Fortran 66 to 77 Conversion: Problem Considerations in an Integrated Environment

Brant L. Kelly
Bradmark Computer Systems
4265 San Felipe
Suite 820
Houston, Texas 77027

Introduction

With the introduction of the Precision Architecture line of computers, HP has made its Fortran 66 compiler obsolete. Along with the introduction of the Fortran 77 compiler, HP provided some tools to help the programmer in converting Fortran 66 source to Fortran 77 source. After converting a system written in Fortran 66 to Fortran 77, I gleaned some knowledge that is not clear in the documentation. The system that I converted is a group of fortran subroutines that calls system intrinsics. The steps outlined in the migration manual for converting Fortran 66 source are clear enough, so this paper is not a "how to" guide. The purpose of this paper is to share my experience in a conversion of interacting programs in a system. The conversion facility from HP is a good tool if the programmer has some previous knowledge on its use. This paper covers internal representation of Fortran 77 data types with the interest of helping the programmer in calling external routines that do not use the same default data sizes as Fortran 77.

I. After the Migration Aid: What Now?

You spent hours of reading and experimenting to understand just what the Migration Aid will do. After taking the plunge, you now have a file of fresh FORTRAN 77/V source code just itching to be compiled. Will it work? If it is a stand-alone program that does not call other programs or system intrinsics, then it will work (probably). If your program is a routine that interfaces with other programs on your HP3000, then there is an excellent chance that it will not PREP. The default size of integers in FORTRAN 77/V is two words. The default size of integers on the HP3000 is one word. This is where the majority of problems arise.

Another problem is the new logical type. You can not use logical arrays to store non logical data such as IMAGE parameters.

The other road block that is thrown in your path when you

are converting is FORTRAN 77/V's implementation of character strings. A character string has two parameters associated with it. The first is the byte address of the string and the second is a one word integer whose value is the length of the string. When you pass a string to a called procedure, FORTRAN 77/V will pass these two parameters. FORTRAN 66/V passes one parameter per string.

II. Integer Data: One Word or Two?

FORTRAN 77/V's default integer size is two words. Repeat the last sentence until it becomes forever burned in your mind. Understanding this will help you in a smooth conversion. HP has provided you with some control on how programs are compiled that will allow you to specify the size of integers. The fool proof ones are the $SHORT and $LONG compiler directives. If your FORTRAN 66/V program uses INTEGER declarations, as opposed to INTEGER*2, then the $SHORT will force the creation of one word integers in FORTRAN 77/V.

Another way to specify one word integers in the above example, is to globally change all occurrences of "INTEGER " to "INTEGER*2". The result is the same as FORTRAN 66/V. Integer constants are another problem. If you specify $SHORT in the code, then all your constants will be one word, except when they are two words. An example (exclamation points are comments in FORTRAN 77/V):

```
$SHORT
       program test
       integer short_integer    !one word integer
       integer*4 long_integer   !two word integer

       short_integer=32000      !the constant is a
                                !short integer

       i=32000                  !the constant is short as
                                !well as the undefined
                                !variable

       long_integer=32000       !even though $SHORT is
                                !declared, the constant takes
                                !on the size of the left hand
                                !side of the equation

       stop
       end
```

The rule of thumb I use for constants is: The constant will take on the characteristics of the left hand side of the equation. This rule also works where there is no apparent

left hand side. An example:

```
        program test
        integer*2 parameter,i

        i=32000                          !short constant

        call something(parameter,i+2) !the second parameter
                                      !is a DOUBLE integer
        stop
        end
```

The left hand side of the second parameter in the procedure call is defined as the parameter in the called procedure. FORTRAN has no knowledge of the called procedure in this example, so the compiler generates a double integer because the default integer size in this code is two words (no $SHORT). The constant in the parameter is a double and it forces the expression to evaluate as a double. If the called procedure expects a short integer then there is a problem. There are two ways to solve the problem. The first way is to force the constant to be a short integer and the second way is to use the $SHORT declarative. Examples are:

```
        call something(parameter,i+2i) !the 'i' following the
                                       !constant forces it to
                                       !be a single integer
                                       !2j will be a double

$SHORT
   .    .
   .    .
      call something(parameter,i+2) !$SHORT forced the
                                    !expression to evaluate
                                    !as a single integer
```

