# Fortran 2000

**Bill Long**, *Cray Inc*

**ABSTRACT:** *Fortran 2000 is the popular name for the upcoming revision of the Fortran programming language. This revision contains several new features to address shortcomings of the current standard, as well as major new additions to the language in the areas of interoperability with C, object oriented programming, I/O, and support for the IEEE standard. The new features are reviewed along with the current implementation plan for a standard conforming compiler on the main product line Cray systems.*

## 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 a focus on scientific computing and computational efficiency. Major milestones were the 1966 (f66), 1978 (f77) and 1991 (f90) standards, with a minor revision in 1997 (f95). Currently nearing completion is another major revision of the standard. It is popularly referred to a Fortran 2000, although the final version of the standard will not be formally adopted until 2004. The contents of this paper are based on the 29-April-2003 Working Draft of the standard document. There are potentially two additional drafts of the standard yet to come, so some of the details presented here may differ slightly from the official language. However, the focus of the standard and the major features are unlikely to change significantly from what is presented here.

Fortran 2000 retains backward compatibility with f95, with very few exceptions, while including a significant number of changes. These fall mainly into two groups. The first group contains a large number of changes that address deficiencies or shortcomings in f95 that are based in the experience of users and implementers of the current language. The change to allow allocatable components in derived types is an example in this group. The second group of changes are new features for capabilities thought to be useful for a class of applications, but missing from the current language. These are better interoperability with C, features to support object oriented programming (OOP), new types of I/O, and a standard mechanism for accessing the facilities specified in the IEEE floating point standard.

The remainder of this paper is organized into broad categories of features that generally follow the sections of a program. First is a brief description of additions to the basic language syntax that apply to all the examples. Subsequent sections detail changes in Declarations, Procedure specification, Basic Operations, and I/O. The final section contains a map of the standard process from here to completion, and some speculation on what might be considered for standards farther into the future.

Each section includes comments on the implementation plans for that feature using four categories. Some features are actually just a standardization of existing extensions that have been implemented for years. The remaining three groups of features are: “already implemented”, (included in ftn version 5.0 to be released soon), “early implementation” (ftn version 5.1 or 5.2, to be released through the first quarter of 2004), and “later implementation”.

This paper focuses on the differences between Fortran 2000 and the current Fortran standard. There is an implicit assumption that the reader is already familiar with f95.

## 2. Basic Syntax

Statement form in Fortran 2000 relaxes some of the restrictions of the current Fortran. Names of objects (variables, procedures, common blocks, types, etc.) may contain up to 63 characters, up from 31. The number of continuation lines for a single statement is 255, up from 39. These changes were made based on user comments that the old limits were too restrictive, especially for the case of source code created by other programs. These are early implementation changes.

The use of [ ] as the array constructor was proposed as early as the 1980’s. This was dropped from the f90 standard because some keyboards lacked these characters at the time. This is largely no longer the case, and the Fortran 2000 standard brings these back as an alternative to the current (/ /) array constructor. The issue is still being debated, though now for a different reason. There are proposals to reserve the square brackets for other syntax in future versions of Fortran. Examples below will include this syntax, but

implementation of this feature in our compiler will be postponed until after the final standard is approved.

Named constants as parts of a complex constant have been allowed in Cray Fortran for several years. This is now part of the standard. 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. These characters are part of the English language version of the standard ASCII character set, and hence are not new on Cray systems.

## 3. Declarations

New types of data structures make up a large part of the enhancements in Fortran 2000. The associated changes in syntax are in declaration statements. This section describes the new syntax and the motivation for the changes.

### *PROTECTED attribute*

Module data in f95 had a visibility outside the module based on their declaration as either public or private. These attributes apply to 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 values of objects. 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. The standardization of the protected attribute was promoted by Cray and has existed as an extension in our past compilers. Example:

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

### *Mixed public and private attributes*

Derived types defined in a module can be either public or private. New rules allow individual components of the type to be public or private. 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. Mixed public and private components are already implemented. Public object of a private type is an early implementation feature. 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 dynamically sized components of a derived type be declared as a pointer. Because a compiler cannot determine all the possible aliases for pointer target data, optimization of expressions involving such data is restricted. The new standard allows allocatable components, which do not have this performance problem. Status: already implemented. Example:

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

### *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. New syntax for the ALLOCATE statement is provided for this feature. Status: early implementation. Example:

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

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

### *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 2000 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. The syntax for specifying an intrinsic

module has been implemented in the Cray compilers. Examples for using each of the new modules:

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

The features in the three ieee intrinsic modules are more closely tied to execution control rather than declarations, so appear later in the basic operations section.

***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 defines 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 unit 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. Also specified is the default error output unit, 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. The iso_fortran_env module is scheduled for early implementation.

