Porting Fortran Between Cray and the IBM RS/6000

By Emad ElHamahmy and Luc Chamberland

XL Fortran Version 3 includes several extensions that enable developers to port their CRAY Fortran code with minimal recoding. Using the compiler options and routines described in this article will make your move from CRAY1 to the RISC System/6000 as easy as possible.

Most of the language extensions discussed in this article can be coded using Fortran 90, which is supported in XL Fortran Version 3. When you are writing new code, adhering to Fortran 90 is preferable because it ensures portability across any platform that supports Fortran 90. Of course, to port code that has already been written, using the extensions would likely require fewer changes than recoding your applications.

You can use the following XL Fortran compiler options and features to meet your porting needs.

Luc Chamberland

Emal ElHamahmy

The -qintsize Option

The `-qintsize` compiler option sets the default size of integer and logical data entities whose sizes are not explicitly declared. This option allows you to port code easily from 16-bit and 64-bit machines to 32-bit machines.

For example, the following declaration explicitly specifies that `x` has a size of 8 bytes:

```
integer(8) x
```

But the next declaration specifies that the default integer size be provided for `x`. For XL Fortran, the default size of integer/logical entities is four bytes, while for CRAY it is eight bytes.

```
integer x
```

If `-qintsize=8` is specified at compile time, `x` is declared as a 64-bit entity. This ensures that very large numbers (larger than the maximum `integer(4)` value) are not inadvertently truncated because of a different default size.

The `-qintsize` option affects the following:

- Integer/logical literals that do not specify `kind` type parameters
- Default integer/logical data objects and functions
- Intrinsic function results of type default integer/logical
- Typeless constants in integer contexts

Fortran 90 alternative to the `-qintsize` option: The Fortran 90 `KIND` intrinsic function returns the `kind` type parameter of an argument. It specifies the representation method for the argument. You can then use the `kind` type parameter when declaring entities. For example,

```
integer (kind=kind(0)) x
```

indicates that the size of `x` is the default integer size for the compiler. Compiled by a CRAY compiler, `x` is 8 bytes long; compiled by XL Fortran, `x` is 4 bytes long.

To ensure that entities have similar value ranges without specifying platform-specific information during type declaration (such as a `kind` type parameter), use the Fortran 90 `SELECTED_INT_KIND` intrinsic function. This function returns a `kind` type parameter value of an

[1]All references to CRAY in this article refer to the Y-MP™ hardware line and the CF77™ and CF90™ Fortran compilers.

integer data type that represents all integer values n in the range $-10^R < n < 10^R$. For example:

```
integer (kind=selected_int_kind(18)) x
   ! XL Fortran kind type parameter: 8
   ! CRAY kind type parameter: 8
```

The -qrealsize Option

Similar to -qintsize, the -qrealsize compiler option sets the default size of real and complex entities whose sizes are not explicitly declared. The default real size in XL Fortran is 4 bytes, while the default on the CRAY products is 8 bytes.

Keep in mind that the precision of entities of type DOUBLE PRECISION is twice that of the default real size. Thus, if you specify -qrealsize=8, DOUBLE PRECISION entities have a default size of 16 bytes. Similarly, the size of complex and double complex entities is affected because they are formed from parts of type real.

Fortran 90 alternative to the -qrealsize option: As for integer entities, the KIND intrinsic function is an alternative way to ensure portability for real entities:

```
real (kind=kind(0.0)) r
```

KIND(0.0) always returns the kind type parameter of the processor's default real size.

Similarly, the SELECTED_REAL_KIND intrinsic function returns the kind type parameter value of a real data type with a minimal decimal precision and exponent range.

The -qautodbl Option

Use the -qautodbl compiler option to convert single-precision entities to double-precision entities, and double-precision entities to extended-precision entities. Figure 1 shows suboptions.

To help maintain proper storage relationships when code is ported, some suboptions will pad as well as promote objects, shown in Figure 2.

NOTE: The -qrealsize is disregarded if you compile with both the -qautodbl and -qrealsize options.