Let me show you an example of an integer overflow:

```
$SHORT
        program test
        integer*4 i
        integer*2 a,b,c

        a=32000
        b=32000
        c=2

        i=(32000i+32000i)*2i

        i=(a+b)*c
```

```
stop
end
```

The first expression will evaluate just fine. The second expression, even though identical to the first, will fail with an integer overflow. In the first expression, the constants are defined as single (remember the "i"'s), yet the compiler makes them double because of the left hand side of the expression. The second expression fails because the intermediate values on the stack are single integers. 32000+32000 will overflow a single integer. Inconsistencies in the compiler like these make programming in FORTRAN 77/V a much more exciting challenge than programming in FORTRAN 66/V.

III. Logical Data: Illogical

Logical data types can be described as 'weird'. FORTRAN 77/V only uses the low order bit of the high order word. When the bit is set, the value is true. This is totally incompatible with anything else on the HP3000. If your application passes logical flags around and you must convert only parts of your application to FORTRAN 77/V, then you have a lot of work to do. One way to solve the logical problem is to pass the flags as integers, testing them for zero or minus one. FORTRAN 66/V logicals and SPL logicals are not compatible to FORTRAN 77/V. The default size of the logical data type is two words.

IV. Character Data: Two Parameters for the Price of One

There is a useful FORTRAN 77/V intrinsic function that returns the length of a string. A further enhancement is dynamic strings in subroutines. FORTRAN 77/V will allow you to write routines that manipulate strings of unknown length. An example:

```
program test
character string*121

call string_sub(string)
stop
end

subroutine string_sub(string)

character string*(*)       !unknown length
integer string_length

string_length=len(string) !len is a FORTRAN 77/V
                          !function that returns the
                          !length of a string
```

```
      print*,'Length of string is=',string_length

      return
      end
```

There is a drawback to this flexibility. As stated before, two parameters are passed with a string. The first parameter is the byte address of the string and the second parameter is a one word integer by value whose value is the length of the string. This is the only case where FORTRAN 77/V will pass a parameter by value on its own (you can force it to pass parameters by value with the alias directive). Calling a non FORTRAN 77/V routine with strings, or being called by a non FORTRAN 77/V procedure with strings will fail. I will describe HP's work-arounds in a later section.

V. Calling FORTRAN 66/V: Mixing the Old and the New

FORTRAN 66/V's default integer size is one word. FORTRAN 66/V does not have the LOGICAL*4 data type. FORTRAN 66/V expects one parameter per character string. The .true. value of a logical parameter is not compatible between the two fortrans.

If you are careful with integer constants then you will have no problem calling either fortrans. LOGICAL*4 can be passed to FORTRAN 66/V into an INTEGER*4 parameter if you relax the parameter checking.

You can pass character strings to FORTRAN 66/V in two ways. The first way is to use the compiler directive $FTN66_3000 CHARS ON. This directive tells the compiler to pass only the address of the string to any routine it calls. The other way is to define the called routine with an $ALIAS compiler directive. In the ALIAS directive you specify the language of the called routine as FORTRAN 66 then the compiler does the rest.

Calling a FORTRAN 77/V routine from a FORTRAN 66/V routine with a character parameter has the same problem only reversed. You have two ways to solve this problem as well. The first way is for you to provide the extra length parameter to the FORTRAN 77/V routine. An example:

```
C FORTRAN 66/V
      program test
      character string*20
C
C length parameter is by value
C
```

```
      call string77(string,/20/)
      stop
        end

C FORTRAN 77/V
      subroutine string77(string)
      character string*20          !or string*(*)

      string='this is FORTRAN 77/V'
      return
      end
```

The second way is to surround the subroutine statement with the $FTN3000_66 CHARS ON and OFF directives. The above subroutine rewritten in this way is as follows:

```
$FTN3000_66 CHARS ON
       subroutine string77(string)
$FTN3000_66 CHARS OFF
       character string*20  !you can not use the *(*)
                            !construct here

       string='this is FORTRAN 77/V'
       return
       end
```

You may not use the unknown string length definition in the above example. The length of the string is not expected by the subroutine, so the length parameter is not needed in the calling routine. This is the why you can not use the *(*) construct for the above string. This method is the easiest way to integrate 77 code with 66 code. You do not have to recompile the 66 code that calls a routine that you have replaced with FORTRAN 77/V if character strings are passed to the replaced code.