***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 or structure components corresponding to C data and function pointers. The C_FUNPTR type is scheduled for later implementation. The constants and C_PTR type in the iso_c_binding module are already implemented.

***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 name of the corresponding global C object 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.

The bind(c) attribute and binding to C global variables have already been implemented.

***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 2000 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,length :: 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 LENGTH 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. Paramterized types are scheduled for later implementation.

***Extended types***

Derived types are often extended from a general parent type to a larger type that contains additional variables for a more specific case. In f90 this was typically done be defining a new type for the specific case and including a component of the parent 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 2000 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 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 multi-part access to the parent types in the f90 style. This can be useful in cases where dummy argument type matching requires an object of the parent type.

Most derived types may be extended, though 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. Type extension is scheduled for later implementation.

***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. The volatile attribute has existed with his same meaning as a language extension for several years, including in Cray compilers. Example:

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

***Enhanced initialization expressions***

The values of initialisation 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 2000. 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. Status: later implementation. Example:

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

***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 for the dummy argument declarations in the interface. Past standards have required that the definitions are either replicated in the interface or accessed by a USE of the 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 provides access to names visible in the surrounding host. If no names are specified in the import statement all the surrounding host names are visible. Status: already implemented. 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
```

***International character set***

Fortran 2000 provides a standardized method for declaring character variables with values from the ISO 10646 standard character set. The ISO 10646 standard 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 (1) since the default character set is ASCII. The selected_char_kind intrinsic is scheduled for early implementation, but full support for 10646 characters are deferred to later implementation. Example:

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

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

c = usc4_"     "
```

## 4. Procedures

Improved flexibility in specifying procedures and procedure interfaces, as well as type bound procedures as part of the object oriented programming model, are significant features of the Fortran 2000 standard.

***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 2000 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. Status: already implemented. 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 2000 and already implemented in the Cray compiler. Example:

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

***Intent for pointer dummy arguments***

Dummy arguments with the pointer attribute could not have an intent attribute in f95. Fortran 2000 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. Intent for pointer arguments is already implemented in the Cray compiler. 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).

***Interoperating with C functions.***

Interoperation with C functions with standard syntax is a major new feature of Fortran 2000. 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. If there is no name clause in the bind(c) attribute, the C name of the procedure is the Fortran name in lower case letters. 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 dummy arguments with the value attribute are defined within the subroutine, the corresponding actual arguments are 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 to describe the function interface. Status: already implemented. 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);
```

***PROCEDURE statement***

The PROCEDURE statement is an extension of the module procedure statement from f90, used to define a generic interface. The specific procedures named in a procedure statement 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. Status: early implementation. 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 of actual procedures. A procedure statement may reference either an abstract interface or a normal interface. Status: later implementation. 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. Status: later implementation. 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.

***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 2000. Procedures are declared with PROCEDURE, GENERIC, or FINAL statements. The type contains only the declaration for the procedure. The actual procedure is defined elsewhere. Only 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. Status: later implementation. 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.

***Polymorphic objects***

The CLASS type specifier is used to declare polymorphic objects. These declarations must be for dummy arguments, or 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. Status: later implementation. 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. Status: later implementation. 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
```

***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, or the state of a unsaved local variable at procedure exit. Finalizers are specified with the FINAL declaration. Status: later implementation. 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
```

***Some new and changed intrinsic procedures***

Several of the intrinsic functions have additional features and there are some new intrinsics. The intrinsics that are part of the C interoperability feature are described in the next section. The following section describes the new environmental intrinsics. The remaining changes for intrinsics are described in this section.

The MIN and MAX functions are extended to accept character arguments. Status: already implemented.

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. Status: early implementation.

Two new functions support OOP features. 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. Status: later implementation.

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 Crays, this is achar(10). Status: early implementation.

The MOVE_ALLOC function changes the address of an allocatable object descriptor. This is used to implement array reallocation, which is described in a later section. Status: early implementation.

***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 or as values of components of derived type objects interoperating with C structs. The three functions dealing with data pointers are already implemented. The two functions dealing with procedure pointers are scheduled for later implementation.

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.

***New environmental intrinsics***

Six new intrinsic procedures are provided to obtain information about the execution environment. All of these are scheduled for early implementation.

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.

## 5. Basic Operations

***Derived type constructor keywords***

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

***Assignment for allocatable objects***

The addition of allocatable components to Fortran 2000 required 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 size, then an array copy is done. If the left hand side array is not allocated, it is allocated with the correct size and then the array copy is done. If the current left hand side is allocate with the wrong side, it is deallocated and then reallocated with the correct size and the array copy is done.

The above assignment algorithm for allocatable variables is extended to all 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 size before the data is moved. Similarly, if the left hand side variable is allocated with the wrong size (not conforming to the f95 standard) then it is reallocated with the correct size 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. Status: early implementation. 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.

***Allocatable character assignment***

The new rules for assignment of allocatable arrays described in the previous section 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 caused 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. Status: early implementation. 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 comment uses the new assignment rule for allocatable characters, and reallocates string to have length 5.

***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 cluttered 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. Status: early implementation. 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
```

***Pointer assignment lower bounds***

Pointer assignment of a pointer array to a target array section in f95 always resulted in the lower bounds of the pointer array set to one. Fortran 2000 allows the specification of the lower bounds 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. Status: early implementation. 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 form 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. Status: later implementation. 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.