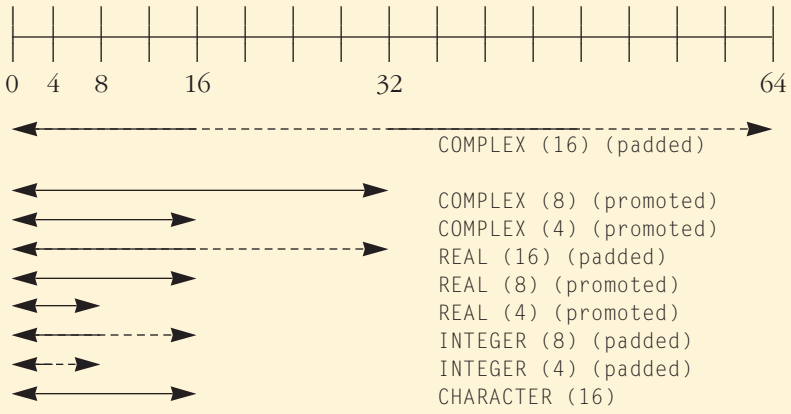
Figure 3 illustrates promotion and padding using -qautodbl=dblpad. Figure 4 shows how data types are mapped in memory when -qautodbl=dblpad is used.

Subscription	Function
none	Does not promote or pad any objects that share storage. This is the default for -qautodbl
dbl4	Promotes 4-byte floating-point objects, including objects composed of these objects, to 8-byte objects. For example, a complex(4) object is promoted to complex(8).
dbl8	Promotes 8-byte floating-point objects (including objects composed of these objects) to 16-byte objects.
dbl	Performs the same promotions as both dbl4 and dbl8.

Figure 1. Suboptions to promote objects

Subscription	Function
dblpad4	Performs dbl4 promotion, and also pads objects of other types (except character) if they possibly share storage with promoted objects.
dblpad8	Performs dbl8 promotion, and also pads objects of other types (except character) if they possibly share storage with promoted objects.
dblpad	Performs the promotions and padding done by both dblpad4 and dblpad8.

Figure 2. Suboptions to pad objects



The dashed lines represent the padding.

Figure 3. Storage relationships with -qautodbl=dblpad

```
@process autodbl(dblpad)
      complex(4) x8      /(1.123456789e0,2.123456789e0)/
      real(16) r16(2)    /1.123q0,2.123q0/
      integer(8) i8(2)   /1000,2000/
      character*5 c(2)  /"abcde","12345"/
      common /named/ x8,r16,i8,c
      end
@process autodbl(none)
      subroutine s()
         complex(8) x8
         real(16) r16(4)
         integer(8) i8(4)
         character*5 c(2)
         common /named/ x8,r16,i8,c
           x8    = (1.123456789d0,2.123456789d0)   ! promotion occurred
          r16(1) = 1.123q0                         ! padding occurred
          r16(3) = 2.123q0                         ! padding occurred
           i8(1) = 1000                            ! padding occurred
           i8(3) = 2000                            ! padding occurred
            c(1) = "abcde"                         ! no padding occurred
            c(2) = "12345"                         ! no padding occurred
      end subroutine s
```

Figure 4. Mapping of data types with -qautodbl=dblpad

Integer Pointers

Integer pointers are the XL Fortran version of CRAY pointers. XL Fortran integer pointers reference the address of a variable, which enables a program to step through part of storage or overlay parts of storage. The following example represents an XL Fortran integer pointer:

```
pointer (p,x)
```

The `p` is the integer pointer, which references the address of `x`, a variable or array declarator (generally called the *pointee*)

An integer pointer is a scalar variable of type `integer(4)` that cannot have a type explicitly assigned to it. The integer pointer can be used in any expression or statement in which an `integer(4)` can be used.

The XL Fortran compiler does not allocate storage for the pointee. Storage is associated with the pointee at execution time by assigning the address of a block of storage to the pointer.

The following subsections describe the differences you must consider when porting Fortran code between CRAY and the RISC System/6000.

Pointer Size

XL Fortran integer pointers are 4 bytes in size, whereas CRAY pointers are 8 bytes. If an absolute address is assigned to a pointer, ensure that the address can fit in four bytes of storage and that it is valid on the RS/6000.

Incrementing/Decrementing Pointers

Adding or subtracting an integer *n* to an integer pointer increments (or decrements) the value of the pointer by *n* bytes. Adding or subtracting an integer *n* to a CRAY pointer increments (or decrements) the value of the pointer by *n* words. To maintain compatibility when porting, multiply the value of *n* (CRAY words) by the size of the RS/6000 machine word size (8 bytes). Figure 5 shows the differences between a pointer incrementing using XL Fortran and CRAY Fortran.

