

NAG Fortran Compiler Release 7.0 Release Note

July 25, 2022

1 Introduction

Release 7.0 of the NAG Fortran Compiler is a major update.

Customers upgrading from a previous release of the NAG Fortran Compiler will need a new licence key for this release.

See `KLICENCE.txt` for more information about Kusari Licence Management.

2 Release Overview

This release contains several major new features:

- parallel execution of coarray programs on shared-memory machines;
- half precision floating-point conforming to the IEEE arithmetic standard, including full support for all exceptions and rounding modes;
- submodules, a Fortran 2008 feature for breaking large modules into separately-compilable files;
- teams, a Fortran 2018 coarray feature for structuring parallel execution;
- events, a Fortran 2018 coarray feature for lightweight single-sided synchronisation;
- atomic operations, a Fortran 2018 coarray feature for updating atomic variables without synchronisation.

This release also contains numerous minor enhancements, including:

- other (minor) features from the Fortran 2008 and Fortran 2018 standards;
- miscellaneous improvements to error detection, and improvements to error messages;
- acceptance of additional non-standard Fortran extensions;
- optional auto-skipping NAMELIST input.

3 Compatibility

3.1 Compatibility with Release 6.2

On MacOS the 32-bit ABI mode accessible via `-abi=32` has been removed; consequently only 64-bit compilation is supported and the `-abi=` switch has been removed entirely.

Other than this, Release 7.0 is fully compatible with Release 6.2 except when coarrays are used, or when the `-C=calls` option is used for a subroutine that has an alternate return. Any program that uses these features will need to be recompiled.

3.2 Compatibility with Release 6.1

Programs which use features from HPF (High Performance Fortran), for example the `ILEN` intrinsic function or the `HPF_LIBRARY` module, are no longer supported.

The previously deprecated `-abi=64` option on Linux x86-64 has been withdrawn. This option provided an ABI with 64-bit pointers but 32-bit object sizes and subscript arithmetic, and was only present for compatibility with Release 5.1 and earlier.

With the exception of HPF support and the deprecated option removal, Release 7.0 of the NAG Fortran Compiler is fully compatible with Release 6.1.

3.3 Compatibility with Release 6.0

With the exception of HPF support and the deprecated option removal, Release 7.0 of the NAG Fortran Compiler is compatible with Release 6.0 except that programs that use allocatable arrays of “Parameterised Derived Type” will need to be recompiled (this only affects module variables and dummy arguments).

3.4 Compatibility with Releases 5.3.1, 5.3 and 5.2

With the exception of HPF support and the deprecated option removal, Release 7.0 of the NAG Fortran Compiler is fully compatible with Release 5.3.1. It is also fully compatible with Releases 5.3 and 5.2, except that on Windows, modules or procedures whose names begin with a dollar sign (\$) need to be recompiled.

For a program that uses the new “Parameterised Derived Types” feature, it is strongly recommended that all parts of the program that may allocate, deallocate, initialise or copy a polymorphic variable whose dynamic type might be a parameterised derived type, should be compiled with Release 7.0.

3.5 Compatibility with Release 5.1

Release 7.0 of the NAG Fortran Compiler is compatible with NAGWare f95 Release 5.1 except that:

- programs that use features from HPF are not supported;
- programs or libraries that use the `CLASS` keyword, or which contain types that will be extended, need to be recompiled;
- 64-bit programs and libraries compiled with Release 5.1 on Linux x86-64 (product NPL6A51NA) are binary incompatible, and need to be recompiled.

3.6 Compatibility with Earlier Releases

Except as noted, the NAG Fortran Compiler release 7.0 is compatible with NAGWare f90 Releases 2.1 and 2.2, as well as with all NAGWare f95 Releases from 1.0 to 5.0, except as noted below.

The following incompatibilities were introduced in Release 5.1:

- The value returned by `STAT=`, on an `ALLOCATE` or `DEALLOCATE` statement, may differ from the pre-5.1 value in some cases. For further information see the `F90_STAT` module documentation.
- Programs that used type extension (`EXTENDS` attribute) in 5.0 need to be recompiled.
- Formatted output for IEEE infinities and NaNs is different, and now conforms to Fortran 2003.
- List-directed output of a floating-point zero now uses F format, as required by Fortran 2003, instead of E format.
- An i/o or format error encountered during `NAMelist` input will now skip the erroneous record. This behaviour is the same as all other formatted input operations including list-directed.

4 Updated Fortran 2003 Support

- The IEEE_SUPPORT functions from IEEE_ARITHMETIC and IEEE_EXCEPTIONS are permitted in constant expressions. For example,

```
LOGICAL,PARAMETER :: minexp = &
    MERGE(MIN_EXPONENT(X)-DIGITS(X),MIN_EXPONENT(X),IEEE_SUPPORT_DENORMAL(X))
```

5 New Fortran 2008 Features

- A specification expression may now use a user-defined operation, provided that operation is provided by a specification function. (A specification function must be a pure function that is not a statement function or internal function, and that does not have a dummy procedure argument.) For example, given the interface block

```
INTERFACE OPERATOR(.user.)
    PURE INTEGER FUNCTION userfun(x)
        REAL,INTENT(IN) :: x
    END FUNCTION
END INTERFACE
```

the user-defined operator `.user.` may be used in a specification expression as follows:

```
LOGICAL mask(.user.(3.145))
```

Note that this applies to overloaded intrinsic operators as well as user-defined operators.

- A specification expression may now use the C_LOC and C_FUNLOC functions from the ISO_C_BINDING module. For example, given a TYPE(C_PTR) variable X and another interoperable variable Y with the TARGET attribute,

```
INTEGER workspace(MERGE(10,20,C_ASSOCIATED(X,C_LOC(Y))))
```

is allowed, and will give `workspace` a size of 10 elements if the C pointer X is associated with Y, and 20 elements otherwise.

- An internal procedure may now be a specific procedure in a generic interface. For example,

```
SUBROUTINE example
    INTERFACE gen
        PROCEDURE internal
    END INTERFACE
    CALL gen
CONTAINS
    SUBROUTINE internal
        PRINT *, 'Hello example'
    END SUBROUTINE
END SUBROUTINE
```

Note that the generic `gen` could refer to other procedures if `gen` is also a generic name in the host scoping unit, or a generic name imported by use association. However, `gen` can only refer to the procedure `internal` when invoked from within the subroutine `example`.

