NAG Fortran Compiler Release 6.2 Release Note

March 15, 2019

1 Introduction

Release 6.2 of the NAG Fortran Compiler is a minor 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.

1.1 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The previously deprecated *−abi=64* option on Linux x86-64 has been withdrawn.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 6.2 of the NAG Fortran Compiler is fully compatible with Release 6.1.

1.2 Compatibility with Release 6.0

With the exception of HPF support and the deprecated option removal, Release 6.2 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).

1.3 Compatibility with Releases 5.3.1, 5.3 and 5.2

With the exception of HPF support and the deprecated option removal, Release 6.2 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 6.2.

1.4 Compatibility with Release 5.1

Release 6.2 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.

1.5 Compatibility with Earlier Releases

Except as noted, the NAG Fortran Compiler release 6.2 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 encounted during `NAMELIST` input will now skip the erroneous record. This behaviour is the same as all other formatted input operations including list-directed.

2 New Features Summary

With the addition of defined input/output, and recursive specification functions, Fortran 2003 is fully supported by Release 6.2. The other major new feature is single image coarray support (Fortran 2008).

Several other new features have been added from Fortran 2008, and some from the draft Fortran 2018 standard. Some other common (obsolete) extensions have been added.

This release also contains additional error checking functionality and other minor enhancements.

3 New Fortran 2003 Features

- A function that is used in a specification expression is now permitted to be recursive (defined with the `RECURSIVE` attribute). For example

```
PURE INTEGER RECURSIVE 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
```

  can now be used in a specification expression. Note that a specification function must not invoke the procedure that invoked it.

- Defined input/output (for derived types) is now available. For details see the Fortran 2003 language documentation, or any good textbook.

4 New Fortran 2008 Features

- Coarray syntax and semantics are accepted (and checked for correctness). Coarrays are part of an SPMD (Single Program Multiple Data) programming model, where multiple copies of a program, called "images", are executed in parallel. In this release of the NAG Fortran Compiler, execution is limited to a single image; that is, without parallel execution.

  For further details on coarrays, see the Fortran 2008 language documentation, or any good textbook.

- The pure inquiry function C_SIZEOF has been added to the intrinsic module ISO_C_BINDING. This function takes one argument (`X`) which must be interoperable, and returns the storage size in bytes, like the C `sizeof` operator. If `X` is an array, the result is the size of the whole array, not just a single element. Note that `X` cannot be an assumed-size array.

- The name of an external procedure with a binding label is now considered to be a local identifier only, and not a global identifier. That means that code like the following is now standard-conforming:

```
SUBROUTINE sub() BIND(C,NAME='one')
  PRINT *,'one'
END SUBROUTINE
SUBROUTINE sub() BIND(C,NAME='two')
  PRINT *,'two'
END SUBROUTINE
PROGRAM test
  INTERFACE
    SUBROUTINE one() BIND(C)
    END SUBROUTINE
    SUBROUTINE two() BIND(C)
    END SUBROUTINE
  END INTERFACE
  CALL one
  CALL two
END PROGRAM
```

- An internal procedure is permitted to have the `BIND(C)` attribute, as long as it does not have a `NAME=` specifier. Such a procedure is interoperable with C, but does not have a binding label (as if it were specified with `NAME=''`).

- The intrinsic functions `MAXLOC` and `MINLOC` now have an additional optional argument `BACK` following the `KIND` argument. It is scalar and of type Logical; if present with the value `.True.`, if there is more than one element that has the maximum value (for `MAXLOC`) or minimum value (for `MINLOC`), the array element index returned is for the last element with that value rather than the first.

  For example, the value of

```
MAXLOC( [ 5,1,5 ], BACK=.TRUE.)
```

  is the array `[ 3 ]`, rather than `[ 1 ]`.

- An `ALLOCATE` statement with the `SOURCE=` clause is permitted to have more than one *allocation*. The *source-expr* is assigned to every variable allocated in the statement. For example,

```
PROGRAM multi_alloc
  INTEGER,ALLOCATABLE :: x(:),y(:,:)
  ALLOCATE(x(3),y(2,4),SOURCE=42)
  PRINT *,x,y
END PROGRAM
```

  will print the value "42" eleven times (the three elements of `x` and the eight elements of `y`). If the *source-expr* is an array, every *allocation* needs to have the same shape.

