Fortran 2003 and Beyond

Bill Long, *Cray Inc*

ABSTRACT: *The recently adopted Fortran standard includes many features that were not present in the old Fortran 95 standard. Selected features are discussed as well as the status of their implementation in the Cray compiler. The proposed feature list for the next standard, Fortran 2008, is also discussed.*

1. Introduction

Fortran was designed almost 50 years ago to be the language of choice for scientific programming. It continues to evolve through a series of revisions that incorporate more modern programming paradigms while retaining the focus on scientific computing and computational efficiency.

The current Fortran standard, commonly referred to as Fortran 2003, was published in November, 2004. This standard deletes and replaces the old standard, commonly know as Fortran 95 or f95. The preliminary list of features for the next standard, Fortran 2008, has been specified by WG5, the ISO Fortran body.

The following sections describe some of the key features of Fortran 2003 and Cray's plans for implementation. The key features proposed for Fortran 2008 are also discussed.

2. Fortran 2003 implementation status

The current cftn compiler release is version 5.4.0.3. The 5.5 release is scheduled for the end of 2005, with additional releases in 2006. The implementation status of the features described in section 3 below falls into three categories: already implemented in 5.4.0.3, planned for 5.5 (end of 2005) or planned for 2006.

Features currently implemented

Basic syntax enhancements
PROTECTED attribute
VOLATILE attribute
INTENT attribute for pointers
mixed PUBLIC and PRIVATE component attributes
allocatable components
allocatable dummy arguments
allocatable function results
allocatable array assignment
ASSOCIATE construct
intrinsic modules
ISO_FORTRAN_ENV module
ISO_C_BINDING module
C interoperability
IMPORT statement
PROCEDURE statement
procedure declaration and abstract interfaces
procedure pointers
pointer assignment lower bounds
pointer rank remapping
FLUSH statement
IOMSG keyword in I/O statements
MAX and MIN with character arguments
NEW_LINE intrinsic
GET_COMMAND intrinsic
COMMAND_ARGUMENT_COUNT intrinsic
GET_COMMAND_ARGUMENT intrinsic
GET_ENVIRONMENT_VARIABLE intrinsic
IS_IOSTAT_END intrinsic
IS_IOSTAT_EOR intrinsic

Features planned for the end of 2005

parameterized derived types
keywords in derived type constructors
type specifiers in array constructors
allocatable character scalars
allocatable character assignment
IEEE_FEATURES module
IEEE_ARITHMETIC module
IEEE_EXCEPTIONS module
Cray IEEE extensions
Asynchronous I/O and WAIT statement
Stream I/O

DECIMAL mode in I/O statements
Rounding mode in I/O statements
Keywords in READ and WRITE statements
Result KIND specifiers in intrinsics
Array reallocation - MOVE_ALLOC intrinsic

Features planned for 2006

Derived type extension
Type-bound procedures
Finalizers
polymorphic objects
SELECT TYPE construct
enhanced initialization expressions
user derived type I/O control
ISO character set support
text encoding selection in I/O

3. Fortran 2003 feature descriptions

Many features in Fortran 2003 were not part of f95 or previous standards. The following sections describe the principal new features.

Basic syntax enhancements

Statement syntax in Fortran 2003 relaxes some of the restrictions of f95. Names of objects (variables, procedures, common blocks, types, etc.) may contain up to 63 characters, up from 31. The maximum number of continuation lines for a single statement is 255, up from 39. The Cray 5.4 compiler allows an unlimited number of continuation lines. These changes were made based on user comments that the old limits were too restrictive, especially in the case of source code created by other programs.

Fortran 2003 allows the cleaner [ ] notation as an alternative to (/ /) for an array constructor.

Named constants as parts of a complex constant are allowed. A simple example is:

```
real,parameter :: zero = 0.0, one = 1.0
complex :: eye
eye = (zero, one)
```

Fortran has always specified a minimal required character set. Traditionally this consisted of all the characters that are required by the basic language syntax. The list of required characters is increased to include \ [ ] { } ` ~ ^ | # @, even though not all of these have a syntax use in the language.

PROTECTED attribute

Module data in f95 had a visibility outside the module based on their declaration as either public or private. These attributes apply the names of the objects, not their values. In many cases it is useful to have the name visible outside the module (public), but prevent procedures outside the module from changing the value of the object. If the object never changes value, it can be declared as a parameter. However, this option is not useful for variables such as overall data sizes that might be initialized at run time. The new PROTECTED attribute applies to the object's value. Module objects with the protected attribute may be defined by procedures in the module, but cannot be defined by statements outside the module. If they are public objects, they may be referenced outside the module. Example:

```
integer,protected :: ncpus
```

VOLATILE attribute

A variable with the volatile attribute may have its value changed by mechanisms not visible to the local program unit. Typically these are variables that may be defined by external means like an asynchronous operating system action or by other threads of a parallel program. Example:

```
integer,volatile :: flag
```

Intent for pointer dummy arguments

Dummy arguments with the pointer attribute could not have an intent attribute in f95. Fortran 2003 has removed this restriction. The intent specification for a pointer argument applies to the association status of the pointer, and not to the definition status of the target of the pointer. A pointer with the intent(in) attribute cannot be pointer associated with a (potentially) new target within the procedure. A pointer with the intent(out) attribute enters the procedure with a disassociated status. Example:

```
subroutine sub(p,dat)
   integer,pointer,intent(in) :: p(:)
   integer,target             :: dat(10)

   p = 1           ! OK
   allocate(p(20)) ! Illegal
```

```
   p => dat        ! Illegal

end subroutine sub
```

In the example above, both the allocate statement and the pointer assignment of p to dat are illegal because they change the target of the pointer p, which is declared with intent(in).

Mixed public and private component attributes

Derived types defined in a module can be either public or private. New rules allow individual components of the type to be private or public. Also, an object of a private type may be declared to be public. Private names of either the type or some or all of the components are not available outside the module. This can be an intentional programming strategy of hiding the details of a structure from the module user. Examples:

```
type,private :: foo
   integer,public  :: bar1
   integer,private :: bar2
end type foo

type(foo),public :: x
```

Allocatable components

One of the least satisfactory aspects of f95 is the requirement that dynamic sized components of a derived type be declared as a pointer. Because a compiler cannot determine all the possible aliases for pointer targets, optimization of expressions involving pointers is restricted. The new standard allows allocatable components, which do not have this performance problem. Example:

```
type :: foo
   real,allocatable :: bar(:)
end type foo
```

Allocatable dummy arguments

The size needed for an actual argument associated with a dummy argument may be computed inside the called procedure. With f95, such an argument had to be a pointer, resulting in the disadvantages of pointers being forced on the programmer. Fortran 2003 allows allocatable dummy arguments, resolving this shortcoming of f95. The storage for an allocatable dummy argument is not automatically deallocated at the end of the procedure. Example:

```
integer,allocatable :: db(:)
call sub(db,nwords)

subroutine sub(db,n)
   integer,allocatable :: db
   integer             :: n

   read *, n
   allocate(db(n))
   read *, db
end subroutine sub
```

Allocatable function results

Function results can be considered equivalent to an additional argument to a subroutine. A natural extension of the allocatable dummy argument feature is the allocatable function result. This is included in Fortran 2003. Example:

```
function foo(x) result (foo_r)
   real,dimension(:),intent(in)  :: x
   real,dimension(:),allocatable ::foo_r
…
end function foo
```

Allocatable array assignment

Fortran 2003 requires the specification of the meaning of default assignment for structures with allocatable components. Array assignment requires that the left hand side variable be allocated and have the same size as the right hand side expression. To avoid having the user explicitly allocate an allocatable component before an assignment statement, the default assignment rule specifies the automatic allocation of the left hand side if necessary. If the current left had side allocatable component is allocated with the correct shape, then an array copy is done. If the left hand side array is not allocated, it is allocated with the correct shape and then the array copy is done. If the current left hand side is allocate with the wrong shape, it is deallocated and then reallocated with the correct shape and the array copy is done.

