ABSTRACT

A multilanguage program is composed of routines written in two or more high-level languages. Multilanguage programs are useful in the IBM® 370 environment for several reasons. Many subroutine libraries in various languages are available to perform standard or specialized functions, such as mathematical or statistical analysis. Additionally, it is often desirable to migrate gradually from one programming language to another, such as from PL/I to C. Thirdly, there are functions that are best performed in a particular programming language, such as producing a complex report with COBOL. A typical multilanguage application will freely combine routines in several languages. This mingling of languages often requires that the programmer have a detailed understanding of the varied environmental requirements of each language, even though the actual programming task does not have such requirements.

There are a number of issues involved in ease of communication between C and other languages. Some of these issues are differing data types, linkage conventions, error handling, parameter and return value handling, and language execution framework requirements.

This paper addresses these issues and illustrates the SAS/C® interlanguage communication support using FORTRAN as a primary example.

INTRODUCTION

Writing a program in multiple high-level languages is often a challenging task in the IBM 370 environment. A major reason is the lack of a common runtime 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 to expect a C programmer to write a C 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.

In this paper we will define and examine the issues involved in writing programs in multiple high-level languages. For purposes of this paper, a high-level language is defined as a third-generation language, such as C, PL/I, COBOL, FORTRAN, or Pascal, including a runtime library of supporting routines provided by the compiler vendor. Communication with Assembler is not addressed, as Assembler is not a high-level language, and the issues are not as complex. This paper describes how the next release of the SAS/C product will approach these issues and illustrates with a brief example of a C program calling FORTRAN.

INTERLANGUAGE COMMUNICATION ISSUES

Execution Framework Requirements

All high-level languages require that a library routine be called to perform initialization before starting program execution. It is the responsibility of this initialization routine to do the following:

- acquire storage for the runtime 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.

Data Types

Although a few data types are generic to all languages, such as 4-byte integers, each language has its own unique set of data types. 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 two-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.
Parameter Passing

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 parameter list containing the addresses of the parameters. This allows the actual parameters to be shared between routines. Pascal/VG can use either method on a parameter-by-parameter basis.

Return Values

A function is a routine that returns a value. A subroutine is a routine that does not return a value. Returning a value is a common practice, both in standard function libraries and for many applications. Some high-level languages (including SAS/C) use the standard IBM 370 convention of placing the integer return value in a general purpose register. FORTRAN, however, returns integer values in register 0. Other languages, such as PL/I and Pascal, pass an extra argument for the return value. The possible solutions to this problem are to write an assembly language bridge to be shared between routines. Other languages, such as COBOL, FORTRAN, and PL/I provide an option (INDep) that allows any C routine to be called. Returns and Execution Framework Cleanup

When a language terminates execution, the library must perform various housekeeping tasks, such as returning allocated storage to the operating system, closing files, removing error-handling routines, and in general, removing all traces of itself. Since there is no common execution framework, when a halt (such as a COBOL STOP RUN) or other preemptive transfer of control is issued in one language, the other languages have no opportunity to clean up their execution frameworks. This is illustrated by the PL/I and C divide-by-zero example above. When PL/I terminates, the entire program may terminate without control being passed through C.

The consequences of unterminated error routines, for example, can be quite serious. If a language’s ESTAE exit is left active after the language halts, the exit may be entered for an unrelated ABEND of some other program in the same job step or virtual machine. This condition will likely cause a new ABEND to occur, making diagnosis of the original problem much more difficult.

Additionally, the rules about where control returns after halts are varied and difficult to understand. For example, to call FORTRAN from another language, the programmer must first establish the FORTRAN execution framework through a call to a library routine, VFEIN#. When EXIT is called from FORTRAN, the return is always to the caller of the caller of VFEIN#. This results in situations where control is returned to entirely unexpected places, depending upon how the calling structure of the program is configured.

POSSIBLE SOLUTIONS

Here are some possible solutions to the problems described above.

• Invest many hours of high-level programming time building an interface that has all or most of the following problems:
  • is dependent upon changes language vendors may make
  • is complex and difficult to maintain
  • invariably involves assembly language
  • may not function as desired in error and shutdown situations.

• Rewrite existing libraries so that everything is in one language.

• Mandate that all applications will be in one language and avoid writing routines in the most suitable language.

• Use the SAS/C Interlanguage Communication Support.

SAS/C INTERLANGUAGE COMMUNICATION SUPPORT

Background

Since SAS/C adheres to standard 370 linkage conventions, calling COBOL routines and FORTRAN subroutines from SAS/C has always been relatively easy. Additionally, the SAS/C compiler provides an option (INDep) that allows any C routine to be called from another language by ensuring that the C execution framework is made accessible at routine entry. However, even with these features, many of the interlanguage communication problems still exist.