- The restrictions that formerly applied to *data-implied-do* loop limits, and to subscripts in a *data-i-do-object*, have been lifted. These restrictions did not permit use of intrinsic functions that were permitted in other constant expressions. For example,

```
DATA (x(i),i=1,SIZE(x))/1,2,3,4,5,6,7,8,9,10/
```

  is now permitted.

- A dummy argument with the `VALUE` attribute is permitted to be an array, and is permitted to be of type `CHARACTER` with length non-constant and/or not equal to one. (It is still not permitted to have the `ALLOCATABLE` or `POINTER` attributes, and is not permitted to be a coarray.)

  The effect is that a copy is made of the actual argument, and the dummy argument is associated with the copy; any changes to the dummy argument do not affect the actual argument. For example,

```
PROGRAM value_example_2008
  INTEGER :: a(3) = [ 1,2,3 ]
  CALL s('Hello?',a)
  PRINT '(7X,3I6)',a
CONTAINS
  SUBROUTINE s(string,j)
    CHARACTER(*),VALUE :: string
    INTEGER,VALUE :: j(:)
    string(LEN(string):) = '!'
    j = j + 1
    PRINT '(7X,A,3I6)',string,j
  END SUBROUTINE
END PROGRAM
```

will produce the output

```
Hello!     2     3     4
     1     2     3
```

5 New Draft Fortran 2018 Features

- The expression in an `ERROR STOP` or `STOP` statement can be non-constant. It is still required to be default Integer or default Character.
- The `ERROR STOP` and `STOP` statements now have an optional `QUIET=` specifier, which is preceded by a comma following the optional stop-code. This takes a Logical expression; if it is true at runtime then the `STOP` (or `ERROR STOP`) does not output any message, and information about any IEEE exceptions that are signalling will be suppressed. For example,

  ```
  STOP 13, QUIET = .True.
  ```

  will not display the usual 'STOP: 13', but simply do normal termination, with a process exit status of 13. Note that this means that the following two statements are equivalent:

  ```
  STOP, QUIET=.True.
  STOP 'message not output', QUIET=.TRUE.
  ```

- The intrinsic subroutine `MOVE_ALLOC` now has optional `STAT` and `ERRMSG` arguments. The `STAT` argument must be of type Integer, with a decimal range of at least four (i.e. not an 8-bit integer); it is assigned the value zero if the subroutine executes successfully, and a nonzero value otherwise. The `ERRMSG` argument must be of type Character with default kind. If `STAT` is present and assigned a nonzero value, `ERRMSG` will be assigned an explanatory message (if it is present); otherwise, `ERRMSG` will retain its previous value (if any).

  For example,

  ```
  INTEGER,ALLOCATABLE :: x(:),y(:)
  INTEGER istat
  CHARACTER(80) emsg
  ...
  CALL MOVE_ALLOC(x,y,istat,emsg)
  IF (istat/=0) THEN
    PRINT *,'Unexpected error in MOVE_ALLOC: ',TRIM(emsg)
  ```

  The purpose of these arguments is to catch errors in multiple image coarray allocation/deallocation, such as `STAT_STOPPED_IMAGE` and `STAT_FAILED_IMAGE`. As this release of the NAG Fortran Compiler only supports single image execution, `STAT` will always be assigned zero, and `ERRMSG` will never be assigned anything.

6 Other Extensions

- Obsolete extension: D (debug) lines in Fixed Source Form.
  A line with the letter 'D' (or 'd') in column one is a D line. If the *−d_lines* option is used, this will be treated as a normal Fortran line, as if the D were a space. Otherwise, it will be treated as a comment line, as if the D were a C.

  For example, in

```
      SUBROUTINE TEST(N)
      INTEGER N
D     PRINT *,'TESTING N'
      ...
```

  the `PRINT` statement will be compiled only if *−d_lines* is used.

