1 Introduction

This document describes those parts of the Fortran 2008 language which are not in Fortran 2003, and which are supported by the latest release of the NAG Fortran Compiler.

The compiler release in which a feature was made available is indicated by square brackets; for example, a feature marked as ‘[5.3]’ was first available in Release 5.3.

2 Overview of Fortran 2008

The new features of Fortran 2008 that are supported by the NAG Fortran Compiler can be grouped as follows:

- data declaration;
- data usage and computation;
- execution control;
- intrinsic procedures and modules;
- input/output extensions;
- programs and procedures.

3 Data declaration [mostly 6.0]

- The maximum rank of an array has been increased from 7 to 15. For example,

  REAL array(2,2,2,2,2,2,2,2,2,2,2,2,2,2,2)

  declares a 15-dimensional array.

- [3.0] 64-bit integer support is required, that is, the result of SELECTED_INT_KIND(18) is a valid integer kind number.

- A named constant (PARAMETER) that is an array can assume its shape from its defining expression; this is called an implied-shape array. The syntax is that the upper bound of every dimension must be an asterisk, for example

  REAL,PARAMETER :: idmat3(*,*) = Reshape( [ 1,0,0,0,1,0,0,0,1 ], [ 3,3 ] )
  REAL,PARAMETER :: yeardata(2000:*) = [ 1,2,3,4,5,6,7,8,9 ]

  declares idmat3 to have the bounds (1:3,1:3), and yeardata to have the bounds (2000:2008).

- The TYPE keyword can be used to declare entities of intrinsic type, simply by putting the intrinsic type-spec within the parentheses. For example,

  TYPE(REAL) x
  TYPE(COMPLEX(KIND(0d0))) y
  TYPE(CHARACTER(LEN=80)) z

  is completely equivalent, apart from being more confusing, to

  REAL x
  COMPLEX(KIND(0d0)) y
  CHARACTER(LEN=80) z
• As a consequence of the preceding extension, it is no longer permitted to define a derived type that has the name DOUBLEPRECISION.

• [5.3] A type-bound procedure declaration statement may now declare multiple type-bound procedures. For example, instead of

```fortran
PROCEDURE,NOPASS :: a
PROCEDURE,NOPASS :: b=>x
PROCEDURE,NOPASS :: c
```

the single statement

```fortran
PROCEDURE,NOPASS :: a, b=>x, c
```

will suffice.

4 Data usage and computation [mostly 5.3]

• In a structure constructor, the value for an allocatable component may be omitted: this has the same effect as specifying NULL().

• [6.0] When allocating an array with the ALLOCATE statement, if SOURCE= or MOLD= is present and its expression is an array, the array can take its shape directly from the expression. This is a lot more concise than using SIZE or UBOUND, especially for a multi-dimensional array.

For example,

```fortran
SUBROUTINE s(x,mask)
REAL x(:,:,:)
LOGICAL mask(:,:,:)
REAL,ALLOCATABLE :: y(:,:,:)
ALLOCATE(y,MOLD=x)
WHERE (mask)
y = 1/x
ELSEWHERE
  y = HUGE(x)
END WHERE
! ...
END SUBROUTINE
```

• In an ALLOCATE statement for one or more variables, the MOLD= clause can be used to give the variable(s) the dynamic type and type parameters (and optionally shape) of an expression. The expression in MOLD= must be type-compatible with each allocate-object, and if the expression is a variable (e.g. MOLD=X), the variable need not be defined. Note that the MOLD= clause may appear even if the type, type parameters and shape of the variable(s) being allocated are not mutable. For example,

```fortran
CLASS(*),POINTER :: a,b,c
ALLOCATE(a,b,c,MOLD=125)
```

will allocate the unlimited polymorphic pointers A, B and C to be of type Integer (with default kind); unlike SOURCE=, the values of A, B and C will be undefined.

• [5.3.1] Assignment to a polymorphic allocatable variable is permitted. If the variable has different dynamic type or type parameters, or if an array, a different shape, it is first deallocated. If it is unallocated (or is deallocated by step 1), it is then allocated to have the correct type and shape. It is then assigned the value of the expression. Note that the operation of this feature is similar to the way that ALLOCATE(variable,SOURCE=expr) works. For example, given

```fortran
CLASS(*),ALLOCATABLE :: x
```
execution of the assignment statement

\[ x = 43 \]

will result in \( X \) having dynamic type Integer (with default kind) and value 43, regardless of whether \( X \) was previously unallocated or allocated with any other type (or kind).