***Array reallocation***

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. Status: early implementation. 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.

This feature was added to the standard draft recently. It is possible that a simpler syntax may emerge to replace the move_alloc routine.

***IEEE features***

Support for IEEE floating point arithmetic is a major new feature in Fortran 2000. 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 modules and constants are scheduled for early implementation. 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
```

Constants omitted from the module 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 an ieee_support_* routine returns

false, referencing other routines that depend on support for that feature 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_status_type, and ieee_flag_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.

## 6. I/O

***Asynchronous I/O***

Fortran 2000 contains syntax support for 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. Status: early implementation. 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. Status: early implementation. Example:

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

***FLUSH statement***

File I/O is typically buffered in memory before actual transfers to or from disks take place. The Cray library has a flush subroutine to force a read or write of the memory buffers before they are full. Fortran 2000 provides a portable syntax for this operation as a Fortran statement. Status: early implementation. 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 variable 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**.**

***DECIMAL mode***

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 2000. 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. Status: early implementation. 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***

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 2000 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. The meaning of ‘processor_defined’ is unspecified, and included for backward compatibility. The default rounding mode is processor dependent. Status: later implementation. Example:

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

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

***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_10646 kind type. The encoding keyword values are ‘utf-8’ and ‘default’. The default is ‘default’ and corresponds to ASCII on Cray systems. Status: later implementation. Example:

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

***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. Status: later implementation. Example:

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

***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. Status: later implementation. Example:

```
character(132) :: msg

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

***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 on 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. Status: later implementation. 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
```

## 7. Future

***Standardization process and schedule***

The completion of a new standard for Fortran involves the collaboration of two standards organizations. The ISO committee, named WG5, creates a list of features to be included in a new standard based on input from member countries and comments from the wider community. Once the specification of requirements is completed, it is transferred to the technical committee, named J3, which is charged with writing the document that defines Fortran. The development of a new standard document goes through a series of drafts.

The current Fortran 2000 standard is nearing completion. A WG5 meeting is scheduled for the end of July, 2003, to review the current draft and make recommendations for minor corrections and changes. A subsequent J3 meeting in August will implement those changes in the document and produce a draft for a four-month vote by WG5 member countries. That vote may result in comments and suggestions for minor changes. Once those comments are processed and a final document is produced there will be a final vote in 2004. That vote is either yes or no, with no comments allowed. Once the final vote is completed (and hopefully the standard is approved), it is published by ISO and becomes official. The plan calls for this to be completed by the end of 2004. A copy of the current draft document, as well as the J3 meeting schedule and membership list, is available at the J3 committee web site: j3-fortran.org.

***Future directions***

As the Fortran 2000 standard nears completion, consideration is being given to the shape of future versions of Fortran. If the pattern of f90 and f95 is followed, the next update should be fairly minor, followed by a major revision near the end of the decade. Three significant features have been discussed for future Fortran.

A proposal to add SUBMODULES to Fortran is already well developed. The basic idea of submodules is to separate the interface from the actual definition of module procedures. This would allow for the definitions of the procedures to be in submodules that are in separate files. The main benefits of this structure are a better environment for large-scale projects with many developers, and as a mechanism to avoid compilation cascades common with the current module structure.

Parallel execution dominates large-scale scientific computations, which is the traditional focus of Fortran. The most attractive candidate for parallel structures within

Fortran is the Co-Array Fortran (CAF) model. Specific proposals to standardize Co-Arrays have been made in the past. These may resurface with the addition of CAF equivalent features to C. Wider adoption of Co-Arrays by the user community would enhance the chances of it being standardized.

Fortran has traditionally focused on numeric computing and has strong support for numeric data types. Emerging fields like bioinformatics have a significant computational component that involves non-numeric bit manipulation. The addition of typeless or bit data types to Fortran will be raised for the next standard. This feature would also simplify some procedure interface issues, better handle hexadecimal, octal, and binary constants, and standardize some of the bit manipulation intrinsics such as popcnt.

## Acknowledgments

The author would like to thank the Cray Fortran compiler group for early implementation of many of the Fortran 2000 features as well as substantial 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 2000. Bill can be reached at Cray Inc., 1340 Mendota Heights Road, Mendota Heights, MN 55120, Email: longb@cray.com.