and Deleting the User-Supported Language Framework

Just as with a standard language, you must create the framework for a user-supported language before calling a routine in that language. You must also delete the framework after all calls to the language have completed.

To create a user-supported language framework from a C function, call the `mkfmwk` function. The first argument to `mkfmwk` must be the language's framework name. You can pass run-time options to the other language if it supports them. (See your interface documentation for any applicable restrictions.) The token returned by `mkfmwk` is later passed to `dlfmwk` when you delete the framework. You may also need to use this token when you call a routine in the other language from C, as described in **Passing a Language Token to a __foreign Routine** later in this chapter.

Except for the differences mentioned above, the descriptions of `mkfmwk` and `dlfmwk` in Chapter 11, "ILC Framework Manipulation Routines," are completely applicable to user-supported languages.

Creating and Deleting the C Framework

Just as with a standard language, when you call C functions from a user-supported language, you must first create the C framework. You must also delete the C framework after all calls to C functions have completed.

To create the C framework from a routine in your language, call the `CFMWK` routine. `CFMWK` expects to receive a call-by-reference argument list. Your interface documentation should describe exactly how to generate this kind of argument list in your language, as well as the correct data type and format for each argument. The token returned by `CFMWK` is later passed to `DCFMWK` when you delete the framework.

In general, the description of `CFMWK` in Chapter 11 applies to user-supported languages as well. For instance, `CFMWK` always accepts four arguments, and the same bits are are always used in the third argument to select run-time options. However, the Chapter 11 description cannot specify the syntax of the call, or the exact data types of the arguments. (For instance, you cannot assume that a string literal can be passed to specify the language name.) Supplying this kind of information is the responsibility of your interface implementor.

`CFMWK` considerations specific to user-supported languages are as follows:

- The first argument to `CFMWK` must be a string containing the framework name for the calling language, followed by a period.
- Some languages may require you to specify the `=multitask` option in the third argument to `CFMWK`. Your interface document should note whether this is required.

Calling C from a User-Supported Language

There are three requirements for calling a C function from a user-supported language after the C framework has been created:

- The C function must be compiled with the `INDep` compiler option.
- The function must be called using IBM 370 standard linkage.
- An argument list must be passed in the format expected by the C function. If the calling language supports call by reference, then you should be able to declare each argument in the C function to be a pointer to an appropriate kind of data. (See your interface documentation for a list of corresponding data types for C and your language.) If you cannot use call by reference, you must ensure that the argument list generated by your language's compiler and expected by the C function are identical. Details and any applicable restrictions should be included in your interface documentation.

Ideally, you should be able to write C functions that return a value to a calling routine in your language. This might be limited to certain kinds of return values, such as scalars only. Any such restrictions should be mentioned in your interface documentation.

Calling a User-Supported Language from C

There are three requirements for calling a user-supported language routine from C after the called language's framework has been created:

- The called routine must be declared in C using the keyword `__foreign`, to inform the compiler that it is in a user-supported language.
- The first argument passed from C must be the called language's token, as returned by **`mkfmwk`**. (In some cases, the token can be omitted, as described below.) This token will be completely processed by the SAS/C library, that is, it will not be present in the argument list received by the `__foreign` routine.
- The remaining arguments must be of types supported by the user-supported language interface, corresponding to the types expected by the called routine. You may be able to use data type conversion macros to pass arguments for which there are no corresponding C types.

When you call a routine in a user-supported language from C, you have more options for argument passing than you do for a call in the other direction because the C compiler knows that the routine to be called is in another language. Although much of the argument processing is language-specific, the following generalizations are valid:

- Any argument to your language that is a C pointer (other than a string literal) is passed unchanged to the called routine.
- Any argument that is not a pointer, or that is a string literal, is passed to your language in a manner appropriate to its type, as determined by the implementor of the interface. Passing an argument of an unsupported type results in a warning message, and the argument address is passed unchanged.
- Any argument that is a data type conversion macro call is passed to your language in an appropriate way, as determined by the

implementor of the interface. Incorrect results might be obtained if the macro is not used properly. Depending on the language, you might use either existing data type conversion macros, such as `_STRING`, or new macros developed by the implementor of the interface to support unique data types of your language.

More specific details on the handling of each of these cases should be in your interface documentation.

Ideally, you should be able to call a FUNCTION in your language that returns a value, and correctly receive the return value in C. This might be limited to certain kinds of return values, such as scalars only. Any such restrictions should be mentioned in your interface documentation.

Passing a Language Token to a `__foreign` Routine

The language token passed to a `__foreign` routine is used by the library to determine which language is being called because the `__foreign` keyword might apply to several languages. If only one high-level language other than C is in use, the token can be omitted.

Note: If you choose not to pass a language token to a `__foreign` routine, it is possible that the first argument will be misinterpreted as a token, with unpredictable results. This can only occur if the first argument is a pointer whose value has the high-order bit set. Because C pointers do not normally have this bit set, this is exceedingly unlikely.

If your language is the main language of the program, you may need to add a dummy call to `mkfmwk` for your language in your C code, since this function is the only way to obtain a language token. Note that such a dummy call requires a corresponding call to `dlfmwk` before the C framework can be terminated.

Using ILCLINK with a User-Supported Language

Whenever you use an ILCLINK control statement, such as the LANGUAGE or FIRST statement, that has a language name as an operand, you must specify the ILCLINK name of your language. Only the first three characters of that name are significant.

ILCLINK has no specific knowledge of languages other than FORTRAN, COBOL, PL/I, and Pascal. In particular, ILCLINK cannot determine the proper entry point for a program whose first routine is in a user-supported language. For this reason, you may not use a FIRST statement with an entry point specification of *. Your interface documentation should describe the correct entry point for a program whose main routine is written in your language. For details on using ILCLINK, see Chapter 8, "Linking Multilanguage Programs with the ILCLINK Utility."

15 User-Supported Language Implementation Background

Introduction

This chapter provides an overview of the key concepts of the SAS/C support for communication with user-supported languages. It is divided into the following sections:

- an overview of the required tasks for adding a user-supported language
- a discussion of language names and routine names
- an overview of the routines you will need to write in order to support an additional language
- an overview of the processes that implement ILC, how they communicate, and how they interact with your support routines.

Chapter 16, "Implementing ILC with a User-Supported Language," expands on these topics in great detail, but it is important that you understand the fundamental concepts before moving on.

In order to extend SAS/C ILC support, you must have in-depth knowledge of the internals of your target language and be familiar with C and assembler language. This chapter and Chapter 16 assume this knowledge.

Implementation Tasks

Adding support for a new language to SAS/C ILC can be divided into three tasks:

1. You must add your language to the library's supported language table. The supported language table is used by the library to record the names and attributes of all the supported languages. Each entry in the table is identified both by the name of the language and by a language number assigned when the table is generated.
2. You must write a number of support routines that will be called by SAS/C library routines to perform language-specific processing. For instance, you must provide a routine that creates the framework for your language.

3. You must provide documentation for users of your language. The importance of this step cannot be overstated. In the course of implementing the support, you will need to make many decisions about equivalent data types, data type conversion macros, restrictions, and so on. Because these decisions will be made by you, not by SAS/C ILC, they need to be documented by you as well.

Language Names and Routine Names

When you implement support for a new language in SAS/C ILC, you must assign three names to the language: a *framework name*, a *generic name*, and an *ILCLINK name*. The first two names are referenced by the entries in the supported language table, while the ILCLINK name is used only by ILCLINK.

The framework name is the name for the language as passed to the framework creation routines **mkfmwk** and **CFMWK**. This name can be up to eight characters long. It is stored in the supported language table to identify the table entry for the language.

The generic name for a language is used to construct ILC support routine names. This name is limited to three characters. ILC support routines have names of the form L$I-*nam*-*p*, where *nam* is the generic language name, and *p* identifies the purpose of the routine. For instance, if your language's generic name is MLA, the name of the routine you must write to create your framework would be L$IMLAF. The entry for your language in the supported language table will include external references to these routines.

The ILCLINK name for a language is used by ILCLINK to include the support routines for that language. These names can be up to eight characters long, but only the first three characters are significant. These names are used to generate the names of TEXT files under CMS, or PDS members under OS, containing the object code for your support routines. The same naming convention is used as for the routine names. For instance, if the ILCLINK name of your language is ML2, the file or member name for the object code for your Framework-routine should be L$IML2F.

If possible, the ILCLINK name and the generic name should be the same, but this may be impossible if you must support multiple versions of the same language. For instance, if you have different support routines for Version 1 and Version 2 of an Ada compiler but the differences do not affect the way your interface is used, you could use ILCLINK names of AD1 and AD2 to identify the two versions. Both the framework name and the generic name could be ADA. Note that in this case, the routine name and the filename are different; the object files named L$IAD1F and L$IAD2F each define a routine whose entry point name is L$IADAF. (The library's support for FORTRAN, COBOL, and PL/I, each of which has two major versions, is arranged in this way.)

Users of your support do not need to be aware of the generic language name. They do need the framework name to call the framework routines correctly and the ILCLINK name to code correct ILCLINK control statements.

Overview of User-Language Support Routines

When you add support for a new language to ILC, you must code a number of support routines. **Table 15.1** lists the required routines. Each routine must be in the indicated language, either assembler, C, or your target language. Each routine has an associated single-letter function code, which is used in forming the routine name, and has the name of the object file or member. For instance, the code for the Framework-routine is F; therefore, the routine's name will be L$I*nam*F, where *nam* is the generic name for your language.

Table 15.1
ILC Support Routines

Routine	Language	Description
Begin-	assembler	return control to C after framework initialization
Comm-	assembler	control calls between C and the target language
Framework-	assembler	create target language framework, passing run-time options
Locate-	assembler	locate target language control blocks
Main-	target language	cause framework initialization
Prep-	assembler	intercept calls to C from the target language
Quit-	target language	terminate target language execution
Xform-	C	transform C argument list into one for the target language

Under OS, the object code for these routines should be stored in the SASC.ILCOBJ (object module) and SASC.ILCSUB (load module) data sets. Under CMS, you can place the object code in TEXT files on any accessible minidisk. (Usually, you will either put them on the SAS/C minidisk or on a minidisk associated with your target language.)

The linkage conventions and functional details of these routines are described in detail in Chapter 16. Note that examples of these routines for the FORTRAN language are provided in source form in the SASC.SOURCE library under OS and in LSU MACLIB under CMS. Macros used by the samples are in SASC.MACLIBA under OS and LCUSER MACLIB under CMS. Because your language is probably different from FORTRAN, you will most likely have to write new routines rather than modify the examples. Nevertheless, they should probably prove useful in elucidating the ways such routines can be written.

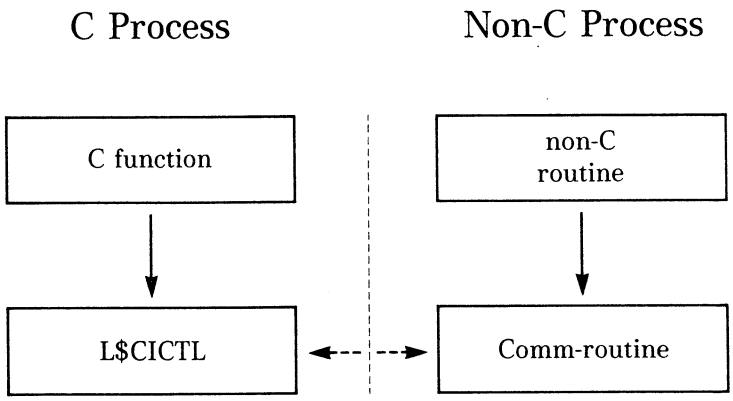
Processes and Process Communication

The best model for the SAS/C ILC support is that each language runs as a separate process. The processes communicate by sending messages to each other, rather than through a standard call-return mechanism. Except during framework creation or deletion, communication occurs between two specific routines, a C control routine (L$CICTL) that runs as part of the C process, and your Comm-routine (L$I*nam*C), that runs as part of your language's process. The Comm-routine uses the CCOMM assembler macro to send messages to and receive messages from the C process.

Control switches from one process to another when one process needs to ask the other to perform a specific action. For instance, the C process may send the non-C process a QUIT message to ask the non-C process to terminate, or a CALL message to ask the non-C process to call a non-C routine for C. Message transfers between processes, as well as process creation and deletion, are handled by a library component called the *ILC framework manager*.

Figure 15.1 illustrates the normal way in which the two processes communicate. The solid arrows represent subroutine calls, while the broken arrows represent inter-process communications.

Figure 15.1
Process Communication



Note that the role of L$CICTL and the Comm-routine above is to act as "stand-ins" for code in the other language. For instance, imagine the following calling sequence: X1 (non-C) -> Y1 (C) -> Y2 (C) -> X2 (non-C). The actual calling sequence in the non-C process is X1->L$I*nam*C->X2; the actual calling sequence in the C process is L$CICTL->Y1->Y2->L$CICTL. The Comm-routine takes the place of all called C functions in the non-C process, and L$CICTL takes the place of all user-language routines in the C process.

This situation is illustrated in **Figure 15.2.** The numbers indicate the order in which events occur.

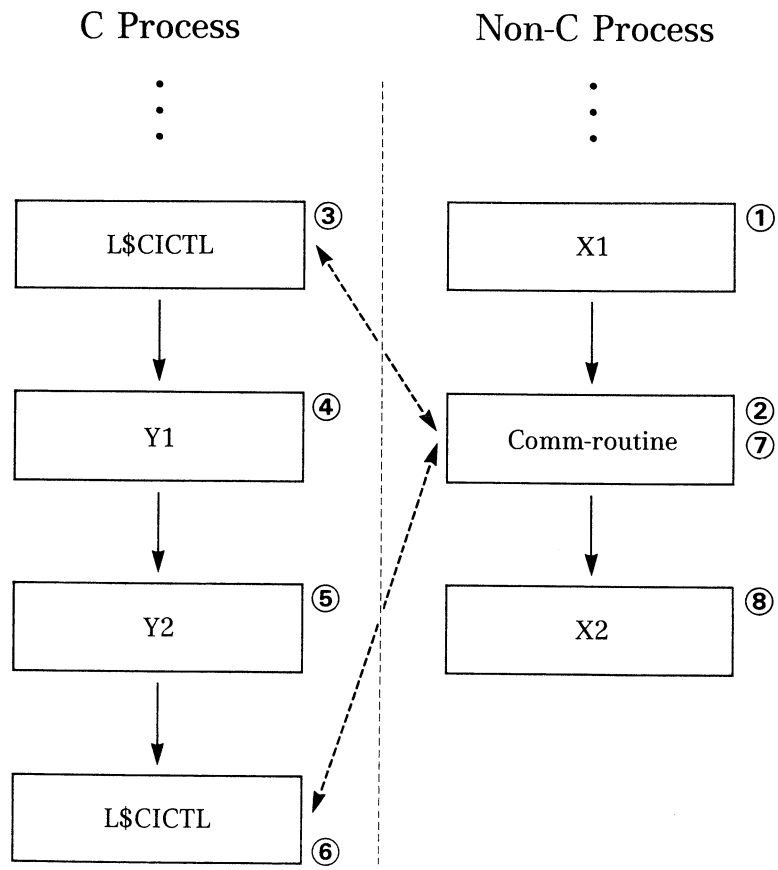


Figure 15.2
Process Communication Example

The order of events depicted in **Figure 15.2** is as follows.

1. The non-C routine X1 calls the C function Y1. This causes control to pass to your Comm-routine (for a detailed explanation, see Chapter 16.)
2. The Comm-routine sends a CALL request to L$CICTL, running in the C process.
3. L$CICTL calls the C function Y1.
4. Y1 calls the C function, Y2.
5. Y2 calls the non-C routine X2, which causes control to pass to L$CICTL.
6. L$CICTL sends a CALL request back to the non-C process.
7. The Comm-routine receives the CALL request.
8. The Comm-routine calls X2.

Now that you have the fundamental concepts of how SAS/C ILC works, you are ready to move on to actually implementing support for your language. The next chapter shows in detail how to accomplish this task.

16 Implementing ILC with a User-Supported Language

Introduction

This chapter describes how to extend SAS/C ILC support to include communication with additional languages. Before reading this chapter, you should have read Chapters 1 through 3, 8, 9, 11, 14, 15, and at least one of Chapters 4 through 7. (Chapter 4 is especially useful as an aid to understanding the FORTRAN sample routines.) In other words, you should be very familiar with the characteristics of SAS/C ILC and with C in general. You also need an expert-level understanding of your target language. If you do not have this prerequisite knowledge, you will probably have great difficulty understanding this chapter.

Before you can add support for your language to SAS/C ILC, you will need to understand the control flow of ILC processing. Then, you must

- update the supported language table
- implement the necessary support routines
- document the interface.

This chapter covers all these topics in detail, and also discusses some miscellaneous issues you will need to decide during development of your interface.