5 Execution control [mostly 6.0]

- [5.3] The BLOCK construct allows declarations of entities within executable code. For example,

```
Do i=1,n
  Block
    Real tmp
    tmp = a(i)**3
    If (tmp>b(i)) b(i) = tmp
  End Block
End Do
```

Here the variable \( tmp \) has its scope limited to the BLOCK construct, so will not affect anything outside it. This is particularly useful when including code by INCLUDE or by macro preprocessing.

All declarations are allowed within a BLOCK construct except for COMMON, EQUIVALENCE, IMPLICIT, INTENT, NAMELIST, OPTIONAL and VALUE; also, statement function definitions are not permitted.

BLOCK constructs may be nested; like other constructs, branches into a BLOCK construct from outside are not permitted. A branch out of a BLOCK construct “completes” execution of the construct.

Entities within a BLOCK construct that do not have the SAVE attribute (including implicitly via initialisation), will cease to exist when execution of the construct is completed. For example, an allocated ALLOCATABLE variable will be automatically deallocated, and a variable with a FINAL procedure will be finalised.

- The EXIT statement is no longer restricted to exiting from a DO construct; it can now be used to jump to the end of a named ASSOCIATE, BLOCK, IF, SELECT CASE or SELECT TYPE construct (i.e. any named construct except FORALL and WHERE). Note that an EXIT statement with no construct-name still exits from the innermost DO construct, disregarding any other named constructs it might be within.

- In a STOP statement, the stop-code may be any scalar constant expression of type integer or default character. (In the NAG Fortran Compiler this also applies to the PAUSE statement, but that statement is no longer standard Fortran.) Additionally, the STOP statement with an integer stop-code now returns that value as the process exit status (on most operating systems there are limits on the value that can be returned, so for the NAG Fortran Compiler this returns only the lower eight bits of the value).

- The ERROR STOP statement has been added. This is similar to the STOP statement, but causes error termination rather than normal termination. The syntax is identical to that of the STOP statement apart from the extra keyword ‘ERROR’ at the beginning. Also, the default process exit status is zero for normal termination, and non-zero for error termination.

For example,

```
IF (x<=0) ERROR STOP 'x must be positive'
```

6 Intrinsic procedures and modules

6.1 Additional mathematical intrinsic functions [mostly 5.3.1]

- The elemental intrinsic functions ACOSH, ASINH and ATANH compute the inverse hyperbolic cosine, sine or tangent respectively. There is a single argument \( X \), which may be of type Real or Complex; the result of the function has the same type and kind. When the argument is Complex, the imaginary part is expressed in radians and lies in the range \( 0 \leq \text{im} \leq \pi \) for the ACOSH function, and \( -\pi/2 \leq \text{im} \leq \pi/2 \) for the ASINH and ATANH functions.

For example, ACOSH(1.543081), ASINH(1.175201) and ATANH(0.7615942) are all approximately equal to 1.0.
The elemental intrinsic function **HYPOT** computes the “Euclidean distance function” (square root of the sum of squares) of its arguments \(X\) and \(Y\) without overflow or underflow for very large or small \(X\) or \(Y\) (unless the result itself overflows or underflows). The arguments must be of type Real with the same kind, and the result is of type Real with that kind. Note that **HYPOT**(\(X, Y\)) is semantically and numerically equal to **ABS**(\(\text{CMPLX}(X, Y, \text{KIND}(X))\)).

For example, **HYPOT**(3e30, 4e30) is approximately equal to 5e30.

The array reduction intrinsic function **NORM2**(\(X, \text{DIM}\)) reduces Real arrays using the \(L_2\)-norm operation. This operates exactly the same as **SUM** and **PRODUCT**, except for the operation involved. The \(L_2\) norm of an array is the square root of the sum of the squares of the elements. Note that unlike most of the other reduction functions, **NORM2** does not have a **MASK** argument. The **DIM** argument is optional; an actual argument for **DIM** is not itself permitted to be an optional dummy argument.

The calculation of the result value is done in such a way as to avoid intermediate overflow and underflow, except when the result itself is outside the maximum range. For example, **NORM2**(\([X, Y]\)) is approximately the same as **HYPOT**(\(X, Y\)).

• **[6.0]** The elemental intrinsic functions **ERF**, **ERFC** and **ERFC** compute the error function, the complementary error function and the scaled complementary error function, respectively. The single argument \(X\) must be of type real. The error function is the integral of \(-t^2\) from 0 to \(X\), times \(2/\sqrt{\pi}\); this rapidly converges to 1. The complementary error function is 1 minus the error function, and fairly quickly converges to zero. The scaled complementary error function scales the value (of \(1\) minus the error function) by \(\exp(X*82)\); this also converges to zero but only very slowly.