The above assignment algorithm for allocatable components is extended to all allocatable objects, not just components. It is now allowed to assign a value to an unallocated array. If that is done, the array is automatically allocated with the correct shape before the data is moved. Similarly, if the left hand side variable is allocated with the wrong shape (not conforming to the f95 standard) then it is reallocated with the correct shape rather than causing an error. This new rule may result in different behavior for programs that were illegal in f95, but seemed to work anyway. This feature is incorporated in the Cray compiler, but is not turned on be default. To enable this behavior, specify the –ew option on the compile command. Examples:

```
type foo
   integer,allocatable :: bar(:)
end type foo

type(foo) :: f1,f2
allocate(f1%bar(100))
f1%bar(:) = 1

f2 = f1
```

In this example, f2%bar is automatically allocated with a size of 100, and the values of f1%bar are copied to f2%bar.

```
real,allocatable :: a(:),b(:),c(:)

allocate(a(10),b(20))
a = 1.10
b = 1.20
c = a        ! Line 1
c = b        ! Line 2
c(:) = a(:)  ! Line 3 – illegal
```

In the statement with comment Line 1, the array c is allocated with a size of 10. In Line 2, c is reallocated with a size of 20. The statement in Line 3 is illegal. The expression c(:) is an array section and not an allocatable array. The reallocation rules apply only to allocatable objects.

ASSOCIATE construct

The ASSOCIATE construct provides a shorthand notation for expressions and derived type objects that appear in statements. Using an associate name can greatly simplify the appearance of otherwise complicated statements. An associate name is specified by an associate statement, and can also be specified in a select type statement. The name is identified with the associate expression at the entry to associate construct or select type construct, and is not affected by later redefinitions of a part of the expression. The associate name assumes the type and type parameters of the associate expression and has the scope of the construct. It is unrelated to any object outside the construct that has the same name. Example:

```
! Old code

do i=1,genome(ng)%chr(nc)%dblen
   genome(ng)%chr(nc)%db(i) = &
   iand(genome(ng)%chr(nc)%db(i),255)
end do

! New code

associate (x=>genome(ng)%chr(nc))
   do i=1,x%dblen
      x%db(i) = iand(x%db(i),255)
   end do
end associate
```

Intrinsic modules

Intrinsic modules are supplied as part of the language and are intended to provide information to the programmer that may be implementation dependent. Fortran 2003 specifies five intrinsic modules. The iso_c_binding module contains definitions of constants and procedure interfaces for the C interoperability features. The iso_fortran_env module contains definitions of constants that characterize memory and I/O sizes. The remaining three intrinsics modules, ieee_features, ieee_exceptions, and ieee_arithmetic, contain definitions of constants and procedure interfaces to support the IEEE floating point arithmetic standard. Details of these modules are in the following sections. Examples for using each of the new modules:

```
use,intrinsic :: iso_c_binding
use,intrinsic :: iso_fortran_env
use,intrinsic :: ieee_features
use,intrinsic :: ieee_exceptions
use,intrinsic :: ieee_arithmetic
```

iso_fortran_env module

The iso_fortran_env intrinsic module contains named constants for characteristics of the hardware and I/O systems use by the program. The standard specifies the concept of a numeric storage unit (essentially the memory associated with a default integer), a character storage unit (the memory associated with a length one character), and a file storage unit (the units used for the RECL values in I/O statements). The sizes of these units, measured in bits, are specified in the iso_fortran_env module as numeric_storage_size, character_storage_size, and file_storage_size. On Cray systems, the numeric_storage_size is either 32 or 64 depending on compiler options, and the character_storage_size and file_storage_size are both 8. The module also specifies the Fortran unit numbers corresponding to the * units in I/O statements. These are input_unit, output_unit, and error_unit. Finally, the module specifies the values returned for end of file and end of record conditions in iostat variables. These are iostat_end and iostat_eor. Note that on Cray systems there are multiple end of file values, depending of the type of end of file. The iostat_end represents the most common end of file return value. In a

later section intrinsic functions are described that provide a better alternative to using iostat_end.

iso_c_binding module

The iso_c_binding intrinsic module contains definitions for constants and types that provide a way to portably link with programs written using the system's C compiler. KIND values are defined that link Fortran intrinsic data types to corresponding C data types. For example, C_INT is defined to be the kind value for which an integer(c_int) declaration specifies a data object that has the same size as an int object in C. Constants are defined for all the C data types that have analogs in Fortran. If the Fortran processor does not support a particular combination of type and kind, the corresponding constant in the iso_c_binding module is –1. The module also defines certain standard character constants widely used in C programs, such as C_null_char, and C_new_line. Finally, the module defines new types. C_PTR and C_FUNPRT. These are used to specify variables that can be used as actual arguments corresponding to C data and function pointers.

C interoperability - C global objects

Names of objects in the data part of a module can be linked to C global data using the bind(c) attribute. This allows Fortran and C routines to have access to shared data using standard syntax. The external name of the data defaults to the Fortran name in lower case letters. Optionally, the user can specify a different name with a character constant. Data of any Fortran intrinsic type may be shared. In addition, a derived type may be specified to interoperate with C with certain restrictions. Interoperable derived types must not have the SEQUENCE attribute, allocatable or Fortran pointer components, or derived type components that are not interoperable. Derived types can be specified to replicate the form of a C structure. Examples illustrating the new syntax:

```
!  First example  ----------

module global_data
use,intrinsic :: iso_c_binding
   type,bind(c) :: flag_type
     integer(c_long) :: ioerror_num
     integer(c_long) :: fperror_num
   end type flag_type

   type(flag_type),bind(c):: error_flags

end module global_data

! The name of error_flags is specified
!  in C as

typedef  struct{
                 long ioerror_num;
                 long fperror_num;
               } flag_type

flag_type  error_flags;

!  Second example --------

module global_data2
use,intrinsic :: iso_c_binding

integer(c_int),bind(c,name='Fc')::fc

common /block/ r,s
common /tblock/ t
real(c_float) :: r,s,t
bind(c) :: /block/, /tblock/

end module global_data2

! The corresponding C declarations are:

int Fc;
struct {float r,s;} block;
float tblock;
```

The first example illustrates specification of an interoperable derived type and a data object of that type. The value of c_long is obtained from the iso_c_binding module.

The second example shows how to connect common block variables to C global variables. The global symbol is the name of the common block. The names of the entries in the common block are local, and may be different in different Fortran modules. As is the case with most attributes, the bind(c) attribute can be used either as a qualifier in a type declaration or as a separate statement. The separate statement form must be used for common blocks.

C interoperability - Interoperating with C functions.