Because of the popularity of writing multilanguage applications and the problems encountered by users as previously described, SAS Institute decided to provide specific support for interlanguage communication. The next release of the SAS/C compiler will provide a usable approach to all these problems with its interlanguage communication (ILC) support. A summary of the support features follows.
Overview of SAS/C Interlanguage Communication Support
• Support is provided for PL/I, FORTRAN, 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 programs with the SAS/C compiler follows. Then some of the features of the next SAS/C release are illustrated using C and FORTRAN examples.

Calling C from Other Languages
To call C routines from another language, the following steps are normally taken:
• Declare the parameters to the C routine as pointers.
• Ensure all arguments passed are of a type usable by C (for example, do not pass packed decimal data).
• Ensure the C routine returns the data type expected by the calling routine.
• Compile the C routine with the INDep option.
• Before calling any C routines, call the CFMWK routine to create the 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 its framework. The value returned from CFMWK is passed to DCFMWK to identify the framework.

Calling Other Languages from C
To call routines in another language from C, the following steps are normally taken:
• Declare the 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 forfc that returns a 4-byte integer, use the following statement:
  _fortran int forfc();
• Declare the 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.

The appendix of this paper contains a sample table of equivalent data types and macros. 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.

ILLUSTRATION
The following example is a segment of a C routine that calls a function from a hypothetical library of FORTRAN plotting routines with the label for an axis.

```c
#include <stdio.h>
#define MAXLEN 40

int main()
{
    char axis[MAXLEN + 1];
    int rc;
    void *fmwk;

    forfc();
    // fortran label() returns the data type expected by the calling routine.

    // CFMWK routine to create the execution framework.
    CFMWK returns a value that is later used to destroy the framework.
    // mkfmwk routine to create the execution framework.
    // mkfmwk returns a value that is later used to destroy the framework.
    // DCFMWK routine to destroy its framework.
    // The value returned from mkfmwk is passed to dlfmwk to identify the framework.

    // mkfmwkroutine to create the execution framework.
    mkfmwk("FORTRAN", "");
    // mfmk routine to create the execution framework.
    // mfkmk returns a value that is later used to destroy the framework.
    // DCFMWK routine to destroy its framework.
    // The value returned from mkfmwk is passed to dlfmwk to identify the framework.

    // read axis label from stdin */
    // if no input, pass a standard label */
    // if input, use compiler to compute length */
    // otherwise, pass the string and allow compiler to compute length */
    // else
    // if (rc)
    // printf("Error from LABEL - return code %d", rc);
    // dlfmwk();
    return rc;
}
1. Declare the function "label" to be in FORTRAN using the _fortran keyword.

2. Initialize the other language framework, identifying it as FORTRAN. The null second argument indicates no runtime options are being passed to FORTRAN. Save the framework token returned by mkfmwk.

3. If no axis label input was entered, pass a standard literal string.

4. If input was entered, pass the input string, and let the compiler compute the string length (indicated by a string length of 0).

5. Use the return value from the FORTRAN function.

6. Delete the FORTRAN framework, using the token returned from mkfmwk.

The relevant FORTRAN declarations for the label function are as follows:

```
C SAMPLE FORTRAN PLOTTING ROUTINE DECLARATIONS

INTEGER FUNCTION LABEL(AXIS)
CHARACTER*(*) AXIS

C AND ETC.

LABEL
RETURN
END
```

The following example shows a C routine that calls the FORTRAN library ITPACK 2C. ITPACK 2C is a library of FORTRAN subroutines for solving large sparse linear systems, such as those used in developing VLSI systems. The purpose of this example is to illustrate the similarity of calling a package such as this from C and from FORTRAN.

```c
#include <ilc.h>
define N 4
#define NW 24
#define NZ 8

fortran void dfault();
---.fortran void vfill(N, u, EO);
---.fortran void jcg();

/* header file */
/* order of linear system */
/* dimension available for wksp */
/* row pointers for ja and a */
/* column numbers for a */
/* nonzero entries of matrix */
/* right-hand side of linear system */
/* guess to solution on input */
/* latest approximate solution on output */
/* integer work space */
/* real work space */
/* initialization for integer parms */
/* initialization for real parms */
/* error flag */

int max = 4;
level = 1;
idgts = 2;

float EO = 0.0;

/* initialize FORTRAN environment */
fmwk = mkfmwk("FORTRAN", "");

/* use ITPACK default parameter values */
default(parm, rparm);
parm[1]= level; /* messages control level */
parm[11]= idgts; /* error analysis level */

/* fill vector c with 0 */
vfill(N, u, EO);

/* call the Jacobi Conjugate Gradient iterative method routine */
jcg(N, ia, ja, a, rhs, u, wksp, nw, rparm, parm, ier);

dfmwk(fmwk);
return ier;
```

1. Declare the FORTRAN subroutines using the _fortran keyword.
2. Initialize the FORTRAN framework, save the framework token, and pass no runtime arguments.
3. Call the ITPACK 2C routines. Since the expected parameter types in FORTRAN (INTEGER and REAL) have C equivalents (int and float), no macros are needed. The compiler and library ensure that the parameters are passed by reference.
4. Delete the FORTRAN framework, using the token returned from mkfmwk.

Here is the equivalent FORTRAN code:

