A multilanguage program is composed of routines written in two or more high-level languages. Multilanguage programs are useful because they facilitate

- use of existing subroutine libraries
- migration of an application from one language to another
- use of languages most applicable to a particular programming task.

Writing and packaging a multilanguage program can be difficult in the IBM® 370 environment. The SAS/C® Interlanguage Communication feature minimizes the difficulties involved. This paper addresses the passing of data among multilanguage programs, with examples using COBOL, FORTRAN, and PL/I. Additionally, because the linking of multilanguage programs is an area for frequent errors, examples of using the ILCLINK utility are provided.

INTRODUCTION

Writing a program in multiple high-level languages is often a challenging task in the IBM 370 environment. This is largely due to the lack of a common run-time environment, such as that found under the VAX/VMS® operating system.

Despite the complexity of the task, writing multilanguage programs is often desirable for the following reasons:

- the natural suitability of a specific language for a specific task
- the desire to migrate from one language to another
- the need to use existing subroutine libraries.

For instance, an engineering system written in FORTRAN might call some C functions to invoke system interfaces. Or, a computer performance monitoring system written in C might call FORTRAN subroutines to perform mathematical analysis and COBOL routines to perform report writing.

Unfortunately, writing a multilanguage program is not as simple as it might appear. Consider the following multilanguage problem. It is not unreasonable for a C programmer to write a program that calls FORTRAN mathematical functions. However, at the very least, this task requires that the programmer possess an understanding of C and FORTRAN data types, conventions for passing parameters and return values, error handling conventions, and execution framework requirements. An assembly language interface routine may even be required, thereby turning a seemingly simple task into a major project. Additionally, the programmer must have a detailed understanding of the linkage editor and the various libraries provided by the language product vendors.

For reasons such as these, SAS Institute developed the SAS/C Interlanguage Communication (ILC) feature. Now, users of the compiler need only call SAS/C library routines to manage the various language environments. Additionally, users follow data type equivalence tables provided in the documentation for passing data elements among the languages. Detailed understanding of the various data formats is rarely required. The ILCLINK utility program makes

in-depth knowledge of the complex linkage process unnecessary. This paper discusses the SAS/C Interlanguage Communication support, concentrating on the proper use of data communication and ILCLINK. Examples using COBOL, FORTRAN, and PL/I are provided.

INTERLANGUAGE COMMUNICATION ISSUES

Execution Framework Requirements

All high-level languages require that an appropriate execution framework exist prior to the execution of the program code. Normally, when a main program begins execution, the execution framework is created by a run-time library initialization routine. It is the responsibility of this initialization routine to do the following:

- acquire storage for the run-time library and establish addressability
- acquire storage for program variables
- open various standard files, such as an error message file
- locate any library routines that are not linked with the program
- establish error handling.

These functions, once performed, constitute the language execution framework. Unfortunately, the details of the execution framework vary widely from language to language. Therefore, each language must have its own framework, and, when calling from one language to another, the frameworks must be switched. As this means that machine registers must be set up to address the components of the framework, this must almost always be done in assembly language.

The SAS/C ILC support provides routines to create and delete execution frameworks for C and the languages supported. For example, prior to calling a routine in another language, a C routine calls the C library mkfmwk routine to create the framework for the language being called. After all the calls to the language are complete, the C routine calls the dfmfwk routine to delete the framework. During execution of the program code, the C library switches and restores frameworks automatically each time an interlanguage call is performed.

Data Types and Parameter Conventions

Although a few data types, such as 4-byte integers, are generic to all languages, each language has its own unique set of data types. This diversity causes a problem in communicating among the various languages. One common example is the packed decimal data type (COMP-3 in COBOL and FIXED DEC in PL/I) that has no corresponding type in C. Another interesting data type problem is that of the character string. C does not have a string data type; a string is a special case of an array of characters. The end of a string is indicated in C by a terminating null (zero) character. FORTRAN and PL/I support a string data type of fixed length that is declared at compile time. Pascal/VS and PL/I support a varying length character string. This type of string is composed of a 2-byte length indicator, followed by the string, and may vary in length during program execution.
Therefore, when communicating between high-level languages, it is
normally necessary to understand the physical representations
of data types for each language involved. Additionally, it is often
necessary to transform data from one physical representation to
another.