• **[6.0]** The elemental intrinsic functions **GAMMA** and **LOG_GAMMA** compute the gamma function and the natural logarithm of the absolute value of the gamma function respectively. The single argument \(X\) must be of type real, and must not be zero or a negative integer.

The gamma function is the extension of factorial from the integers to the reals; for positive integers, **GAMMA**(\(X\)) is equal to \((X−1)\)!, i.e. factorial of \(X−1\). This grows very rapidly and thus overflows for quite small \(X\); **LOG_GAMMA** also diverges but much more slowly.

### 6.2 Additional intrinsic functions for bit manipulation [mostly 5.3]

• The elemental intrinsic functions **BGE**, **BGT**, **BLE** and **BLT** perform bitwise (i.e. unsigned) comparisons. They each have two arguments, \(I\) and \(J\), which must be of type Integer but may be of different kind. The result is default Logical.

For example, **BGE**(\(\text{INT}(Z'FF', \text{INT8}), 128\)) is true, while **INT(Z'FF', \text{INT8}) >= 128** is false.

• **[5.3.1]** The elemental intrinsic functions **DSHIFTL** and **DSHIFTR** perform double-width shifting. They each have three arguments, \(I\), \(J\) and **SHIFT** which must be of type Integer, except that one of \(I\) or \(J\) may be a **BOZ** literal constant – it will be converted to the type and kind of the other \(I\) or \(J\) argument. \(I\) and \(J\) must have the same kind if they are both of type Integer. The result is of type Integer, with the same kind as \(I\) and \(J\). The \(I\) and \(J\) arguments are effectively concatenated to form a single double-width value, which is shifted left or right by **SHIFT** positions; for **DSHIFTL** the result is the top half of the combined shift, and for **DSHIFTR** the result is the bottom half of the combined shift.

For example, **DSHIFTL**(\(\text{INT}(B'11000101', 1), B'11000101', 2\)) has the value **INT**(\(B'00010111', 1\)) (decimal value 23), whereas **DSHIFTR**(\(\text{INT}(B'11000101', 1), B'11000101', 2\)) has the value **INT**(\(B'011110010', 1\)) (decimal value 114).

• The array reduction intrinsic functions **IALL**, **IANY** and **IPARITY** reduce arrays using bitwise operations. These are exactly the same as **SUM** and **PRODUCT**, except that instead of reducing the array by the \(\oplus\) or \(\otimes\) operation, they reduce it by the **IAND**, **IOR** and **IEOR** intrinsic functions respectively. That is, each element of the result is the bitwise-and, bitwise-or, or bitwise-exclusive-or of the reduced elements. If the number of reduced elements is zero, the result is zero for **IANY** and **IPARITY**, and **NOT**(zero) for **IALL**.

• The elemental intrinsic functions **LEADZ** and **TRAILZ** return the number of leading (most significant) and trailing (least significant) zero bits in the argument \(I\), which must be of type Integer (of any kind). The result is default Integer.
The elemental intrinsic functions `MASKL` and `MASKR` generate simple left-justified and right-justified bitmasks. The value of `MASKL(I,KIND)` is an integer with the specified kind that has its leftmost I bits set to one and the rest set to zero; I must be non-negative and less than or equal to the bitsize of the result. If `KIND` is omitted, the result is default integer. The value of `MASKR` is similar, but has its rightmost I bits set to one instead.

[5.3.1] The elemental intrinsic function `MERGE_BITS(I,J,MASK)` merges the bits from Integer values I and J, taking the bit from I when the corresponding bit in `MASK` is 1, and taking the bit from J when it is zero. All arguments must be BOZ literal constants or of type Integer, and all the Integer arguments must have the same kind; at least one of I and J must be of type Integer, and the result has the same type and kind.

Note that `MERGE_BITS(I,J,MASK)` is identical to `IOR(IAND(I,MASK),IAND(J,NOT(MASK)))`.

For example, `MERGE_BITS(INT(B'00110011',1),B'11110000',B'10101010')` is equal to `INT(B'01110010')` (decimal value 114).

The array reduction intrinsic function `PARITY` reduces Logical arrays. It is exactly the same as `ALL` and `ANY`, except that instead of reducing the array by the `.AND.` or `.OR.` operation, it reduces it by the `.NEQV.` operation. That is, each element of the result is `.TRUE.` if an odd number of reduced elements is `.TRUE..`

The elemental intrinsic function `POPCNT(I)` returns the number of bits in the Integer argument I that are set to 1. The elemental intrinsic function `POPPAR(I)` returns zero if the number of bits in I that are set to 1 are even, and one if it is odd. The result is default Integer.

