INTERFACING IBM VS FORTRAN VERSION 2 WITH
SAS/CTM VERSION 3.00F

Stephen L. Maier, HADRON, Inc.

ABSTRACT

This paper addresses the interfacing obstacles and solutions identified while developing a library of 'C' programs to be used as a subroutine library for a VS FORTRAN program. The initial difficulty in interfacing program modules written in these two languages is in the calling conventions and the run-time environment. However, the task does not stop there. Basic language differences such as data structures and format complicate the issue. The SAS/C Compiler addresses many of these issues through the use of a compiler option to generate independent code that can be called from any host language which conforms to the IBM 370 linkage conventions. To complicate the issue even more, IBM's VS FORTRAN does not fully conform to the IBM 370 linkage convention. The result requires a sophisticated interface library of subroutines which 'bridge' the interface between the 'C' functions and the FORTRAN main program. The intricacies and problems in developing the 'bridge' routines is presented.

Introduction

The task of interfacing any two languages requires a great deal of analysis to determine whether or not they conform to the same standard, and if not, just how the interface on both ends is designed. This analysis must include the assembler code generated by the compiler. Two items of interest need to be determined, first, the transition of control from program to subroutine, and second, the method of transferring arguments or program variables from the program to the subroutine. Both the IBM VS FORTRAN compiler and the SAS/C Compiler conform to the IBM 370 linkage convention for transfer of control, so it is not considered an issue in this paper. The argument passing conventions do differ, and are covered in detail in the following section. One other problem that arises is the difference in environment required by the two languages which is discussed in the section on Run-Time environments. Finally, the arguments themselves passed from calling program to subroutine differ widely between the languages and are discussed in the section on Data Structures.

Calling Conventions

The calling conventions used between the two compilers to pass arguments, values, variables, etc. differ significantly. First, FORTRAN passes all arguments by reference. This means that every argument identified in the calling argument list is passed to the called subroutine by providing the memory address of the argument. The 'C' compiler on the other hand allows both call by reference and call by value. Thus, all interface routines written in 'C' which are called from FORTRAN must expect their arguments as pointers (addresses) to their respective data types. The structure of the FORTRAN argument list reveals some additional information which is usable in creating the bridge routines.

IBM VS FORTRAN Compiler

VS FORTRAN essentially creates two separate argument lists for each call to a function or subroutine. The first, and primary, argument list is a list of the addresses of the variables to be passed to the subroutine. The last argument in the list has its most significant bit (MSB) set to identify
it as the last argument to the subroutine. The address of this list is then put into REGISTER 1 prior to the transfer of control. The receiving subroutine saves this value to use in accessing the passed arguments. Adjacent to this first argument list in memory, is placed a second list which contains the lengths of the arguments passed in bytes. The last length argument is also flagged by setting the MSB. A subroutine call which passes three (3) arguments actually passes 6 values to the subroutine, 3 variable addresses, and 3 length addresses. Figure 1 shows the format of the argument list as described.

The FORTRAN compiler also allows a subroutine to determine an alternate processing path through the use of alternate return addresses. These addresses identified in the argument list in the form *[label] are removed by the compiler from the argument list passed to the subroutine. The programmer may place any number of alternate return arguments in the list, and they may be interspersed freely among the other types of arguments. The order of the alternate return arguments in the parameter list identifies the conditions upon which each alternate return is taken. FORTRAN does not maintain the alternate return arguments as an argument list, but generates code to conditionally branch to the labels identified by the alternate return address based on the value of a condition code returned by the subroutine. The condition code is taken from general register 15 upon return from the subroutine, and branches to the first alternate return if the value returned was 4, the second if the value returned was 8, etc. A value of zero in register 15 results in the execution of the first instruction immediately following the subroutine call. Therefore, a subroutine can control the execution of its caller through this mechanism.

<table>
<thead>
<tr>
<th>Addr N</th>
<th>Address Argument 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Address Argument 2</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td>1</td>
<td>Address Argument M</td>
</tr>
<tr>
<td></td>
<td>Address Length Argument 1</td>
</tr>
<tr>
<td></td>
<td>Address Length Argument 2</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td>1</td>
<td>Address Length Argument M</td>
</tr>
</tbody>
</table>

Figure 1

IBM VS FORTRAN Argument List Data Structure

SAS/C Compiler

The 'C' language is a stack based language. Arguments to a 'C' function or subroutine are placed on the stack prior to each function call, then removed upon return. In addition, the 'C' language supports call by value as well as call by reference. Therefore, algorithms which are recursive in nature can be implemented in 'C' very easily. Compilation of a 'C' subroutine with the 'independent' compiler option results in a 'C' subroutine which can be called from and receive arguments from, a FORTRAN calling program. The receiving 'C' function must declare all dummy arguments as addresses. The 'C' function may then perform it's processing functions and return to the calling program. Upon return from a SAS/C function, the epilogue code places the function return value into general register 15. VS FORTRAN expects a return value in general register 0. The result is that the calling FORTRAN program receives random values left in register 0 by the 'C' subroutine, and attempts to use the function return value as the condition code for determining the alternate return path to take. The actual 'C'

1 FORTRAN may use a register pair or a floating point register depending upon the type of the return value, but still primarily uses register 0.