Distinct from the problem of data types is the problem that different
languages use different conventions for passing parameters. C is
unusual among the common high-level languages in that C uses the
pass-by-value convention. This means that a copy is made of each
parameter and stored in a parameter block for the called routine.
Other languages, such as COBOL, FORTRAN, and PL/I, use pass-
by-reference. This means that the called routine is given a parame-
ter list containing the addresses of the parameters. This allows the
actual parameters to be shared between routines. Pascal/VS can
use either method on a parameter-by-parameter basis.

Whenever possible, the SAS/C ILC support converts arguments
passed from a C routine to a supported language to the appropriate
data type and uses the appropriate argument passing convention.
For example, when a C function calls a FORTRAN FUNCTION and
passes a C string constant, the argument is converted to a
FORTRAN format string and is passed by reference. In other cases,
the programmer can use type conversion macros to specifically indi-
cate the desired format of the argument. For example, the _ARRAY
macro can be used to pass an array as an array rather than a
pointer.

When C is called from another language, all call-by-reference argu-
ments must be passed as pointers to the appropriate type. The
SAS/C ILC documentation contains data type equivalence tables to
simplify this task.

SAS/C INTERLANGUAGE COMMUNICATION
SUPPORT

Overview
• Support is provided for PL/I, FORTRAN, Pascal, and
COBOL. Additionally, provisions are made for users to add
support for other languages, such as APL.
• Several high-level languages in addition to C may be used in
the same program.
• No assembly code is required for communication with the
supported languages.
• Most data types are handled by the compiler, or by data
type macros.
• Parameters are passed as expected by the called language
through support provided by the compiler in conjunction with
the library.
• Return values from functions are supported.
• Error handling is integrated and controlled so that the
language that should handle the error does so.
• Execution frameworks are coordinated so that preemptive
transfers of control do not cause unpleasant and
unexpected results.
• Functions are provided to establish and remove execution
frameworks for each language. The functions allow for
passing of runtime options.
• A utility program is provided to handle many of the details of
linking multilanguage programs automatically.

• Both the C debugger and other debuggers (such as the VS
FORTRAN debugger) can be used simultaneously.

A brief description of what is involved in writing multilanguage pro-
grams with the SAS/C Compiler follows. Then some of the details
data sharing and linking are explained and illustrated with examples.

Calling C from Other Languages

To call C routines from another language, the following steps are
normally taken:
• Declare the parameters to each C routine as pointers.
• Ensure all arguments passed are of a type usable by C (for
dexample, do not pass packed decimal data).
• Ensure that each C routine returns the data type expected
by the calling routine.
• Compile each C routine called from another language with the
INDep option.
• Before calling any C routines, call the CFMWK routine to
create the C execution framework. CFMWK returns a value
that is later used to destroy the framework.
• When all calls to C have been completed, call the DCFMWK
routine to destroy the framework. The value returned from
CFMWK is passed to DCFMWK to identify the C
framework.

Calling Other Languages from C

To call routines in another language from C, the following steps are
normally taken:
• Declare each called routine to the SAS/C Compiler as a
routine written in that language. Language keywords, such
as _fortran, are provided. For example, to declare the
FORTRAN FUNCTION FORTFC that returns a 4-byte
integer, use the following statement:
...fortran int fortfc();
• Declare each routine to return the correct type of data.
• Verify that the data types of the parameters match C data
types. If there is no ambiguity, then the parameters may be
passed as if another language were not involved. For
example, an _int in C may be passed directly to FORTRAN,
and declared in FORTRAN as INTEGER * 4.