  Note that if the initial line of a statement is a D line, any continuation lines it may have must also be D lines. Similarly, if the initial line of a statement is not a D line, any continuation lines must not be D lines.

  A D line can use TAB format, with the TAB expanding to one less space as the letter D already accounts for a space.

- Obsolete ("dusty deck") extension: named `COMMON` blocks with different sizes.
  With the *−dusty* option, named `COMMON` blocks with different sizes (in the same file) are permitted. The effect is that all the copies of that `COMMON` block are increased to the maximum size. Note that if a `COMMON` block in a separately compiled file has a different size, the results are indeterminate, especially if the `COMMON` block is initialized in a `BLOCK DATA` subprogram where it has a smaller size.

  Use of this feature is **strongly** disrecommended. Also, a `COMMON` block that is OpenMP `THREADPRIVATE` is still required to be the same size everywhere, even with the *−dusty* option.

- When overriding a byte-length specifier for non-CHARACTER type, the syntax "`*(`*integer*`)`" is accepted; previously, only "`*`*integer*" was accepted. For example,

```
REAL X*4, Y*(8)
```

  Note that byte length specifiers are an obsolete extension. Kind type parameters should be used instead.

7 Additional error checking

- The runtime option **show_dangling** enables tracing of dangling pointers, for code compiled with *−C=dangling*.

  Runtime options are controlled by the `NAGFORTRAN_RUNTIME_OPTIONS` environment variable. If **show_dangling** is specified, messages will be produced on the runtime error file when a dangling pointer is created, reassociated with something else, nullified, or ceases to exist. For example,

```
[file.f90, line 20: Dangling pointer P detected (number 1), associated at file.f90, line 18]
[file.f90, line 7: Dangling pointer P (number 1) has been reassociated]
[file.f90, line 9: Dangling pointer Q (number 2) has been nullified]
[file.f90, line 21: Dangling pointer R (number 3) no longer exists]
```

  The dangling pointer number is incremented every time a dangling pointer is detected. If an array with dangling pointer components ceases to exist, a message will be produced for each dangling pointer component of each element; however, the element subscripts will not be shown, instead '`(...)`' will be produced to indicate that it is an array element, e.g.

```
[file.f90, line 44: Dangling pointer X(...)%A (number 8) no longer exists]
```

- A "Questionable" warning message is now produced if a DO index variable (in a DO statement, or an i/o-implied-do) is liable to accidental modification. That is, if it has the `POINTER` or `TARGET` attribute, is in a `COMMON` block or `EQUIVALENCE`, or is a non-local variable being accessed by use or host association.

- A warning message is now produced if the result of an intrinsic function or operation underflows to zero. Previously this warning only appeared for exponentiation.
- The *−C=do* option has been extended to check for modifying an active DO index variable via host association. With this option, the example

```
Program example
  Do i=1,10
    Call inner
    Print *,i
  End Do
Contains
  Subroutine inner
    i = 999
  End Subroutine
End Program
```

  will produce the output

```
Runtime Error: example.f90, line 8: Assignment to active DO index I
Program terminated by fatal error
```

- Use in a `READ` statement of a character constant format or a `FORMAT` statement that has a character string edit descriptor is now detected at compile time. For example,

```
   Read 1,n
1  Format("Oops",I10)
```

  will produce an error at compile time.

- When an input/output unit number is a 64-bit integer, having an invalid value such as being negative or greater than `HUGE(0_int32)` is now more reliably detected. Negative values will now always raise `IOERR_BAD_UNIT`, and positive values that are out of range will be treated as a nonexistent unit (thus raising `IOERR_BAD_UNIT` except for `CLOSE`, `INQUIRE`, and `WAIT` with no `ID=` specifier).
- The new option *−C=alias* enables checking for violation of the dummy argument aliasing rules; specifically, assignment to a scalar dummy argument is checked to determine whether it affects another scalar dummy argument; if so, a runtime error is produced. For example, if the file `sub.f90` contains the following program,

```
Subroutine sub(a,b)
  a = 1
  b = 2
End Subroutine
Program test
  Call sub(x,x)
  Print *,x
End Program
```

  and it is compiled with *−C=alias*, the output

```
Runtime Error: sub.f90, line 2: Assignment to A affects dummy argument B
Program terminated by fatal error
```

  will be produced at runtime.