Interoperation with C functions with standard syntax is a major new feature of Fortran 2003. To correctly link with a C function, as caller or callee, the compiler needs to know the correct interface information. This is specified by extensions to the interface block syntax. The bind(c) attribute identifies an external procedure as conforming to the C calling conventions. The external routine could be written in a language other than C, as long as the interface conforms to the C rules. The constants from the

iso_c_binding module are used in dummy argument declarations. A new attribute, VALUE, is optional for dummy arguments. If a dummy argument with the value attribute is defined within the subroutine, the corresponding actual argument is not changed. The value attribute effectively causes the argument to be passed by copy-in value. The dummy arguments in an interface for a bind(c) procedure must be interoperable with C data types. It is always possible to write a corresponding C prototype for describe the function interface. Example:

```
use,intrinsic :: iso_c_binding
interface
   function foo(ptr,val)       &
            bind(c,name='Foo') &
            result(bar)
      import :: c_int, c_long
      integer(c_int) :: ptr, bar
      integer(c_long),value :: val
   end function foo
end interface
integer(c_int) :: x,n
integer(c_long) :: y
   …
n = foo(x,y)

Corresponding C interface:

   int Foo( int *ptr, long val);
```

C interoperability - Intrinsics

Five new intrinsic functions are provided as part of the iso_c_binding module. These are used to create and test C style pointers that are sometimes needed as actual arguments to C functions.

C_LOC(fortran_data_arg) returns a type(C_PTR) pointer to the data argument.

C_ASSOCIATED(cp1, [cp2]) returns true if the C pointer cp1 is associated, or if the two arguments are associated with the same target. This is analogous to the associated intrinsic function for Fortran pointers.

C_F_POINTER is a subroutine that associates the target of a C data pointer with a Fortran pointer.

C_FUNLOC(fortran_proc_arg) returns a type(C_FUNPTR) pointer to the Fortran procedure argument.

C_F_PROCPOINTER is a subroutine that associates the target of a C function pointer with a Fortran procedure pointer.

IMPORT statement

Interface blocks are their own scoping units and thus to not have direct access to definitions in the surrounding host scoping unit. This has been especially cumbersome when derived type definitions are required in the dummy argument declarations in the interface. Past standards have required that the definitions are either repeated in the interface or accessed by a USE of a module containing the definition. If the interface is in the same module as the type definition the USE option is not available. The new standard provides a solution to this problem with the import statement. The import statement allows importing type information from the surrounding host. Example:

```
type :: foo
   integer :: foo_int
end type foo

interface
   function  bar(x) result(bar_res)
      import foo
      type(foo) :: x
      integer :: bar_res
   end function bar
end interface
```

PROCEDURE statement

The PROCEDURE statement is an extension of the module procedure statement from f90, used to define a generic interface. The specific procedures do not have to be contained in the module, as is the case with the module procedure statement. Interfaces for the procedures do need to be visible. Example:

```
interface sgemm
   procedure sgemm_44, sgemm_48
   procedure sgemm_84, sgemm_88
   procedure cgemm_44, cgemm_48
   procedure cgemm_84, cgemm_88
end interface

interface dgemm
   procedure sgemm_44, sgemm_48
   procedure sgemm_84, sgemm_88
   procedure cgemm_44, cgemm_48
   procedure cgemm_84, cgemm_88
end interface
```

The example illustrates a mechanism for making the BLAS matrix multiply routine completely generic. The numbers at the ends of the specific routine names indicate the kind values for integer and real (or complex) arguments. Interfaces for the generic names cgemm and zgemm would

be written in the same way. Interfaces for all of the specific routines need to be visible.

Procedure declarations and abstract interfaces

The procedure statement can declare names to be of external procedures and identify an interface. An abstract interface specifies the interface information for a hypothetical procedure, and hence the procedure name itself is not made external. Abstract interfaces are used as templates for the interfaces for actual procedures. A procedure statement may reference either an abstract interface or a normal interface. Examples:

```
abstract interface
   function fun_r(x)
      real,intent(in) :: x
      real            :: fun_r
   end function fun_r
end interface

procedure(fun_r) :: gamma, Bessel

interface
   subroutine sub_r(x)
      real :: x
   end subroutine sub_r
end interface

procedure(sub_r) :: sub
procedure(real) :: psi
```

The declarations for gamma and Bessel use the abstract interface fun_r. The declaration for sub uses the explicit interface for sub_r. The declaration for psi uses an implicit interface, and is equivalent to real,external :: psi.

Procedure pointers

The procedure statement may be used to declare procedure pointers. The pointer name may be used in place of the target name in CALL statements, function references, or as an actual argument. Procedure pointers may be components of derived types. Examples, assuming the abstract interface for fun_r above:

```
procedure(fun_r),pointer :: &
                  special_fun => null()
special_fun => gamma
```

The name special_fun is effectively an alias for gamma. Prior to the pointer assignment statement, special_fun was default initialized to disassociated.

```
type proc_ptr
   procedure(fun_r),pointer :: special
end type proc_ptr

type(proc_ptr) :: special(10)

ans = special(i)%fun(arg)
```

The second example defines a list of 10 procedure pointers, and the syntax for referencing a procedure pointer.

Pointer assignment lower bounds

Pointer assignment of a pointer array to a target array section in f95 always resulted in the lower bound of the pointer array of one. Fortran 2003 allows the specification of the lower bound in the pointer as part of the pointer assignment syntax. This feature simplifies programming by allowing the pointer and its target to have corresponding subscript values. Example:

```
real,pointer  :: p(:)
real,target   :: t(100)

p     => t(2:5)  ! old syntax
p(2:) => t(2:5)  ! new syntax
```

Executing the old syntax for of the pointer assignment associates p(1) with t(2). The new syntax form associates p(2) with t(2).

Pointer rank remapping

Pointers of any rank can have rank-1 targets through pointer remapping. The rank-1 target may be more useful in some circumstances, such as an argument to an f77 function, while the higher rank version may be clearer in computation expressions. Example:

```
real,pointer :: p(:,:)
real,target   :: t(100)

p(1:10,1:10) => t
```

The data in the array t can be referenced either through t as a length 100 vector, or through p as a 10 by 10 array.

FLUSH statement

File I/O is typically buffered in memory before actual transfers to or from disks take place. The Cray library has provided a flush subroutine to force a read or write of the memory buffers before they are full. Fortran 2003 provides a portable syntax for this operation as a Fortran statement. Example:

```
flush 10

flush(unit=10, iostat = n)
```

The first form of the flush statement parallels the backspace and endfile statements. The second form also accepts iomsg and err keywords. The value returned for the iostat variable is zero if no error occurs, a positive value if there was an error, and a negative value if the flush operation is not supported for the specified unit. If the iostat value indicates an error, and the iomsg optional keyword is supplied, then the iomsg value is set to a printable error message. The optional err keyword is similar to err on other I/O statements, specifying a statement number as a branch target if there is an error.

Error message text

A new keyword, iomsg, is provided for most I/O statements. If there is a error, end of file, or end of record, in the execution of the I/O operation, the character variable specified by iomsg is set to a text message describing the error or condition. This message could be used to provide more useful output in the case of an error. This option is typically used in conjunction with iostat to ensure that an error condition does not abort the program before the user has a chance to print the iomsg value. Example:

```
character(1024) :: msg

read(10,iomsg=msg,iostat=n) x
```

Intrinsic procedures implemented in cftn 5.4.0.3

Several of the intrinsic functions have additional features and there are some new intrinsics. The intrinsics that are part of the C interoperability feature were described earlier. Additional intrinsics that are not yet implemented are described in later sections.

The MIN and MAX functions are extended to accept character arguments.