ILC Control Flow

Creating the Non-C Framework

When a multilanguage program begins execution, only a single process is active. If the initial language is C, a non-C process is created when a C function calls `mkfmwk`, as illustrated in **Figure 16.1.**

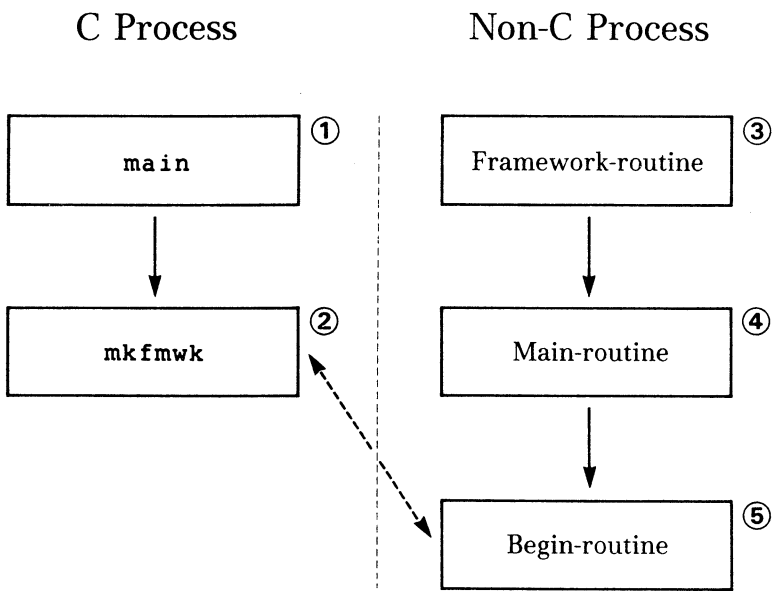


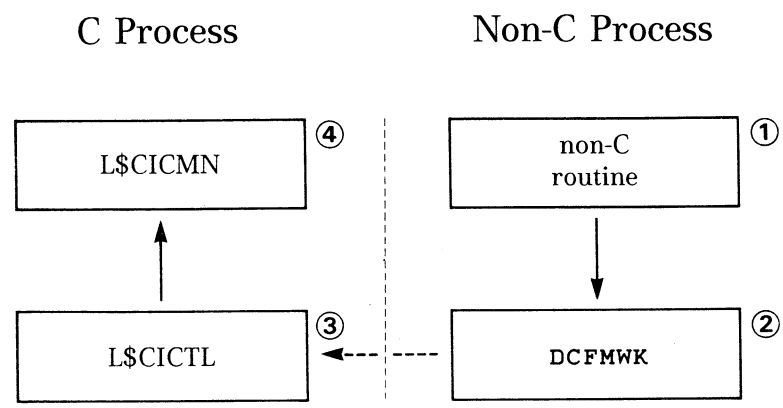
Figure 16.1 *Non-C Process Initialization*

1. A C function calls the `mkfmwk` function to create the non-C framework.
2. `mkfmwk` creates the non-C process.
3. The Framework-routine is called by the ILC framework manager to begin execution of the non-C process.
4. The Framework-routine calls your Main-routine. This causes your language's framework to be created. The Main-routine then calls the Begin-routine.
5. The Begin-routine uses the CCOMM macro to inform `mkfmwk` that the non-C framework has been established.

Creating the C Framework

When the initial language of a program is a non-C language, the effect of calling `CFMWK` is very similar to the previous case. See **Figure 16.2.**

Figure 16.2
C Process Initialization



1. A non-C routine calls the `CFMWK` routine to create the C framework.
2. `CFMWK` creates the C process.
3. The C framework is created by a call to L$CICMN, a special C main routine in the run-time library.
4. L$CICMN calls the ILC control routine L$CICTL, which sends a message to the non-C process to inform `CFMWK` that the C framework has been successfully established.

Calls from a Non-C Routine to a C Function

When a non-C routine calls a C function, the compiler-generated code for the called function transfers control to the entry point L$UPREP. When you use ILC, L$UPREP is an alias for the Prep-routine for the calling language. The Prep-routine locates the C Run-time Anchor Block (CRAB), and transfers control to the Comm-routine.

Note that your Prep-routine is used only when there are only two languages (including C) in use. For a program using more than two languages, a more complicated version of L$UPREP, supplied by the library, must be used. This more complicated version performs the same function (finding the CRAB and calling the Comm-routine) but with more overhead. (This version of L$UPREP has to make operating system calls to locate previously saved information about the current framework because it cannot determine which framework is active when it is called.)

The sequence of events for a call from a non-C routine to a C function is shown in **Figure 16.3.**

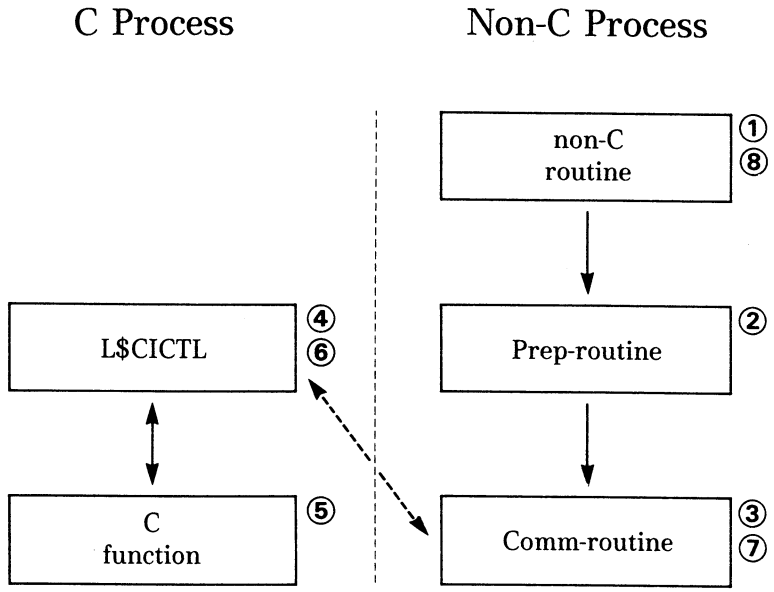


Figure 16.3 *Calls from a Non-C Routine to a C Function*

1. The non-C routine calls the C function.
2. The compiler-generated code for the C function calls L$UPREP, which is the name of the entry point to your Prep-routine. The Prep-routine locates the CRAB and calls your Comm-routine.
3. The Comm-routine sends a CALL message to the C process to request a call to the C function.
4. The CALL message is received by L$CICTL.
5. L$CICTL calls the C function, which executes and returns.

6. L$CICTL sends a RET message to the non-C process to inform it that the C function has returned.
7. The Comm-routine receives the RET message.
8. The Comm-routine returns to the non-C routine, bypassing the Prep-routine.

Calls from a C Function to a Non-C Routine

The flow of control when a C function calls a non-C routine is similar to the flow when a non-C routine calls C, but it is generally necessary to remap the argument list before the non-C routine can be called.

When the compiler processes a call to a routine declared (using the `__foreign` keyword) to be in another high-level language, it generates different code from the normal function call code. The generated code calls L$CILCL (an entry point to L$CICTL) and passes a structured list of tokens and pointers. The information passed to L$CILCL includes the address of the non-C routine to call, the types of all non-pointer arguments, and the type of return value expected. L$CILCL passes this list to your Xform-routine, which must build an output argument list in the format expected by the non-C language and return it to L$CILCL.

The flow of control between processes when a C function calls a non-C routine is illustrated in **Figure 16.4**. Note that because L$CILCL is an entry point to L$CICTL and most of the code is common to the two, the name L$CICTL is used for either, except in contexts that apply only to L$CILCL.

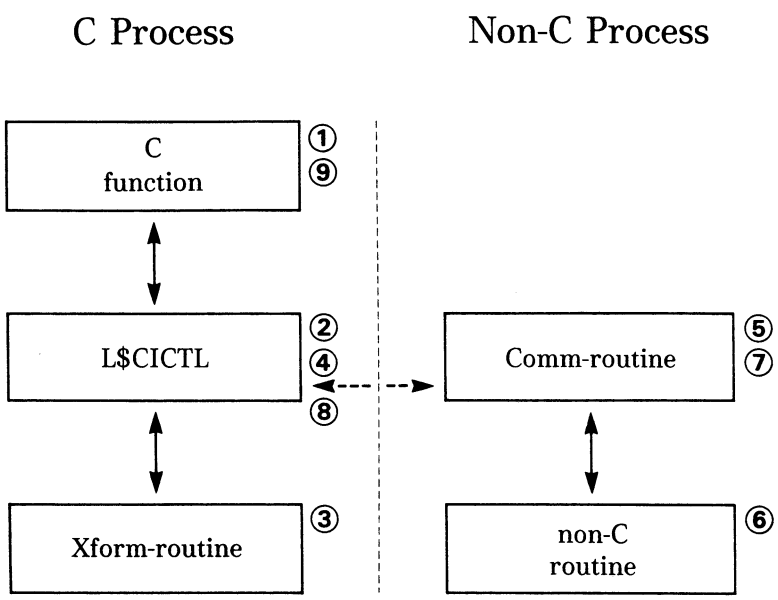


Figure 16.4
Calls from a C Function to a Non-C Routine

1. The C function calls the non-C routine, declared in C as `__foreign`. The compiler generates code to call L$CILCL.
2. L$CICTL calls your Xform-routine.
3. Your Xform-routine transforms the argument list built by the compiler into an argument list in your language's format and returns to L$CICTL.
4. L$CICTL sends a CALL message to the non-C process.
5. The CALL message is received by the Comm-routine.
6. The Comm-routine calls the non-C routine, which executes and returns.
7. The Comm-routine sends a RET message to the C process to inform it that the non-C routine has returned.
8. L$CICTL receives the RET message.
9. L$CICTL returns to the calling C function.

Normal Termination of the C Framework

Normal termination of the C framework occurs during a call to `DCFMWK`. The flow of control is shown in **Figure 16.5.**

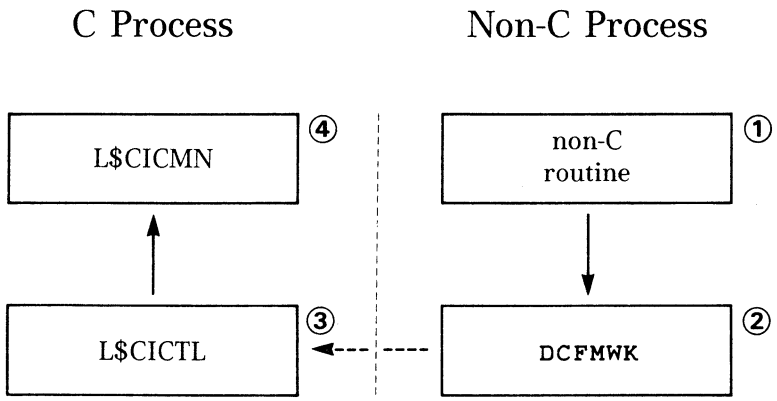


Figure 16.5
Normal Termination of the C Framework

1. A non-C routine calls `DCFMWK` to delete the C framework.
2. `DCFMWK` sends a QUIT message to the C process.
3. L$CICTL receives the QUIT message and returns to L$CICMN.
4. L$CICMN returns to the C run-time library, which deletes the C framework. The library then returns to the ILC framework manager, which deletes the C process.

Normal Termination of the Non-C Framework

Normal termination of the non-C framework occurs during a call to `dlfmwk`. The flow of control is shown below in **Figure 16.6.**

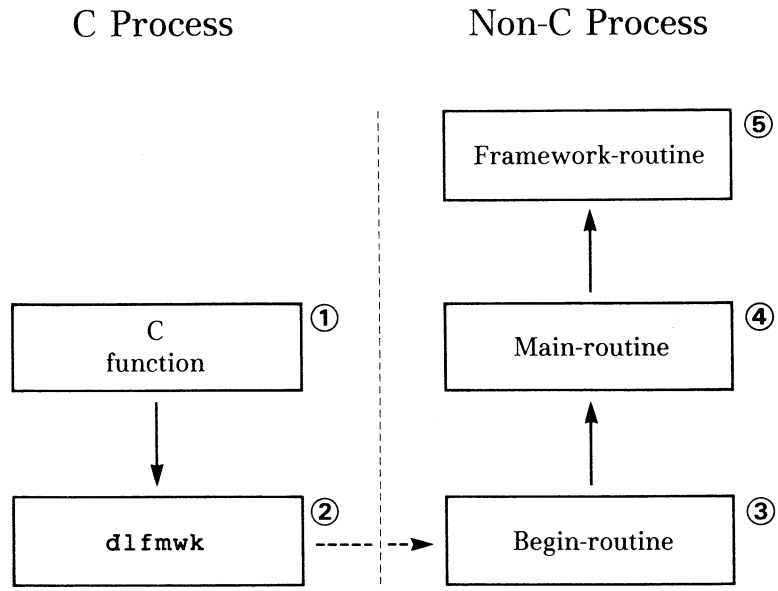


Figure 16.6
Normal Termination of the Non-C Framework

1. A C function calls `dlfmwk` to delete the non-C framework.
2. `dlfmwk` sends a QUIT message to the non-C process.
3. The Begin-routine receives the QUIT message and returns to the Main-routine.
4. The Main-routine returns to the Framework-routine, causing the non-C run-time library to delete the non-C framework.
5. The Framework-routine then returns to the ILC framework manager, which deletes the non-C process.

Unexpected Termination of the C Framework

If the C framework terminates while other language frameworks are active, for instance by a call to `exit`, the library must terminate all the other frameworks. This is implemented as shown in **Figure 16.7.**

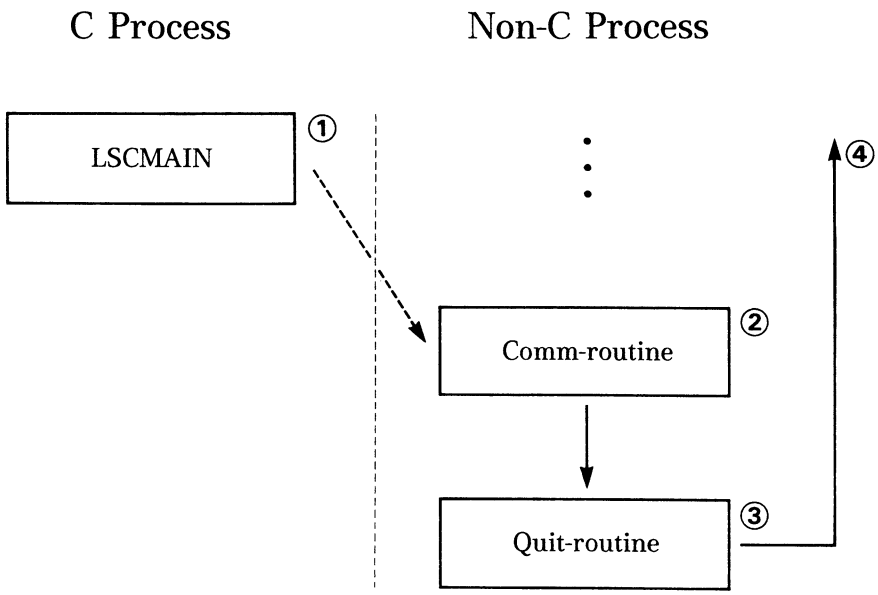


Figure 16.7 *Unexpected Termination of the C Framework*

1. The C library's initialization/termination routine, LSCMAIN, checks during termination for any active non-C frameworks. If one is found, it sends a QUIT message to the non-C process.
2. The Comm-routine receives the QUIT message and calls your Quit-routine.
3. The Quit-routine uses a facility in your language to terminate execution. (For instance, in FORTRAN a STOP statement is used.)
4. The framework is deleted by your language's run-time library, after which the ILC framework manager deletes the non-C process.

Unexpected Termination of the Non-C Framework

If a non-C framework terminates while the C framework is still active, the library must terminate the C framework. This is somewhat more complicated than the previous situation because control flow for termination of a non-C framework does not cause any SAS/C code to be invoked, since the other language's library is unaware of the existence of the C framework.

In order to handle this situation, the C library obtains the address of the save area for the routine (usually the operating system) that called the first non-C routine. It obtains this address by calling the Locate-routine, which loads this save area from a location defined by the target language implementation. (For instance, in PL/I this save area is accessed from the TESA field of the PL/I TCA, which is addressed via register 12.)

After obtaining this save area address, the library copies the register 14 value from the save area and replaces it with the address of the library routine L$CIQIT. This means that, if the non-C framework terminates unexpectedly, processing occurs as shown in **Figure 16.8.**

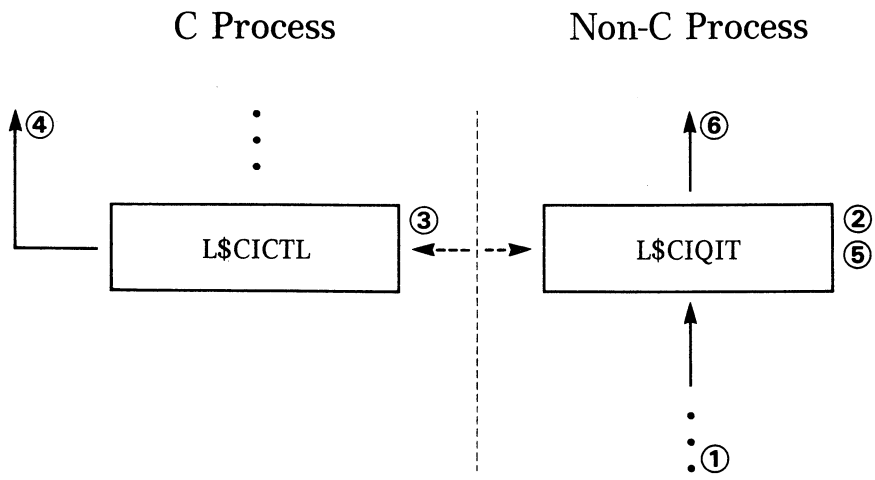


Figure 16.8
Unexpected Termination of the Non-C Framework

1. After your language's run-time library has terminated its framework, it returns control to L$CIQIT rather than to its caller.
2. L$CIQIT sends a QUIT message to the C process.
3. L$CICTL receives the QUIT message and calls the library **exit** function to terminate the C framework.
4. After the C framework has been deleted, the ILC framework manager deletes the C process and informs L$CIQIT that the process has terminated.
5. L$CIQIT loads the original return address into register 14.
6. L$CIQIT returns to the caller of the non-C language.

Note that the above technique assumes that a program in your language is always called using 370 standard linkage, with a return address in register 14. This must be true if programs in your language can be called directly from the operating system, even if your language uses non-standard linkage internally.

Some languages may not provide a way for the Locate-routine to locate the correct save area. In this case, the Locate-routine may return zero as the save area address. If the Locate-routine returns zero, the C library simply will not attempt to handle unexpected terminations of the non-C framework. In this case, a number of errors may occur, depending on circumstances, due to the inability to delete or clean up the C framework. For instance, memory allocations might fail because memory used by C has not been freed.