If a data type is unclear, then consult the SAS/C
documentation and code the parameter as a macro, if
required. For example, to pass a character string defined in
FORTRAN as CHARACTER*133 PARM, use the following
macro:
...STRING (parm, 133)

• In the C program, before calling any routines in the other
language, call the mkfmwk routine to create the execution
framework. mkfmwk returns a value that is later used to
destroy the framework. Note that this call is made once for
each other language called.
• When all calls to the other language have been completed,
call the dlfmwk routine to destroy its framework. The value
returned from mkfmwk is passed to dlfmwk to identify the
framework.
SHARING DATA BETWEEN LANGUAGES

Each high-level language has its own set of data types with associated physical representations (data format). Because the different languages do not recognize the data types of other languages, in general you need to use variables with the same data format when sharing data between languages. The common data types will be discussed, and recommendations for sharing the types among languages will be made.

Common Data Types

The 370 architecture supports three numeric data formats in varying sizes: binary, packed decimal, and floating-point. Different languages support various combinations of the formats and sizes. For example, the usual size for floating-point numbers is 8 in C and Pascal and 4 in FORTRAN and PL/I. While COBOL and PL/I support the packed decimal format, C and Pascal do not.

The boolean type is used in many languages to store truth values. The FORTRAN LOGICAL, PL/I BIT(1), and Pascal BOOLEAN are examples. In C, truth values are simply Integers. C also does not have a bit data type that corresponds to the PL/I BIT(n) ALIGNED or the Pascal SET.

C differs from most languages in its definition of character data. In most languages, a single character is regarded as a character string of length 1. In C, a single character is regarded as a small integer. The character literal 'a' and the string literal "a" are different data types. Character strings in C are represented as a sequence of characters terminated by a null (zero) character. In the other 370 languages, strings are either of fixed-length or varying-length. The fixed-length string format, supported by FORTRAN, COBOL, PL/I, and Pascal, is just a sequence of characters. The varying-length string format, supported by PL/I and Pascal, consists of a 2-byte length indicator followed by a sequence of characters of that length.

All languages define array types that are used to store multiple elements of a single data type. In general, the elements are stored contiguously in memory. Sharing array elements between languages normally requires that the arrays be of the same size and contain elements of the same data type. There are a few language-specific items to be considered when sharing arrays. The first element of a C array is addressed as element 0, while the first element in most other languages is addressed as element 1. FORTRAN stores multi-dimensional arrays in column major order, while C and most other languages store arrays in row major order. Additionally, C is unique in that arrays and pointers are treated synonymously. For example, while a C pointer to int may be considered as an array of int, this is not true in other languages.

Most languages support a structure type consisting of one or more named elements of differing types. The elements are normally stored in consecutive locations in memory, often with padding for alignment of the elements. In general, C structures and structures in other languages have the same format, although the alignment rules for the languages vary.

Pointer types are generally stored as 4-byte memory addresses by the various languages that support them. In C and Pascal, pointers to one data type are different than pointers to another data type. In PL/I, POINTER is a generic type.

Sharing Data with SAS/C ILC

In general, to share data between C and another language you should consult the tables provided in the SAS/C ILC documentation. These tables show the type that should be declared for each C parameter corresponding to the argument in the other language. The tables also document whether there are any special considerations for the argument type.

When calling C from another language, the C function should contain a declaration of each parameter to be a pointer to data of the C type corresponding to the data type being passed. (An exception to this would be for arguments passed by value from Pascal. These would be declared to be the equivalent type in C.) The equivalent type, and any special considerations, are easily determined from the documentation. For example, a FORTRAN REAL*4 argument should be declared as float * in the C routine. A COBOL POINTER could be declared as void ** in C.

More support is available from the SAS/C Compiler when passing data to another language from C. The data type equivalency tables should again be consulted in order to determine the appropriate C type. Most arguments can simply be passed directly and will be converted by the compiler. For example, an unsigned short may be passed to a COBOL COMP PIC 9(4). The compiler understands that COBOL expects a pointer to the argument, and passes it accordingly.

Additionally, any argument except for character strings can be passed using a pointer to the appropriate data type. Any argument passed as a pointer or an address expression (using the & or operators) is not converted. Some arguments should be used passing using a data type conversion macro provided by the ILC support. This is indicated in the data type equivalency table. For example, the pdval macro may be used in C to convert data stored in a char array to packed decimal data for passing to COBOL COMP-3 or PL/I FIXED DECIMAL. The _ARRAY macro may be used to pass an array argument to PL/I with appropriate descriptors.