- The FINDLOC intrinsic function is now available. It is similar to MAXLOC and MINLOC, but instead of finding the location of the maximum or minimum value of an array, it finds a location that is equal to a specified value; thus it is available for all intrinsic types including COMPLEX and LOGICAL. It has one of the following two forms:

```
FINDLOC (ARRAY, VALUE, DIM, MASK, KIND, BACK )
FINDLOC (ARRAY, VALUE, MASK, KIND, BACK )
```

where

ARRAY is an array of intrinsic type, with rank N ;
VALUE is a scalar of the same type (if **LOGICAL**) or which may be compared with **ARRAY** using the intrinsic operator **==** (or **.EQ.**);
DIM is a scalar **INTEGER** in the range 1 to N ;
MASK (optional) is an array of type **LOGICAL** with the same shape as **ARRAY**
KIND (optional) is a scalar **INTEGER** constant expression that is a valid Integer kind number;
BACK (optional) is a scalar **LOGICAL** value.

The result of the function is type **INTEGER**, or **INTEGER(KIND)** if **KIND** is present.

In the form without **DIM**, the result is a vector of length N , and is the location of the element of **ARRAY** that is equal to **VALUE**; if **MASK** is present, only elements for which the corresponding element of **MASK** are **.TRUE.** are considered. As in **MAXLOC** and **MINLOC**, the location is reported with 1 for the first element in each dimension; if no element equal to **VALUE** is found, the result is zero. If **BACK** is present with the value **.TRUE.**, the element found is the last one (in array element order); otherwise, it is the first one.

In the form with **DIM**, the result has rank $N-1$ (thus scalar if **ARRAY** is a vector), the shape being that of **ARRAY** with dimension **DIM** removed, and each element of the result is the location of the (masked) element in the dimension **DIM** vector that is equal to **VALUE**.

For example, if **ARRAY** is an Integer vector with value [10,20,30,40,50], **FINDLOC(ARRAY,30)** will return the vector [3] and **FINDLOC(ARRAY,7)** will return the vector [0].

- Submodules, together with separate module procedures, provide an additional method of structuring a Fortran program.

A “separate module procedure” is a procedure whose interface is declared in the module specification part, but whose definition may provided either in the module itself, or in a submodule of that module. The interface of a separate module procedure is declared by using the **MODULE** keyword in the prefix of the interface body. For example,

```
INTERFACE
  MODULE RECURSIVE SUBROUTINE sub(x,y)
    REAL,INTENT(INOUT) :: x,y
  END SUBROUTINE
END INTERFACE
```

An important aspect of the interface for a separate module procedure is that, unlike any other interface body, it accesses the module by host association without the need for an **IMPORT** statement. For example,

```
INTEGER,PARAMETER :: wp = SELECTED_REAL_KIND(15)
INTERFACE
  MODULE REAL(wp) FUNCTION f(a,b)
    REAL(wp) a,b
  END FUNCTION
END INTERFACE
```

The eventual definition of the separate module procedure, whether in the module itself or in a submodule, must have exactly the same characteristics, the same names for the dummy arguments, the same name for the result variable (if a function), the same *binding-name* (if it uses **BIND(C)**), and be **RECURSIVE** if and only if the interface is declared so. There are two ways to achieve this:

1. Define the procedure in the normal way, and get all the characteristics right; the compiler will check that you have done so. Note that the definition must also include the **MODULE** keyword in the prefix, just like the definition. For example,

```
...
CONTAINS
  MODULE REAL(wp) FUNCTION f(a,b)
    REAL(wp) a,b
    f = a**2 - b**3
  END FUNCTION
```

2. Alternatively, the entire interface may be accessed in the definition without redeclaring everything by using the `MODULE PROCEDURE` statement in this context. For example,

```
...
CONTAINS
  MODULE PROCEDURE sub
    ! Arguments A and B, their characteristics, and that this is a recursive subroutine,
    ! are all taken from the interface declaration.
    IF (a>b) THEN
      CALL sub(b,-ABS(a))
    ELSE
      a = b**2 - a
    END IF
  END PROCEDURE
```

A submodule has the form (*italic square brackets indicate optionality*):

```
submodule-stmt
  declaration-part
[ CONTAINS
  module-subprogram-part ]
END [ SUBMODULE [ submodule-name ] ]
```

The initial *submodule-stmt* has the form

```
SUBMODULE ( module-name [ : parent-submodule-name ] ) submodule-name
```

where *module-name* is the name of a module with one or more separate module procedures, *parent-submodule-name* (if present) is the name of another submodule of that module, and *submodule-name* is the name of the submodule being defined. The submodules of a module thus form a tree structure, with successive submodules being able to extend others; however, the name of a submodule is unique within that module. This structure is to facilitate creation of internal infrastructure (types, constants, and procedures) that can be used by multiple submodules, without having to put all the infrastructure inside the module itself.

The submodule being defined accesses its parent module or submodule by host association; for entities from the module, this includes access to `PRIVATE` entities. Any local entity it declares in the *declaration-part* will therefore block access to an entity in the host that has the same name.

The entities (variables, types, procedures) declared by the submodule are local to that submodule, with the sole exception of separate module procedures that are declared in the ancestor module and defined in the submodule. No procedure is allowed to have a binding name, again, except in the case of a separate module procedure, where the binding name must be the same as in the interface.

For example,

```
MODULE mymod
  INTERFACE
    MODULE INTEGER FUNCTION next_number() RESULT(r)
    END FUNCTION
    MODULE SUBROUTINE reset()
    END SUBROUTINE
  END INTERFACE
END MODULE
SUBMODULE (mymod) variables
  INTEGER :: next = 1
END SUBMODULE
SUBMODULE (mymod:variables) functions
CONTAINS
  MODULE PROCEDURE next_number
    r = next
    next = next + 1
  END PROCEDURE
```

```

END SUBMODULE
SUBMODULE (mymod:variables) subroutines
CONTAINS
  MODULE SUBROUTINE reset()
    PRINT *, 'Resetting'
    next = 1
  END SUBROUTINE
END SUBMODULE
PROGRAM demo
  USE mymod
  PRINT *, 'Hello', next_number()
  PRINT *, 'Hello again', next_number()
  CALL reset
  PRINT *, 'Hello last', next_number()
END PROGRAM

```

Submodule information for use by other submodules is stored by the NAG Fortran Compiler in files named *module.submodule.sub*, in a format similar to that of *.mod* files. The *-nomod* option, which suppresses creation of *.mod* files, also suppresses creation of *.sub* files.