Updating the Supported Language Table

The supported language table source is named L$IMIXD. As supplied, it contains a call to the LANGDEF macro defining the four standard languages:

```
LANGDEF (PLI,PLI,1,NO),
        (FORTRAN,FOR,2,YES),
        (PASCAL,PAS,3,NO),
        (COBOL,COB,4,YES)
```

These four operands are constant and must not be changed. To add support for user-supported languages, you simply add an operand to this macro call for your language, reassemble L$IMIXD, and replace the object module. (The object module is in the SASC.ILCOBJ and SASC.ILCSUB data sets under OS. Under CMS, the object module is in L$IMIXD TEXT on the minidisk on which the SAS/C software has been installed.)

Each operand of LANGDEF has the following format:

```
(framework-name,generic-name,language-number,ILCENTRY-used)
```

The operands are used as follows:

- The framework-name operand is the language name (a maximum of eight characters) as passed to **mkfmwk** or **CFMWK**.
- The generic-name operand is the three-character name used to name entry points to the language support routines.

- The language-number operand is a number between 5 and 255 identifying the language. This number is completely arbitrary, except that no two languages may have the same number.
- The ILCENTRY-used operand must be either YES or NO. This operand indicates whether the Begin-routine and Comm-routine use the ILCENTRY macro for storage management. See the descriptions of these routines later in this chapter for information on ILCENTRY.

Implementing the Support Routines

This section describes implementation details for each of the support routines outlined previously. Information on data type conversion macros, including how to write additional ones, is presented in conjunction with the Xform-routine. Overview information on processing return values from routines in your language is also presented in the Xform-routine section.

You may wish to have listings of the sample FORTRAN support interface routines available as you read these specifications.

Control Block Location (the Locate-routine)

The Locate-routine, named L$I*nam*L, where *nam* is your language's generic name, must be written in assembler language. It is called by the library to obtain two addresses:

- the address of a word in which the CRAB address can be stored. This word is called the *CRAB address word*. It must contain zeroes before the C framework has been created. Considerations for selecting a location for this word are given in the next section.
- the address of the save area addressed by register 13 at the time the first routine in your language was called. This area is called the *original save area*. See **Unexpected Termination of the Non-C Framework** earlier in this chapter for details on the use of this information. Note that you can return a zero save area address if the original save area cannot be located.

When the Locate-routine is called, register 1 addresses an 8-byte area in which return information can be stored. Registers 2 through 13 contain the same values they would if the Locate-routine had been called directly by a caller in your language. The values in these registers may be useful in locating framework control blocks for your language.

When the Locate-routine returns, the first word of the area addressed by register 1 on entry must address the CRAB address word, and the second word must address the original save area or contain zeroes.

Selecting a CRAB address word

The CRAB address word that is returned by the Locate-routine has the following requirements:

- It must contain zeroes before the C framework has been created.
- It must be modifiable by the SAS/C library, and never be modified in any other fashion.

- It must occupy a different location for each distinct framework. (Since a framework can be created only once for a single program invocation, the only way to create multiple frameworks for the same language is to run several multilanguage programs simultaneously.)
- For performance reasons, it should be easy to access whenever your language's framework is active. (Your Prep-routine and Begin-routine normally also require access to this word.)

If reentrancy is not a requirement, you should probably define a unique CSECT as the CRAB address word. (This technique is used by the FORTRAN example because FORTRAN does not support completely reentrant programs.)

If reentrancy is a requirement, this technique cannot be used because the library stores into the CRAB address word. For use in a reentrant application, the CRAB address word must be located in dynamically allocated memory. An ideal location, if your language supports it, is in a *user word* defined by your language's run-time library. For instance, the Pascal ILC implementation uses a user word defined by Pascal in its PCWA control block. If no user word is available, you may be able to *borrow* a word in a control block associated with some unusual language feature (for example, multitasking or telecommunications), and publish a restriction that the feature for which the field is normally used is not available. You should be familiar enough with your language's implementation to know that use of this word will not produce any undesirable side-effects.

Framework Generation (the Framework-routine)

The Framework-routine, named L$I*nam*F, where *nam* is your language's generic name, must be written in assembler language. It is called by the library framework manager to create your language's framework. See **Figure 16.1** for an illustration of the position of the Framework-routine in the task of creating a non-C framework.

The framework for your language will technically be created by your language's run-time library. The purpose of the Framework-routine is to

- create an argument list in the format required by your language. The argument list should specify any run-time options requested by the call to **mkfmwk**, and should make the CRAB address available to the Main-routine.
- create your language's framework by passing control to the Main-routine, a main routine written in your target language. Depending on the conventions of your language, the Framework-routine might call the Main-routine directly, or it might call an intermediate library routine. For instance, the sample FORTRAN routine calls the Main-routine directly, while the corresponding PL/I routine calls the PL/I library routine PLISTART, and PLISTART calls the Main-routine.

On entry to the Framework-routine, register 1 addresses a two-word argument list. The first word contains the address of the CRAB, and the second word addresses a null-terminated string containing the run-time options for your language requested in the program's call to **mkfmwk**. The Framework-routine must make the CRAB address available to the Main-routine and must pass the run-time options to

the run-time library for your language. For a typical language, you perform both functions by building a string containing both the run-time options and the CRAB address, separated by a language-defined separator, and then pass this string to your Main-routine.

It is usually wise to convert the CRAB address to a printable form (such as hexadecimal or zoned decimal) before copying it to the argument string because some languages tokenize or otherwise modify their argument list if it contains special characters that might accidentally be present in binary data.

The sample FORTRAN routine, L$IFORF, does not conform to this pattern because FORTRAN does not permit the main program to receive arguments. The CRAB address is therefore stored in a CSECT that can be accessed by the FORTRAN Main-routine, as a COMMON block.

Upon termination of your language's framework, control returns to your Framework-routine. You should release any dynamically allocated memory and return with the value in register 15 that was returned to you by the Main-routine.

The Main-routine

The Main-routine, normally named L$I*nam*M, where *nam* is your language's generic name, is called, directly or indirectly, by the Framework-routine to cause your language's framework to be created. This routine must be written in your target language and must be defined as a main routine according to the conventions of your language. Note that there is no specific code in the Main-routine to create the framework. Framework creation should happen automatically when the Main-routine is entered, because it is defined as a main routine.

See **Figure 16.1** for an illustration of the position of the Main-routine in the task of creating a non-C framework.

The only explicit action required of the Main-routine is to call the Begin-routine, passing it the CRAB address. The CRAB address is normally passed to the Main-routine as its only argument by your Framework-routine, as described previously.

Some languages may not permit you to name the Main-routine according to the conventions. For instance, some versions of FORTRAN restrict you to six-character identifiers, and COBOL does not allow the $ symbol. You can assign a different name to this routine if necessary, because it is referenced only by the Framework-routine. (For instance, the sample FORTRAN routine is named L$IFOM.) However, the object file must conform to the naming conventions so it can be successfully included by ILCLINK.

The Quit-routine

The Quit-routine, normally named L$I*nam*Q, where *nam* is your language's generic name, is called by your Comm-routine to terminate your language's framework. It should be written in your target language. See **Figure 16.7** for the position of the Quit-routine in the process of handling unexpected termination of the C framework.

The Quit-routine receives a single argument, which is an integer specifying the return code value that should be returned in register 15 after the framework is terminated. The Quit-routine should use whatever statements are necessary in your language to set the return code as requested, and then terminate execution.

Two versions of a FORTRAN Quit-routine are provided, one for VS FORTRAN Version 2, and one for previous versions. (Previous versions of FORTRAN did not provide a way to specify a variable return code, so the version 1 Quit-routine supports only some return codes.)

Some languages may not permit you to name the Quit-routine according to the conventions. For instance, some versions of FORTRAN restrict you to six-character identifiers, and COBOL does not allow the $ symbol. You can assign a different name to this routine if necessary, because it is referenced only by the Comm-routine. (For instance, the sample FORTRAN routines are named L$IF1Q and L$IF2Q.) However, the object file must conform to the naming conventions so it can be successfully included by ILCLINK.

The Pre-Prolog (Prep) Routine

The Prep-routine, named L$I*nam*P, where *nam* is your language's generic name, must be written in assembler language. It is called in place of the C library prolog by any C function compiled with the `INDep` compiler option. The entry point to the Prep-routine must be named L$UPREP. Note that the Prep-routine is used only when a program that uses exactly two languages (C and your language) is executed. If more than two languages are in use, a language-independent version of L$UPREP is used instead. The Prep-routine is responsible for routing control to your Comm-routine when a routine in your language calls a C function.

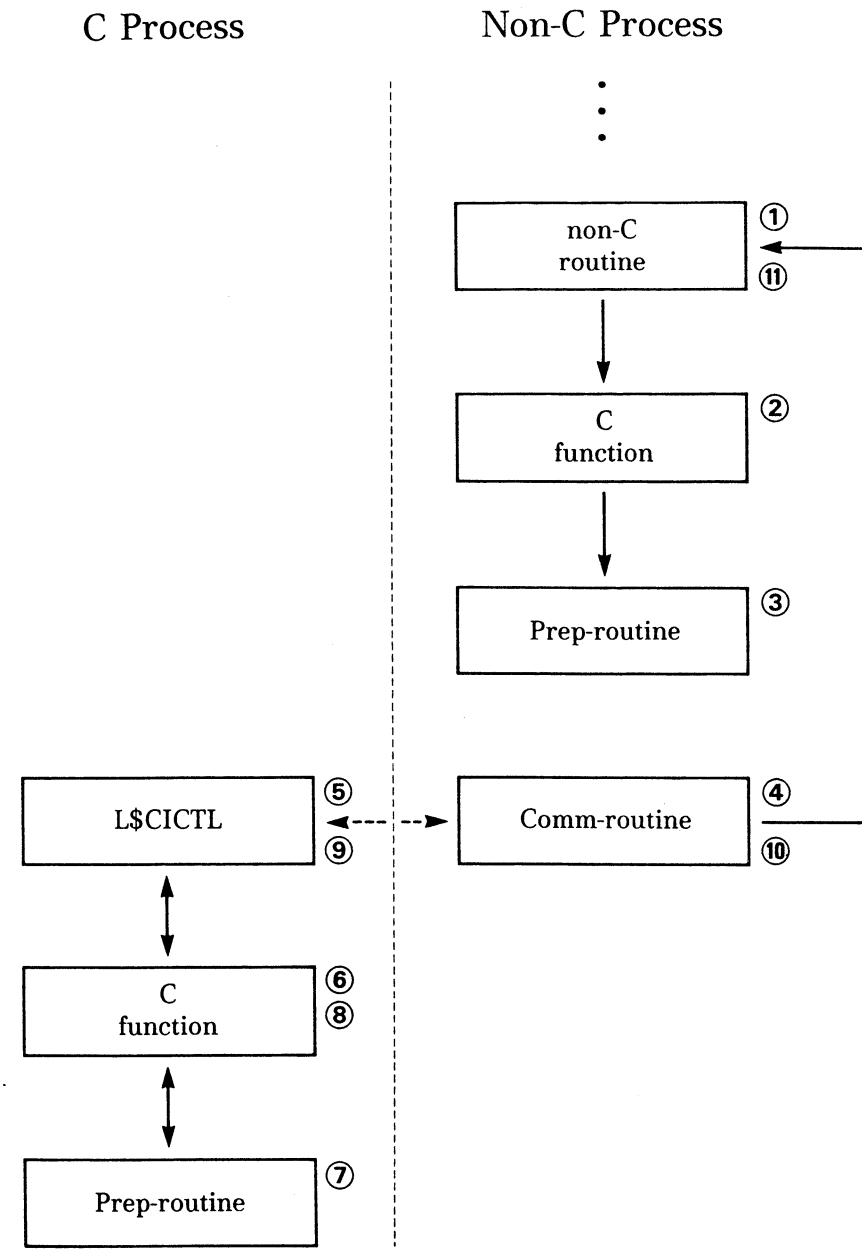
See the *SAS/C Compiler and Library User's Guide* for general information about L$UPREP and the `INDep` compiler option. The use of L$UPREP requires that your language follow some of the 370 standard linkage conventions for its calls, notably,

- register 15 must contain the address of the called routine
- register 14 must contain the return address
- register 13 must address a 72-byte save area in which registers can be saved.

When L$UPREP is entered, all registers except register 14 have the same contents as when the C function was called. Register 14 serves as a base register. The registers have already been saved by compiled code in the save area addressed by register 13, and you must not save them again, or information will be lost. Note that the Prep-routine may be entered while either the C process or the non-C process is running.

The flow of control through the Prep-routine is illustrated in **Figure 16.9.**

Figure 16.9
The Prep-routine



The flow of control represented in the diagram is as follows:

1. A non-C routine calls a C function. The non-C framework and the non-C process are active at this time.
2. Generated code in the **INDep**-compiled C function saves registers and transfers control to the L$UPREP entry point of the Prep-routine.
3. The Prep-routine, as described below, transfers control to your Comm-routine, passing it the address of the called function and its argument list.
4. The Comm-routine sends a CALL message to the C process.
5. The CALL message is received by L$CICTL, which calls the C function passing the requested argument list.
6. Generated code in the **INDep**-compiled C function saves registers and transfers control to the L$UPREP entry point of the Prep-routine.
7. The Prep-routine determines that the C framework is active and, therefore, returns control to the C function.
8. The C function executes and returns to L$CICTL.
9. L$CICTL sends a RET message to the non-C process to inform it that the called function has returned.
10. The Comm-routine receives the RET message.
11. The Comm-routine returns. Because of the linkage used by the Prep-routine, return is made directly to the original non-C function, not to the Prep-routine.

You should implement the above flow of control in the following steps:

1. Check for the constant 'CSA' in the first three bytes of the area addressed by register 13. If this constant is present, the C function was called by another C function, and no ILC processing is required. In this case you should either return directly to the called function or to the C library prolog, as shown in the sample code at the label PREPOK.
2. Locate your language's CRAB address word (as defined by your Locate-routine), and put its address in register 1. (Since step 1 has established that the calling routine is not in C, you can assume that your language's framework is active, and that any registers with defined meanings in that framework are correct.) If the CRAB address word contains zeroes, the C framework has not yet been created. In this case you should load the address of the library routine L$CINCE into register 15 and branch to register 15. This routine writes an appropriate diagnostic and then issues a user ABEND 1234.
3. If the CRAB address word contains a nonzero value, load register 15 with the address of your Comm-routine, reload register 14 from the save area, and branch to register 15. Before you call the Comm-routine, you may want to copy the saved register 15 and register 1 values to the CRABDWK field of the CRAB. This will be necessary if you have to save registers in the register 13 save area before allocating an area of your own in which to save these values. See **Comm-routine memory management** later in this chapter for more information on this situation.

Because you reload register 14 before calling the Comm-routine, control does not return to your Prep-routine when the Comm-routine's processing is complete.

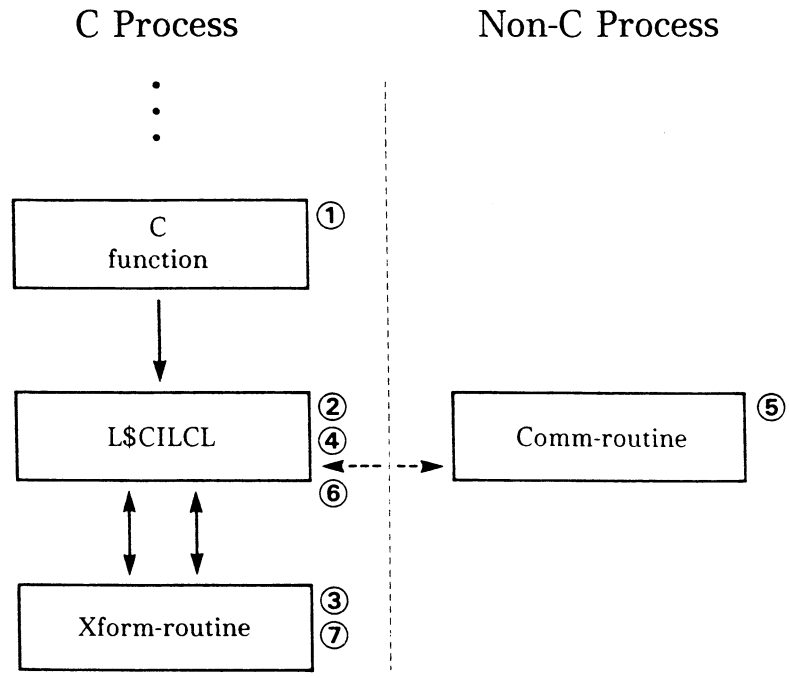
Note that these linkage conventions must be followed exactly, or the results are unpredictable. This is required so that the same results will be obtained whether your Prep-routine or the language-independent version of L$UPREP is used.

Argument Transformation (the Xform-routine)

The Xform-routine, named L$I*nam*X, where *nam* is your language's generic name, is called by L$CICTL when a C function calls a routine in your language. Its purpose is to transform the argument list generated by the C compiler for a routine in your language into one that can be passed to the called routine. Unlike the other support routines, the Xform-routine runs as part of the C process and should be written in C. (It may also be written in assembler, subject to the rules for mixing assembler routines with C described in the *SAS/C Compiler and Library User's Guide*.)

The position of the Xform-routine in the task of calling a non-C routine from C is shown in **Figure 16.10.**

Figure 16.10
The Xform-routine



1. A C function calls a `__foreign` routine. (The compiler generates code to call the library routine L$CILCL.)
2. L$CILCL is entered and calls your Xform-routine, passing it the compiler-generated argument list.

3. Your Xform-routine transforms the argument list and returns the address of a list of arguments to be passed to the called routine. (The Xform-routine may also set a flag requesting to be called again after the called routine returns.) Control is returned to L$CILCL.
4. L$CILCL sends a CALL message to the non-C process, passing the address of the non-C function and of the argument list constructed by the Xform-routine.
5. The Comm-routine calls the requested routine. On completion of the called routine, it sends a RET message back to the C process.
6. L$CILCL receives the RET message and, if requested, calls the Xform-routine again.
7. On the second call, the Xform-routine may free the argument list that was passed to the called routine.