### 6.3 Other new intrinsic procedures [mostly 5.3.1]

- The intrinsic subroutine `EXECUTE_COMMAND_LINE` passes a command line to the operating system’s command processor for execution. It has five arguments, in order these are:
  ```
  CHARACTER(*),INTENT(IN) :: COMMAND — the command to be executed;
  LOGICAL,INTENT(IN),OPTIONAL :: WAIT — whether to wait for command completion (default true);
  INTEGER,INTENT(INOUT),OPTIONAL :: EXITSTAT — the result value of the command;
  INTEGER,INTENT(OUT),OPTIONAL :: CMDSTAT — see below;
  CHARACTER(*),INTENT(INOUT),OPTIONAL :: CMDMSG — the error message if CMDSTAT is non-zero.
  
  CMDSTAT values are zero for success, −1 if command line execution is not supported, −2 if WAIT is present and false but asynchronous execution is not supported, and a positive value to indicate some other error. If CMDSTAT is not present but would have been set non-zero, the program will be terminated. Note that Release 5.3.1 supports command line execution on all systems, and does not support asynchronous execution on any system.
  
  For example, CALL EXECUTE_COMMAND_LINE(‘echo Hello’) will probably display ‘Hello’ in the console window.
  ```

- The intrinsic function `STORAGE_SIZE(A,KIND)` returns the size in bits of a scalar object with the same dynamic type and type parameters as A, when it is stored as an array element (i.e. including any padding). The `KIND` argument is optional; the result is type Integer with kind `KIND` if it is present, and default kind otherwise.

  If A is allocatable or a pointer, it does not have to be allocated unless it has a deferred type parameter (e.g. `CHARACTER(*)`) or is `CLASS(*)`. If it is a polymorphic pointer, it must not have an undefined status.

  For example, `STORAGE_SIZE(13,1)` is equal to 8 (bits).

- [6.0] The intrinsic inquiry function `IS_CONTIGUOUS` has a single argument `ARRAY`, which can be an array of any type. The function returns true if `ARRAY` is stored contiguously, and false otherwise. Note that this question has no meaning for an array with no elements, or for an array expression since that is a value and not a variable.

### 6.4 Changes to existing intrinsic procedures [5.3.1]

- The intrinsic functions `ACOS`, `ASIN`, `ATAN`, `COSH`, `SINH`, `TAN` and `TANH` now accept arguments of type Complex. Note that the hyperbolic and non-hyperbolic versions of these functions and the new `ACOSH`, `ASINH` and `ATANH` functions are all related by simple algebraic identities, for example the new `COSH(X)` is identical to the old `COS((0,1)*X)` and the new `SINH(X)` is identical to the old `(0,-1)*SIN((0,1)*X)`.

- The intrinsic function `ATAN` now has an extra form `ATAN(Y,X)`, with exactly the same semantics as `ATAN2(Y,X)`.

- The intrinsic function `SELECTED_REAL_KIND` now has a third argument `RADIX`; this specifies the desired radix of the Real kind requested. Note that the function `IEEESELECTED_REAL_KIND` in the intrinsic module `IEEE_ARITHMETIC` also has this new third argument, and will allow requesting IEEE decimal floating-point kinds if they become available in the future.
6.5 ISO_FORTRAN_ENV additions [5.3]

The intrinsic module ISO_FORTRAN_ENV contains additional named constants as follows.

- The additional scalar integer constants INT8, INT16, INT32, INT64, REAL32, REAL64 and REAL128 supply the kind type parameter values for integer and real kinds with the indicated bit sizes.
- The additional named array constants CHARACTER_KINDS, INTEGER_KINDS, LOGICAL_KINDS and REAL_KINDS list the available kind type parameter values for each type (in no particular order).

7 Input/output extensions [mostly 5.3]

- The NEWUNIT= specifier has been added to the OPEN statement; this allocates a new unit number that cannot clash with any other logical unit (the unit number will be a special negative value). For example,

  ```fortran
  INTEGER unit
  OPEN(FILE='output.log', FORM='FORMATTED', NEWUNIT=unit)
  WRITE(unit,*) 'Logfile opened.'
  ```