  This option is included in *−C=all*, but not included in the plain *−C* option, as when it is extended to cover other aliasing cases (in particular, arrays), it may have a large performance impact.

- Assignment of an out-of-range value (e.g. a large magnitude double precision value to a single precision variable) now produces a warning at compile time.
- An actual argument that is not simply contiguous when the corresponding dummy argument is a CONTIGUOUS pointer now produces an error message.
- Better error messages are now produced when a symbol has been accessed by USE association before an attempt to IMPORT it.

8 Miscellaneous enhancements

- The interface generator ("`nagfor =interfaces`") now outputs `USE` statements in a standardised form, with entities required by each resulting interface block imported using the `ONLY` clause and with unneeded module imports omitted. For example, given the code

```
Subroutine s(n, x)
  Use iso_fortran_env
  Integer (int32) :: n
  Real (real64) :: x(n)
  Print *, compiler_options()
  x = 42._real64
End Subroutine
```

  an interface module of the form

```
Module interfaces
  ! Interface module generated on ...
  Interface
    Subroutine s(n, x)
      Use, Intrinsic :: iso_fortran_env, Only: int32, real64
      Integer (int32) :: n
      Real (real64) :: x(n)
    End Subroutine
  End Interface
End Module
```

  is created.

- Callgraph output (from "`nagfor =callgraph`") now indicates when an actual procedure for a non-optional dummy could not be found in the input source.

- The polisher ("nagfor =polish") and dependency analyser ("nagfor =depend") now accept the *−maxcontin=* option, and so can be used on programs with more than 255 continuation lines.

- The default polish setting *−name_scopes=Insert* has been changed to *−name_scopes=Keywords*.

- There is a new polish option *−dcolon_in_decls=X* which controls the optional double colon in declaration and specification statements. *X* can be one of: 'Asis', to make no change, 'Insert', to insert an optional double colon when it is missing, and 'Remove', to remove the double colon when it is optional.

  For example, with the code

```
Real x
Real :: y
Real, Save :: z
```

  the *−dcolon_in_decls=Insert* option produces

```
Real :: x
Real :: y
Real, Save :: z
```

  and the *−dcolon_in_decls=Remove* option produces

```
Real x
Real y
Real, Save :: z
```

- The new enhanced polisher tool provides advanced options for polishing files that are compilable. These options are:

–add_arg_keywords

Add keywords to actual arguments in references to user-defined procedures with an explicit interface and at least two dummy arguments, and in references to intrinsic procedures and intrinsic module procedures with at least three dummy arguments (except for `MAX` and `MIN`, where it is at least three actual arguments).

Keywords are not added to arguments that precede a label argument. The order of the arguments is unchanged.

–add_arg_keywords=*proc_class_list*

Add keywords to actual arguments in procedure references, when the procedure has an explicit interface, for the classes of procedure listed in *proc_class_list*, which is a comma-separated list that may contain the following suboptions:

all	(all classes of procedure),
bound	(object-bound and type-bound procedures),
dummy	(dummy procedures),
external	(external procedures),
internal	(internal procedures),
intrinsic	(intrinsic procedures and intrinsic module procedures),
module	(non-intrinsic module procedures),
user	(procedures other than intrinsic procedures and intrinsic module procedures).

Keywords are not added to arguments that precede a label argument. The order of the arguments is unchanged. Procedure pointer components are also known as "object-bound procedures", and thus included in *–add_arg_keywords=bound*; named procedure pointers are treated as external procedures and thus included in *–add_arg_keywords=external*.

A suboption name may be followed by a single nonzero digit (e.g. "intrinsic3"); this specifies that for procedures covered by that suboption, keywords are only to be added if the procedure has at least that many dummy arguments. For type-bound and object-bound procedures, the passed-object dummy argument does not count towards the limit (as it never appears in the argument list). The intrinsic `MAX` and `MIN` functions use the number of actual arguments instead.