The linkage conventions for the Xform-routine are described in C as follows:

```
int L$I namX(unsigned *cargs, char *cret, char ***outargs,
             char **outret, int *after);
```

An illustration of the argument list structure is shown in **Figure 16.11.**

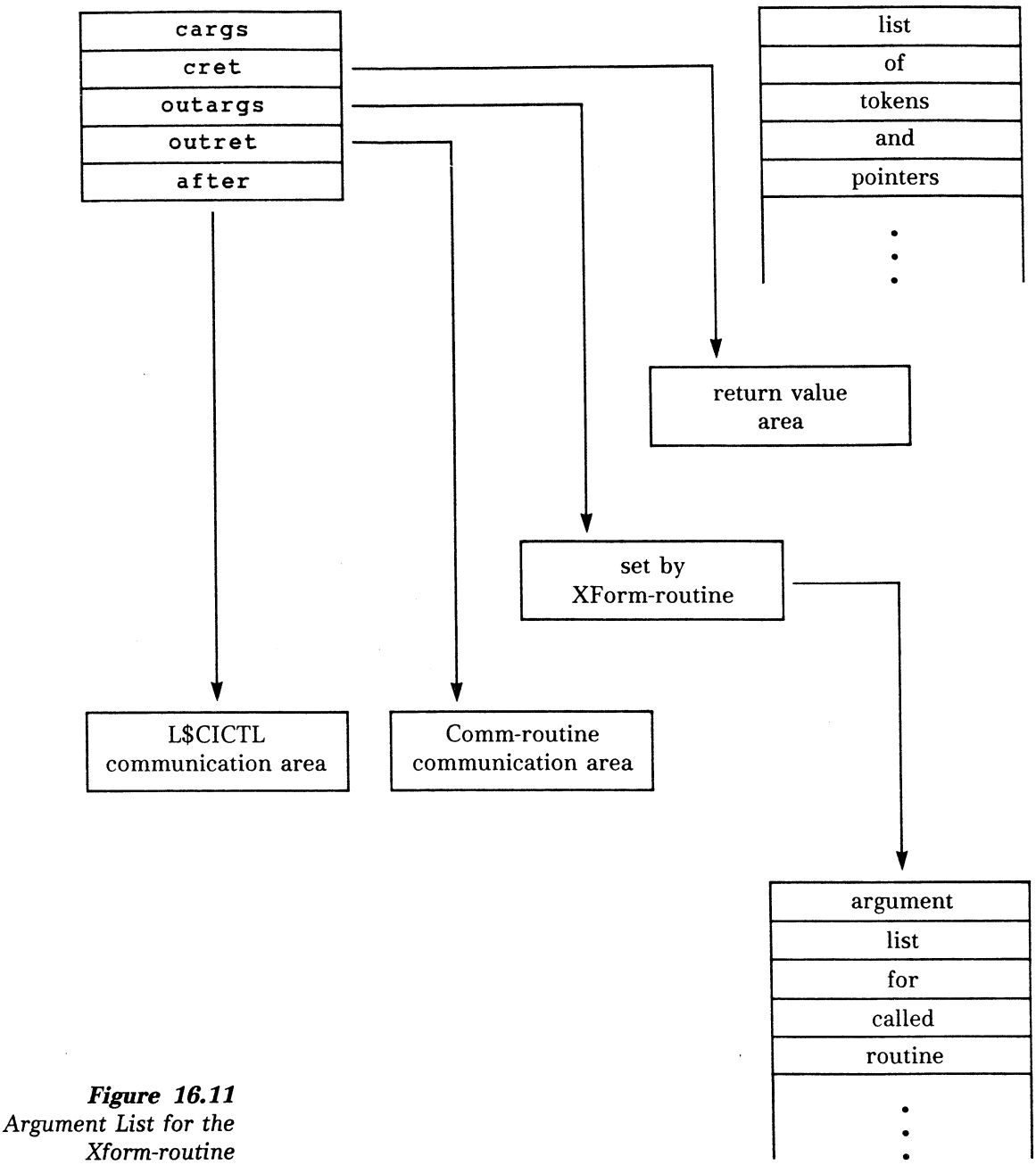


Figure 16.11
Argument List for the Xform-routine

A summary of the arguments is as follows:

`cargs`
: is a pointer to a list of tokens and pointers generated by the compiler that completely describes the call, including argument types, argument values, and the return value type. See below for a complete description of how to interpret this list. This argument, and the list it addresses, is input only; it must not be modified.

`cret`
: is a pointer to an area in which L$CICTL expects any return value from the called function to be stored. This argument is input only.

`outargs`
: is the address of a fullword in which the Xform-routine can store the address of the transformed argument list to be passed to the called routine in your language.

`outret`
: is the address of a fullword in which the Xform-routine can store a value to be passed to the Comm-routine. The word addressed by `outret` is normally used to communicate return-value information to the Comm-routine. See **Return value processing** later in this chapter.

`after`
: is a pointer to a fullword flag that distinguishes calls to the Xform-routine. The flag is set both by L$CILCL and by the Xform-routine. When the Xform-routine is initially called, the flag's value is zero. The Xform-routine sets the flag's value to nonzero to request another call after the called non-C routine has completed execution. If this flag is set on return from the Xform-routine, L$CILCL will call the Xform-routine again after the called routine has returned, with the flag containing the value stored by the previous call. This second call is normally used to free storage allocated at the time of the first call.

The Xform-routine should return zero if it was able to complete successfully. If it was unable to complete, for instance, because it was unable to allocate memory, it should return the value of `errno` associated with the failure. If the Xform-routine returns a nonzero value, L$CICTL writes an explanatory message and issues user ABEND 1235.

Interpreting the compiler's token list

This section describes the structure of the `cargs` argument passed to your Xform-routine. Your Xform-routine should include the header file `<ilctok.h>` in order to obtain definitions of the symbols used in this discussion.

The `cargs` pointer addresses a list of *tokens* and pointers. There are several types of tokens, identified by a *token id* stored in the first byte. A value in the `cargs` list that does not begin with a defined token id is a pointer, not a token. (Each token id has the 0x80 bit set. Since C pointers do not normally have this bit set, there is little chance of confusion between tokens and pointers.)

For most token types, the second byte of the token (called a *tag*) identifies a data type, and the last two bytes specify a length. The header file `<ilctok.h>` defines two macros: `Tag`, which combines a token id and a tag, and `Token`, which builds a token from a token id, a tag, and a length. These macros can be used to improve the readability of code that processes the tokens generated by the compiler. See the sample Xform-routine for examples.

The layout of the `cargs` list is as follows:

Element 0
: is the address of the routine in your language to be called. If this routine was declared in C as a function pointer rather than as a function, the 0x80000000 is set in this word. (See **Function Pointer Implementation** later in this chapter for more information.) You do not ordinarily have to process this element.

Element 1
: is a *return token* whose token id is named `TOK_RET`. The remainder of this token specifies the type of the return value, which will be `void` if the routine is not expected to return a value.

Element 2
: may be a *language token*. For a `__foreign` routine, a language token is present only if it was passed by the user in his function call. The token id for a language token is named `TOK_LANG`. The rest of the token contains information that is meaningful only to the library.

Last element
: is an *end-of-list token*, whose token id is `TOK_END`. The other bytes of this token are zeroes.

Other elements
: (after any language token and before the end of list token) are either *value tokens*, *macro tokens*, or pointers. A value token has a token id of `TOK_ARG`, and a macro token has a token id of `TOK_MAC`. A value token supplies information about the data type of a non-pointer argument to a non-C routine. Each value token is followed by a pointer to the argument described by the token. A macro token supplies information from a data type conversion macro. The order of macro tokens, and any following data, depends on the generating macro.

  The tag values in return tokens, value tokens, and macro tokens specify a data type or class of data types. The defined tag values and the corresponding C types are:

  `TYP_VOID`
  : specifies `void`. This is valid only for a return token.

  `TYP_INT`
  : specifies either an integer or a pointer type. (Because the compiler does not generate tokens for pointer arguments, a pointer type could occur only for a return or macro token.) The size of the associated value is specified in the token's length field. For example, `Token(TOK_ARG, TYP_INT, 2)` is a value token for a `short int` argument.

`TYP_FLT`
: specifies a floating-point type. The size of the associated value is specified in the token's length field. For example, `Token(TOK_RET, TYP_FLT, 8)` is a return token for a routine that returns a `double`.

`TYP_STRING`
: specifies a fixed-length string structure type (`struct {char text [n];}`). If the compiler option `VString` was not used, `TYP_STRING` is also used for an argument that is a string literal. Finally, a macro token with tag `TYP_STRING` is generated by the `_STRING` data type conversion macro. The size of the structure or string is specified in the token's length field.

`TYP_VSTR`
: specifies a varying-length string structure type (`struct {short len; char text [n];}`). The size of the structure (including the length field) is specified in the token's length field. For example, `Token(TOK_RET, TYP_VSTR, 22)` is a return token for a varying-string structure containing a `char` array of length 20.

`TYP_STRLIT`
: specifies a string literal argument in a function compiled with the `VString` compiler option. The pointer following the token addresses the string data, which is preceded by a compiler-generated halfword prefix containing the string length. The string's length, not including the prefix, is specified in the token's length field.

`TYP_STRUCT`
: specifies a structure or union type, other than one of the two string structure types. The size of the structure or union is specified in the token's length field.

`TYP_CFUNC`
: specifies a C function or function pointer argument, that is, one not declared with the `__asm` keyword or any of the ILC keywords such as `__pli` or `__foreign`. See **Function Pointer Implementation** later in this chapter for details.

`TYP_FFUNC`
: specifies a `__foreign` function pointer. The argument is a function pointer, not a routine name. See **Function Pointer Implementation** later in this chapter for details.

`TYP_AFUNC`
: specifies an `__asm` routine or function pointer, or a `__foreign` routine. The next element in the list is the address of the code for the routine. See **Function Pointer Implementation** later in this chapter for details.

`TYP_BIT`
: specifies a bit string or set argument. These tokens are generated by the `_BIT` and `_SET` data type conversion macros.

`TYP_DIM`
: defines a dimension of an array argument. These tokens are generated only by the use of the `_ARRAY` macros. The number of elements for this dimension of the array is specified in the token's length field. See **Output of the _ARRAY Macros** later in this chapter for further information.

`TYP_ELEM`
: defines an array element. The type of the element is not specified, but the size of the element in bytes is present in the token's length field.

`TYP_INV`
: defines an argument to a `__foreign` function that was found to be invalid during compilation. For instance, a `__pli` function pointer passed to a `__foreign` routine generates this sort of token.

user-defined tokens
: may have tags with numeric values from 24 through 30, for use by your own data type conversion macros. All other tag values are reserved for use in future SAS/C software releases.

Token list examples Following are two examples of calls to routines declared as `__foreign`, and of the token lists generated by the compiler and passed to the Xform-routine as a result. The first is a simple example:

```
__foreign void simple();

char val;

simple(&val, 14, "some text");
```

The token list generated by the compiler for this call contains the following words (in mixed assembler and C notation). The example assumes the compiler `VString` option was not used.

```
V(SIMPLE)
Token(TOK_RET, TYP_VOID, 0)
&val          (no preceding value token for a pointer argument)
Token(TOK_ARG, TYP_INT, 4)
@14           (a pointer to an integer 14)
Token(TOK_ARG, TYP_STRING, 9)
"some text"  (a pointer to the first character of the literal)
Token(TOK_END, 0, 0)
```

Here is a more complicated example, using a **typedef**:

```
typedef struct
{
   short len;
   char text [20];
} vstr20;
__foreign vstr20 (*complex)();
vstr20 ans;

struct node
{
   struct node *next;
   char *name;
} head;
double a [20][20];
char flags [5];

ans = (*foreign)(head, _ARRAY2(a, 20, 20),
                 _BIT(flags, 36));
```

The token list generated by this example is

```
A(X'80000000'+&complex)
Token(TOK_RET, TYP_VSTR, 22)
Token(TOK_ARG, TYP_STRUCT, 8)
&head
Token(TOK_MAC, TYP_DIM, 20)
Token(TOK_MAC, TYP_DIM, 20)
Token(TOK_MAC, TYP_ELEM, 8)
a             (a pointer to the first element)
Token(TOK_MAC, TYP_BIT, 36)
flags         (a pointer to the first element)
Token(TOK_END, 0, 0)
```

Adding your own data type conversion macros If your language has data types with no exact C equivalent, or several data types that all have the same equivalent, you may want to define your own data type conversion macros. (Guidelines to establishing equivalent data types are given in **Miscellaneous User-Supported Language Issues** later in this chapter.) Tag values 24 through 30 are reserved for your use.

Each data type conversion macro replaces its argument or arguments by two or more values separated by commas. This does not represent a C comma-expression and, therefore, must not be enclosed in parentheses. When a conversion macro is replaced, its output is interpreted as several arguments in the called routine's argument list. Usually, all generated values other than the last are tokens, and the last value is the address of the data to be passed. (See **Output of the _ARRAY macros** later in this chapter for another possibility.) Tokens must be cast to `void *` by the macro so that the compiler will insert them unchanged into the argument list. Note that the compiler generates value tokens for any non-pointer expression generated by your macros, so be sure to use an addressing operator (`&` or `@`) on such expressions unless these tokens are useful.

A typical conversion macro, `_BIT`, is implemented as follows:

```
#define _BIT(x, size) (void *) (0xd5040000+(size)),\
                      (void *)(x)
```

Thus, when the macro call `_BIT(ptr, 32)` appears in a `__foreign` argument list, it is replaced by two arguments: `0xd5040020` and `(void *)(ptr)`. Because both of these values are pointers, the compiler stores them unchanged in the argument list passed to L$CILCL. The first argument is a macro token with tag `TYP_BIT`, while the second is the address of the bit data.

Output of the _ARRAY macros If your language, like C, passes an array as a pointer to the first element, no data type conversion macro is required to pass an array correctly. For a language such as PL/I, where information on array bounds must be passed with the array, use of a data type conversion macro is generally necessary. Four such macros are defined in `<ilc.h>`: `_ARRAY`, `_ARRAY2`, `_ARRAY3` and `_STRARRAY`. You can define other such macros yourself, if necessary.

The output of the `_ARRAY` set of macros consists of a list of `TYP_DIM` tokens, one for each dimension, followed by a `TYP_ELEM` token giving the element size, followed by a pointer to the first array element. Note the macro's use of the compiler's builtin operator `__sizelem`, which returns the element size for an array argument (and is equivalent to `sizeof` for a scalar).

There may be array types that you want to support for which simply having the dimensions and element size is insufficient. The `_STRARRAY` macro can be used as a model for these cases. `_STRARRAY` assumes that its argument is an array of string structures, either fixed-length or varying-length.

This macro is defined in `<ilc.h>` as follows:

```
#define _STRARRAY(x, dim) (void *) (0xd5800000+(dim)), *(x)
```

The first item in its expansion is a macro token with tag `TYP_DIM`. The second item in the expansion is a fixed-length or varying-length string structure for which the compiler generates a value token of appropriate type, followed by a pointer to `*x`, that is, of course, identical to the address of the first element of `x`.

Return value processing

Two methods of processing return values are commonly used in 370 high-level languages: either the return value is returned in a register, or it is returned in memory. Which technique is used (and which register is used) can depend on the data type. For instance, FORTRAN returns integer and floating-point return values in registers, but returns strings in memory.

The implementation of SAS/C ILC as several communicating processes does not allow return values to be passed between languages in registers. It is always necessary for such return values to be passed through memory, whether or not this is required when only a single language is used. When SAS/C ILC is used, return values are always returned via a *return value area*, allocated by the process of the

calling language. For instance, in a call from C to a non-C routine, the return value area is allocated by the C process.

In the case of a call from a C function to a non-C routine, the return value is handled in the following way. (This list is intended to describe a typical case. Some languages may require additional or different steps, depending on their conventions.)

1. The code generated for the call to the non-C routine invokes L$CILCL. If the non-C routine returns a scalar, the compiled C code expects the result to be returned in a register. If the non-C routine returns a structure, the argument list is preceded by a pointer to a memory area in which the return value should be stored.
2. If the non-C routine returns a scalar type, L$CILCL allocates an 8-byte return value area. If it returns a structure, the return value area is the area passed by the calling C function.
3. The return value area address is passed to the Xform-routine as the `cret` argument. If the non-C language returns this type of data in memory, the Xform-routine generally needs only to put the address of the return value area at a defined location in the argument list.

   If the non-C language returns this type of data in registers, the location of the return value area must be communicated to the non-C process. In this case, the Xform-routine generally copies the return area's address to the fullword addressed by the `outret` argument.
4. When L$CILCL sends a CALL message to the non-C process, it passes the contents of the `*outret` fullword. (By the previous step, this is ordinarily the return area address.)
5. The Comm-routine calls the requested non-C routine. When the called routine returns, if the return value is in a register, the Comm-routine stores it into the return area address passed from the C process.
6. The Comm-routine sends a RET message to the C process to inform it that the call is complete. If the return value type is a structure, the return value has already been stored. If not, L$CILCL loads the value from the return value area into the correct register before returning to C.

The sequence of events for a call from a non-C routine to a C function is similar. Because the Xform-routine is not involved in this process, it is presented in the Comm-routine discussion later in this chapter.

Xform-routine return value handling Two of the arguments to the Xform-routine are intended for use in return value handling. The `cret` argument addresses an area in which the return value is to be stored by the non-C process unless the called routine is declared as returning `void`, in which case `cret` is zero. If the called routine returns a scalar value, `cret` addresses an 8-byte area, regardless of the size of the return value. If the called routine returns a structure, `cret` addresses an area whose size is that of the structure. Note that information about the size and type of return value is contained in the second token of the `cargs` list.

The `outret` argument is the address of a fullword in which you can store information to be passed to your Comm-routine about the

call. This word is normally used to pass the address of the return value area, using the assignment `*outret = cret`. If your Comm-routine needs to know the return value's type, you can also pass the return token by storing it into the return value area, via an assignment such as `*(unsigned *) cret = cargs [1]`. This technique is used by the FORTRAN Xform-routine.