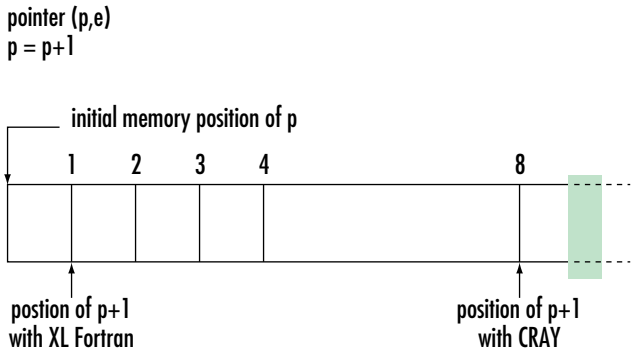


Figure 5. Pointer arithmetic: XL Fortran and CRAY Fortran

To port the code segment in Figure 5 to XL Fortran, modify the assignment statement as follows:

```
integer, parameter :: word_size = 8
p = p + (word_size*1)
```

LOC Intrinsic

The LOC intrinsic function returns the address of a pointee. This function is useful for finding the address of a variable and assigning it to the integer pointer of another pointee (overlaying the storage of the two variables).

Figures 6 and 7 show the modifications required when porting code from CRAY to XL Fortran.

Fortran 90 alternative to Integer Pointers: To dynamically dimension arrays in Fortran 90, use deferred-shape array specifications, and then explicitly allocate memory for the arrays as shown below:

```
integer, dimension(:), allocatable ::
   array
I = 5
allocate(array(5))
array = (/1,2,3,4,5/)
deallocate(array)
end
```

Service and Utility Procedures

XL Fortran supports several CRAY procedures as industry extensions. Some of these are described below. These procedures (except for CLOCK_) have the same names as the corresponding CRAY routines.

clock_: The clock_ function returns the time in hh:mm:ss format. (The XL Fortran routine is CLOCK_, not clock, because the libc library of AIX already contains a clock routine).

date: The date function returns the system date in mm/dd/yy format.

irtc: The irtc function returns an integer(8) value of the number of nanoseconds elapsed since the initial value of the machine's real-time clock, as follows:

```
integer(8) :: a,b,elapsed,irtc,x=3
a = irtc()
do m = 1,20000
  x = x**2
end do
b = irtc()
elapsed = b-a      ! elapsed = 33052928
```

rtc: The rtc function returns a real(8) value of the number of seconds elapsed since the initial value of the machine's real-time clock. Figure 8 shows an example.

```
integer big_array(100)
integer element, dim_size
integer array1(5), array2(5), array3(10)
pointer (p1,array1)
pointer (p2,array2)
pointer (p3,array3)
! Assign storage to array1
p1 = loc(big_array)
! Initialize array1
array1 = (/1,3,5,7,9/)
! Assign storage to array2
p2 = p1 + 5
! Initialize array2
array2 = (/2,4,6,8,10/)
! array3 holds data of array1 and array2
p3 = loc(big_array)
if ((array3(3) .ne. 5) .or. (array3(8) .ne. 6)) stop 1
! Increment the area of storage where array1 begins by 2 words
p1 = p1 + 2
element = array1(5)
if (element .ne. 4) stop 2
end
```

Figure 6. Original CRAY code

timef: The timef function returns the elapsed time in milliseconds since the first instance timef was called. The first instance timef is called and the value 0.0d0 is returned. Figure 9 shows an example.

jdate: The jdate function returns the system Julian date in yyddd format.

Fortran 90 alternatives to CRAY Fortran functions: You can use the DATE_AND_TIME intrinsic function to access date and time information, and the SYSTEM_CLOCK intrinsic function to inquire on your system clock.

CVMGx Procedures

XL Fortran supports conditional vector merge functions similar to the CRAY. These functions take three arguments. The values returned depend on the result of the test that is done on the third argument. The first argument is returned if the test done on the third argument is true. The second argument is returned if the test done on the third argument is false. When the arguments are arrays, testing is done on an element-by-element basis.

The conditional vector merge functions are as follows:

CVMGP	Test for positive values or zero
CVMGM	Test for negative values
CVMGZ	Test for zero values
CVMGN	Test for nonzero values
CVMGT	Test for logical true values

```
      @process intsize(8)               !<—- Added code
      integer big_array(100)
      integer element, dim_size
      integer array1(5), array2(5), array3(10)
      pointer (p1,array1)
      pointer (p2,array2)
      pointer (p3,array3)
      integer word                      !<—- Added code
      parameter (word = 8)              !<—- Added code
      ! Assign storage to array1
      p1 = loc(big_array)
      ! Initialize array1
      array1 = (/1,3,5,7,9/)
      ! Assign storage to array2
      p2 = p1 + 5*word                  !<—- Modified code
      ! Initialize array2
      array2 = (/2,4,6,8,10/)
      ! array3 holds data of array1 and array2
      p3 = loc(big_array)
      if ((array3(3) .ne. 5) .or. (array3(8) .ne. 6)) stop 1
      ! Increment the area of storage where array1 begins by 2 words
      p1 = p1 + 2*word                  !<—- Modified code
      element = array1(5)
      if (element .ne. 4) stop 2
      end
```