As you can see, calling FORTRAN 77/V and FORTRAN 66/V from each other is not that much of a problem except for logical data types.

VI. Calling System Intrinsics: Is FORTRAN 77/V on Speaking Terms WIth MPE?

MPE intrinsics require parameters that FORTRAN 77/V can not provide; such as value parameters and byte address of logical arrays. HP has provided two mechanisms that will allow you to call system intrinsics and other SPL routines. The nicest one is the SYSTEM INTRINSIC declarative. Ninety-nine percent of your problems in calling system intrinsics are solved with this declarative. The compiler will use the SPLINTR file to set up the call to the intrinsic. The parameters will be passed in the proper manner; by value or

reference, by word or byte address. Always declare the system intrinsics that you use in your programs. Failure to do so will crash programs.

The other mechanism is the $ALIAS directive. The ALIAS directive is best used to call SPL routines, although it can be used to call system intrinsics. The directive allows you to define the passing method for each parameter. Based on your definition of the procedure from the ALIAS directive, the compiler will create the code to call the procedure.

You can not use logical arrays to store parameters for system intrinsics in FORTRAN 77/V. Moving a logical array to another logical array will not move all the data from the source array. Only the low order byte of the high order word is moved. An example will explain this better:

```
program test
logical*2 l_array1(10)  !the default size of logical
                        !is two words, so I used '*2'

character string1*20
equivalence (l_array1,string1)

logical*2 l_array2(10)
character string2*20
equivalence (l_array2,string2)

string1='abcdefghijklmnopqrst'
string2='                    '

do i=1,10
   l_array2(i)=larray1(i)
end do

print*,string2
stop
end
```

The output will be 'a c e g i k m o q s'. FORTRAN 77/V will allow you to use integer arrays in place of logical arrays to store your parameters. The system intrinsic declarative will take care of the necessary conversions for you. Do not put parameters in logical arrays, period. The Migration Aid will not place parameters in integer arrays, you must.

The FORTRAN 77/V compiler seems to relax parameter checking at PREP time for system intrinsics. Therefore, you do not have to worry about having a mixture of FORTRAN 66/V and FORTRAN 77/V code that calls intrinsics with different parameter types such as logical arrays in 66 and integer arrays in 77 for the same intrinsic call.

There was one interesting bug that happened to me in a call to FREAD. The length field that I used to receive the value of the FREAD function was defined as a double integer. Immediately after the FREAD call, I tested the condition word. Every time the condition word was CCG, indicating an end-of-file state. I knew this error condition was wrong because the file's record pointer was updated to the next logical record in the file and the pointer was not greater than or equal to the actual end-of-file. The cause of the problem was that the FORTRAN 77/V compiler generated some code that converted the short integer returned by FREAD to the double integer. The code affected the condition word before I could test the result of the FREAD.

Conclusion

Should you convert to FORTRAN 77/V? Yes. Even with the problems associated with a 32 bit language on a 16 bit machine, FORTRAN 77/V is well worth the trouble that it causes. With the new control statements - DO WHILE and IF...ELSE blocks - plus powerful string handling constructs, you can write better programs easily. The FORTRAN 77/V Language allows you to write structured programs without any GO TO statements. GO TO's are difficult to read in programs and they allow for careless programming. The only statement labels that are necessary in FORTRAN 77/V are FORMAT labels. The programmers that have to maintain good FORTRAN 77/V code in the future will appreciate the absence of GO TO's. The string handling helps the language to be a more general purpose language. If only they added BCD. If you are interested in having your code run on the 900 series computers, then you need to convert to 77 because there is no native-mode 66 compiler.