If return value processing for your language does not require the use of `outret`, for example, if all return values are returned in memory addressed by the called routine's argument list, you can use the `outret` argument for some other purpose or ignore it.

Sample FORTRAN Xform-routine

The supplied FORTRAN Xform-routine is an example worth considerable study before you attempt to code your own. Your routine will differ in many details from the FORTRAN example, but you will probably want to retain its general structure and some of its algorithms.

The processing of the sample Xform-routine can be divided into the following steps:

1. See if the `after` argument indicates this is a call after a called FORTRAN routine has returned. If so, go to step 7.
2. Make a preliminary scan of the `cargs` list. If there are no tokens in the list, and if the return type is not string, then the argument list after the language token is a valid call-by-reference argument list. In this case, no transformation is necessary, and the Xform-routine can simply return to the Comm-routine after storing the address of the first argument within the `cargs` list into `*outargs`.
3. If there are tokens in the list or a string is returned, compute the size of the argument list that must be passed to FORTRAN.
4. Allocate some space for the Xform-routine's output argument list. The sample routine uses an internal stack for this purpose to avoid a call to `malloc` each time FORTRAN is called.
5. Make a second pass over the `cargs` list, putting the appropriate information in the output list. For most arguments, this simply involves discarding the tokens, and copying over the pointers to the arguments (because FORTRAN normally uses call by reference). If any strings are passed, however, FORTRAN requires a secondary argument list containing string lengths. Information is extracted from any string tokens and used to build this secondary list.

   If string arguments are passed, FORTRAN requires an 8-byte prefix to the argument list containing control information. In this case, the value returned to L$CICTL via `outargs` addresses the first byte after the prefix.
6. Process the return value and return. If the return value is a string, it is passed to FORTRAN as though it were the last argument. If it is not a string, the return token is copied to the return value area, whose address is passed to the Comm-routine via `outret`.
7. On reentry after the FORTRAN routine has returned, remove space allocated from the internal stack for this routine's arguments, and return.

Other noteworthy aspects of the sample routine are:

- The Xform-routine is expected to diagnose the appearance of any unrecognized or unsupported token types, either for the return value or for a function argument. The tokens should then be ignored, and the corresponding argument passed by reference. The library routine L$CIAWN may be called to generate an invalid argument diagnostic. It has two arguments, the name of the language being called and the argument number. An argument number of 0 indicates an unsupported return value type.
- The memory allocation algorithm in the sample routine uses a private external area for its stack. (A stack organization is necessary because of the possibility of nested calls to FORTRAN, each with its own argument list.) Most of the memory allocation code deals with the possibility of stack overflow. Because stack overflow can occur only with greatly nested calls or very long argument lists, you could simplify your routine by removing most of this code.

See **Function Pointer Implementation** later in this chapter for a discussion of the X-routine's processing of `TYP_CFUNC`, `TYP_FFUNC`, and `TYP_AFUNC` tokens.

Beginning Framework Execution (the Begin-routine)

The Begin-routine, normally named L$I*nam*B, where *nam* is your language's generic name, must be written in assembler language. It is called by your Main-routine to complete framework initialization. Its purpose is to issue the CCOMM macro to inform the C process that the other language's framework has been created. The role of the Begin-routine in the initialization of the non-C framework is illustrated in **Figure 16.1.**

After completion of the CCOMM macro, the Begin-routine has the same functionality as the Comm-routine, namely, to communicate with the C process and to call routines in your language as directed. For this reason, the Begin-routine is usually implemented as an entry point to the Comm-routine, allowing most of their code to be shared. Details about the CCOMM macro and the operation of this shared code is discussed in **Communicating with the C Process** later in this chapter.

The Begin-routine expects to receive the CRAB address, in some form, in its argument list. (See the Framework-routine and Main-routine descriptions for details.) The Begin-routine must receive the CRAB address so that it can pass it on to other ILC routines via the CCOMM macro, and so that it can store it in the CRAB address word. The only absolute requirement is that the CRAB address be available to the Begin-routine; some other method of passing it can be used if passing it as an argument from the Main-routine is not practical.

The Begin-routine terminates only when the C process sends a message to quit. When a QUIT message is received, shared code in the Comm-routine checks to see whether execution started at the Begin entry point or the Comm entry point. If execution started at the Begin entry, a QUIT message indicates that a C function called `dlfmwk`. The Begin-routine therefore returns to its caller (the Main-routine), allowing the framework to be terminated. See **Figure 16.6** for an illustration of the role of the Begin-routine in non-C framework termination.

Some languages may not permit you to name the Begin-routine according to the conventions. For instance, some versions of FORTRAN restrict you to six-character identifiers, and COBOL does not allow the $ symbol. You can assign a different name to this routine if necessary, because it is referenced only by the Main-routine. (For instance, the sample FORTRAN routine is named L$IFOB.)

Note that ILCLINK assumes that an INCLUDE statement is not required for the Begin-routine because it is ordinarily an entry point to the Comm-routine. If your Begin-routine is separate from your Comm-routine, you should store the object code for both in a single file, named L$I*nam*C, where *nam* is your language's ILCLINK name.

Controlling Interlanguage Calls (the Comm-routine)

The Comm-routine, named L$I*nam*C, where *nam* is your language's generic name, must be written in assembler language. It is called by either your Prep-routine or by the library's generalized L$UPREP routine to manage a call from your language to a C function. After any necessary preprocessing of the call, the Comm-routine issues the CCOMM macro to inform the C process that a C function should be called. After completion of the CCOMM macro, processing by the Comm-routine depends on data returned by the C process via the CCOMM macro. This processing is common to both the Begin-routine and the Comm-routine.

The position of the Comm-routine in the task of calling a C function from a non-C routine is illustrated in **Figure 16.3**.

The linkage conventions for the Comm-routine are slightly non-standard due to entry from L$UPREP. These conventions may not be changed because they are used by the library's generalized L$UPREP, which can be used with any language. Recall that the Comm-routine is given control when a C function is called from a routine in your language. When the Comm-routine is entered, all registers except 15 and 1 contain the same data as when the C function was called. Register 15 contains the Comm-routine's entry point address, and register 1 contains the address of your language's CRAB address word. The save area addressed by register 13 contains all the registers at the time the C function was called. You can obtain the original register 15 value (the C function's entry point address) and register 1 value (the C function's argument list pointer) from this save area.

Note that if you store registers in the save area addressed by register 13, you overlay the previous register 15 and register 1 values left there by L$UPREP. The library's generalized L$UPREP copies these register 15 and register 1 values to the CRABDWK field of the CRAB so that they can be retrieved even if the Comm-routine saves its registers. If your Comm-routine needs to save registers (for instance, due to use of a language-supplied linkage macro), your Prep-routine should also perform this copy so that your Comm-routine can retrieve this data regardless of which version of L$UPREP was used.

See **Comm-routine memory management** and **Error handling considerations** later in this chapter for further information on managing this non-standard linkage.

The Comm-routine can terminate in one of two ways. If the CCOMM macro receives a QUIT message from the C process, the Comm-routine must call your Quit-routine to terminate the framework. (Such a message is ordinarily sent as the result of the unexpected termination of the C framework.) If the CCOMM macro receives a RET message, it indicates that the called C function has returned. The Comm-routine must handle this message by returning to its caller. If your language returns data in registers, you must load the return value appropriately before returning.

Communicating with the C Process (the Begin- and Comm-routines)

The Begin-routine/Comm-routine combination is the most intricate of all the routines you have to write to support your language because of the number of different tasks in which they participate. The primary task is to communicate with the C process, directing it to call C functions, and calling any non-C routines as directed by C. However, there are important considerations for memory allocation, error handling, return value handling, and function pointer handling, each of which is discussed later in this chapter.

Recall that the Comm-routine "stands in" for called C functions in a non-C process. Because the Comm-routine can call a routine in your language on request from C, it must be able to execute recursively. Because the Comm-routine may be active when an ABEND occurs in a non-C routine, it is important that the Comm-routine not interfere with your language's diagnostic facilities. (For instance, its save area must be in a format that will not confuse your language's library.) These two aspects place unique demands on the Begin- and Comm-routines that do not apply to any of the other ILC support routines.

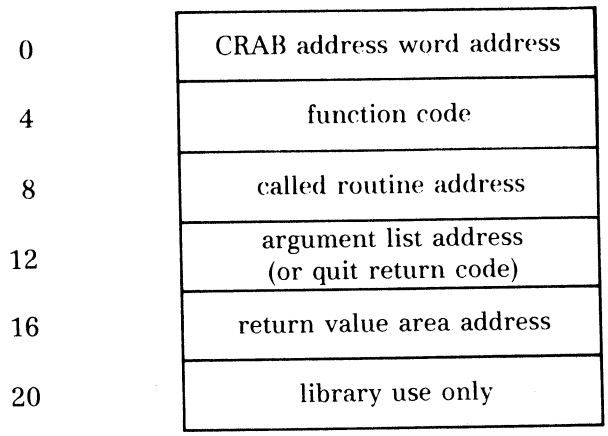
Interlanguage communication packets

Communication between the C process and a non-C process is effected by interchange of Interlanguage Communication Packets (ILCPs). You prepare an ILCP in dynamically allocated memory to transmit using the CCOMM macro. CCOMM transfers the contents of the ILCP to C, and then waits for a response from the C process. The response is also an ILCP, which CCOMM returns by copying it over the input ILCP. (Thus, if you will need the contents of the ILCP after CCOMM completes, you must keep an additional copy.)

An ILCP is 24 bytes long. You must allocate a single area of this size, which you reference for all your uses of CCOMM. When the Comm-routine is invoked recursively, this applies separately to each level. That is, each level must define its own ILCP buffer. Usually, you will want to simply include space for an ILCP in your dynamic work area. (See **Comm-routine memory management** later in this chapter.)

You may not modify the ILCP other than through the CCOMM macro. **Figure 16.12** illustrates the layout of the ILCP.

Figure 16.12 *ILCP Structure*



The function code is an integer specifying the kind of message represented by the ILCP. The codes are defined by the COPY code member CCONS. The message types you may have to process in your Begin- and Comm-routine are "begin", "call", "return", and "quit" requests (symbolic names CCOMBEGN, CCOMCALL, CCOMRET, and CCOMQUIT). For some function codes, the ILCP structure differs slightly from that shown above. These differences are documented in the next section.

Using the CCOMM macro

The CCOMM macro is issued to send a message to the C process, and then to wait for a response packet. The macro completes when a response is received; the response is copied over the input ILCP. Note that the C process may execute for an indefinite period of time during a call to CCOMM, and that the C calling sequence may change radically between the time that CCOMM is issued and the time it completes.

For instance, suppose that the C calling sequence is `main->mkfmwk` at the time your Begin-routine issues CCOMM to inform C that the framework has been created. The C process resumes execution and later, when the calling sequence is `main->sub1->sub2`, `sub2` calls a routine declared as `__foreign`. The full C calling sequence will now be `main->sub1->sub2->L$CICTL` when L$CICTL sends a response packet back to your language's process and execution of your Begin-routine is resumed.

The syntax of CCOMM common to all forms is as follows:

```
label     CCOMM code,CRAB=address,PLIST=address,RESP=reg,
                code-specific-operands
```

where

code
is a function code, specified as one of BEGN, CALL, or RET. This indicates the type of message to be sent to the C process, as follows:

- The BEGN operand is used by the Begin-routine to inform C that the user-language framework has been successfully created. Code-specific operands for use with BEGN are described later in this list.
- The CALL operand is used by the Comm-routine to ask the C process to call a C function. Code-specific operands for use with CALL are described later in this list.
- The RET operand is used by both the Begin-routine and the Comm-routine to inform the C process that a called routine in your language has completed. There are no code-specific operands for use with RET.

CRAB=
is used to pass the address of the CRAB address word. This operand is normally specified as (*reg*), where *reg* is a register addressing the CRAB address word. Alternately, it may be the symbolic name of a fullword pointing to the CRAB address word.

Caution: Pass the address of the CRAB address word, not its contents.

PLIST=
specifies the address of the ILCP area. Each use of CCOMM must specify the same area. This operand may be specified either as the symbolic name of the ILCP or as (*reg*), where *reg* is a register addressing the ILCP.

RESP=
specifies a register number, without parentheses. On completion of the CCOMM macro, this register addresses response data from the C process. (This data begins at offset 8 from the ILCP.)

code-specific operands
OSA=*address* is required when the function code is specified as BEGN.

The OSA operand specifies the address of the original save area. (See **Control Block Location** earlier in this chapter for a discussion of this area.) The address may either be the name of a fullword containing the address of the original save area, or (*reg*), where *reg* is a register addressing the original save area.

If you cannot locate the original save area, you can specify OSA=(*reg*), where the register contains zeroes. However, because the BEGN function of CCOMM is issued only in your Begin-routine, it is used only when your framework was created via your Framework-routine and Main-routine. For this reason, in this case the original save area is the one allocated by your Framework-routine. You may be able to use this fact to make this save area address available, even if the address is not preserved by your language's run-time library.

The following code-specific operands are required for function code CALL:

```
CALL=entry-point
ARGS=argument-list
RETP=return-area
```

CALL=
: specifies the entry point of the C function to be called. This value should be extracted from the register 15 slot of the save area addressed by register 13 on entry to the Comm-routine (or from CRABDWK if supported by your Prep-routine). This argument is normally (*reg*), where *reg* is a register containing the entry point address. It may also be the symbolic name of a fullword pointing to the entry point.

ARGS=
: specifies the value that should be placed in register 1 when the C function is called. Normally, this will be the value stored in the register 1 slot of the save area addressed by register 13 on entry to the Comm-routine. However, if necessary, you can generate a new argument list and specify its address instead. As with the CALL argument, ARGS can specify either the symbolic name of a fullword pointing to the argument list, or a parenthesized register containing the argument list address.

RETP=
: specifies the address of an area (at least 8 bytes long) where a return value from the called C function can be stored. See **Return value processing** earlier in this chapter for more information on this operand. This operand can be specified as the name of a fullword addressing the return value area, or as a parenthesized register containing the return value area address.

When CCOMM completes, register 15 contains the function code from the ILCP returned by the C process. The value will be one of CCOMCALL, CCOMRET or CCOMQUIT. In addition, the RESP register will address other response data, as described previously. See the next section for information on processing the return information.

Post-CCOMM processing

The response to the CCOMM macro includes a function code that should be interpreted as directions from the C process. It is easiest to use the same piece of code after all uses of CCOMM in the Begin- and Comm-routines. In the sample FORTRAN Comm-routine, this code is located at the label FORCJOIN. The list below discusses each of the possible function codes and suggests appropriate processing.

- If the function code in register 15 is CCOMQUIT, the C process has requested your process to terminate. The return code that should be returned to the caller of your language's main routine is in the ILCP, in the slot labeled *quit return code* in **Figure 16.12.** (This will be at offset 4 from the value in the RESP register.) The action you should take in this case depends on whether this level of the

Comm-routine was entered at the Begin entry point or the Comm entry point.

- If entry was through the Begin entry point, process execution should be terminated by returning to the caller of the Begin-routine (the Main-routine). This case occurs when your language's framework is to be terminated due to a call to `dlfmwk` from C. The return code information is not meaningful in this case because any return code set will be discarded when your process is deleted.
- If entry was through the Comm entry point, execution should be terminated by a call to your Quit-routine. You should pass to the Quit-routine the return code from the ILCP as an argument. Because the Quit-routine will be written in your language, you must set up its argument list according to the run-time conventions of your language. This case occurs when the C framework terminates without having deleted your language's framework.

- If the function code in register 15 is CCOMRET, the C function that L$CICTL called at your request completed, and the Comm-routine must now return to its caller. This function code is returned only when the Comm entry point is used.

  If the called C function returns a scalar, its return value has been stored in the area addressed by the RETP operand of the completed CCOMM call. If your language returns values in registers, you need to load the return value appropriately before returning to your caller.

  See **Returning values from C to your language** later in this chapter for further details.
- If the function code in register 15 is CCOMCALL, the C process has issued a call to a `__foreign` routine. In this case, the Comm-routine must call this routine so that it will run in your language's process rather than in the C process. The address of the routine to call, the address of its argument list, and the address of a return value area are all addressable using the RESP register of the completed CCOMM macro call. Note the following additional information about these addresses:

  - The *called routine address* slot of the ILCP normally contains the address of the routine that is to be called. If the high order bit of this value is on, however, it represents a `__foreign` function pointer. In this case, the address points to the function pointer. The format of such a function pointer is determined by you, but it is intended to be your language's equivalent to a C function pointer. For instance, a `__pli` function pointer is simply a PL/I ENTRY variable.

    If the routine to be called is represented by a function pointer, you have to find the actual address of the routine to call. There may also be language-specific actions you have to take in this case. For instance, in PL/I register 5 has to be loaded from the second word of the ENTRY variable before the call is made.

    See **Function Pointer Implementation** later in this chapter for details.
  - The *argument list* slot of the ILCP contains the address that should be loaded into register 1 before the routine is called. It addresses the argument list constructed by the Xform-routine, not the list generated by the compiler.

- The *return value area* slot of the ILCP contains any address stored by the Xform-routine using its `outret` argument. This is normally an area where the return value from the called routine should be stored, but other uses are possible. See the Xform-routine description earlier in this chapter for further details.

When you call a routine in your language, note that this routine might itself call a C function. This call will be intercepted by the Prep-routine and cause a recursive entry to your Comm-routine. Your Comm-routine must not use static data areas in any way that will cause recursive calls to fail.

After the routine you call in response to a CCOMCALL message from C returns, you must inform the C process that it has completed, using the RET function of the CCOMM macro. If the called routine returned a value, this value must be passed back to C. (You need to write code to do this if your language returns values in registers. If it returns values in memory, you can usually arrange for it to store return values directly in the area used by C.)

Upon completion of the CCOMM RET macro, you should branch back to your common CCOMM completion code because the C process may return any of the normal CCOMM responses.