6 New Fortran 2018 Features

- If a dummy argument of a function that is part of an **OPERATOR** generic has the **VALUE** attribute, it is no longer required to have the **INTENT(IN)** attribute.

For example,

```

INTERFACE OPERATOR(+)
  MODULE PROCEDURE logplus
END INTERFACE
...
PURE LOGICAL FUNCTION logplus(a,b)
  LOGICAL, VALUE :: a,b
  logplus = a.OR.b
END FUNCTION

```

- If the second argument of a subroutine that is part of an **ASSIGNMENT** generic has the **VALUE** attribute, it is no longer required to have the **INTENT(IN)** attribute.

For example,

```

INTERFACE ASSIGNMENT(=)
  MODULE PROCEDURE asgnli
END INTERFACE
...
PURE SUBROUTINE asgnli(a,b)
  LOGICAL, INTENT(OUT) :: a
  INTEGER, VALUE :: b
  DO WHILE (IAND(b, NOT(1)) /= 0)
    b = IEOR(IAND(b, 1), SHIFTR(b, 1))
  END DO
  a = b /= 0 ! Odd number of "1" bits.
END SUBROUTINE

```

- With the *-recursive* or the *-f2018* option, procedures are recursive by default. For example, this subprogram

```

INTEGER FUNCTION factorial(n) RESULT(r)
  IF (n > 1) THEN
    r = n * factorial(n-1)
  ELSE
    r = 1
  END IF
END FUNCTION

```

```

ELSE
    r = 1
END IF
END FUNCTION

```

is valid, just as if it had been explicitly declared with the `RECURSIVE` keyword.

This does not apply to assumed-length character functions (where the result is declared with `CHARACTER(LEN=*)`; these remain prohibited from being declared `RECURSIVE`.

Note that procedures that are `RECURSIVE` by default are excluded from the effects of the `-save` option, exactly as if they were explicitly declared `RECURSIVE`.

- Elemental procedures may now be recursive, whether explicitly declared `RECURSIVE` or by default (when the `-f2018` or `-recursive` options are specified). For example,

```

ELEMENTAL RECURSIVE INTEGER FUNCTION factorial(n) RESULT(r)
    INTEGER,INTENT(IN) :: n
    IF (n>1) THEN
        r = n*factorial(n-1)
    ELSE
        r = 1
    END IF
END FUNCTION

```

may be invoked with

```
PRINT *,factorial( [ 1,2,3,4,5 ] )
```

to print the first five factorials.

- The `NON_RECURSIVE` keyword explicitly declares that a procedure will not be called recursively. For example,

```

NON_RECURSIVE INTEGER FUNCTION factorial(n) RESULT(r)
    r = 1
    DO i=2,n
        r = r*i
    END DO
END FUNCTION

```

In Fortran 2008 and older standards, procedures are non-recursive by default, so this keyword has no effect unless the `-recursive` or `-f2018` is being used.

- The `C_FUNLOC` function from the intrinsic module `ISO_C_BINDING` accepts a non-interoperable procedure argument. The `C_FUNPTR` value produced should not be converted to a C function pointer, but may be converted to a suitable (also non-interoperable) Fortran procedure pointer with the `C_F_PROCPOINTER` subroutine from `ISO_C_BINDING`. For example,

```

USE ISO_C_BINDING
ABSTRACT INTERFACE
    SUBROUTINE my_callback_interface(arg)
        CLASS(*) arg
    END SUBROUTINE
END INTERFACE
TYPE,BIND(C) :: mycallback
    TYPE(C_FUNPTR) :: callback
END TYPE
...
TYPE(mycallback) cb
PROCEDURE(my_callback_interface),EXTERNAL :: sub
cb%callback = C_FUNLOC(sub)
...

```

```

PROCEDURE(my_callback_interface),POINTER :: pp
CALL C_F_PROCPONTER(cb%callback,pp)
CALL pp(...)

```

This functionality may be useful in a mixed-language program when the C_FUNPTR value is being stored in a data structure that is manipulated by C code.

- The C_LOC function from the intrinsic module ISO_C_BINDING accepts an array of non-interoperable type, and the C_F_POINTER function accepts an array pointer of non-interoperable type. The array must still be non-polymorphic and contiguous.

This improves interoperability with mixed-language C and Fortran programming, by letting the program pass an opaque “handle” for a non-interoperable array through a C routine or C data structure, and reconstruct the Fortran array pointer later. This kind of usage was previously only possible for scalars.

- The EQUIVALENCE and COMMON statements, and the BLOCK DATA program unit, are considered to be obsolescent in Fortran 2018, and will be reported as such if the `-f2018` option is used (this is not the default).
- The RECL= specifier in an INQUIRE statement for an unconnected unit or file now assigns the value `-1` to the variable. For a unit or file connected with ACCESS='STREAM', it assigns the value `-2` to the variable. Under previous Fortran standards, the variable became undefined.
- Additional intrinsic atomic subroutines provide a means for multiple images to update atomic variables without synchronisation. These are:

```

      ATOMIC_ADD  (ATOM,VALUE,STAT)
      ATOMIC_AND  (ATOM,VALUE,STAT)
      ATOMIC_CAS  (ATOM,OLD,COMPARE,NEW,STAT)
      ATOMIC_FETCH_ADD  (ATOM,VALUE,OLD,STAT)
      ATOMIC_FETCH_AND  (ATOM,VALUE,OLD,STAT)
      ATOMIC_FETCH_OR   (ATOM,VALUE,OLD,STAT)
      ATOMIC_FETCH_XOR  (ATOM,VALUE,OLD,STAT)
      ATOMIC_OR   (ATOM,VALUE,STAT)
      ATOMIC_XOR  (ATOM,VALUE,STAT)

```

The arguments ATOM, COMPARE, NEW and OLD are all INTEGER(ATOMIC_INT_KIND). The ATOM argument is the one that is updated, and must be a coarray or a coindexed variable. The OLD argument is INTENT(OUT), and receives the value of ATOM before the operation. The STAT argument is optional, and must be a non-coindexed variable of type INTEGER and at least 16 bits in size.