The NEW_LINE function returns the character used as a record separator in stream files and in C text files. On almost every system, including the Cray, this is achar(10).

Six new intrinsic procedures are provided to obtain information about the execution environment.

GET_COMMAND returns as a character value the entire command that was issued to execute the program.

COMMAND_ARGUMENT_COUNT returns an integer with the number of arguments in the command issued to execute the program.

GET_COMMAND_ARGUMENT returns the specified command line argument as a character value.

GET_ENVIRONMENT_VARIABLE returns the definition of an input environment variable as a character value.

IS_IOSTAT_END returns true if the argument is one of the iostat values corresponding to an end of file condition. Cray systems do provide for more than one end of file value. One of the values is IOSTAT_END from the iso_fortran_env module. However, the IS_IOSTAT_END function is a more general and robost method for checking end of file values.

IS_IOSTAT_EOR returns true if the argument is one of the iostat values corresponding to an end of record condition. On Cray systems there is only one end of record value, which is the value of IOSTAT_EOR from the iso_fortran_env module.

Parameterized derived types

A major goal of f90 was the ability to write codes with parameterized precision and user specified generic procedures. For codes that required derived types, this sometimes required defining a set of nearly identical types that differed only in the kind parameters of the components. Fortran 2003 allows specification of parameterized types. Type parameters may be either kind type parameters or length type parameters. The value of a kind type parameter must be known at compile time. These are typically used to specify kind values in declarations. Length type parameters may be deferred until run time. Length type parameters are typically used to specify sizes of arrays or character variables. An example of a tri-diagonal matrix type might look like:

```
type(k,n) ::  tridiag
   integer,kind :: k
   integer,len  :: n
   real(k) :: upper(n-1)
   real(k) :: diag(n)
   real(k) :: lower(n-1)
end type tridiag

integer,parameter::rk=8

type(tridiag(8,20)) :: mat20
type(tridiag(rk,:)) :: mat(:)

allocate(type(tridiag(rk,20) :: mat(4))
```

The definition of the type tridiag involves both kind and length parameters, k and n. These must be declared as integer in the type using the KIND and LEN attributes. The variable mat20 is declared as a tridiagonal matrix with 64 bit elements and 20 elements on the diagonal. The declaration of mat uses deferred length parameters. The actual length parameter value is specified in the allocate statement where the array of 4 tridiagonal matricies is created.

Keywords in derived type constructors

Derived type constructors are extended to allow the use of the component names as keywords, similar to the syntax for procedure references. Example:

```
type foo
   integer :: ii
   integer,allocatable :: bar(:)
end type foo

type(foo) :: fobj

fobj = foo( ii = 1, bar = null() )
```

Type specs in array constructors

Array constructors may be used to define an array of constant values. The type and type parameters of the array constant are based on the type and type parameters of the elements. Allowing explicit type specifications in the constructors applies a type cast to each of the constants in the array. This specifies the type and type parameters of the array independent of the forms of the constants. Example:

```
character(7) :: names(3)

names=[character(7)::   &
           'Brian','Melanie','Jeff']
```

Without the type specification in the array constructor, the compiler will complain that the lengths of the character constants do not match, and hence the length parameter for the array constant is not defined.

Allocatable character scalars

Scalar objects of type character can be allocatable. This feature is valuable in contexts where the size of a character is determined by runtime data. The actual size of the variable is specified in the allocate statement. Example:

```
character(len = :),allocatable :: string

allocate (character(16) :: string)
```

Allocatable character assignment

The new rules for assignment of allocatable arrays described earlier also apply to allocatable character scalars. If the length of an allocatable character scalar variable does not match the length of the expression to which it is being assigned, the variable is reallocated with the correct length before data is copied. Note that this is very different from the assignment for non-allocatable characters where different lengths were also allowed, but carried implied truncation or padding of the right hand side expression. The new assignment rule for allocatable characters effectively provides a varying length string facility in Fortran. Example:

```
.character(len = :),allocatable :: string

allocate (character(16) :: string)
string='0123456789abcdef'

string(:) = 'pad'   ! Line 1
string = 'short'    ! Line 2
```

The statement with the Line 1 comment results in a 16-character result padded with 13 spaces on the right because the left hand size is a substring and not an allocatable character variable. The statement with the Line 2 label uses the new assignment rule for allocatable characters, and reallocates string to have length 5.

IEEE features

Support for IEEE floating point arithmetic is a major new feature in Fortran 2003. This is optional in the sense that the features are not required on systems that do not have hardware support for particular modes or functions. The IEEE_FEATURES intrinsic module contains constants that are defined if the processor supports the indicated feature. The full list of constants is

```
ieee_datatype
ieee_nan
ieee_inf
ieee_denormal
ieee_rounding
ieee_sqrt
```

```
ieee_halting
ieee_inexact_flag
ieee_invalid_flag
ieee_underflow_flag
```

Undefined constants correspond to unsupportable features. A USE of the module with an ONLY clause can detect the absence of a feature at compile time.

IEEE arithmetic control

The IEEE_ARITHMETIC intrinsic module defines a type, ieee_class_type, and constants of that type corresponding to the possible values of ieee floating point numbers:

```
ieee_signaling_nan
ieee_quiet_nan
ieee_negative_inf
ieee_negative_normal
ieee_negative_denormal
ieee_negative_zero
ieee_positive_zero
ieee_positive_denromal
ieee_positive_normal
ieee_positive_inf
ieee_other_value
```

The module also defines a type, ieee_round_type, and constants of that type corresponding to the ieee rounding modes:

```
ieee_nearest
ieee_up
ieee_down
ieee_to_zero
ieee_other
```

IEEE arithmetic functions

The IEEE_ARITHMETIC intrinsic module also defines a set of functions to inquire about support for various features, get and set rounding modes, and perform ieee conforming operations. If the ieee_support_standard routine returns false, referencing the other routines may not be meaningful. The functions defined in the module are:

```
ieee_support_datatype
ieee_support_denromal
ieee_support_divide
ieee_support_inf
ieee_support_io
ieee_support_nan
ieee_support_rounding
ieee_support_sqrt
ieee_support_standard
ieee_support_underflow_control

ieee_class
ieee_copy_sign
ieee_is_finite
ieee_is_nan
ieee_is_normal
ieee_is_negative
ieee_logb
ieee_rem
ieee_rint
ieee_scalb
ieee_unordered
ieee_value

ieee_selected_real_kind

ieee_get_rounding_mode
ieee_set_rounding_mode
ieee_get_underflow_mode
ieee_set_underflow_mode
```

IEEE exception control

The IEEE_EXCEPTIONS intrinsic module defines two new data types: ieee_flag_type, and ieee_status_type. The ieee_status_type should be used to declare a variable that holds the current value of the floating point status. The constants of type ieee_flag_type defined in the module are:

```
ieee_overflow
ieee_divide_by_zero
ieee_invalid
ieee_underflow
ieee_inexact
```

The module also includes several routines to get and set values of exception flags:

```
ieee_support_flag
ieee_support_halting
ieee_get_flag
ieee_set_flag
ieee_get_halting_mode
ieee_set_halting_mode
ieee_get_status
ieee_set_status
```

If the ieee_support_flag or ieee_support_halting routines return false for a particular flag, referencing the corresponding get and set routines is not meaningful.

Cray IEEE extensions

The IEEE module procedures provide a mechanism for controlling and recording exceptions that result from floating point operations as well as controlling the rounding of inexact operations. In the hardware of the X1 series systems, two processor registers are used by these routines: C0, the floating point control register (FPCR), and C1, the floating point status register (FPSR).