Comm-routine memory management

Because the Comm-routine may be invoked recursively, it must use dynamically allocated memory for its ILCP buffer and other work areas. The run-time system for your language may provide an interface by which assembler routines can obtain dynamic work areas (or *stack space*) on entry. For instance, Pascal/VS provides a PROLOG macro to allocate a work area, and an EPILOG macro to free the work area and return. If your language provides such a facility, you should use it in the Begin- and Comm-routines, especially since use of this interface probably improves the ability of your language's run-time library to process errors that occur while your routine is active.

If you use an interface or macro provided by your language implementation for this purpose, note the following:

- The Begin- and Comm-routines should have work areas with identical size and identical mapping, to facilitate the use of common code. You should allocate a switch in your work area that can be used to indicate which entry point was used, since processing of a CCOMQUIT response from C varies depending on the entry point.
- Because argument information is present in the register 13 save area on entry to your Comm-routine, you should not save registers yourself unless you copy the saved register 15 and register 1 values elsewhere. If you use a macro such as the Pascal PROLOG macro at your Comm entry point, this may cause problems. The recommended solution is to copy the saved register 15 and register 1 values to the CRABDWK fields in your Prep-routine. Later, the CRAB can be located via the CRAB address word (passed in register 1), and the values copied to your work area. These values must be saved before you issue the CCOMM macro, which may modify the contents of CRABDWK.
- Some languages require the presence of a routine name in character form at a fixed offset (sometimes negative) from the entry point. You should include the name of your Comm-routine in this fashion

if at all possible, in order to get more useful tracebacks and diagnostics from your language.

The ILCENTRY and ILCEXIT macros If your language does not provide a dynamic memory allocation interface, or if, as with FORTRAN, it does not support recursion, you must use another memory allocation technique. The SAS/C library provides ILCENTRY and ILCEXIT macros for this purpose. When you use these macros, you must note that you use them in your entries in the supported language table.

Even though ILCENTRY and ILCEXIT have some aspects that make them more convenient than using macros such as Pascal's PROLOG, you should use the language-supported macros rather than ILCENTRY and ILCEXIT whenever possible. This is because ILCENTRY and ILCEXIT may interfere with your language's diagnostic capabilities; any native macros are unlikely to have such adverse effects.

The ILCENTRY macro is intended to be used as the first instruction of your Begin-routine or Comm-routine. The syntax of ILCENTRY is as follows:

```
label ILCENTRY SYS=system,TYPE=type,CRABREG=reg,
               CRABFMT=format,WKREGS=(reg1,reg2,reg3),
               BASE=basereg,DSA=DSA-size,OFLOW=flow
```

The ILCENTRY operands have the following meanings:

SYS=
: is required. It must specify either SYS=OS or SYS=CMS to indicate the operating system in use.

TYPE=
: is required. It must specify TYPE=BEGIN for use in your Begin-routine, or TYPE=CALL for use in your Comm-routine.

CRABREG=
: specifies a register into which the macro can load the CRAB address. If not specified, CRABREG=12 is assumed.

CRABFMT=
: is used only for a TYPE=BEGIN call. This call assumes that register 1 on entry addresses a call-by-reference argument list where the first argument is the CRAB address in some format. If CRABFMT=BIN is specified, the first argument is assumed to address a fullword containing the CRAB address in binary. If CRABFMT=CHAR is specified, the first argument is assumed to address an 8-byte area containing the CRAB address in zoned decimal. (If you require some other format, you can easily update the macro to convert from this format to binary.)

WKREGS=
: specifies three registers that can be used as work registers. If WKREGS is omitted, WKREGS=(2,3,4) is assumed.

BASE=
: specifies the number of a register to be used as a base register. If BASE is omitted, BASE=9 is assumed.

DSA=

specifies the size of the dynamic area to be allocated. This operand is required. At least 96 bytes (72 bytes for a save area plus 24 bytes for an ILCP) are normally required.

OFLOW=

specifies whether code should be generated to handle overflow of the ILCENTRY stack. OFLOW=YES generates substantially more code than OFLOW=NO. When a framework is created, an initial area of 4K is allocated for use by ILCENTRY, and overflow occurs only if this area is filled. If OFLOW=NO is specified and an overflow occurs, it will not be detected, and other storage areas will be overlaid, or an 0C4 ABEND will occur. If OFLOW is omitted, OFLOW=YES is assumed.

Note that, at the completion of the ILCENTRY macro with TYPE=CALL, register 15 addresses the entry point of the C function to be called.

The ILCEXIT macro should be used at any point in your Begin- or Comm-routine where you want to return control to your caller. The syntax of ILCEXIT is as follows:

```
label ILCEXIT CRABREG=reg,WKREGS=(reg1,reg2),PREVSA=reg
```

The ILCEXIT operands have the following meanings:

CRABREG=

specifies the number of a register containing the CRAB address. If this operand is omitted, CRABREG=12 is assumed.

WKREGS=

specifies two registers that can be used as work registers by the macro. If this operand is omitted, WKREGS=(2,3) is assumed.

PREVSA=

specifies a register containing the value in register 13 when the Comm-routine was entered. (If your language returns values in registers, you have to locate this area to store the return value.) If this operand is omitted, ILCEXIT assumes that this value is not in a register and loads it from the standard chain field (offset 4) in the current save area.

Error-handling considerations

Depending on your language and its implementation, you may have to take special actions in your Begin- and Comm-routines to avoid interfering with your language's ability to produce diagnostics and tracebacks. This is more likely to be necessary with these routines than with any others because the Comm-routine can call any other routine in your language and, therefore, you will not always be able to prevent errors or ABENDs.

Language implementations differ so much from one to another in this area that no absolute rules can be given, but you should investigate the following:

- If possible, you should make the Begin-routine and the Comm-routine appear to be written in your language. For instance, if your language provides a macro like the Pascal PROLOG macro, you should use it if at all possible. (Some languages, like Pascal, may not let you call subroutines from assembler unless you do this.) If your language does not provide a macro or document an equivalent interface, you may want to list compiler-generated code and copy the compiler-generated code for subroutine entry and exit into your assembler routine.
- If you cannot emulate the compiler-generated linkage for your language (for instance, because your language does not support recursion), you should mark your save area to identify it as reserved for an assembler routine rather than a high-level language routine. By convention, a value of zero in the word at offset zero from register 13 indicates an assembler routine. The ILCENTRY macro automatically sets this word to zero.
- When the Comm-routine is entered, the address of the called C function is in the register 15 slot of the register 13 save area. Many languages' traceback algorithms use this value to find the name of each active routine. Since the format for the entry point to a C function is unlikely to be the same format as for your language, this may cause gibberish in the traceback or other side-effects. After you have stored the C function's entry point somewhere else, you may want to replace the value in the register 15 slot with the entry point address for your Comm-routine, allowing the Comm-routine to appear under its own name in the traceback. Note that for this to work, the Comm-routine's entry point must have the format expected by your language's run-time library.
- Except when you run under OS using the C run-time option `=multitask`, all the language processes run in the same OS task or CMS virtual machine. Each language process may use the ESTAE or ABNEXIT macro to handle ABENDs. Due to the way these macros are defined by OS and CMS, if any language process ABENDs, all the ESTAE and ABNEXIT routines for all processes are called. This is usually good because it allows each language to produce a diagnostic and a traceback.

  However, some languages may attempt to retry an ABEND that occurs in another language's process. (PL/I is an example of a language with this unpleasant habit.) Such a retry is doomed to fail, because the retrying language's framework may not be established correctly, and the active save area chains will be formatted according to C conventions rather than the retrying language's conventions. Usually, a second ABEND occurs during such a retry, but only after the other language has done enough processing to completely hide the cause and location of the original ABEND. Any diagnostics from C or a third language then refer to the new ABEND, not the original one, and are completely useless.

  Any solution to this will be, of necessity, language dependent. The best approach is to prevent your language from retrying after an ABEND, perhaps by use of a run-time option or by setting error-handling flags in a language control block. If you modify control blocks to prevent ABEND retry, note that retry is a problem only when some language other than your language is running. For this

reason, it is sufficient to inhibit retry only during the execution of the CCOMM macro, thereby allowing your language's normal recovery procedures to be used when an ABEND occurs during its own processing.

Returning values from your language to C

This section summarizes the considerations for support of calls from C to routines in your language that return a value. Additional details may be found in **Return value processing** in the section **Argument Transformation (the Xform-routine)** earlier in this chapter.

- When a `__foreign` routine is declared in the C program as returning a type other than `void`, the compiler informs your Xform-routine of the type of return value expected by means of a return token in the generated token list. This token does not provide a complete description of the type of return value. For instance, integers and pointers are not distinguished. Two special structure types corresponding to common 370 string implementations are distinguished, but for all other structure types, only size information is passed.

  If your language distinguishes types that the C compiler does not, you must impose restrictions to resolve ambiguities. (You should choose the restrictions so as to support your language's most commonly used data types.) For instance, if your language returns integers and pointers in different ways, you cannot support FUNCTIONs of both kinds. If your language needs more information about a returned structure than its size, you must forbid FUNCTIONs that return a structure.
- Note that C does not support functions that return arrays. Even if your language supports FUNCTIONs that return ARRAYs, you cannot call them from C unless you can support them in C as functions returning a structure with an array member.
- When C calls a routine that returns a structure, it places the address of an area where the structure value should be stored 4 bytes before the parameter block addressed by register 1. Since structure values are too large to return in registers, you probably can simply copy the pointer to the return area to an appropriate location in the argument list created by your Xform-routine to cause the called routine to return its result correctly.
- C expects non-structure return values to be returned in registers. L$CICTL takes care of the detail of loading such return values into the correct register during the processing of a return from a FUNCTION in another language. Before calling your Xform-routine, L$CICTL allocates an 8-byte area in which it expects the returned value from the called routine to be stored, and passes the address of this area to the Xform-routine.

  If your language returns such values in memory, you can probably simply copy the address of the L$CICTL-supplied area to an appropriate location in the argument list created by the Xform-routine to cause the called routine to return its result correctly.

  If your language returns some or all such values in registers, your Comm-routine will have to store the value returned by a routine in your language in the L$CICTL-provided area. Normally, the Xform-routine passes the Comm-routine this address via its `outret` argument. Your Comm-routine will probably require type

information about the return value to do this correctly. For instance, different registers will be used for integer and floating-point return values. The normal technique for making type information available is for the Xform-routine to store the return token in the first 4 bytes of the L$CICTL-provided area.

The return value must be stored left-justified in the return value area; only the number of bytes required for the returned type should be stored. For instance, two bytes should be stored for a `short` return value. Note that you need not distinguish between `signed` and `unsigned` return values, as this is handled automatically by the compiled code for the calling C function.

Returning values from C to your language

This section summarizes the considerations for how to support calling C functions that return a value from your language.

- When your Comm-routine processes a call from your language to a C function, much less information is available about return value type than for a call in the other direction. The only information available is a flag byte in the compiler-generated prolog code for the called function.

  The address of the prolog code will be in the register 15 slot of the save area addressed by register 13 when your Comm-routine is entered. It can be mapped by the CPROLOG DSECT. The byte named CPROTYPE contains return value bit flags, with names and meanings defined as follows:

  - The flag CPROSTRC indicates that the function returns a structure or union.
  - The flag CPRODBL indicates that the function returns `double` or `float`.
  - The flag CPROVOID indicates that the function returns no value.
  - If none of CPROSTRC, CPRODBL, or CPROVOID is set, the function returns an integral or pointer value.
  - The flag CPROSHRT indicates that the function returns a `short` or `float` value. (If CPRODBL is also set, the function returns `float`.)
  - The flag CPROCHAR indicates that the function returns a `char`.

  Note that information on the size or mapping of a returned structure is not available. This may force you to forbid altogether calls to C functions returning structures, or to assume that such a function returns one of the special string structures.
- When the Comm-routine processes a call to a C function that returns a structure or union, you must pass C an argument list with the return value address located 4 bytes before the argument list. If this is not the format used by your language, you may need to make a copy of the argument list to pass to C.
- When the Comm-routine processes a call to a C function that returns some scalar type, you must inform the C process of the address of an area where L$CICTL can store the value returned by the C function. The RETP argument of the CCOMM CALL macro is used to identify this area. The return value is always stored left-justified. For instance, if the called C function returns a `short`, the return value is stored in the first two bytes of the return area. Note

that you can use RETP with CCOMM even for a function returning a structure or `void`, as the RETP value is ignored by L$CICTL in these cases.

- When CCOMM CALL completes, if your language expects values to be returned in registers, you must load the return value from memory before returning.

Sample Begin- and Comm-routines

Because of the large number of issues to consider when writing a Begin-routine and a Comm-routine, the sample versions of these routines are not easy to follow. Now that most of the considerations have been presented, it is possible to summarize the sample routines. Note that these routines are only samples. Your routines may require steps that are not needed for FORTRAN, and you may be able to omit some steps that are unique to FORTRAN.

The FORTRAN Begin-routine is named L$IFOB to conform to FORTRAN naming conventions, so it can be called by the FORTRAN Main-routine. It performs the following steps:

- Use the ILCENTRY macro to obtain a dynamic work area. The CRABFMT=BIN operand is used because the Main-routine passes the CRAB address in binary. Note that use of ILCENTRY is necessary because the FORTRAN library does not provide dynamic memory allocation or recursion support.
- Record that the entry was at the Begin entry point.
- Store the CRAB address in the FORTRAN CRAB pointer word, which is simply a CSECT, since FORTRAN does not support reentrancy.
- Call the FORTRAN Locate-routine to find the address of the original save area. For most languages, no such call is necessary, and the save area address can just be loaded from a language control block. However, finding this save area for FORTRAN is highly dependent on the release of FORTRAN, and calling the Locate-routine allows the release-dependencies to be isolated to this module.
- Issue the CCOMM BEGN macro to inform C that the FORTRAN framework has been created. On completion of the CCOMM macro, control is transferred to the label FORCJOIN to perform processing common to both entry points.

The FORTRAN Comm-routine performs the following steps:

- Use the ILCENTRY macro to get a dynamic work area, and copy the C function's entry point and argument list to registers.
- Modify the register 15 slot of the previous save area to address the Comm-routine's entry point, to improve FORTRAN traceback information.
- Record that the entry was at the Comm entry point.
- Examine the called C function's prolog to determine the return value type.
- If the C function returns a structure, it is assumed that the FORTRAN caller expects a CHARACTER*n string to be returned. In this case, the last argument address passed by FORTRAN points to the area in which the return value is to be stored. The Comm-routine copies this value to the word before the argument list to make it conform to C conventions. (Due to the layout of FORTRAN argument lists, this word before the argument list will always be allocated and modifiable.)

- If the C function returns a scalar, pass the address of a doubleword in the work area for C to copy the return value to.
- Issue the CCOMM CALL macro to instruct the C process to call the required routine.

The remaining processing is common to both the Begin-routine and the Comm-routine, beginning at the label FORCJOIN.

- Determine the function code returned by the CCOMM macro.
- If the function code is CCOMRET, the called C function has returned. If the function returned a scalar, load the return value into register 0 (if an integer) or floating register 0 (if floating-point). The CPROTYPE byte of the called function is used to determine the return value type.

  FORTRAN treats a nonzero value in register 15 at return as indicating use of a FORTRAN RETURN n statement in the called routine. Therefore, zero is loaded into register 15 before return to avoid any such misinterpretation.
- If the function code is CCOMCALL, the called C function is in turn calling a FORTRAN routine. The Comm-routine loads registers 15 and 1 with data from the ILCP and calls the FORTRAN routine. On return, it checks the token in the return value area to determine the type of return value expected by C, and stores the correct register (either general or floating register 0) in the return value area. It then issues the CCOMM RET macro to inform C that the called routine has returned, and branches to the start of common processing to handle C's response.

  Note that the X'80000000' bit may be set in the FORTRAN routine address to indicate a `__fortran` function pointer rather than an actual routine address. `__fortran` function pointers are the same as FORTRAN EXTERNALs; that is, they simply contain the address of the routine to be called. Therefore, the only special requirement is an extra load to get the address of the routine, rather than the address of the function pointer. See **Function Pointer Implementation** later in this chapter for further discussion of function pointer issues.
- If the function code is CCOMQUIT, the C process is attempting to terminate the FORTRAN process. If entry was via the Begin entry point, it issues ILCEXIT to return control to the Main-routine. If not, the Comm-routine calls the FORTRAN Quit-routine, passing it the requested return code. The Quit-routine is referenced indirectly, since different routines are used for VS FORTRAN Version 1 and VS FORTRAN Version 2. This allows the Comm-routine to avoid any version dependency of its own.

Miscellaneous User-Supported Language Issues

This section discusses a number of issues that you may need to decide as you implement ILC with a user-supported language.

Defining Equivalent Data Types

Perhaps the most important issue for you to decide and document as part of your ILC interface is that of compatible data types. In many cases, this is easy. For instance, your language probably has a fullword binary integer type, which is clearly equivalent to the C `int`.

Some considerations to take into account in less clear-cut cases are as follows:

- Remember that the data formats of corresponding types should be the same. You must be careful not to be misled by superficial similarities of types with different formats. For instance, your language may have an enumerated data type. Recall that SAS/C uses a fullword of memory for `enum` values and, therefore, you cannot consider your language's enumerations and C's to be equivalent unless yours also occupy a fullword.
- The most likely source of difficulty is with string types. Your language probably does not support null-terminated strings as a data type. If your language represents strings as a simple array of characters, you should be able to treat the fixed-length string structure type `struct {char text [n];}` as an equivalent. Similarly, if your language supports a string implementation like PL/I VARYING strings (a halfword length followed by an array of characters), you can treat the C type `struct {short len; char text [n];}` as an equivalent. If this format is your language's primary string type, you should recommend the C compiler option `VString` to the users of your interface so that C string literals are passed to routines in your language with a halfword length prefix.