  The NEWUNIT= specifier can only be used if either the FILE= specifier is also used, or if the STATUS= specifier is used with the value 'SCRATCH'.
- Recursive input/output is allowed on separate units. For example, in

  ```fortran
  WRITE (*,Output_Unit) f(100)
  ```

  the function f is permitted to perform i/o on any unit except Output_Unit; for example, if the value 100 is out of range, it would be allowed to produce an error message with

  ```fortran
  WRITE (*,Error_Unit) 'Error in F: ',n,' is out of range'
  ```
- [6.0] A sub-format can be repeated an indefinite number of times by using an asterisk (*) as its repeat count. For example,

  ```fortran
  SUBROUTINE s(x)
    LOGICAL x(:)
    PRINT 1,x
  1   FORMAT('x = ',*(',L1'))
  END SUBROUTINE
  ```

  will display the entire array x on a single line, no matter how many elements x has. An infinite repeat count is only allowed at the top level of the format specification, and must be the last format item.
- [6.0] The GO and GO.d edit descriptors perform generalised editing with all leading and trailing blanks (except those within a character value itself) omitted. For example,

  ```fortran
  PRINT 1,1.25,.True.,"Hi !",123456789
  1 FORMAT(*(G0.,'))
  ```

  produces the output

  1.250000,T,Hi !,123456789,
8 Programs and procedures [mostly 5.3]

- An empty internal subprogram part, module subprogram part or type-bound procedure part is now permitted following a CONTAINS statement. In the case of the type-bound procedure part, an ineffectual PRIVATE statement may appear following the unnecessary CONTAINS statement.

- [6.0] An internal procedure can be passed as an actual argument or assigned to a procedure pointer. When the internal procedure is invoked via the dummy argument or procedure pointer, it can access the local variables of its host procedure. In the case of procedure pointer assignment, the pointer is only valid until the host procedure returns (since the local variables cease to exist at that point).

For example,

```fortran
SUBROUTINE mysub(coeffs)
   REAL,INTENT(IN) :: coeffs(0:) ! Coefficients of polynomial.
   REAL integral
   integral = integrate(myfunc,0.0,1.0) ! Integrate from 0.0 to 1.0.
   PRINT *,'Integral =',integral
   CONTAINS
   REAL FUNCTION myfunc(x) RESULT(y)
      REAL,INTENT(IN) :: x
      INTEGER i
      y = coeffs(UBOUND(coeffs,1))
      DO i=UBOUND(coeffs,1)-1,0,-1
         y = y*x + coeffs(i)
      END DO
   END FUNCTION
   END SUBROUTINE
```

- The rules used for generic resolution and for checking that procedures in a generic are unambiguous have been extended. The extra rules are that
  - a dummy procedure is distinguishable from a dummy variable;
  - an ALLOCATABLE dummy variable is distinguishable from a POINTER dummy variable that does not have INTENT(IN).

- [6.0] A disassociated pointer, or an unallocated allocatable variable, may be passed as an actual argument to an optional nonallocatable nonpointer dummy argument. This is treated as if the actual argument were not present.

- [5.3.1] Impure elemental procedures can be defined using the IMPURE keyword. An impure elemental procedure has the restrictions that apply to elementality (e.g. all arguments must be scalar) but does not have any of the “pure” restrictions. This means that an impure elemental procedure may have side effects and can contain input/output and STOP statements. For example,

```fortran
Impure Elemental Integer Function checked_addition(a,b) Result(c)
   Integer,Intent(In) :: a,b
   If (a>0 .And. b>0) Then
      If (b>Huge(c)-a) Stop 'Positive Integer Overflow'
   Else If (a<0 .And. b<0) Then
      If ((a+Huge(c))+b<0) Stop 'Negative Integer Overflow'
   End If
   c = a + b
End Function
```

When an argument is an array, an impure elemental procedure is applied to each element in array element order (unlike a pure elemental procedure, which has no specified order). An impure elemental procedure cannot be referenced in a context that requires a procedure to be pure, e.g. within a FORALL construct.

Impure elemental procedures are probably most useful for debugging (because i/o is allowed) and as final procedures.
• [6.0] If an argument of a pure procedure has the VALUE attribute it does not need any INTENT attribute. For example,

```fortran
PURE SUBROUTINE s(a,b)
    REAL, INTENT(OUT) :: a
    REAL, VALUE :: b
    a = b
END SUBROUTINE
```

Note however that the second argument of a defined assignment subroutine, and all arguments of a defined operator function, are still required to have the INTENT(IN) attribute even if they have the VALUE attribute.

• [5.3.1] The FUNCTION or SUBROUTINE keyword on the END statement for an internal or module subprogram is now optional (when the subprogram name does not appear). Previously these keywords were only optional for external subprograms.

• ENTRY statements are regarded as obsolescent.

• [1.0] A line in the program is no longer prohibited from beginning with a semi-colon.

9 References