The VALUE argument must be INTEGER but can be of any kind; however, both VALUE and the result of the operation must be representable in INTEGER(ATOMIC_INT_KIND).

The *_ADD operation is addition, the *_AND operation is bitwise and (like IAND), the *_OR operation is bitwise or (like IOR) and the *_XOR operation is bitwise exclusive or (like IEOR).

ATOMIC_CAS is an atomic compare-and-swap operation. If ATOM is equal to COMPARE, it is assigned the value NEW; otherwise, it remains unchanged. In either case, the value before the operation is assigned to OLD. Note that both COMPARE and NEW must also be INTEGER(ATOMIC_INT_KIND).

If the ATOM is a coindexed variable, and is located on a failed image, the operation fails and an error condition is raised; the OLD argument becomes undefined, and if STAT is present, it is assigned the value STAT_FAILED_IMAGE; if STAT is not present, the program is terminated. If no error occurs and STAT is present, it is assigned the value zero.

- The intrinsic function COSHAPE returns a vector of the co-extents of a coarray. It has the form

```
COSHAPE( COARRAY , KIND )
```

where

COARRAY is a coarray of any type; if it is ALLOCATABLE, it must be allocated;
if it is a structure component, the rightmost component must be a coarray component;
KIND (optional) is a scalar INTEGER constant expression that is a valid Integer kind number.

The result of the function is type INTEGER, or INTEGER(KIND) if KIND is present.

The size of the result is equal to the co-rank of COARRAY. For example, if a coarray is declared

REAL x[5,*]

and there are eight images in the current team, COSHAPE(x) will be equal to [5,2].

- The intrinsic elemental function `IMAGE_STATUS` enquires whether another image has stopped or failed. It has the form

`IMAGE_STATUS(IMAGE , TEAM)`

where

`IMAGE` is a positive integer that is a valid image number;

`TEAM` (optional) is a scalar `TEAM_TYPE` value that identifies the current or an ancestor team.

The result of the function is default `INTEGER`, with the value `STAT_FAILED_IMAGE` if the image has failed, `STAT_STOPPED_IMAGE` if the image has stopped, and zero otherwise. The optional `TEAM` argument specifies which team the image number applies to; if it is not specified, the current team is used.

- The intrinsic function `STOPPED_IMAGES` returns an array listing the images that have initiated normal termination (i.e. “stopped”). This function has the form

`STOPPED_IMAGES(TEAM , KIND)`

where

`TEAM` (optional) is a scalar `TEAM_TYPE` value that identifies the current or an ancestor team;

`KIND` (optional) is a scalar `INTEGER` constant expression that is a valid Integer kind number.

The result of the function is a vector of type `INTEGER`, or `INTEGER(KIND)` if `KIND` is present. The elements of the array are in ascending order.

- The type `EVENT_TYPE` in the intrinsic module `ISO_Fortran_ENV`, along with new statements and the intrinsic function `EVENT_QUERY`, provides support for **events**, a lightweight one-sided synchronisation mechanism.

Like type `LOCK_TYPE`, entities of type `EVENT_TYPE` are required to be variables or components, variables of type `EVENT_TYPE` are required to be coarrays, and variables with noncoarray subcomponents of type `LOCK_TYPE` are required to be coarrays. Such variables are called **event variables**. An event variable is not permitted to appear in a variable definition context (i.e. any context where it might be modified), except in an `EVENT POST` or `EVENT WAIT` statement, or as an actual argument where the dummy argument is `INTENT(INOUT)`.

An event variable on an image may have an event “posted” to it by means of the image control statement `EVENT POST`, which has the form

`EVENT POST (event-variable [, sync-stat]...)`

where the optional *sync-stats* may be a single `STAT=stat-variable` specifier and/or a single `ERRMSG=errmsg-variable` specifier; *stat-variable* must be a scalar integer variable that can hold values up to 9999, and *errmsg-variable* must be a scalar default character variable. Posting an event increments the variable’s “outstanding event count” (this count is initially zero). The *event-variable* in this statement will usually be a coindexed variable, as it is rarely useful for an image to post an event to itself.

If `STAT=` appears and the post is successful, zero is assigned to the *stat-variable*. If the image on which the *event-variable* is located has stopped, `STAT_STOPPED_IMAGE` is assigned to the *stat-variable*; if the image has failed, `STAT_FAILED_IMAGE` is assigned, and if any other error occurs, some other positive value is assigned. If `ERRMSG=` appears and any error occurs, an explanatory message is assigned to the *errmsg-variable*. Note that if `STAT=` does not appear and an error occurs, the program will be error-terminated, so having `ERRMSG=` without `STAT=` is useless.

Events are received by the image control statement `EVENT WAIT`, which has the form

`EVENT WAIT (event-variable , event-wait-spec-list)`

where the optional *event-wait-spec-list* is a comma-separated list that may contain a single `STAT=stat-variable` specifier, a single `ERRMSG=errmsg-variable` specifier, and/or a single `UNTIL_COUNT=scalar-integer-expr` specifier. Waiting on an event waits until its “outstanding event count” is greater than or equal to the `UNTIL_COUNT=` specifier value, or greater than zero if `UNTIL_COUNT=` does not appear. If the value specified in `UNTIL_COUNT=` is less than one, it is treated as if it were equal to one.

The *event-variable* in this statement is not permitted to be coindexed; that is, an image can only wait for events posted to its own event variables. There is a partial synchronisation between the waiting image and the images that contributed to the “outstanding event count”; the segment following execution of the **EVENT WAIT** statement follows the segments before the **EVENT POST** statement executions. The synchronisation does not operate in reverse, that is, there is no implication that execution of any segment in a posting image follows any segment in the waiting image.

The **STAT=** and **ERRMSG=** operate similarly to the **EVENT POST** statement, except of course that **STAT_FAILED_IMAGE** and **STAT_STOPPED_IMAGE** are impossible.

Finally, the intrinsic function **EVENT_QUERY** can be used to interrogate an event variable without waiting for it. It has the form

```
EVENT_QUERY ( EVENT, COUNT, STAT )
```

where **EVENT** is an event variable, **COUNT** is an integer variable at least as big as default integer, and the optional **STAT** is an integer variable that can hold values up to 9999. **EVENT** is not permitted to be a coindexed variable; that is, only the image where the event variable is located is permitted to query its count. **COUNT** is assigned the current “outstanding event count” of the event variable. If **STAT** is present, it is assigned the value zero on successful execution, and a positive value if any error occurs. If any error occurs and **STAT** is not present, the program is error-terminated.