LINKING MULTILANGUAGE PROGRAMS

Creating an executable load module from routines in several languages can be very difficult for several reasons. There is no comprehensive set of documentation available for linking multilanguage load modules; the linking must often be done in several steps; and rules and restrictions for the various languages may affect the others. The SAS/C ILC feature provides the ILCLINK utility to assist in creating multilanguage load modules.

ILCLINK control statements allow the user to specify the following types of information:

- the languages involved and which language is to be invoked first
- the link utilities to be run (such as CLINK, GENMOD, the OS linkage editor, etc.), their options, and any control statements (such as INCLUDE)
- the names of the autocall libraries and any allocate commands required.

ILCLINK invokes the various link utilities required, using the necessary libraries, as directed by control statements provided by the user. ILCLINK has unique knowledge about the SAS/C libraries and the various other language products that enable it to generate appropriate commands for the various link utilities.

The following hints will prevent common errors made when using ILCLINK:

- Be sure not to combine object and load libraries in the same autocall concatenation. The most common occurrence of this is when using the SYSLIB DDname. ILCLINK dynamically concatenates libraries specified in the
AUTOCALL statement to SYSLIB. If, for example, you allocate the SAS/C load libraries to SYSLIB, and specify your object file library as autocall input to CLINK, CLINK will abend.

- When using the LOAD command under CMS, it is wise to specify NOAUTO to prevent autocall from occurring from old TEXT files.

- It is always useful to run ILCLINK with the ECHO option to understand what commands ILCLINK is generating to the various utilities that it is invoking.

- It is important to specify only the SAS/C standard and base libraries and the language libraries needed as AUTOCALL libraries. Including extraneous libraries can result in unwanted routines being included in the load module.

- When linking C and PL/I, it is necessary to run the C code through CLINK in order to remove the pseudoregisters.

- ILCLINK control statements begin in position 2.

EXCEPTIONS

FORTRAN

The following program is a simple example of a FORTRAN program that calls a C routine passing it a string to be printed. The C routine is invoked using the SAS/C Source Level Debugger.

C FORTRAN PROGRAM NAMED MIX

PROGRAM MIX

C TOKEN REPRESENTING THE C FRAMEWORK

INTEGER TOKEN, ERR

CHARACTER*12 S

C CREATE C FRAMEWORK AND USE THE C DEBUGGER

CALL CFKWX( 'FORTRAN' , 'DEBUG' ,0, TOKEN)

IF (TOKEN.EQ.D) STOP 16

S='HELLO, FORTRAN WORLD!'

CALL WRITER(S)

DELETE C FRAMEWORK

CALL DCFHWK(TOKEN, ERR)

IF (ERR.NE.D) STOP 8

STOP

END

"C routine name writer */
#include <stdio.h>
#include <dl.h>
#include fortmath.h
void writer(double x) {
  double cexp,fexp;
  char *s = "C says FORTRAN says";
  for (last = message; last != '');
  printf("%s
",last,message+1,s); return;
}

Here are the ILCLINK control statements that create the module under CMS. In this case, the FORTRAN program is input to the CMS loader, and autocall is performed from the SAS/C and FORTRAN load libraries.

FIRST MIX(FORTRAN)

LANGUAGE C, FORTRAN

PROCESS LOAD MIX (LOADMODEL)

AUTOCALL LC370STL, YLCLINK

INCLUDE WRITER

PROCESS GENMOD MIXER

Here are the ILCLINK control statements that create the load module under MVS. In this case, the SAS/C load libraries are allocated to the LC370 DDname and the VS FORTRAN library to the FORTVS DDname. The FORTRAN and C routines are both specifically INCLUDED to avoid any concatenation of object and load libraries.