A suboption name followed by a digit may be further followed by the letter 'a' (e.g. "intrinsic3a"; this specifies that the argument limit applies to the number of actual arguments in a reference to the procedure, not the number of dummy arguments (the number of actual arguments will be less than the number of dummy arguments when an optional argument is omitted).

Note that suboptions are parsed from left to right, and later suboptions override earlier ones.

–intrinsic_case=*analogy*

Specifies whether the case of an intrinsic procedure name should be the same as other names (**as_names**), or the same as language keywords (**as_keywords**).

–remove_intrinsic_stmts

Specifies that intrinsic procedure names that were not passed as actual arguments should be removed from `INTRINSIC` statements, and that if all the names in an `INTRINSIC` statement are removed in this way, the `INTRINSIC` statement itself should be removed. Any comments associated with the `INTRINSIC` statement will remain.

- The precision unifier now has more accurate analysis of EQUIVALENCE statements, improving the usefulness of its warning messages.

- Assigning a scalar value of intrinsic (non-Character) type to a contiguous array of the same type and kind has improved performance in many cases.

- The new *–kind=unique* option makes all intrinsic kind type parameter values unique across the intrinsic types (except that the kinds of Real and Complex are the same). This means that using the kind type parameter of one intrinsic type to declare the kind of a different intrinsic type will produce a compile-time error. For example,

```
Integer,Parameter :: rkind = Selected_Real_Kind(15)
Integer,Parameter :: ikind = Selected_Int_Kind(9)
Real(rkind) :: x
Integer(rkind) :: n
```

  will produce an error for the '`Integer(rkind)`' statement.

- The new *–Warn=class* option causes additional warning messages to be produced according to *class* as follows:

allocation warn if an intrinsic assignment might cause allocation of the variable (or a subcomponent thereof) being assigned to;

constant_coindexing
warn if all the cosubscripts in an *image-selector* are constant;

reallocation warn if an intrinsic assignment might cause reallocation of an already-allocated variable (or a subcomponent thereof) being assigned to;

subnormal warn if an intrinsic function or operation with normal operands produces a subnormal result (reduced precision, less than `TINY(...)`).

Reallocation only occurs when the shape of an array, the value of a deferred type parameter, or the dynamic type (if polymorphic), differs between the variable (or subcomponent) and the expression (or the corresponding subcomponent). Allocation can occur also when the variable (or subcomponent) is not allocated prior to execution of the assignment (except for broadcast assignment). Note that *−Warn=allocation* thus subsumes *−Warn=reallocation*.

For example, in

```
Subroutine s(x,y)
  Integer,Allocatable :: x,y(:)
  x = 123
  y = [ 1,2,3 ]
End Subroutine
```

both assignments might cause allocation (and thus produce warnings with *−Warn=allocation*), but only the assignment to `y` can cause reallocation (and thus produce a warning with *−Warn=reallocation*).

- Conversion errors in `PARAMETER` declarations now produce more accurate line number information. Previously these errors were reported on the use of the parameter, rather than its declaration. Similarly, errors in initialisation of a variable could sometimes be reported with no line information; these are now reported at the line of the initialisation. For example, if a source file contains the following code,

```
Program bad
  Use Iso_Fortran_Env
  Integer(int8),Parameter :: x = 1000
  Print *,x
End Program
```

the conversion error for `X` will be reported at line 3 rather than at line 4.

Furthermore, if an overflow arises converting a floating-point constant value to an integer in such a context, this is now considered to be an error rather than a warning. (The previous behaviour can still be obtained with the *−dusty* option.) For example, if a source file contains

```
Subroutine s
  Integer :: x = 1d300
  x = x + 1
  Print *,x
End Subroutine
```

the conversion overflow will be reported as an error at line 2.

- The fpp preprocessor now allows unrecognised directives in unprocessed sections. For example, if the file "test.ff90" contains the text

```
Program test
#if 0
#unknown directive
#endif
  Print *,'ok'
End Program
```

it will now compile successfully instead of producing an error message, and print 'ok' when executed.

- The full compiler manual is now available in HTML. It is located in `html/manual/compiler.html`.