Note that event posts in unordered segments might not be included in the value assigned to count; that is, it might take some (communication) time for an event post to reach the variable, and it is only guaranteed to have reached the variable if the images have already synchronised. Use of **EVENT_QUERY** does not by itself imply any synchronisation.

- The type **TEAM_TYPE** in the intrinsic module **ISO_FORTRAN_ENV**, along with new statements and intrinsic procedures, provides support for **teams**, a new method of structuring coarray parallel computation. The basic idea is that while executing inside a team, the coarray environment acts as if only the images in the team exist. This facilitates splitting coarray computations into independent parts, without the hassle of passing around arrays listing the images that are involved in a particular part of the computation.

Unlike **EVENT_TYPE** and **LOCK_TYPE**, functions that return **TEAM_TYPE** are permitted. Furthermore, a variable of type **TEAM_TYPE** is forbidden from being a coarray, and assigning a **TEAM_TYPE** value from another image (e.g. as a component of a derived type assignment) makes the variable undefined; this is because the **TEAM_TYPE** value might contain information specific to a particular image, e.g. routing information to the other images. Variables of type **TEAM_TYPE** are called **team variables**.

Creating teams

The set of all the images in the program is called the **initial team**. At any time, a particular image will be executing in a particular team, the **current team**. A set of subteams of the current team can be created at any time by using the **FORM TEAM** statement, which has the form

```
FORM TEAM ( team-number , team-variable [ , form-team-spec ]... )
```

where *team-number* is a scalar integer expression that evaluates to a positive value, *team-variable* is a team variable, and each *form-team-spec* is **STAT=**, **ERRMSG=**, and **NEW_INDEX=index-value** specifier. At most one of each kind of *form-team-spec* may appear in a **FORM TEAM** statement. All active images of the current team must execute the same **FORM TEAM** statement. If **NEW_INDEX=** appears, *index-value* must be a positive scalar integer (see below). The **STAT=** and **ERRMSG=** specifiers have their usual form and semantics.

The number of subteams that execution of **FORM TEAM** produces is equal to the number of unique *team-number* values in that execution; each unique *team-number* value identifies a subteam in the set, and each image belongs to the subteam whose team number it specified. If **NEW_INDEX=** appears, it specifies the image number that the image will have in its new subteam, and therefore must be in the range 1 to *N*, where *N* is the number of images in that subteam, and must be unique. If **NEW_INDEX=** does not appear, it is processor-dependent what the image number in the new subteam will be.

For example,

```
TYPE(Team_Type) oddeven
myteamnumber = 111*(MOD(THIS_IMAGE(),2) + 1)
FORM TEAM ( myteamnumber, oddeven )
```

will create a set of two subteams, one with team number 111, the other with team number 222. Team 111 will contain the images with even image numbers in the current team, and team 222 will contain the images with odd image numbers in the current team. On each image, the variable `oddeven` identifies the subteam to which that image belongs.

Note that the team numbers are completely arbitrary (being chosen by the program), and only have meaning within that set of subteams, which are called “sibling” teams.

Changing to a subteam

The current team is changed by executing a `CHANGE TEAM` construct, which has the basic form:

```
CHANGE TEAM ( team-value [ , sync-stat-list ] )
      statements
END TEAM [ ( [ sync-stat-list ] ) ]
```

where *team-value* is a value of type `TEAM_TYPE`, and the optional *sync-stat-list* is a comma-separated list containing at most one `STAT=` and `ERRMSG=` specifier; the `STAT=` and `ERRMSG=` specifiers have their usual form and semantics. Execution of the *statements* within the construct are with the current team set to the team identified by *team-value*; this must be a subteam of the current team outside the construct. The setting of the current team remains in effect during procedure calls, so any procedure referenced by the construct will also be executed with the new team current.

Transfer of control out of the construct, e.g. by a `RETURN` or `GOTO` statement is prohibited. The construct may be exited by executing its `END TEAM` statement, or by executing an `EXIT` statement that belongs to the construct; the latter is only possible if the construct is given a name (this is not shown in the form above, but consists of “*construct-name*:” prefix to the `CHANGE TEAM` statement, and and a “*construct-name*” suffix to the `END TEAM` statement).

While executing a `CHANGE TEAM` construct, image selectors operate using the new team’s image indices, the intrinsic functions `NUM_IMAGES` and `THIS_IMAGES` return the data for the new team, and `SYNC ALL` synchronises the new team only.

There is an implicit synchronisation of all images of the new team both on the `CHANGE TEAM` statement, and on the `END TEAM` statement, and all active images must execute the same statement at this time.

Synchronising parent or ancestor teams

While executing within a `CHANGE TEAM` construct, the effects of `SYNC ALL` and `SYNC IMAGES` only apply to images within the current team. For `SYNC ALL` to synchronise the parent team, it would be necessary to first exit the construct. This may be inconvenient when the computation following the synchronisation would be within the team.

For this purpose, the `SYNC TEAM` statement has been added, with the form

```
SYNC TEAM ( team-value [ , sync-stat-list ] )
```

where *team-value* identifies the current team or an ancestor thereof, and *sync-stat-list* is the usual comma-separated list containing at most one `STAT=` specifier and at most one `ERRMSG=` specifier (these have their usual semantics and so are not further described here).

The effect is to synchronise all images in the specified team.

Team-related intrinsic functions

- The intrinsic function `GET_TEAM` returns a value of type `TEAM_TYPE` that identifies a particular team. (This is the only way to get a `TEAM_TYPE` value for the initial team.) The function has the form

```
GET_TEAM( LEVEL )
```

where the optional `LEVEL` argument is a scalar integer value that is equal to one of the named constants `CURRENT_TEAM`, `INITIAL_TEAM` or `PARENT_TEAM`, in the intrinsic module `ISO_FORTTRAN_ENV`. This argument specifies which team the returned `TEAM_TYPE` value should identify; if it is absent, the value for the current team is returned. If the current team is the initial team, the `LEVEL` argument must not be equal to `PARENT_TEAM`, as the initial team has no parent.