  If your language uses some other string representation, such as a linked list of characters, you may be forced to document your language's format for the users of your interface and require them to convert from your language's format to C format. This is especially true for calls from your language to C; for calls in the other direction, your Xform-routine may be able to perform the necessary conversions. With the Xform-routine, you probably will be able to arrange for string literals and arguments passed using the `__STRING` conversion macro to be successfully converted to your language's format. (However, see the next item.)
- If possible, you should avoid having two different equivalents to the same type, one for use when C calls your language and one when your language calls C. This is frequently tempting because of the ability of the X-routine to remap arguments when your language is called by C, but it also may be confusing. It might also force users to convert between the two representations themselves in a function that is called by one routine in your language and calls another language.
- If your language supports structure or record types, verify that your language's alignment rules are the same as C's. If they are not, you will need to investigate how identical alignment can be forced and document the results. Possible techniques include the use of the C `__noalignmem` keyword or `BYtealign` option, similar features of your language or, at worst, the introduction of padding members.

  Note that C does not support self-defining records (such as ones defined with the PL/I REFER option), but so-called "variant records" (as implemented in Pascal and Ada) can often be represented as unions.

Data Sharing Considerations

In addition to issues raised by the different data types and formats of other languages, there are those associated with different conventions for passing arguments, defining external data, and so on. Some of these issues are discussed below:

- If your language supports both call by value and call by reference, you should encourage the use of call by reference. Call by value conventions frequently vary from language to language, especially with regard to alignment of the arguments. (The SAS/C conventions are described in Chapter 3, "Communication with Other Languages.") If your language uses different conventions from SAS/C, call by value will be difficult for all but the simplest data types. On the other hand, with call by reference, the argument list format must, in general, be a simple list of fullword pointers.
- If, like PL/I, your language passes arguments using descriptors (or *dope vectors*), it is best to suppress them through use of a language facility like the PL/I OPTIONS(ASM). If no such facility exists, you must document for your users how to decode descriptors passed from your language in C.

  For calls from C to your language, you are in many cases able to build any necessary descriptors yourself in your Xform-routine.
- Note argument passing conventions for arrays. If your language allows arrays of unknown size to be passed, it probably has its own convention for passing the array size as an additional argument. If this is the case, you should probably tell your users to use the **__ARRAY** set of data type conversion macros to pass arrays to your language, and in your Xform-routine build any supplementary arguments required.
- Note whether your language's implementation of external variables always defines storage for them, or whether there is a form that generates references without storage. This determines whether external variables referenced from your language can be defined in C, or whether they can only be defined in your language.

  Also note if your language defines any unusual kinds of external variables that are not implemented as external symbols in the object module so you can document for your users that these cannot be shared directly.

Function Pointer Implementation

For some applications, it is desirable to be able to pass the address of a routine in one language to a routine in another language. SAS/C ILC offers some support for this, depending on the facilities available in your language. With appropriate support, it is possible to do the following:

- Pass a routine in your language the address of a C or assembler function and call this function from your language.
- Pass a C function the address of a routine in your language and call this routine from C.
- Pass a routine in your language the name or address of a routine in your language, and call it from your language.

Note that your language might support facilities such as this in one of two ways. Some languages, such as C and PL/I, treat routine addresses as a full-fledged data type. Others, such as FORTRAN and Pascal, limit you to passing routine addresses to other routines. The method used changes the way you describe passing routines as

arguments but has little effect on the way you have to implement your support. The rest of this discussion uses the term *routine descriptor* to describe your language's internal format for routines passed as arguments, whether or not that format is associated with a special data type. The term *routine address* is used to describe the physical address of a routine's entry point.

To implement support for passing routine descriptors, you must be familiar with their format in your language. If a routine descriptor in your language is simply the location of its entry point, you are unable to support calls to C functions in other load modules from your language. If, like Pascal or PL/I, your language provides auxiliary information in routine descriptors, you can probably support these calls.

In addition to being familiar with your language's routine descriptor format, you should also be familiar with the SAS/C function pointer representation. This is described in the *SAS/C Compiler and Library User's Guide.*

Passing routine descriptors from your language to C

A user of your interface who refers in C to a routine descriptor in your language must usually declare it as a `__foreign` function pointer. The size of a `__foreign` function pointer is 4 bytes; the compiler makes no assumptions about its contents.

When a routine descriptor is passed from your language to C, none of the interface routines can tell that this is happening and, therefore, there is no opportunity to change its data format. The corresponding argument in the called C function must therefore be declared so that its format matches the descriptor format. This argument should be declared as a `__foreign` function pointer if either of the following is true:

- routine descriptors in your language are 4 bytes long
- routine descriptors are passed by reference. In this case, the `__foreign` function pointer will be the descriptor address rather than the descriptor value.

If your language uses more than a fullword for routine addresses and passes them by value, you have to define some special type other than a function pointer type to represent them. Usually, this type will be a structure type including a `__foreign` function pointer as a component. See the discussion of `__pascal` function pointers in Chapter 7, "Communication with Pascal," for an example of this kind of representation.

Using a __foreign function pointer in C

Ordinarily, the programmer passes a routine descriptor to a C function so that the routine can be called from C. Because the C compiler does not know the format of such descriptors, it cannot determine the location of the routine to call. Therefore, the information built by the compiler about the call (and passed to your Xform-routine and Comm-routine) must contain a pointer to the `__foreign` function pointer rather than the location of the routine to call. The high-order bit should be set in the address to indicate that this is a call using a function pointer.

Your Comm-routine must check for this case before calling a routine in your language. If the function pointer bit is set, the Comm-routine

must locate the called routine's actual entry point, based on the format of routine descriptors in your language. Such indirect calls may require additional actions not needed for ordinary calls. For instance, PL/I requires you to load register 5 with data from an ENTRY variable before calling the routine it references.

Passing function pointers from C to your language

Your Xform-routine is primarily responsible for passing function pointers from C to your language. When the compiler processes a call to a `__foreign` routine where an argument is a function or function pointer, it generates a token describing the type of value being passed. Your Xform-routine needs to process each case differently.

- The token tag `TYP_CFUNC` indicates a C function name or C function pointer was passed. The value following the token is the function pointer value, which addresses an 8-byte area containing the physical address of the function and its pseudoregister vector (PRV) address. (If a function name is passed, the compiler promotes it to a function pointer.) To support passing this type, you have to transform the C function pointer format into your language's routine descriptor format. If your language's format includes only the routine address, you have to discard the second word of the C function descriptor (the PRV address) and, therefore, will be unable to support calls between load modules. Note that some support for C function pointers must also be present in your Comm-routine to support calls between load modules, as described in the next section.
- The token tag `TYP_FFUNC` indicates a `__foreign` function pointer was passed. The word after the token is the address (not the value) of the fullword function pointer. In most cases, your Xform-routine simply copies either the address or the value of the pointer to its outgoing argument list. Which one you choose depends on your language's calling conventions.
- The token tag `TYP_AFUNC` indicates that an `__asm` or `__foreign` routine name or an `__asm` function pointer was passed. The word after the token is a pointer to a fullword containing the physical address of the routine. Your X-routine must build a routine descriptor in your language's format for an argument of this sort.

Calling a C function pointer from your language

In addition to the Xform-routine support described above, you also need to consider calls via C function pointers in your Comm-routine. Recall that when you pass a C function pointer to a `__foreign` routine, the Xform-routine passes a routine descriptor in your language's format, including the location of the C function. When a C function is called from C using a function pointer, one of the required actions is to replace the current PRV address (in the CRABPRV field of the C Run-time Anchor Block) with the PRV for the called routine, and to restore it after the called routine returns. For you to support calls to C functions in another load module, your Comm-routine must do the same thing.

For the Comm-routine to switch PRVs, the PRV address must be available. This means two things. First, the Xform-routine must save the PRV address somewhere in the descriptor it constructs. Then, your Comm-routine must locate this information in order to update

the CRAB. How to locate the information depends very much on your language's linkage conventions.

An example is the conversion of C function pointers to PL/I ENTRY variables in the SAS/C PL/I interface. A PL/I ENTRY variable contains two words. When the Xform-routine converts a function pointer to an entry variable, it places the PRV address in the second word. When PL/I calls the ENTRY variable, it loads the second word into register 5. The PL/I Comm-routine can locate the register 5 value from the previous save area and use it to update the PRV address in the CRAB.

CFMWK and DCFMWK Considerations

Note that you need to provide users of your language with information on how to call the `CFMWK` and `DCFMWK` routines. These routines expect a call-by-reference argument list. The first two arguments to `CFMWK` must be in either fixed-length or varying-length string format. (That is, they must be an array of characters, possibly preceded by a halfword prefix containing the array size.) If varying-length string format is used, note that the string length must be less than 256 because `CFMWK` distinguishes the two formats by checking for a zero byte at the start of the string. Most languages allow you simply to call `CFMWK` and `DCFMWK` in the natural way and to pass string literals as arguments.

If this is not the case, you will need to describe and illustrate the correct techniques for your language. If this is exceedingly tedious, for instance, if your language uses an arcane string representation, another possibility is to write your own front ends for these functions and document the front ends. In this case, the front ends can do the work of transforming argument lists from your language's format to the format expected by `CFMWK` and `DCFMWK`.

Error Handling and the C Run-Time Option =multitask

When the C run-time option `=multitask` is used, the C framework manager uses an expensive technique to assure that a program check is handled by the program check handler for the proper language. Under OS, this requires running each language in a separate OS task; under CMS, the SPIE macro is used at every framework switch to reinstate the current framework's program check handling.

When `=multitask` is not specified, a more efficient technique is used based on assumptions of normal use of the (E)SPIE and (E)STAE macros by the other languages' run-time libraries. (Note that under CMS, the use of ABNEXIT by the other language is not relevant due to convenient aspects of the definition of ABNEXIT.) If these assumptions are not valid for your language, you must tell your users to use the `=multitask` option, at least if accuracy of error-handling is a requirement.

When `=multitask` is not used, the C library assumes the following:

- The other language's library issues any (E)SPIE or (E)STAE macro it requires when the language's framework is created.
- The other language does not issue (E)SPIE or (E)STAE during later processing, except as the result of handling a program check or ABEND.
- The other language's library does not overlay or replace any (E)STAE exit defined by its caller.

Note that it is possible that your language might violate these restrictions only under certain well-defined circumstances. For

instance, Pascal/VS violates these conditions only when the Pascal/VS debugger is used. In this case, you can document the need for `=multitask` as a restriction of the specific feature or features, rather than of the entire language.

Documenting Your Interface

The documentation for your user-supported language interface is at least as important as the code. No one will be able to use your interface without careful, comprehensive documentation. Chapter 14 provides only an overview of using a user-supported language; your documentation must provide the details.

In preparing your documentation, look over Chapters 4 through 7 of this book, which will prove helpful in suggesting topics and organization. Also look at Chapter 14 to see how your documentation will be used. Be sure to include frequent examples. Note that, if possible, you should use the same terminology as this book so that users can use both sets of documentation together without becoming confused.

The list below suggests numerous issues that should be covered in your documentation. By necessity, the list is very general and will probably not include all the necessary items for your language. A good way to evaluate your documentation is to use it, without reference to any source code, to write a modest multilanguage application. By taking on the user role in this fashion, you can discover several items that did not occur to you while you were developing the interface.

Important Items to Document

- □ both the framework name and ILCLINK name of your language.
- □ what versions of your language are supported, and any differences between versions.
- □ how to call `CFMWK` and `DCFMWK` to create and delete the C framework from your language.
- □ whether or not your language supports passing run-time options from C.
- □ whether unexpected termination of your language's framework can be handled successfully.
- □ whether or not the C run-time option `=multitask` is required, and under what circumstances.
- □ what data types are equivalent, both for calls from your language to C and from C to your language.
- □ any special alignment rules for sharing structures between your language and C.
- □ what data types may be returned from your language to C, and from C to your language.
- □ any unusual calling conventions that users need to be aware of to pass arguments correctly.
- □ what data type conversion macros may be used in calls to your language, and directions for their use.
- □ any restrictions on naming or sharing external variables.
- □ whether or not function pointers and routine descriptors may be passed between your language and C.
- □ the rules for defining the entry point of a program whose main routine is in your language.

- □ any restrictions on the use of ILCLINK with your language. (For instance, if your language implementation uses pseudoregisters, you may need to require or recommend the use of CLINK to avoid combining your language's pseudoregister vector with C's.)
- □ any special error-handling requirements or restrictions.
- □ any features of your language that cannot be used in multilanguage applications.
- □ any special requirements for using your language's debugger in a multilanguage program.

Part 3
Appendices

Appendix 1
ILC Library Diagnostic Messages

Introduction

Library messages are classified as ERRORs, WARNINGs, and NOTEs. An ERROR message is associated with a condition that forces program termination, usually with an ABEND. A WARNING message describes a condition that permits program execution to continue, where the routine that detected the condition returns an error indication to its caller. (Most library messages are WARNINGs.) A NOTE describes a condition that permits program execution to continue and is not communicated to the caller of the routine that detected the condition.

The library messages generated by ILC support and their severity levels and meanings are as follows. (Terms shown in angle brackets in the messages are replaced by applicable information when the message is issued.)

LSCX033 **Internal error in interlanguage communication.**

Error Level: ERROR

This message indicates an internal library error. Please call the Technical Support Department at SAS Institute.

LSCX035 **Fatal interlanguage communication usage error.**

Error Level: ERROR

This message is issued during processing of a library user 1235 ABEND, which is issued for various unrecoverable usage errors. It should always be accompanied by another message describing the nature of the error.

LSCX051 **ABEND <code> in <language>.**

Error Level: ERROR

This message is sent when an ABEND occurs in a language other than C to identify the ABEND code and the language. Some languages, such as FORTRAN, modify the ABEND code during their own processing, and the ABEND code printed by C in such a case may be the modified rather than the original code.

<language> may also be CDEBUG for an ABEND that occurs in the C debugger.

LSCX052 **ABEND < code > reinstated as 0C6.**

Error Level: ERROR

This message may occur when an 0C*x* ABEND occurs in a language that does not define a handler for this particular ABEND. In this situation, the C framework manager gets control and determines that the ABEND should be allowed to proceed. In some cases, it is not possible for the framework manager to allow the ABEND to proceed without either changing the ABEND code or the perceived location of the ABEND. Because changing the location of the ABEND can cause language diagnostic information to be incorrect, the framework manager instead changes the ABEND code. More exactly, it loads the registers at the time of ABEND and branches to an odd address within a byte of the original ABEND location. Thus, information such as the number of the line that failed should be correct.

LSCX053 **Interlanguage call or return attempted during program termination.**

Error Level: ERROR

This message indicates that the framework for another language had begun to terminate, and then attempted to call (or return to) C. This can occur only if an interlanguage call or return is attempted after the use of some language facility that allows program termination to be intercepted, such as a PL/I FINISH ON-unit.

LSCX054 **Internal error in C library interlanguage communication routines.**

Error Level: ERROR

This is a SAS/C library error. Please call the Technical Support Department at SAS Institute.

LSCX055 **Invalid inter-language call to C.**

Error Level: ERROR

This message is issued when a C function is called from another language but the C framework is already active. The most likely cause of this error is a non-C routine that was not declared with a keyword such as `__cobol`. In some cases, a misdeclared routine of this sort may execute successfully; however, if it calls another C function, this error results.

LSCX270 **The language " < language > " is not defined to the C library.**

Error Level: WARNING

A call to `mkfmwk` or `CFMWK` specified a language that was unknown to the library. Check the spelling of the language. (For instance, specify "PLI", not "PL1".)

The function that failed returns or stores a token value of zero after this error.

LSCX271 **C framework initialization failed due to lack of memory.**

Error Level: WARNING

A call to `CFMWK` failed because there was insufficient memory to initialize the C framework. `CFMWK` stores a zero token after this error.

LSCX272 **Inter-language call failed - target language not defined.**

Error Level: ERROR

A call was made to a routine declared with the `__foreign` keyword, but more than one user-supported language was active, and no "language token" was passed to allow the correct language to be determined.

LSCX273 **Unexpected termination of language framework for <language>.**

Error Level: ERROR

This message indicates an internal library error. Please call the Technical Support Department at SAS Institute.

LSCX274 **Unsupported control structure used in multi-language program.**

Error Level: ERROR

An interlanguage call or return has occurred, but the routine making the call or return is not consistent with the chain of routines that were active at the time of the last interlanguage call or return. This could be caused by a storage overlay. However, it is more likely that use of a GOTO-like control structure in one language (such as the C `setjmp`) has terminated routines in another language. This condition can usually only be detected at the time of an interlanguage call or return; therefore, the actual error may have occurred much earlier than the point at which the message was issued.

LSCX275 **Unable to create language framework for <language>.**

Error Level: WARNING

The framework for the named language failed to initialize. The other language's run-time library should have generated one or more diagnostics describing the condition that prevented initialization. `mkfmwk` returns a zero token to indicate that the framework could not be initialized.

LSCX276 **Argument to dlfmwk is not a valid language token.**

Error Level: WARNING

An argument was passed to `dlfmwk` that is not a valid language token for any active language. This may be the result of an uninitialized variable. It can also occur if an attempt is made to delete a framework more times than it has been created. `dlfmwk` returns a nonzero error code to indicate that the call failed.

LSCX277 **Unable to terminate execution of <language>.**

Error Level: WARNING

A call to `dlfmwk` or `DCFMWK` failed because the framework could not be terminated. Usually, the reason for this message is that one or more routines in the named language were still executing. For example, this message is generated if a C function called from FORTRAN uses `dlfmwk` to attempt to delete the FORTRAN framework.

A nonzero error code will be stored by `dlfmwk` or `DCFMWK` to reflect the failure.

LCX278 **Inter-language communication not permitted for more than one coprocess.**

Error Level: WARNING, ERROR

An attempt was made to create a framework or call a routine in another language in more than one coprocess. If the error is detected by `mkfmwk`, the message is issued as a WARNING, and a zero token is returned to indicate that the framework could not be created. If the error is detected during a call to another language, the message is issued as an ERROR, and a user 1235 ABEND is issued because there is no way to communicate to the program that the call has failed.

LSCX279 **Inter-language call failed - no framework created for <language>.**

Error Level: ERROR

An attempt was made to call a routine in another language, but its framework had not been successfully created. Perhaps a call to `mkfmwk` or `CFMWK` was omitted, or perhaps it failed and the program neglected to check for failure. In the latter case, a previous message should describe the reason for the failure. After this message is printed, the SAS/C library issues a user 1234 ABEND.