There are five types of exceptions for floating point operations (with examples):

Overflow (huge(x)*1000.)
Underflow (tiny(x)*0.0001)
Divide_by_zero (1./0.)
Invalid (0./0., or NaN-NaN)
Inexact (1./3.)

The X1 hardware flushes the results of operations that underflow to 0. Denormal values are not supported on the X1. Typically underflows and inexact operations are not recorded or trapped by default.

There are four hardware rounding modes:

Round to nearest (the default)
Round up
Round down
Round towards zero

The FPCR contains three sets of bits to specify the following:

Is recording of an exception enabled? (5 bits)
Does a trap on an exception occur? (5 bits)
What is the current rounding mode? (4 bits)

The FPSR contains two sets of bits that record the results of floating point operations, under control of the corresponding bits in the FPCR:

Did an exception signal (i.e. occur)? The bit for a particular exception will be set only if the corresponding recording bit is set in the FPCR. (5 bits)

Trap bits in the FPSR trigger an exception fault. If an exception occurs and the corresponding trap bit in the FPCR is set, then the bit for that exception will be set in the trap field of the FPSR and the hardware will cause an interrupt and branch to a trap handler in the operating system. That handler can look at the contents of the FPSR trap field to determine what exception caused the trap. (5 bits)

The routines in the IEEE modules that access the FPSR always include an "lsync fp" instruction to ensure that all the outstanding floating-point operations have completed before the status bits are accessed.

Certain combinations of the settings of the control register bits are used frequently. The Cray version of the IEEE_EXCEPTIONS module includes 3 additional predefined constants of type ieee_status_type that can be used as the argument to the ieee_set_status subroutine. All three of these cause all 10 bits in the FPSR register to be cleared and the rounding mode to be set to round to nearest. These constants affect the FPCR as follows:

ieee_cri_silent_mode: no recording, no traps

ieee_cri_nostop_mode: all exceptions record, no traps

ieee_cri_default_mode: record and trap for overflow, divide_by_zero, and invalid. Do not record or trap underflow or inexact.

A user would typically save the current state of both registers with a call to the ieee_get_status subroutine, and then set one of the modes above by calling ieee_set_status. Later, the original status can be restored by calling ieee_set_status with the value returned from the original call to ieee_get_status.

Asynchronous I/O and WAIT statement

Fortran 2003 contains syntax support asynchronous input and output operations. An asynchronous read or write statement initiates the operation but allows the program to continue before the operation is finished. A separate WAIT statement forces the program to wait until the operation is completed. The functionality is essentially the same as that provided by the old buffer in and buffer out statements. Example:

```
open(10,…,asynchronous='yes',…)

read(10,…,asynchronous='yes',id=idw,…)
…
wait(10, id = idw)
```

Without the id clause in the wait statement all currently outstanding operations on the unit must complete. Executing a close or inquire operation on the unit has an implied wait if the file was opened as asynchronous.

Stream I/O

Part of the improved interoperability with C includes support for stream I/O. Files opened for stream I/O do not have internal record structure information. Formatted files may have embedded newline characters, matching the convention used by C programs to delimit records. Unformatted files do not contain internal record size information. The current location within the file can be obtained or specified with a POS= keyword in the I/O statement. Example:

```
open (unit=10, … access = 'stream', …)
```

DECIMAL mode in I/O statements

Support for the use of a comma, rather than a period, as the character that separates the fractional part and whole number part of a formatted real number is included as part of the internationalization features of Fortran 2003. A new DECIMAL keyword for the open statement is used to specify the mode. A DECIMAL keyword may be specified in a read or write statement, overriding the value specified in the open statement. Example:

```
open(unit=10, …, decimal="comma", … )

open(unit=11, …, decimal="point", … )
```

The internal value of 4.3 would be written to unit 10 as "4,3", and to unit 11 as "4.3". The default mode is "point". If the comma mode is used then list directed I/O operations use a semi-colon for the value separator.

Rounding mode in I/O statements

When real values are written to a file the conversion between the internal binary form and the external character string is usually inexact. The method used to determine the value of the final character(s) is determined by a convention on rounding. Fortran 2003 gives the user control over what rounding mode is used. The mode is specified with the ROUND keyword in the open statement. The keyword may also be supplied in a read or write statement which overrides the value specified in the open statement. The supported values for the rounding value are 'up', 'down', 'zero', nearest', 'compatible', and 'processor_defined'. The 'zero' mode means round toward zero. For systems that support IEEE arithmetic, the 'nearest' mode must conform to the IEEE nearest rounding rules. The 'compatible' mode differs from 'nearest' for the case where the actual value is exactly half way between possible output values. The 'compatible' mode rounds such values away from zero. This is designed for compatibility with some systems. The 'nearest' mode on IEEE machines rounds tie cases to even which is the standard science rounding convention. Example:

```
open (unit=10, …, round='down', …)

write (10, '(g20.7)', round='nearest') x
```

Keywords in read and write statements

Six of the keywords that can be specified in open statements to describe I/O modes may also be specified in a read or write statement. The value specified in the read or write statement overrides the corresponding value specified in the open statement, and affects only the I/O performed by that statement. The allowed changeable mode keywords are BLANK, DECIMAL, DELIM, PAD, ROUND, and SIGN. Example:

```
write (unit=10,fmt=*,decimal='comma') x
```

Result KIND specifiers in intrinsics

Several of the intrinsics that return integer results now include an optional KIND argument to specify the precision of the result. This overcomes an existing problem of the result being too big to fit in a default integer. The SIZE intrinsic is a typical example of a function with a new KIND argument.

Array reallocation – MOVE_ALLOC intrinsic

The goal of array reallocation is to end up with a new array that is larger or smaller than the old array of the same name, and containing (possible not all of) the values from the old array. This is now possible with fewer statements and memory operations by using the new MOVE_ALLOC intrinsic subroutine. MOVE_ALLOC effectively changes the address of an allocatable object descriptor. Example:

```
integer,allocatable :: x(:),tmp(:)

allocate(x(20))
! Want to expand x to 40 elements

! Old method
allocate(tmp(20))
tmp = x
deallocate(x)
allocate(x(40))
x(1:20) = tmp
deallocate(tmp)

! New method
allocate(tmp(40))
tmp(1:20) = x
```

```
call move_alloc(tmp,x)
```

Because the x argument to move_alloc is intent(out) the old storage for x is deallocated in move_alloc.

Derived type extension

Derived types are often extended from a general base type to a larger type that contains additional components for a more specific case. In f95 this was typically done by defining a new type for the specific case and including a component of the base type. This technique requires a multiple part reference for the components of the base type. If the specific type is extended again, the complexity of references to the parent types increases. Fortran 2003 allows explicit extension of a type such that the parent components are also components of the extended type. The parent components are "inherited" by the extended type. This eliminates the reference part explosion, and is also more in keeping with the style of modern object oriented programming. Example:

```
type :: dna
   integer,allocatable :: ascii_text(:)
   integer  :: length
end type dna

type(extends(dna)) :: ocdna
   integer :: ssdid
   integer :: ssdsize
   integer :: state
end type ocdna
```

The derived extended type ocdna (out of core version of dna) contains five components, the three specified along with the two inherited from the parent type dna. There is also an implied component named dna that allows indirect access to the parent types in the f95 style. This can be useful in cases there dummy argument type matching requires an object of the parent type.