```
ROUTINE TO CALL ITPACK 2C

INTEGER FUNCTION ITPCKF
INTEGER IA(N+1),JA(N+1),IPARM(12),IWKSP(12)
REAL A(N+1),RHS(4),U(4)
REAL WKSP(24),RPARM(12)

USE ITPACK DEFAULT VALUES
CALL DFAULT(IPARM, RPARM)
IPARM(1) = ITMAX
IPARM(2) = LEVEL
IPARM(12) = IDGTS

C FILL VECTOR U WITH 0
CALL VFILL(N, U, E0)

C CALL JACOBI CONJUGATE GRADIENT ITERATIVE METHOD ROUTINE
CALL JCG(N, IA, JA, A, RHS, U, IWKSP, NW, WKSP, IPARM, RPARM, IER)
ITPCKF = IER
RETURN
END
```

CONCLUSION

Writing a multilanguage program for the IBM 370 environment is rarely as simple as it might appear. Achieving cooperative interaction among languages is difficult, and it often involves assembly language programming. The goal of the SAS/C interlanguage communication support is to insulate the programmer from many of the issues and to reduce the effort required to achieve a robust and maintainable multilanguage program.

APPENDIX

The following sample table illustrates some of the data types from the supported languages, the corresponding C data type (if any), and the format required to pass the type from C.
<table>
<thead>
<tr>
<th>FORTRAN Type</th>
<th>C Equivalent</th>
<th>How to Pass</th>
<th>COBOL Type</th>
<th>C Equivalent</th>
<th>How to Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER*4</td>
<td>int i;</td>
<td>foutfc(i);</td>
<td>COMP PIC 59()</td>
<td>int i;</td>
<td>compfc(i);</td>
</tr>
<tr>
<td>INTEGER*2</td>
<td>long l;</td>
<td>(l);</td>
<td>COMP PIC 53()</td>
<td>long l;</td>
<td>(l);</td>
</tr>
<tr>
<td>REAL*4</td>
<td>float f;</td>
<td>(f);</td>
<td>COMP PIC 59()</td>
<td>short s;</td>
<td>(s);</td>
</tr>
<tr>
<td>REAL*2</td>
<td>double d;</td>
<td>(d);</td>
<td>COMP PIC 9()</td>
<td>unsigned short ush;</td>
<td>(ush);</td>
</tr>
<tr>
<td>LOGICAL*4</td>
<td>char c;</td>
<td>(c);</td>
<td>COMP PIC 9()</td>
<td>unsigned long ul;</td>
<td>(ul);</td>
</tr>
<tr>
<td>CHARACTER*1</td>
<td>char s[8];</td>
<td>(.STRING(s, 8));</td>
<td>PIC X</td>
<td>char c[ ];</td>
<td>(.STRING(c, 8));</td>
</tr>
<tr>
<td>CHARACTER*8</td>
<td>char *;</td>
<td>(.STRING(s, 8));</td>
<td>PIC X</td>
<td>char *s[ ];</td>
<td>(.STRING(c, 8));</td>
</tr>
<tr>
<td></td>
<td>array</td>
<td>(<em>literal</em>);</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PL/I Type</th>
<th>C Equivalent</th>
<th>How to Pass</th>
<th>COBOL Type</th>
<th>C Equivalent</th>
<th>How to Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIXED BIN(3)</td>
<td>short sh;</td>
<td>plifc(sh);</td>
<td>COMP PIC 59()</td>
<td>int i;</td>
<td>compfc(i);</td>
</tr>
<tr>
<td>FIXED BIN(31)</td>
<td>int i;</td>
<td>(i);</td>
<td>COMP PIC 53()</td>
<td>long l;</td>
<td>(l);</td>
</tr>
<tr>
<td>FLOAT DEC(6)</td>
<td>float f;</td>
<td>(f);</td>
<td>COMP PIC 59()</td>
<td>short s;</td>
<td>(s);</td>
</tr>
<tr>
<td>FLOAT DEC(16)</td>
<td>double d;</td>
<td>(d);</td>
<td>COMP PIC 9()</td>
<td>unsigned short ush;</td>
<td>(ush);</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>char s[n]</td>
<td>(.STRING(s, n));</td>
<td>PIC X</td>
<td>char s[ ];</td>
<td>(.STRING(c, n));</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>struct {char val[n];} s;</td>
<td>(<em>literal</em>);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIT(n)</td>
<td>unsigned char</td>
<td>(.BIT(b, n));</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pointer</td>
<td>type *p;</td>
<td>(p);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>array</td>
<td>type area;</td>
<td>(are);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type a[n];</td>
<td>(.ARRAY(a, n));</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type *a;</td>
<td>(.ARRAY(a, n));</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type a[n][n2];</td>
<td>(.ARRAY(a, n, n2));</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type *a;</td>
<td>(.ARRAY(a, n, n2));</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

REFERENCE


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

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

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