LSCX280 **Attempt to terminate language frameworks out of order.**

Error Level: WARNING

Calls to `dlfmwk` or `DCFMWK` were made in an incorrect order. The language whose framework was created last must always be deleted first. After this message, `dlfmwk` or `DCFMWK` stores a nonzero return code.

LSCX281 **<language> framework terminated unexpectedly - attempting to halt execution.**

Error Level: NOTE

The named language terminated while other frameworks were still active. There are many possible causes for this message, including the following:

- A routine in the named language terminated as the result of executing a termination request, such as a C `exit` call or a FORTRAN STOP statement.

- The main routine completed execution without successfully deleting all of the frameworks it created.
- The run-time library for the named language terminated the framework due to an error detected by it, such as a PL/I ON-condition.
- The framework was terminated as the result of debugging. For instance, the SAS/C debugger's `EXIT` command was used.

The SAS/C library responds to this condition by terminating all frameworks and, thereby, program execution.

LSCX282 **Inter-language call or return attempted during program/process termination.**

Error Level: ERROR

This message indicates that the C program had begun to terminate and then attempted to call (or return to) another language. This can occur only if an interlanguage call or return is attempted after `blkjmp` or `atexit` is used to intercept program termination.

This message can also occur if an interlanguage call is attempted during termination of a coprocess, using `blkjmp` or `atcoexit`.

LSCX283 **Additional errors may occur due to <language>'s premature termination.**

Error Level: ERROR

This message is generated after message LSCX281 if the framework that terminated unexpectedly is not the framework most recently created. In this case, the unexpected termination interferes with the SAS/C framework manager's error-handling code. This interference is only temporary in the sense that if termination completes successfully, there should be no residual effects of the interference. However, if an error or program check occurs during termination of the other frameworks, it will probably cause a cascade of errors, with little reliable information about cause or location.

Since this situation cannot be avoided, this message is issued before anything else can go wrong because the chances of issuing it after an error are slim.

LSCX284 **Argument <n> to <language> routine is not a supported data type.**

Error Level: NOTE

An argument passed to a non-C routine is of a data type not supported by the called language. (For instance, a structure was passed to FORTRAN.) The SAS/C library passes the argument by reference, unaltered. (If the call is correct, you can inhibit this message by adding an `&` or `@` operator to the argument yourself, thereby passing it as a pointer.)

LSCX285 **Return value from <language> routine is not a supported data type.**

Error Level: NOTE

The declared return value type for a non-C routine called from C is not a type supported by the called language. (For instance, a PL/I routine cannot return a structure.) The SAS/C library allows the call to proceed, but the handling of the return value is unpredictable.

LSCX286 **Unable to create framework. Too many frameworks currently active.**

Error Level: WARNING

A library table used to contain framework information has filled up, and no more frameworks can be created. This table is shared by all C programs in a virtual machine, or under a single OS TCB. The table contains entries for 20 frameworks. You can normally correct this condition by running fewer C programs simultaneously in a single virtual machine or task.

When this condition occurs, `mkfmwk` or `CFMWK` stores a zero token value to indicate the failure.

LSCX287 **Calling coprocess terminated unexpectedly - attempting to halt other languages.**

Error Level: NOTE

This message is produced when a coprocess (other than the main coprocess) that has created frameworks for other languages terminates without deleting the frameworks. These frameworks are all deleted, but execution of the C program continues.

LSCX288 **<language> framework terminated unexpectedly - attempting to halt calling coprocess.**

Error Level: NOTE

This message is produced when a language whose framework was created by a C coprocess, other than the main coprocess, terminates unexpectedly. All other non-C frameworks, and the coprocess that created the framework, are terminated, but other coprocesses continue to execute if there are no errors during this termination.

LSCX289 **Inter-language call failed due to previous error - errno = <code>.**

Error Level: ERROR

This message is produced when an unexpected error occurs during a call from C to a non-C routine. The previous message will describe this error more fully. The <code> value in the message is the value stored in `errno` by the previous error. Usually, this value will be **ENOMEM**, indicating that there was no memory available to perform the call.

After this message is printed, the library issues a user 1235 ABEND.

Appendix 2
ILCLINK Diagnostic Messages

Introduction

ILCLINK diagnostic messages are divided into five severity levels. Each severity level is associated with a return code. The levels and their associated return codes are shown in **Table A2.1.**

Table A2.1
Severity Levels and Return Codes

Severity Level	Return Code
Note	0
Warning	4
Error	8
Severe	12
Internal	(ABEND)

A *Note* usually displays information about expected behavior. *Warning* messages indicate an error has occurred that will not affect ILCLINK's processing but, nevertheless, should be examined by the user. An *Error* or *Severe* message is produced if ILCLINK encounters a condition that will not allow processing to continue. *Error* messages indicate a problem that is within the user's control, such as an invalid control statement. *Severe* error messages indicate a problem with the environment, such as an out-of-memory condition. Finally, an *Internal* error message (of which LSCI038 is the only example) indicates that ILCLINK has failed to process a correct input file. ILCLINK issues a user ABEND in this situation. ILCLINK will continue processing if no diagnostics are produced with a severity level greater than *Warning.*

The rest of this chapter gives the ILCLINK diagnostic messages and their severity levels, causes, and resolutions. Terms in angle brackets in the messages are replaced by applicable information when the message is issued.

LSCI000 **Not enough memory to continue.**

Error Level: Severe

Systems: All

Explanation: An attempt to allocate memory failed.

Response: Under CMS, increase the size of the virtual machine. Under TSO and OS-batch, increase the region size.

LSCI001 **Internal error in <function>. Code <nn>.**

Error Level: Severe

Systems: All

Explanation: An internal error was detected in the specified function.

Response: Report the function name and code to the Technical Support Department at SAS Institute.

LSCI002 **Input file name truncated to 8 characters.**

Error Level: Warning

Systems: CMS

Explanation: The filename given as the name of the input file contains more than 8 characters. The name will be truncated to 8 characters.

Response: CMS filenames cannot have more than 8 characters.

LSCI003 **Missing input file name.**

Error Level: Error

Systems: CMS

Explanation: No input filename was specified on the command line.

Response: Re-invoke ILCLINK with the name of the input file as the first parameter.

LSCI004 **Unknown option <option> ignored.**

Error Level: Warning

Systems: All

Explanation: The specified option is not an option. The option may have been misspelled.

Response: Re-invoke ILCLINK with the correct option name.

LSCI005 **Missing right parenthesis.**

Error Level: Error

Systems: TSO, OS-batch

Explanation: An option requiring a parenthesized value does not have the terminating right parenthesis.

Response: Re-invoke ILCLINK with the correct form of the option.

LSCI006 **Unable to open <file>.**

Error Level: Error, Warning

Systems: All

Explanation: ILCLINK could not open the specified file. The error level is Warning if the file is an output file, and Error if the file is the input file.

Response: Refer to library messages on `stderr` for more information about the error.

LSCI007 **Invalid parameter <parameter>.**

Error Level: Error

Systems: CMS

Explanation: More than one non-option parameter was used.

Response: Re-invoke ILCLINK using the filename of the input file as the only non-option parameter.

LSCI008 **Unable to determine operating system.**

Error Level: Severe

Systems: All

Explanation: ILCLINK cannot determine the operating system.

Response: Contact the Technical Support Department at SAS Institute.

LSCI009 **Expecting <statement-types> statement.**

Error Level: Error

Systems: All

Explanation: The input file contains control statements that are not in the expected order. The message specifies the types of statements that can occur in the current context.

Response: Correct the statements in the input file.

LSCI010 **Incorrect or missing data in statement.**

Error Level: Error

Systems: All

Explanation: The statement contains values that are misspelled, in the wrong order, or otherwise incorrect, or the statement does not contain an expected value or keyword.

Response: Correct the statements.

LSCI011 **Only one entry point language may be specified.**

Error Level: Error

Systems: All

Explanation: More than one language name was found in the FIRST statement.

Response: Correct the FIRST statement so that only one language name is used.

LSCI012 **Reading input file.**

Error Level: Error

Systems: All

Explanation: An error occurred while reading the input file.

Response: Refer to library messages on `stderr` for more information about the error.

LSCI013 **Statement longer than < nn > characters.**

Error Level: Error

Systems: All

Explanation: A statement in the input file is longer than the maximum allowed. This can occur if the input file has a logical record length greater than 255.

Response: Reformat the input file to use only statements less than or equal to 255 characters.

LSCI014 **No PROCESS statements in input file.**

Error Level: Error

Systems: All

Explanation: No PROCESS statement was found in the input file. A PROCESS statement must be used to produce useful results.

Response: Add PROCESS statements as required to link the program.

LSCI015 **Only one FIRST statement may be used.**

Error Level: Error

Systems: All

Explanation: More than one FIRST statement was found in the input file. There may be only one FIRST statement.

Response: Remove extra FIRST statements from the input file.

LSCI016 **Writing to <file>.**

Error Level: Error

Systems: All

Explanation: An error occurred while writing to the specified output file.

Response: Refer to library messages on `stderr` for more information about the error.

LSCI017 **No non-C languages defined.**

Error Level: Note

Systems: All

Explanation: No languages other than C were named in any FIRST or LANGUAGE statement.

Response: None. This message is informatory only.

LSCI018 **Unknown PROCESS keyword <keyword>.**

Error Level: Error

Systems: All

Explanation: The PROCESS statement keyword is not CLINK, LINK, LKED, LOAD, or GENMOD. One of these keywords must be used in a PROCESS statement.

Response: Correct the PROCESS statement.

LSCI019 **Opening utility file.**

Error Level: Error

Systems: All

Explanation: An error occurred while opening the utility file.

Response: Refer to library messages on `stderr` for more information about the error.

LSCI020 **Return code from <command> was <nn>.**

Error Level: Note, Warning, Error

Systems: All

Explanation: The return code from the specified operating system command or utility is the value shown in the message.

Response: If the return code is unexpected, consult the appropriate operating system documentation for the meaning of the return code. The operating system may have issued other messages, or the library may have issued a diagnostic message on `stderr`.

LSCI021 **SYSTEM command not issued due to <reason>.**

Error Level: Warning

Systems: All

Explanation: Either an error or attention prevented the command in the SYSTEM statement from completing.

Response: If an error occurred, refer to the library messages on `stderr` for more information.

LSCI022 **Entry point name longer than 8 characters. Using <name> as the entry point name.**

Error Level: Warning

Systems: All

Explanation: The entry point name specified in a FIRST statement, linkage editor ENTRY statement, or the RESET option for the LOAD command is longer than 8 characters.

Response: Shorten the entry point name.

LSCI023 **Languages in program are <language-list>.**

Error Level: Note

Systems: All

Explanation: The list contains the language names specified in the FIRST and LANGUAGE statements.

Response: None. This message is informatory only.

LSCI024 **Name <library> truncated to eight characters.**

Error Level: Warning

Systems: All

Explanation: A library name (a filename under CMS, or a DDname under OS-batch and TSO) in an AUTOCALL statement is longer than eight characters. The maximum length of these names is eight.

Response: Correct the AUTOCALL statement.

LSCI025 **No more than <nn> TXTLIBs may be GLOBALed.**

Error Level: Error

Systems: CMS

Explanation: Prior to VM/SP Release 5, the maximum number of GLOBALed TXTLIBs is 8. The total number of TXTLIB names found in the AUTOCALL statement exceeds this number.

Response: Remove TXTLIB names from the AUTOCALL statements. If the application requires more than 8 TXTLIBs, you must merge some.

LSCI026 **Echo < command >.**

Error Level: Note

Systems: All

Explanation: This message is issued when the ECHO option is in effect. ILCLINK has issued the specified operating system command.

Response: None. This message is informatory only.

LSCI027 **Unknown CLINK option < option >.**

Error Level: Error

Systems: All

Explanation: The specified option in a PROCESS CLINK statement is not a CLINK option. The option may have been misspelled.

Response: Refer to CLINK documentation for a list of options.

LSCI028 **Entry point < name1 > in < statement1 > conflicts with entry point < name2 > spec via < statement2 >.**

Error Level: Error

Systems: All

Explanation: Two conflicting entry point names have been detected. The name of the entry point may be specified by a FIRST statement, a linkage editor ENTRY statement, or the RESET option in a PROCESS LOAD statement.

Response: Check the specified statements and ensure that the entry point names do not conflict.

LSCI029 **You must specify an explicit entry point name for < language > programs.**

Error Level: Warning

Systems: All

Explanation: The FIRST statement specified a default entry point name and no default can be chosen. An explicit entry point name must be specified when the FIRST language is COBOL, FORTRAN, or a user-supported language.

Response: Determine the entry point name and add it to the FIRST statement.

LSCI030 **< DDname > FILEDEF is already in effect.**

Error Level: Warning

Systems: CMS

Explanation: A FILEDEF for a TXTLIB named in an AUTOCALL statement has already been issued. ILCLINK will not reissue the FILEDEF.

Response: Clear conflicting FILEDEFs before invoking ILCLINK.

LSCI031 **Concatenated SYSLIB FILEDEFs are not supported by the LKED command.**

Error Level: Warning

Systems: CMS

Explanation: More than one AUTOCALL library name was specified for a PROCESS LKED statement. As of Release 5 of CMS, the LKED command does not support concatentated SYSLIB libraries. ILCLINK will issue the FILEDEFs anyway.

Response: Put all autocalled object code into a single TXTLIB.

LSCI032 **Invalid DDname prefix <prefix> specified by FILES option.**

Error Level: Warning

Systems: TSO, OS-batch

Explanation: The value specified by the FILES option was greater than 3 characters long, or the first character was not A–Z, #, $, or @.

Response: Specify a valid DDname prefix with the FILES option.

LSCI033 **Dataset is already open.**

Error Level: Error

Systems: TSO, OS-batch

Explanation: A data set referred to in the current statement is already open.

Response: Ensure that all data sets that will be used are closed before invoking ILCLINK.

LSCI034 **Specified DDname not found.**

Error Level: Error

Systems: TSO, OS-batch

Explanation: A DDname specified in the current statement has not been allocated.

Response: Ensure that all DDnames that will be used are allocated before invoking ILCLINK.

LSCI035 **In dynamic allocation. Return code <nn>, reason code <nn>, information reason <nn>.**

Error Level: Severe

Systems: TSO, OS-batch

Explanation: An unexpected error occurred while executing SVC 99 (dynamic allocation).

Response: Report the return code, reason code, and information reason code to the Technical Support Department at SAS Institute.

LSCI036 **Dynamic allocation request denied by installation. Return code <nn>, reason code <nn>, information reason <nn>.**

Error Level: Severe

Systems: TSO, OS-batch

Explanation: An SVC 99 request for dynamic allocation failed due to an installation restriction.

Response: Consult a systems programmer at the site for more information.

LSCI037 **JOBLIB, STEPLIB, JOBCAT, or STEPCAT specified on <statement-type> statement.**

Error Level: Error

Systems: TSO, OS-batch

Explanation: One of the above DDnames was used in a statement.

Response: Use another DDname to refer to the data set.

LSCI038 **Invalid dynamic allocation parameter list.**

Error Level: Internal

Systems: TSO, OS-batch

Explanation: ILCLINK created an invalid SVC 99 parameter list. ILCLINK will issue a user ABEND.

Response: Report the error, including the traceback, to the Technical Support Department at SAS Institute.

LSCI039 **Required DDname unavailable.**

Error Level: Error

Systems: TSO, OS-batch

Explanation: A DDname in the current statement has not been allocated.

Response: Ensure that the DDnames used in the input file have been allocated before invoking ILCLINK.

LSCI040 **Unable to allocate <dsname> OLD. Allocated to another job or user.**

Error Level: Error

Systems: TSO, OS-batch

Explanation: The specified data set has already been allocated.

Response: Free any conflicting allocation(s) before invoking ILCLINK.

LSCI042 **Deconcatenation of DDname <ddname> would result in duplicate DDnames.**

Error Level: Error

Systems: TSO, OS-batch

Explanation: A data set concatenation created by ILCLINK contains a DDname that has since been allocated.

Response: Use unique DDnames in ALLOCATE commands issued via the SYSTEM statement.

LSCI043 **SYSTEM statement too long to process.**

Error Level: Error

Systems: All

Explanation: A command specified in a SYSTEM statement exceeds the limit of 500 characters.

Response: Reduce the number of characters in the command. This error is most likely caused by excessive blank space on the command.

LSCI044 **SYSTEM statements may not be issued in OS-batch.**

Error Level: Warning

Systems: OS-batch

Explanation: ILCLINK will not issue a command in a SYSTEM statement under OS-batch.

Response: Remove the statement.

LSCI045 **GENMOD command too long to issue.**

Error Level: Error

Systems: CMS

Explanation: A PROCESS GENMOD statement requires that a GENMOD command longer than 240 bytes be created. ILCLINK cannot create commands longer than 240 bytes.

Response: Re-code the PROCESS GENMOD statement. This error is most likely the result of excessive blank space in the statement.

LSCI046 **Cannot determine first CSECT name.**

Error Level: Error

Systems: CMS

Explanation: An error occurred while reading the LOAD MAP file.

Response: Refer to library messages for more information about the error.

LSCI047 **<utility> abnormally terminated.**

Error Level: Error

Systems: TSO, OS-batch

Explanation: The utility (for example, linker) invoked for the current process ABENDed.

Response: Refer to diagnostics issued by the utility for more information.

Function Index

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

Q

R

S

T

U

V

W

X

Z

Special Characters

0

1

2

3

Your Turn

If you have comments or suggestions about the *SAS/C Compiler Interlanguage Communication Feature, Release 4.00*, or the SAS/C compiler, please send them to us on a photocopy of this page.

Please return the photocopy to the Publications Division (for comments about this book) or the Technical Support Department (for suggestions about the compiler) at SAS Institute Inc., SAS Circle, Box 8000, Cary, NC 27512-8000.

SAS Institute Inc.
SAS Circle □ Box 8000
Cary, NC 27512-8000

ISBN 1-55544-323-0