Figure 7. CRAY code (Figure 6) ported to the RS/6000

```
real(8) :: a,b,elapsed,rtc,x=2.0
a = rtc()
do m = 1,20000
  x = x**2
end do
b = rtc()
elapsed = b-a   ! loop required 0.8463557684564229 78e-02
```

Figure 8. Using the rtc function

```
real(8) elapsed, timef
elapsed = timef()     ! elapsed = 0.0d0
do m = 1,20000
  a = a**2
end do
elapsed = timef()     ! elapsed = 15.2616500997857362
```

Figure 9. Using the timef function

```
! CRAY
r = cvmgp(1.0,2.0,3.0)    ! r = 32-bit representation of 1.0
i = cvmgp(1.0,2.0,3.0)    ! i = 32-bit representation of 1.0
                          ! i = 1065353216
! XL Fortran
r = cvmgp(1.0,2.0,3.0)    ! r = 32-bit representation of 1.0
i = cvmgp(1.0,2.0,3.0)    ! i = int(1.0)
                          ! i = 1
```

Figure 10. CVMGx procedure

Be cautious when you are porting code to XL Fortran. Although the CVMGx routines accept integer and real arguments, these arguments and the function return type are interpreted as Boolean data types on a CRAY. Since XL Fortran does not have a Boolean data type, the argument and function return types are treated as integer, logical, or real intrinsic data types (as shown in Figure 10). The only place the user would see a difference between the XL Fortran and CRAY behavior is if the function is referenced in an assignment or expression where type conversion is expected to occur.

Fortran 90 alternative to CRAY CVMGx intrinsic functions: Use the Fortran 90 `MERGE` intrinsic function instead of the CVMGx intrinsic functions. Figure 11 shows the same function implemented using the XL Fortran extensions and Fortran 90 intrinsic functions.

Summary

XL Fortran Version 3 supports a variety of features to facilitate your porting needs, including several industry extensions found in CRAY products. Also, because XL Fortran fully supports Fortran 90, you can modify your code for portability to any platform that supports this latest Fortran standard.

For more information about porting, contact AIX Support Family at callaix@vnet.ibm.com or 1-800-CALL-AIX (U.S. and Canada only).

Emad ElHamahmy, IBM Canada Ltd., 844 Don Mills Road, North York, Ontario, Canada, M3C 1V7. Internet: emad@vnet.ibm.com. Mr. ElHamahmy is a development analyst currently working on the XL Fortran compiler. He has co-authored a workshop on Fortran 90 using the XL Fortran compiler. Mr. ElHamahmy received a BSc in Computer Science and Mathematics from the University of Toronto.

Luc Chamberland, IBM Canada Ltd., 844 Don Mills Road, North York, Ontario, Canada, M3C 1V7. Mr. Chamberland is an information developer on the XL Fortran team, focusing primarily on Fortran language documentation. He recently co-authored and taught a workshop on Fortran 90. Mr. Chamberland received both his BA in English and MA in Religious Studies from the University of Toronto.

Code Using CVMGx Routines

```
integer i
integer i_a(3)
integer array_t(3), array_f(3), mask(3)
i = cvmgp(1.0,2.0,3.0)        ! i = 1
array_t(1) = 1
array_t(2) = 2
array_t(3) = 3
array_f(1) = 4
array_f(2) = 5
array_f(3) = 6
mask(1)=7
mask(2)=0
mask(3)=8
i_a = cvmgz(array_t,array_f,mask)   ! i_a(1)=4, i_a(2)=2,
                                    ! i_a(3)=6
end
```

Fortran 90 Alternative

```
integer :: i
integer, dimension(3) :: i_a
integer, dimension(3) :: array_t, array_f, mask
i = merge(1.0,2.0,3.0 >= 0.0)           ! i = 1
array_t = (/1,2,3/)
array_f = (/4,5,6/)
mask = (/7,0,8/)
i_a = merge(array_t,array_f,mask == 0)  ! i_a = (/4,2,6/)
end
```

Figure 11. CVMGx functionality in Fortran 90