- The intrinsic function `TEAM_NUMBER` returns a team number value (that was used in `FORM TEAM` by the executing image). It has the form

```
TEAM_NUMBER( TEAM )
```

where the optional `TEAM` argument specifies which team to return the information for; it must identify the current team or an ancestor team, not a subteam or unrelated team. If `TEAM` is absent, the team number for the current team is returned. The initial team is considered to have a team number of `-1` (except for the initial team, all team numbers are positive values).

Information about sibling and ancestor teams

The intrinsic functions `NUM_IMAGES` and `THIS_IMAGE` normally return information relevant to the current team, but they can return information for an ancestor team by using the optional `TEAM` argument, which takes a `TEAM_TYPE` value that identifies the current team or an ancestor. Similarly, the `NUM_IMAGES` intrinsic can return information for a sibling team by using the optional `TEAM_NUMBER` argument, which takes an integer value that is equal to the team number of the current or a sibling team. (Note that because the executing image is never a member of a sibling team, `THIS_IMAGE` does not accept a `TEAM_NUMBER` argument.) The intrinsic function `NUM_IMAGES` thus has two additional forms as follows:

```
NUM_IMAGES( TEAM )
NUM_IMAGES( TEAM_NUMBER )
```

For `THIS_IMAGE`, the revised forms it may take are as follows:

```
THIS_IMAGE( TEAM )
THIS_IMAGE( COARRAY , TEAM )
THIS_IMAGE( COARRAY, DIM , TEAM )
```

The meanings of the `COARRAY` and `DIM` arguments is unchanged. The optional `TEAM` argument specifies the team for which to return the information.

Establishing coarrays

A coarray is not allowed to be used within a team unless it is **established** in that team or an ancestor thereof. The basic rules for establishment are as follows:

1. a nonallocatable coarray with the `SAVE` attribute (explicit or implicit) is always established;
2. an unallocated coarray (with the `ALLOCATABLE` attribute) is not established;
3. an allocated coarray is established in the team where it was allocated;
4. a dummy coarray is established in the team that executed the procedure call (this may be different from the team where the actual argument is established).

Allocating and deallocating coarrays in teams

If a coarray with the `ALLOCATABLE` attribute is already allocated when a `CHANGE TEAM` statement is executed, it is not allowed to `DEALLOCATE` it within that construct (or within a procedure called from that construct).

If a coarray with the `ALLOCATABLE` attribute is unallocated when a `CHANGE TEAM` statement is executed, it may be allocated (using `ALLOCATE`) within that construct (or within a procedure called from that construct), and may be subsequently deallocated as well. If such a coarray remains allocated when the `END TEAM` statement is executed, it is automatically deallocated at that time.

This means that when using teams, allocatable coarrays may be allocated on some images (within the team), but unallocated on other images (outside the team), or allocated with a different shape or type parameters on other images (also outside the team). However, when executing in a team, the coarray is either unallocated on all images of the team, or allocated with the same type parameters and shape on all images of the team.

Accessing coarrays in sibling teams

Access to a coarray outside the current team, but in a sibling team, is possible using the `TEAM_NUMBER=` specifier in an image selector. This uses the extended syntax for image selectors:

```
[ cosubscript-list [ , image-selector-spec-list ] ]
```

where *cosubscript-list* is the usual list of cosubscripts, and *image-selector-spec-list* contains a `TEAM_NUMBER=team-number` specifier, where *team-number* is the positive integer value that identifies a sibling team. The *image-selector-spec-list* may also contain a `STAT=` specifier (this is described later, under Fault tolerance).

When the `TEAM_NUMBER=` specifier is used the cosubscripts are treated as cosubscripts in the sibling team. Note that access in this way is quite risky, and will typically require synchronisation, possibly of the whole parent team. The coarray in question must be *established* in the parent team.

Accessing coarrays in ancestor teams

Access to a coarray in the parent or more distant ancestor team is possible using the `TEAM=` specifier in an image selector. This uses the extended syntax for image selectors:

```
[ cosubscript-list [ , image-selector-spec-list ] ]
```

where *cosubscript-list* is the usual list of cosubscripts, and *image-selector-spec-list* contains a **TEAM=team-value** specifier, where *team-value* is a value of type **TEAM.TYPE** that identifies the current team or an ancestor. The *image-selector-spec-list* may also contain a **STAT=** specifier (this is described later, under Fault tolerance).

When the **TEAM=** specifier is used the cosubscripts are treated as cosubscripts in the specified ancestor team, and the image thus specified may lie within or outside the current team. If the access is to an image that is outside the current team, care should be taken that the images are appropriately synchronised; such synchronisation cannot be obtained by **SYNC ALL** or **SYNC IMAGES**, as they operate within a team, but may be obtained by **SYNC TEAM** specifying an ancestor team, or by using locks or events. The coarray in question must be *established* in the specified (current or ancestor) team.

Coarray association in **CHANGE TEAM**

It is possible to associate a local *coarray-name* in a **CHANGE TEAM** construct with a named coarray outside the construct, changing the codimension and/or coextents in the process. This acts like a limited kind of argument association; the local *coarray-name* has the type, parameters, rank and array shape of the outside coarray, but does not have the **ALLOCATABLE** attribute. The syntax of the **CHANGE TEAM** construct with one or more such associations is as follows:

```
CHANGE TEAM ( team-value , coarray-association-list [ , sync-stat-list ] )
```

where *coarray-association-list* is a comma-separated list of

```
local-coarray-name [ explicit-coshape-spec ] => outer-coarray-name
```

and *explicit-coshape-spec* is

```
[ [ lower-cobound : ] upper-cobound , ]... [ lower-cobound : ] *
```

(The notation *[something]...* means *something* occurring zero or more times.)

The cobounds expressions are evaluated on execution of the **CHANGE TEAM** statement.

Use of this feature is not encouraged, as it is less powerful and more confusing than argument association.

- Fault tolerance features for coarrays are supported. These consist of the **FAIL IMAGE** statement, the named constant **STAT_FAILED_IMAGE** in the intrinsic module **ISO_FORTRAN_ENV**, the **STAT=** specifier in an image selector, and the intrinsic function **FAILED_IMAGES**.

The form of the **FAIL IMAGE** statement is simply

```
FAIL IMAGE
```

and execution of this statement will cause the current image to “fail”, that is, cease to participate in program execution. This is the only way that an image can fail in NAG Fortran 7.0.