FIRST MIX(FORTRAN)

LANGUAGE C, FORTRAN

PROCESS LOAD MIX (LOADMODEL)

AUTOCALL LC370STL, YLCLINK

INCLUDE WRITER

NAME MIXER(R)

The following example illustrates a C main program that dynamically loads a module consisting of C and FORTRAN. Additionally, this example illustrates the following points:

- C passes a string argument to FORTRAN.
- The FORTRAN subroutine returns a value to C.
- The C and FORTRAN routines call the appropriate versions of the EXP function.

C main routine that performs the dynamic load */
#include <stdio.h>
#include <dynam.h>
#include <string.h>

int (*fp)();

int main( ) {
  int (*fp)();
  if (fp) {
    printf("now calling the C/FORTRAN module
");
    (*fp)(d);
    unload(fp);
  } else
    printf("EXPTST failed to load\n");
    exit(0);
}

Here is the loaded C routine named sugdynam in the user's object library */
#include <stdio.h>
#include <sic.h>

#include fortmath.h to rename C exp function to _exp to avoid link conflict with FORTRAN EXP */
#include <fortmath.h>

void _exp(double x) {
  double cexp,fexp;
  char *s = "C says FORTRAN says";
  for (last = message; last != '');
  printf("%s
",last,message+1,s); return;
}

The following example illustrates a C main program that dynamically loads a module consisting of C and FORTRAN. Additionally, this example illustrates the following points:

- C passes a string argument to FORTRAN.
- The FORTRAN subroutine returns a value to C.
- The C and FORTRAN routines call the appropriate versions of the EXP function.

C main routine that performs the dynamic load */
#include <stdio.h>
#include <dynam.h>
#include <string.h>

int (*fp)();

int main( ) {
  int (*fp)();
  if (fp) {
    printf("now calling the C/FORTRAN module
");
    (*fp)(d);
    unload(fp);
  } else
    printf("EXPTST failed to load\n");
    exit(0);
}

Here is the loaded C routine named sugdynam in the user's object library */
#include <stdio.h>
#include <sic.h>

#include fortmath.h to rename C exp function to _exp to avoid link conflict with FORTRAN EXP */
#include <fortmath.h>

void _exp(double x) {
  double cexp,fexp;
  char *s = "C says FORTRAN says";
  for (last = message; last != '');
  printf("%s
",last,message+1,s); return;
}
printf("exp = \'%.16E\' \n exp, exp);
}  

C HERE IS THE FORTRAN FUNCTION EXPF
FUNCTION EXPF(X, S)
REAL'S EXPF, X
CHARACTER· t' 
WRITE (6,1) S
FORMAT (',A)
EXPF=EXP(X)
RETURN
END

Here are the CMS ILCLINK control statements necessary to create
the load module containing the C and FORTRAN code. The calling
C load module is linked as for a normal C program. The loaded C
object code, SUGDYNAM, is linked with the SASfC object libraries
using the CLINK utility. The output from CLINK is passed by
ILCLINK to the OS-style linkage editor. The FORTRAN function,
EXPF, is INCLUDED from the DDname ALIB, and the VS FORTRAN
library is used for autocall.

FIRST IDYAMN(C)
LANGUAGE C,FORTRAN
PROCESS CLINK {NOAUTO
AUTOCALL LC370BAS,LC310STD
INCLUDE SUGDYNAM
SYSTEM FILEDEF ALIB DISK MYLIB TXTLIB A
PROCESS LKED (LIBE NEWLIB NAKE EXPTST XREF MAP
AUTOCALLVFORTLIB
INCLUDE ALIB( EXPF)

COBOL

The following example illustrates a simple technique that can be
used to bypass the ILC restriction that the COBOL dynamic loading
feature cannot be used to load a C function. This application con­
sists of a COBOL main program that dynamically loads a load mod­
ule consisting of a COBOL shell calling C using the SAS/C ILC
feature.

IDENTIFICATION DIVISION.
PROGRAM-ID. COBOMAIN.
REMARKS.
This COBOL program dynamically loads a load module
composed of COBOL and C. This program must be
compiled with the DYNAM option.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 PARM-AREA.
03 PARM-LEN PIC 9(11) COMP.
03 PARM-TXT PIC X(3).
PROCEDURE DIVISION USING PARM-AREA.
999-STOP.
STOP RUN.