Most derived types may be extended, although sequence and bind(c) types are not extendable. A type can inherit components from only one parent, commonly known as single inheritance. However, several extended types may have the same parent.

Two new intrinsic functions support type extension. The EXTENDS_TYPE_OF is true if the type of the first argument is an extension of the type of the second argument. The SAME_TYPE_AS function returns true if the two arguments have the same type.

Type-bound procedures

Procedures can be bound to a type, automatically carrying along interface information with each variable of that type. Type-bound procedures are part of the overall OOP features of Fortran 2003. Procedures are declared with PROCEDURE, GENERIC, or FINAL statements. The type contains only the declaration for the procedure. The actual procedure is defined elsewhere. The interface for the procedure must be visible to the type definition. A type-bound procedure may have an implied argument of the containing type, specified with the PASS attribute, Example:

```
type strange_int
    integer :: n
contains
    generic :: operator(+)=> strange_add
end type
```

The interface for strange_add must be either supplied by an interface block, or by defining the function in an accessible module.

Finalizers

Finalizers are a special type of type-bound procedure that is executed when an object of the containing derived type becomes undefined. A variable may become undefined by various means, including the initial state of an intent(out) dummy argument, of the state of a unsaved local variable at procedure exit. Finalizers are specified with the FINAL declaration. Example:

```
type foo
    real,pointer :: bar(:)
contains
   final :: foo_cleanup
end type

subroutine foo_cleanup(x)
   class(foo) :: x
   deallocate(x%bar)
end subroutine foo_cleanup
```

Polymorphic objects

The CLASS type specifier is used to declare polymorphic objects. These declarations must be for dummy arguments, or the declared object must have the allocatable or pointer attribute. The primary use of polymorphic objects is as dummy arguments. Actual arguments of the type specified, or any extension of that type, are type compatible with the corresponding dummy argument. Assuming the

subprogram uses only components from the base type, all extensions of that type will also have those components and hence be a reasonable type for actual arguments. The specification of a polymorphic dummy argument allows the routine to be called with arguments of the base type or any of the extended types. It is possible to declare something CLASS(*), or unlimited polymorphic. Such an object is type compatible with any type object. Use of an unlimited polymorphic object is limited to allocate statements or statements within a select type construct, where more information about the actual type can be determined. Example:

```
function strange_add (a,b) result (c)
   class(strange_int),intent(in) :: a,b
   type(strange_int)             ::c

   c%n = iand(a%n+b%n, 1)
end function strange_add
```

This function is assumed to be in the same module that defines the type strange_int above.

Select Type construct

The select type construct allows alternate execution paths based on the actual type of a polymorphic object. The selection clauses are TYPE IS, CLASS IS, or CLASS DEFAULT. If the type of the argument specified in the select type statement matches one of the types specified in a TYPE IS clause statement, then the code block following that statement is executed. If none of the TYPE IS types match the type of the selector, then the CLASS IS clauses are tried. The most extended type that matches is selected. If none of the CLASS IS statements has a compatible type, the CLASS DEFAULT block is executed. CLASS IS (*) is not allowed because it is redundant with CLASS DEFUALT. Example, assuming the definition of strange_int from above:

```
type,extends(strange_int) ::strange_mint
   integer :: m
end type strange_mint

class(strange_int) :: a,b,c

select type(a)
type is (strange_int)
   c%n = iand(a%n+b%n,1)
class is (strange_int)
   i = min(a%m.b%m)
   c%n = iand(a%n + b%n, 2**i-1)
   c%m = i
end select
```

Enhanced initialization expressions

The values of initialization expressions must be computable at compile time and are commonly used to provide kind values or values of parameters. The restrictions on these expressions have been relaxed in Fortran 2003. In particular, most of the language intrinsic functions may be referenced in initialization expressions. This feature is especially useful in the portable definitions of parameters. Example:

```
   real,parameter :: pi = acos(-1.0)
```

Derived type I/O control

The default mechanism for handling a derived type variable in an I/O list is to effectively expand the item into a list of items, one for each component of the type. Fortran 2000 allows the user to specify subroutines that handle the I/O operations to be performed in a derived type variable. Up to four routines may be specified with these generic names: read(formatted), write(formatted), read(unformatted), and write(unformatted). These are typically type-bound procedures and are invoked for formatted I/O if the format contains a corresponding DT format specifier. Examples:

```
type :: dna
   integer,allocatable :: ascii_text(:)
   integer             :: length
contains
   generic :: write(formatted) => fw_dna
end type dna

type(dna) :: hs_chr20
…
write (10,'(dt)') hs_chr20
```

In printing out the dna string for human chromosome 20, only the text should appear, and not the length. In this case the default derived type I/O would not provide the desired result. It would be possible to write out the individual component, but this is not in the OOP spirit. The form of the user written function, fw_dna, must have a specific interface since this will be called by an I/O library routine. Interfaces are detailed in the standard for all 4 of the possible routines. For this example:

```
subroutine fw_dna( dtv,    &
                   unit,   &
                   iotype, &
                   vlist,  &
                   iostat, &
                   iomsg)
```

```
class(dna),intent(in) ::dtv ! hs_char20
integer,intent(in)    :: unit   ! 10
character(*),intent(in):: iotype ! "DT"
integer,intent(in)    :: vlist
integer,intent(out)   :: iostat
integer,intent(inout) :: iomsg

! Write out the first dtv%length
! characters in dtv%ascii_list.
! Set iostat based on results of the
! write.
! Set iomsg if iostat was non-zero.
! The vlist argument is not used in
! this example

end subroutine fw_dna
```

ISO character set support

As part of the internationalization of Fortran the new standard provides a standard method for declaring character variables based on the ISO 10646 standard. The ISO 10646 character set defines 32 bit characters and includes characters for most of the world's languages. A new selected_char_kind intrinsic is provided to return the kind value appropriate for the 10646 character set, or to indicate that is it not supported by the compiler. Selected_char_kind accepts three argument values: "ASCII", "DEFAULT", and "ISO_10646". For Cray systems, "ASCII" and "DEFAULT" will return the same result value since the default character set is ASCII. Example:

```
integer,parameter :: usc4  = &
     selected_char_kind('iso_10646')

character(len=5,kind=usc4) :: c

c = usc4_"     "
```

Text encoding

Support for character variables containing ISO 10646 characters is complemented by I/O support for files containing a standard encoding, called Unicode, of the ISO 10646 characters. Records from a file containing these characters should be transferred to and from character variables with the ISO kind type. The encoding keyword values are 'utf-8' and 'default'. The default is 'default' and corresponds to ASCII on Cray systems. Example:

```
open(unit=10, …, encoding='utf-8', …)
```

4. Fortran 2008

With the completion of the Fortran 2003 standard, the attention of the Fortran standardization process has turned to the next standard, tentatively referred to as Fortran 2008. Proposals have been submitted to WG5, the ISO level policy setting committee for Fortran, over the past two years. A meeting of WG5 was held in Delft, The Netherlands, during the week of May 9, 2005 to decide on an initial list of features for the next standard. Proposals that were submitted were put into one of 4 groups. The Group A proposals were the highest priority and should be put into the next standard. The group B proposals were desirable and should be included if time permits in the document editing process. Group C proposals were those on which consensus could not be reached, and Group D proposals were rejected at this time. A second WG5 meeting will be held in February, 2006, to access the editing progress and set the final list. Before that meeting there will be two meetings of J3, the body tasked with writing the standard document, where creating the edits needed to add the new features will be the main focus. The likelihood that all of the Group A features will be in the final standard is very high. The fate of the Group B features will be determined by the amount of time available.