If all images have failed or stopped, program execution will terminate. NAG Fortran will display a warning message if any images have failed.

An image selector has an optional list of specifiers, the revised syntax of an image selector being (where the normal square brackets are literally square brackets, and the italic square brackets indicate optionality):

```
[ cosubscript-list [ , image-selector-spec-list ] ]
```

where *cosubscript-list* is a comma-separated list of cosubscripts, one scalar integer per codimension of the variable, and *image-selector-spec-list* is a comma-separated containing at most one **STAT=stat-variable** specifier, and at most one **TEAM=** or **TEAM_NUMBER=** specifier (these were described earlier). If the coindexed object being accessed lies on a failed image, the value **STAT_FAILED_IMAGE** is assigned to the *stat-variable*, and otherwise the value zero is assigned.

The intrinsic function **FAILED_IMAGES** returns an array of images that are known to have failed (it is possible that an image might fail and no other image realise until it tries to synchronise with it). This function has the form

```
FAILED_IMAGES( TEAM , KIND )
```

where

- TEAM** (optional) is a scalar **TEAM_TYPE** value that identifies the current or an ancestor team;
- KIND** (optional) is a scalar **INTEGER** constant expression that is a valid Integer kind number.

The result of the function is a vector of type **INTEGER**, or **INTEGER(KIND)** if **KIND** is present. The elements of the array are in ascending order.

In order to be able to handle failed images, the following semantics apply:

- writing a value to a variable on a failed image is permitted (but may have no effect);
- reading a value from a variable on a failed image is permitted, but the result is unpredictable;
- execution of a **CHANGE TEAM**, **END TEAM**, **FORM TEAM**, **SYNC ALL**, **SYNC IMAGES** or **SYNC TEAM** statement with a **STAT=** specifier is permitted, and performs the team change, creation, or synchronisation operation on the non-failed images, assigning the value **STAT_FAILED_IMAGE** to the **STAT=** variable.

The latter effect in particular allows the program to form a team of all the non-failed images, and keep executing normally. However, since the data on the failed images is lost (reading the data produces garbage), the program would need to be carefully designed to “checkpoint” its work periodically, so that it can roll the computation state back to a known good value to recover.

The fault tolerance features are in principle intended to permit recovery from hardware failure, with the **FAIL IMAGE** statement allowing some testing of recovery scenarios. NAG Fortran 7.0 does not recover from hardware failure.

- The module **IEEE_ARITHMETIC** has new functions **IEEE_NEXT_DOWN** and **IEEE_NEXT_UP**. These are elemental with a single argument, which must be a **REAL** of an IEEE kind (that is, **IEEE_SUPPORT_DATATYPE** must return **.TRUE.** for that kind of **REAL**). They return the next IEEE value, that does not compare equal to the argument, in the downwards and upwards directions respectively, except that the next down from $-\infty$ is $-\infty$ itself, and the next up from $+\infty$ is $+\text{sym}\{\text{inf}\}$ itself. These functions are superior to the old **IEEE_NEXT_AFTER** function in that they do not signal any exception unless the argument is a signalling NaN (in which case **IEEE_INVALID** is signalled). For example, **IEEE_NEXT_UP(-0.0)** and **IEEE_NEXT_UP(+0.0)** both return the smallest positive subnormal value (provided subnormal values are supported), without signalling **IEEE_UNDERFLOW** (which **IEEE_NEXT_AFTER** does). Similarly, **IEEE_NEXT_UP(HUGE(0.0))** returns $+\infty$ without signalling overflow.
- The module **IEEE_ARITHMETIC** has new named constants **IEEE_NEGATIVE_SUBNORMAL**, **IEEE_POSITIVE_SUBNORMAL**, and the new function **IEEE_SUPPORT_SUBNORMAL**. These are from Fortran 2018, and reflect the change of terminology in the IEEE arithmetic standard in 2008. They are equivalent to the old functions **IEEE_NEGATIVE_DENORMAL**, **IEEE_POSITIVE_DENORMAL** and **IEEE_SUPPORT_DENORMAL**.
- The requirement that the **FLAG.VALUE** argument to **IEEE_GET_FLAG** and **IEEE_SET_FLAG**, the **HALTING** argument to **IEEE_GET_HALTING_MODE** and **IEEE_SET_HALTING_MODE**, and the **GRADUAL** argument to **IEEE_GET_UNDERFLOW_MODE** and **IEEE_SET_UNDERFLOW_MODE**, be default **LOGICAL** has been dropped; any kind of **LOGICAL** is now permitted.

For example,

```
USE F90_KIND
USE IEEE_ARITHMETIC
LOGICAL(byte) flags(SIZE(IEEE_ALL))
CALL IEEE_GET_FLAG(IEEE_ALL,flags)
```

will retrieve the current IEEE flags into an array of one-byte **LOGICAL**s.

7 Other Extensions

- The VAX FORTRAN **OPEN** statement specifier **ACCESS='APPEND'** is now supported, as an aid to porting old programs (it was superseded in 1991 by the **POSITION='APPEND'** specifier). It is equivalent to using **ACCESS='SEQUENTIAL', POSITION='APPEND'**.

For example,

```
OPEN(17,FILE='my.log',ACCESS='APPEND')
```

has the same effect as

```
OPEN(17,FILE='my.log',POSITION='APPEND')
```

- The VAX FORTRAN TYPE statement is now supported. This statement has identical syntax and semantics to the PRINT statement, except that the keyword TYPE is used instead of PRINT. Some forms of this statement where the *format* begins with a name are ambiguous with respect to a derived type definition, and those forms are only treated as a TYPE statement if that name is used or declared earlier in the scoping unit; otherwise, it is treated as a derived type definition.

For example,

```
TYPE *, 'Hello'
```

is equivalent to

```
PRINT *, 'Hello'
```

Processing a source file containing VAX FORTRAN TYPE statements with the enhanced polisher will turn all TYPE statements into PRINT statements. The ordinary polisher will not change any TYPE statements; furthermore, if one of the ambiguous forms is used, the remainder of the file will be incorrectly indented, as the ordinary polisher does not have semantic analysis and therefore assumes the ambiguous form is the beginning of a type definition.

- The “nX” edit descriptor is detected as an individual token when it is followed by another edit descriptor instead of punctuation. This produces better error messages, and enables acceptance of the format with the *—dusty* option. For example,

```
PRINT 1,42
1      FORMAT(7XIO)
```

will print

bbbbbbb42

where *b* represents a blank character. (This is not, and has never been, standard Fortran, but is a common extension.)