13
return value will be lost to the FORTRAN calling program. In addition, the FORTRAN code will incorrectly branch to an alternate return if (1) the return value coincides with the value required to pass the conditional branch test, and (2) the function was called with alternate return addresses.

Run-Time Environment

The 'C' language requires a very different run-time environment than FORTRAN. The 'C' language makes extensive use of a run-time stack, permits automatic allocation of variables, and many other features which take advantage of global information, all of which must be available upon execution of any 'C' function. This run-time is not available in FORTRAN, and must be initialized by the SAS/C prologue code upon execution of the first 'C' subroutine or function. This environment set-up code is normally inserted into the 'C' main program, however, since this main program is in FORTRAN, the environment setup code is inserted, by the compiler, into each function or subroutine which is compiled with the 'independent' compiler option. This setup code check for the existence of the 'C' run-time environment before creating one, but still requires extra overhead to perform the check. For this reason, it is desirable to limit the number of 'C' functions compiled with this option to only those which are directly called from another environment.

Data Structures

Both the FORTRAN and 'C' compilers support multiple data types. For the basic data types of integer, floating point, and byte, both languages agree on supporting the hardware defined formats, however, beyond that the differences abound. Character strings in FORTRAN are padded with blanks up to the length of the string. The 'C' language requires that a character string be terminated with a NULL (0) character. This simple difference can cause many 'C' programs to fail catastrophically. FORTRAN supports a data value known as a complex number. 'C' on the other hand must define this value to be a 'structure' or user defined data type as follows:

```c
struct complex
{
    float real;
    float imaginary;
};
```

The 'C' program must deal with the parts individually, where the FORTRAN compiler can handle them as simply as floating point numbers. The 'C' language can create unbelievably complex data structures using the above 'struct' declaration, and a similar 'union' declaration. These differences can make the interface programming task very difficult if they must be converted in the interface process. Both languages support arrays, however, FORTRAN implements multi-dimensional arrays using column major order (left index varies most frequently), and 'C' implements them as row major (right index varies most frequently). This difference can only be efficiently be implemented through modification of the program algorithms to adjust for the inter-language differences.

Bridge Routine Conventions

The solution to the interface problem is to generate small 'C' functions which act as a bridge between VS FORTRAN and SAS/C. Each bridge routine should be declared as a function returning type integer. If the calling program will never use alternate return path arguments, the bridge routine should only return with a value of zero (0). If alternate return path arguments are to be used in the call from FORTRAN, the bridge code should determine what value (0, 4, 8, ...) is to be returned and under what conditions. All calls to 'C' functions will then be programmed as subroutine calls in FORTRAN using the syntax:

```
CALL SUBXYZ (A, B, ...)"
```

The bridge routine must convert character string arguments into the appropriate 'C' null terminated format. This can be achieved easily by allocating sufficient space for the string in automatic or dynamic storage, then using the 'C' strncpy function to copy the non-null terminated string into the temporary space, and then setting the end of string position to the position following the last non-blank character. Prior to returning to the calling FORTRAN subroutine this
space should be freed. The address of the length of the character string will
be in the argument list, at position N + M, where N is the number of arguments
in the calling list, and M is the position of the character string in the
calling list. It will appear to novice programmers that the argument lists do
not match, and that a return value is being lost in the bridge routines. This
should serve as a warning to document the bridge code extensively.

FORTRAN MAIN
PROGRAM

'C' BRIDGE
ROUTINE

'C' FUNCTION

Figure 2
Layering Methodology for Bridge Routines

Recommendations

Developing a set of 'bridge' routines in 'C' will reduce the effort required to interface these two languages to achieve maximum potential out of both of them. The bridge routines should be set up as a library separate from the 'C' function library to ensure portability for future projects. Compile all of the 'bridge' routines with the 'independent' compiler option to ensure correct operation and do not compile the 'C' function library with the independent option to maintain as much efficiency as possible.

Avoid passing complex data types between FORTRAN and 'C' which must be manipulated by both languages. Also avoid multiple dimensioned arrays to the extent possible, 'C' function code which deals directly with FORTRAN arrays will most likely be not portable.

Wherever possible, pass character strings from FORTRAN to 'C' as string literals, or FORTRAN parameters. This will reduce the task of removing trailing blanks from character strings.

Investigate the length of external subroutine names acceptable by the FORTRAN compiler. VS FORTRAN Version 2.0 limits function names to six (6) characters, and the 'C' bridge routines must also conform to this limit.

Conclusion

It is possible to interface these two very powerful programming languages using the guidelines defined above. Selection of which language to be used should be based on which language offers the greatest potential programmer productivity and machine efficiency. Some discipline defining the number and function of 'C' routines to be called from FORTRAN will result in a smaller, more manageable bridge routine library. The result of this capability will be increased productivity through the use or selection of a language appropriate to the function to be accomplished.

For more information, contact:
Stephen L. Maier
Hadron, Inc.
9841 Broken Land Parkway
Suite 208
Columbia, MD 21076