The Group A features are:

Submodules
Co-arrays
DO Concurrent construct
Contiguous attribute
Intent(scratch) attribute
internal procedures as arguments and targets
pointer valued functions as actual arguments
require 64-bit integer support
standard form of 'call system()'
rank up to 15
additional math intrinsics
enhanced STOP statement
decimal arithmetic
allow empty contains section
allow type(intrinsic-type)
allow ASCII as arguments to lexical intrinsics
move ENTRY to the obsolescent list
move statement functions to the deleted list

The Group B features are:

Parameterized modules
BITS intrinsic data type
C interoperability with allocatable and pointers
C interoperability with optional arguments
EXIT from additional labeled constructs

non-null pointer initialization
pointer valued function ref on lhs
assignment for allocatable polymorphics
additional intrinsics from libm and HPF
CSV file support
parameter size from initialization expression
generic resolution from pointer/allocatable
generic resolution from procedure/variable
conformance to the IEEE 754R standard
separate definition of complex parts
IO_UNIT data type

In the following sections the Group A and Group B features will be discussed in more detail. The Group C and D proposals will not be discussed.

5. Fortran 2008 Group A features

Submodules

Submodules, also referred to as Enhanced Module Facilities, have already been specified in detail in ISO/IEC TR 19767:2004. Submodules provide a mechanism to separate the interface and implementation parts of a module procedure. Submodules are described in more detail in a companion paper in the CUG 2005 proceedings: "Programming for High Performance Computing in Modern Fortran". Submodules enhance programmer productivity and help reduce program maintenance costs, especially for large projects.

Co-arrays

Co-arrays provide a direct and simple syntax for exchanging data between images of a parallel execution, and intrinsic image synchronization procedures. The inclusion of co-arrays in Fortran 2008 fundamentally changes Fortran into a parallel language, recognizing that in that time frame most systems will contain multiple processor cores. Co-arrays are the most significant new feature in the Fortran 2008 standard. They are discussed in more detail in a companion paper in the CUG 2005 proceedings: "Programming for High Performance Computing in Modern Fortran". Co-arrays are a significant programmer productivity feature as well as a performance feature of Fortran 2008.

DO Concurrent construct

The DO CONCURRENT construct is a variant of the DO construct. It uses a loop control syntax like that of the FORALL construct, but allows a much wider class of statements than does FORALL. The iterations of the construct may be executed in any order, or in parallel. The concurrent construct provides a portable mechanism to replace vendor specific directives with the same basic meaning. The goal is to provide the compiler with significant optimization flexibility that could not be deduced from the form of the loop statements. The DO CONCURRENT construct is a performance feature.

Contiguous attribute

A pointer or assumed-shape variable declared with the contiguous attribute may be assumed to occupy a contiguous region of memory. This allows the compiler to use some optimizations, such as loop collapse, that are otherwise available only to allocatable, explicit-shape, or assumed-size arrays. The contiguous attribute is a performance feature.

Intent(scratch) attribute

Specifying intent(scratch) for a dummy argument indicates that the initial value of the associated actual argument on entry is not needed and that the final value of the argument need not be stored. This is a performance feature.

Internal procedures as arguments and pointer targets

In previous versions of Fortran, an internal procedure could be referenced only from the host scoping unit or another internal procedure in the same host. Fortran 2008 relaxes that restriction and allows internal procedures to be used as actual arguments and to be the targets of procedure pointers. This is a programmer productivity feature.

Pointer valued functions as actual arguments

If a pointer is supplied as an actual argument corresponding to a dummy argument that is not a pointer, the dummy argument is associated with the target of the pointer actual argument. If the dummy argument has the intent(out) attribute then the target of the pointer actual argument shall be definable. If the actual argument is a function reference that returns a pointer result, then the dummy argument is still associated with the target of that pointer. However, the current wording of the Fortran standard does not allow this in the case where the dummy argument has intent(out) because the actual argument has the form of an expression. This feature is intended to correct that conflict in the wording of the standard, and explicitly allow the pointer valued function reference as the actual argument. This feature is a maintenance correction to the current standard.

Require 64-bit integer support

Essentially all compilers currently support a 64-bit integer type, though it is not explicitly required by the standard. This feature makes that requirement explicit, so that programmers can specify 64-bit integers and remain within coding guidelines that require standard conformance. If the processor's default integer is already 64-bits, then there is no effective change in the standard. If the default is different, usually 32 bits, than the new standard will require a second integer kind be supplied. This feature standardizes existing practice.

Standard form of 'call system()'

Most processors provide a mechanism to trigger the execution of another program, often by calling a subroutine named 'system'. The proposal is to require an intrinsic subroutine named EXECUTE_COMMAND_LINE that provides a portable means to execute a separate program. This feature standardizes a common and useful capability.

Rank up to 15

The rank of an array, or the combined rank and co-rank of a co-array, is extended to 15 with this feature. It has been several decades since the rank limit was changed from 3 to 7. With much larger modern systems, it seemed reasonable to increase the limit again. This is nominally a programmer productivity feature.

Additional math intrinsics

Intrinsic hyperbolic, inverse hyperbolic, and inverse trig functions with complex arguments are added so that the list of Fortran math intrinsics is aligned with those required by C99. This is primarily a maintenance update to Fortran.

Enhanced STOP statement

The STOP statement currently allows an optional trailing integer value of up to 5 digits. This proposal relaxes the limit on the size of the integer and allows named integer constants (parameters). In addition, for a processor-defined range of values, expected to be small, the value is actually returned by the program as its exit status. This is a programmer productivity feature.

Decimal arithmetic

The pending IEEE 754R standard includes support for decimal floating-point formats and arithmetic. This feature will provide a means for obtaining KIND values for REAL decimal objects. The proposal is to add a RADIX= argument to the selected_real_kind() intrinsic function. On systems that do not support decimal arithmetic in hardware, the implementation will likely return a negative value from selected_real_kind if radix=10 is specified. At this time only IBM has expressed its intention to support radix 10 reals. This is part of a larger Group B feature to track the evolving 754R standard.

Allow empty contains section

A contains statement separates the specification part of a module from the procedure part. It is also used to separate parts of a derived type. This proposal is to allow a contains statement even if the part that comes after is empty. This was a request from authors of automatic code generators.

Allow type(intrinsic-type)

The type() statement is used to declare variable of a derived type. Derived type names shall not be the same as an intrinsic type name, so there is no syntax conflict to allow, for example, 'type(real)' to be an alternate spelling for 'real'. This proposal removes the restriction that intrinsic type names are not allowed in type() statements. This was a request from authors of automatic code generators.

Allow ASCII as arguments to lexical intrinsics

Four intrinsic functions, LLT, LLE, LGE, and LGT, provide character comparisons according to the ordering of the ASCII character set, independent of the kind of the default character set. The arguments are currently restricted to be default character. This proposal expands this to also allow ASCII character arguments in the case where they are not the default. While this is very unusual today, in the future some systems might use the ISO characters as the default kind. This is a maintenance update to the standard.

Move ENTRY to the obsolescent list

The Fortran standard maintains a list of obsolescent features in Annex B. These features are still part of the full standard, but are candidates for deletion in some future release. This proposal moves the ENTRY statement to the list of obsolescent features. This is a maintenance update to the standard.