- Auto-skipping NAMELIST input is optionally available. The Fortran standard requires that when namelist input is performed, the name after the ampersand in the input record must match the namelist group name (in the READ statement). Normally, the NAG Fortran system raises an i/o error condition if the names do not match, but when auto-skipping namelist input is in effect, instead it skips records until it reaches the end of the file or finds a record that begins with an ampersand and the correct name.

For example, given the program

```
PROGRAM asnl
  INTEGER x,y
  NAMELIST/name/x,y
  READ(*,name)
  PRINT *, 'Result', x,y
END PROGRAM
```

and the input data

```
&wrong x = 999 y = -999 /
&name x = 123 y = 456 /
```

with auto-skipping namelist it will print

```
Result 123 456
```

Auto-skipping namelist is controlled by runtime options. The environment variable NAGFORTRAN_RUNTIME_OPTIONS contains a comma-separated list of runtime options; auto-skipping namelist is enabled by the option **autoskip_namelist** or **log_autoskip_namelist**. The latter option produces an informative message to standard error, displaying where the namelist input occurred, for example:

```
[example.f90, line 5: Looking for namelist group NAME, skipping WRONG]
```

Auto-skipping namelist is not standard Fortran, but it is a very common extension.

8 Half precision floating-point

Half precision (16-bit) floating-point is supported for values and variables of type `REAL` and `COMPLEX`. This floating-point kind conforms to the IEEE arithmetic standard (ISO/IEC/IEEE 60559:2011).

The intrinsic function `SELECTED_REAL_KIND(3)` and intrinsic module function `IEEE_SELECTED_REAL_KIND(3)` return the kind value for half precision. In *—kind=byte* mode, the value will be two; in *—kind=sequential* mode, it will be 16 (this unusual value was chosen to maintain upward compatibility of kind numbers).

The largest finite half-precision value is 65504.0, the smallest normal half-precision value is 0.00006103515625, and the smallest subnormal value is 0.000000059604644775390625.

Scalar half-precision operations are evaluated in single precision, and only rounded to half precision when assigned to a variable or passed as an actual argument to a non-intrinsic or non-mathematical procedure (e.g. `SQRT` is mathematical, but `NEAREST` is not). This can be controlled by the *—round_hreal* option; if used, all half-precision operations will be rounded to half precision, both at compile time and run time.

Because of all the conversions needed, half precision is slower than single precision; its sole benefit is halving the memory and file storage requirements.

9 Additional error checking

- The *—C=intouv* now detects integer overflow in the intrinsic function `INT` when applied to `COMPLEX` values.

For example, in

```
CALL sub((128,0))
...
SUBROUTINE SUB(c)
  USE iso_fortran_env
  COMPLEX,INTENT(IN) :: c
  PRINT *,INT(c,int8)
  ...
```

the conversion of the `COMPLEX` variable `C` to 8-bit `INTEGER` is out of range and will be reported with a message similar to

```
Runtime Error: b6a.f90, line 10: Overflow converting 128.0 to INTEGER(int8)
```

- Common block storage sequence inconsistencies between program units in the same file are now detected, even when the total size of the common block is the same in both places. For example, in

```
SUBROUTINE s1
  COMMON/c/x,y,z
  REAL x(5)
  INTEGER y
  CHARACTER z(4)
END SUBROUTINE
SUBROUTINE s2
  COMMON/c/xx,zz,yy
  REAL xx(5)
  INTEGER yy
  CHARACTER zz(4)
END SUBROUTINE
```

a message similar to the following will be produced (the message has been wrapped for this document):

```
Error: test.f90: Definitions of COMMON block /C/ in program units S1 and S2 differ
at storage unit 6,
  variable Y in S1 provides a numeric storage unit,
  but variable ZZ in S2 provides a character storage unit
```

This error may be downgraded to a warning with the *—dusty* option.

10 Miscellaneous enhancements

- An empty source file, or a source file containing only comments, may now be compiled. A warning is produced.
- The `-f2018` option requests Fortran 2018 semantics. This means suppressing extension messages for the use of Fortran 2018 features, producing Obsolescent/Deleted messages for the use of features that Fortran 2018 has made obsolescent or deleted, and making procedures `RECURSIVE` by default (the same as the `-recursive` option).
- The runtime option **`suppress_underflow_warning`** suppresses the usual warning on program termination if the floating-point underflow flag is set. The runtime option **`underflow_warning`** requests production of the usual warning on program termination if the floating-point underflow flag is set. These runtime options are set in the environment variable `NAGFORTRAN.RUNTIME.OPTIONS`. They override the appearance or non-appearance of the `-no_underflow_warning` option at compilation time.
- There is a new polish option `-dcolon_column=N` to control the alignment of double colon in declaration and specification statements. Supplying `-dcolon_column=0` is synonymous with option `-nodcolon_column` and performs no alignment, which is the default.

For example, with the code

```
Subroutine s(x, y)
  Integer, Intent (In) :: x
  Real, Intent (Out) :: y(1)
  Logical :: l1, l2, l3, l4, l5, l6, l7, l8, l9, l10, l11, l12, l13

  y = (/ Real :: x /)
End Subroutine
```

the `-dcolon_column=30` option produces

```
Subroutine s(x, y)
  Integer, Intent (In)      :: x
  Real, Intent (Out)       :: y(1)
  Logical                   :: l1, l2, l3, l4, l5, l6, l7, l8, l9, l10, l11, &
                           l12, l13

  y = (/ Real :: x /)
End Subroutine
```

- The precision unifier ("`nagfor =unifyprecision`") now allows additional control over the level of conversion it applies to floating-point and complex entities. To modify only those entities that don't already have a kind specifier, use `-pu_floats=Default_Kinds`.

For example, with the code

```
Use working_precision, Only: rp
Real (Kind=rp) :: x_single = real(42, kind=rp)
Double Precision :: x_double = 42.0D0
```

applying the tool and using the `-pu_floats=Default_Kinds` option produces

```
Use working_precision, Only: rp
Use working_precision, Only: wp
Real (Kind=rp) :: x_single = real(42, kind=rp)
Real (Kind=wp) :: x_double = 42.0E0_wp
```