COBOL

The following example illustrates a simple technique that can be
used to bypass the ILC restriction that the COBOL dynamic loading
feature cannot be used to load a C function. This application con­
sists of a COBOL main program that dynamically loads a load mod­
ule consisting of a COBOL shell calling C using the SAS/C ILC
feature.

IDENTIFICATION DIVISION.
PROGRAM-ID. COBOMAIN.
REMARKS.
This COBOL program dynamically loads a load module
composed of COBOL and C. This program must be
compiled with the DYNAM option.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 PARM-AREA.
03 PARM-LEN PIC 9(11) COMP.
03 PARM-TXT PIC X(3).
PROCEDURE DIVISION USING PARM-AREA.
999-STOP.
STOP RUN.

IDENTIFICATION DIVISION.
PROGRAM-ID. COBOMAIN.
REMARKS.
This COBOL program dynamically loads a load module
composed of COBOL and C. This program must be
compiled with the DYNAM option.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 PARM-AREA.
03 PARM-LEN PIC 9(11) COMP.
03 PARM-TXT PIC X(3).
PROCEDURE DIVISION USING PARM-AREA.
999-STOP.
STOP RUN.

IDENTIFICATION DIVISION.
PROGRAM-ID. COBOMAIN.
REMARKS.
This COBOL program dynamically loads a load module
composed of COBOL and C. This program must be
compiled with the DYNAM option.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.

PL/I

The following example illustrates a C main program that passes a
structure to various PL/I subroutines:

#include <stdio.h>

main()
{
    /* declare the PL/I routines */
    extern _pli void vstrpl1();
    extern _pli void vstrpl2();
    extern _pli void vstrpl3();
    void*pil;token;
    int intp = 0;
    struct
      short len;
      char data[17];
      vstrpl1();
      printf("returned from 1st call
      
    vstrpl2(intp, vstrp);
    printf("returned from 2nd call
    
    vstrpl3(intp, vstrp, &structpp);
    printf("returned from 3rd call

    if (!pli_token)
      exit(16);
    return;
}

Here is the ILCLINK data file to link the COBOULC module under
MVS. The C code is linked with the SAS/C libraries allocated to the
LC370 DDname using the CLINK utility. The output is passed to the
linkage editor, where the COBOL shell is included, and the load
module is given the name the COBOL main program uses in the
CALL statement.

FIRST IDYAMN(C)
LANGUAGE C,FORTRAN
PROCESS CLINK
AUTOCALL LC370
INCLUDE MYLIB (CPROG)
PROCESS LKED (XREF LIST CALL
AUTOCALL COBOL
INCLUDE MYLIB(COBESHEL)
NAME COBESHEL(S)
Here are the ILCLINK control cards required. Note that the CLINK process should always be used when linking C and PL/I code so that the C pseudo-registers are removed.

```plaintext
FIRST HAIN(C)
LANGUAGE C,PLI
PROCESS CLINK (PREM

The important DD statements for the ILCLINK job step above are as follows:

```
/ILC370 DD DSN=SASC.C450F.STDOBJ  
// DD DSN=SASC.C450F.BASBOBJ  
// DD DSN=SASC.C450F.ILCOBJ  
II MYLIB DO DSN=NUSRID.C.OBJ  
II LOADLIB DO DSN=NUSRID.LOAD  
II DSN=SYSPLIBASE  
```

CONCLUSION

The SAS/C Interlanguage Communication support makes writing a multilanguage program a fairly straightforward process if the programmer follows the rules presented in the documentation. The support includes data equivalency tables, data conversion support when calling from C, and a utility to aid in the ILC linkedit process. The resulting program functions in a reliable fashion under the various error situations that can be encountered in the production environment.

ACKNOWLEDGMENTS

The author wishes to thank George Wilder of SAS Institute for his help with the examples in this paper.

SAS/C is a registered trademark of SAS Institute Inc., Cary, NC, USA.

IBM is a registered trademark of International Business Machines Corporation, Armonk, NY.

VAX and VMS are trademarks of Digital Equipment Corporation, Maynard, MA.