Move statement functions to the deleted list

The Fortran standard maintains a list of deleted features in Annex B. The normative text describing these features is removed from the main standard and vendors are no longer required to support them in a standard conforming compiler. As a practical matter, they are usually still used in older codes and vendors do not actually delete them. This proposal moves statement functions to the deleted list. While having no practical affect on users, it will simplify maintaining the text of the standard as other new features are added. This is a maintenance update to the standard.

6. Fortran 2008 Group B features

Parameterized modules

Parameterized modules provide generic programming functionality to Fortran, and complete the object oriented programming aspects of the language introduced in Fortran 2003. A parameterized module can have arguments that correspond to actual arguments that are constants or type specifiers. The dummy arguments are treated like macros in the definition of the module. When an instance of the module is created through a USE or instantiate statement, the actual arguments are substituted and the resulting module is compiled for later use as an ordinary module. The model for this feature comes from Ada. It addresses the same need as the template facility in C++. This is a large feature aimed at programmer productivity.

BITS intrinsic data type

A new intrinsic data type, BITS, provides support for sequences of bits. The bit_size of a data object is set at compile time but is not restricted to word sizes. A separate kind parameter determines alignment of the data to ensure that an efficient implementation is possible. The current BOZ (binary, octal, hexadecimal) constants will be of type BITS. Assignment of integer, real, or complex data to a BITS variable just moves the bits with no format change. Argument association and pointer association involving BITS objects also do not change the bit patterns of the data. The operators .and., .or., .xor., and .not. are defined for BITS objects and perform the usual bit-wise operations. Comparison operators (<, <=, ==, /=, >=, and >) are defined for BITS objects and treat a 1 as larger than 0, with the comparison starting with the leftmost bits in the object. List-directed I/O uses Z format for BITS objects. Several new intrinsic functions are included as part of this feature, including popcnt, leadz, poppar, trailz, faster shift intrinsics, and intrinsics for mask generation. This is a large feature that is aimed at high performance in non-numeric computations, improved programmer productivity, and standardization of several existing extensions.

C interoperability with allocatables and pointers

The current C interoperability feature in Fortran does not allow interoperation with allocatable, pointer, or assumed-shape Fortran objects. This feature provides a set of C function prototypes that a C programmer could use to interoperate with these classes of Fortran objects. Vendors would provide the actual library routines described by the prototypes. These routines effectively provide a mechanism for C programmers to create and extract information from the Fortran processor's dope vectors. This feature should improve program portability and programmer productivity.

C interoperability with optional arguments

Fortran subprograms with optional arguments are not currently interoperable with C. This feature extends C interoperability to include Fortran optional dummy arguments. This feature should improve program portability and programmer productivity.

EXIT from additional labeled constructs

The EXIT statement allows a user to branch to the statement following the end of a DO construct. This allows users to avoid GOTO statements and the associated statement numbers. This feature extends the use of EXIT to similar use in labeled IF, select case, select type, and associate constructs. Exits from the critical construct in the co-arrays feature would also be allowed. Constructs such as FORALL that do not currently allow a GOTO statement would not allow EXIT either. This is a maintenance update and supports improved code maintainability.

Non-NULL pointer initialization

The only value allowed for a pointer initialization is a reference to the NULL() intrinsic. This feature would allow other valid pointer targets as initialization values. Both data and procedure pointers are included in the proposal. This is a programmer productivity feature.

Pointer valued function ref as left-hand-side

A pointer on the left hand side of the equals sign in an ordinary assignment statement causes the value of the expression on the right had side of the equals sign to define

the target of the pointer. This feature extends the same functionality to the case where the left hand side is a reference to a function that returns a pointer. This functionality is equivalent to pointer assigning the function result to a temporary pointer and then using that temporary as the left hand side of the assignment. This is a maintenance feature to make use of pointers more consistent in the standard.

Assignment for allocatable polymorphic variable

An allocatable array used as the variable in an assignment will be automatically allocated or reallocated if its current shape does not match that of the expression. This feature extends that idea to polymorphic allocatable variables. In the current standard, a polymorphic allocatable variable gets its dynamic type when it is allocated with an allocate statement. With this new feature, if the current dynamic type of the variable does not match the actual type and type parameters of the expression then the variable is allocated or reallocated with the correct dynamic type and type parameters before the assignment is done. This feature improves the internal consistency of the standard and improves programmer productivity.

Additional intrinsics from limb and HPF

Selected special functions available in libm (Bessel functions, error function, and the gamma function) often appear in Fortran programs and are standardized as language intrinsics by this feature. The function names become generic, simplifying coding. In addition, the functionality provided by selected bitwise reduction functions and prefix and suffix reduction functions from HPF is added through new language intrinsics by this feature. This feature mainly standardizes the functionality of existing extensions.

CSV file support

Comma Separated Value (CSV) files are commonly produced and consumed by applications like spreadsheets. Fortran can already read such files with list-directed input statements. This feature specifies a new form for an output file that ensures the values are written as a comma-separated list without repeat factors. This will ease the use of Fortran in conjunction with such applications.

Parameter size from initialization expression

The length of a character parameter may be specified as * with the actual length taken from the length of the initialization expression. This feature extends that capability to the shape of a parameter array. The array's actual shape is taken from the shape of the initialization expression. This is a programmer productivity feature.

Generic resolution from pointer/allocatable

In the specific interfaces that make up a generic interface, the dummy argument characteristics must differ according to a set of rules that ensures that an actual reference to the generic name can be resolved to only one of the specific names. This feature relaxes those rules such that a dummy argument declared with the pointer attribute is distinguished from an otherwise similar dummy argument with the allocatable attribute. This is a maintenance update to the standard.

Generic resolution from procedure/variable

This feature is similar to the one above, except that the generic resolution rules are relaxed such that a dummy argument declared to be a procedure is distinguished from an otherwise similar argument that is a data object. This is a maintenance update to the standard.

Conformance to the IEEE 754R standard

The current standard contains a large section (Section 14) that provides optional conformance with the IEEE 754 floating point standard. That standard is being updated to IEEE 754R. At this time the 754R standard is not completed, but this feature proposes to modify Section 14 as needed to conform to the new standard when it is available. The current timetable calls for the content of 754R to be available in November, 2005. This is a maintenance update of the standard.

Separate definition of complex parts

This feature specifies a notation that will allow the real or imaginary part of a complex variable to appear on the left of the equals sign in an assignment. This allows each part to be defined separately. This is a programmer productivity feature.

IO_UNIT data type

A long-standing problem encountered when using Fortran I/O is not knowing which I/O unit numbers are available for use. This feature introduces a derived type defined in the iso_fortran_env module. A variable of that type can be used instead of an integer file number. An OPEN statement that included a FILE specifier would return a value to a variable of this type that is then used in later I/O statements to refer to the file. The expectation is that the derived type will have the form of a system dependent file

handle that is directly usable by the I/O library. Since the system supplies the IO_UNIT value, the user does not need to keep track of unit numbers. This is a programmer productivity feature.

Acknowledgments

The author would like to thank the Cray Fortran compiler group for early implementation of many of the Fortran 2003 features as well as input on the future implementation schedule.

About the Author

Bill Long represents Cray as a primary member of the J3 Fortran standard committee. He is also the primary author of the Cray Bioinformatics Library, most of which is written in Fortran 2003. Bill can be reached at Cray Inc., 1340 Mendota Heights Road, Mendota Heights, MN 55120, Email: longb@cray.com.