NAG Fortran Compiler Release 7.2 Release Note

March 8, 2024

1 Introduction

Release 7.2 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

Release 7.2 supports all of Fortran 2018, and so the default language level is now *−f2018*.

Partial support for OpenMP 4.0 and 4.5 is included in the initial release of 7.2. An update will follow in early 2024 to upgrade that to full support.

3 Compatibility

3.1 Compatibility with Release 7.1

Release 7.2 is fully compatible with Release 7.1.

3.2 Compatibility with Release 7.0

Release 7.2 is compatible with Release 7.0, except that files compiled with the *−C=calls* option will need to be recompiled if they contain a procedure with a procedure pointer argument, or a reference to such a procedure.

3.3 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 7.2 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.

3.4 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The previously deprecated *−abi=64* option on Linux x86-64 has been withdrawn.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 7.2 of the NAG Fortran Compiler is fully compatible with Release 6.1.

3.5 Compatibility with Release 6.0

With the exception of HPF support and the deprecated option removal, Release 7.2 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).

3.6 Compatibility with Releases 5.3.1, 5.3 and 5.2

With the exception of HPF support and the deprecated option removal, Release 7.2 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 7.1 or later.

3.7 Compatibility with Release 5.1

Release 7.2 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.

4 New Fortran 2018 Features

- The `GENERIC` statement provides a concise way of declaring generic interfaces. It has the syntax:

  `GENERIC` *[, access-spec]* `::` *generic-spec* `=>` *procedure-name-list*

  where the optional *access-spec* is either `PUBLIC` or `PRIVATE`, the *generic-spec* is a generic identifier (name, `ASSIGNMENT(=)`, `OPERATOR(`*op*`)`, or `{READ|WRITE}({FORMATTED|UNFORMATTED})`), and the *procedure-name-list* is a comma-separated list of named procedures.

  The *access-spec* is only permitted if the `GENERIC` statement is in the specification part of a module. Each named procedure in the list must have an explicit interface; that is, it must be an internal procedure, module procedure, or be declared with an interface block or procedure declaration statement that specifies an explicit interface. Collectively, the procedures must satisfy the usual generic rules about all being functions or all being subroutines, and being unambiguous.

  Apart from the optional *access-spec*, the `GENERIC` statement has the same effect as

  ```
  INTERFACE generic-spec
      PROCEDURE procedure-name-list
  END INTERFACE
  ```

  The only advantage is that it is a couple of lines shorter, and can declare the accessibility in the same line. This syntax is the same as for a *generic-binding* in a derived type definition, except that the list of names is of ordinary named procedures instead of type-bound procedures.

  For example, the program

  ```
  Module print_sqrt
      Private
      Generic,Public :: g => s1, s2
  ```

```
    Contains
        Subroutine s1(x)
            Print '(F10.6)',Sqrt(x)
        End Subroutine
        Subroutine s2(n)
            Print '(I10)',Nint(Sqrt(Real(n)))
        End Subroutine
    End Module
    Program test
        Use print_sqrt
        Call g(2.0)
        Call g(127)
    End Program
```

will print

```
  1.414214
        11
```

- The `E0` exponent width specifier can be used on all edit descriptors that allow the exponent width to be specified (thus `E`, `EN` et al, but not `D`). This specifies formatting with the minimal width for the exponent. For example,

```
    Print '(7X,3ES10.2E0)', 1.23, 4.56E24, 7.89D101
```

will print

```
          1.23E+0  4.56E+24 7.89E+101
```

- The `E`, `D`, `EN` and `ES` edit descriptors may have a width of zero on output. This provides minimal width editing similarly to the `I` and other edit descriptors; that is, the processor chooses the smallest value for the width w that does not result in the field being filled with asterisks. This means that leading blanks will be suppressed, and for `E` and `D`, when the scale factor is less than or equal to zero, the optional zero before the decimal symbol is suppressed.

For example, printing 12.3 with the formats shown below will display the results below with no leading or trailing blanks.

`E0.4`	`.1230E+02`
`E0.4E3`	`.1230E+002`
`1PE0.4`	`1.2300E+00`
`EN0.4`	`12.3000E+00`
`ES0.4`	`1.2300E+01`
`E0.4E0`	`.1230E+2`

Note that field width has no effect on the format of the exponent; that is, to print a number in the smallest width requires use of exponent width zero as well (as shown in the last example above).

There is no means of eliminating trailing zeroes in the mantissa part for these edit descriptors (though there is for the new `EX` edit descriptor).

- The `G0.`*d* edit descriptor is permitted for types Integer, Logical, and Character; in Fortran 2008, this was an i/o error. In Fortran 2018 it has the same effect as `I0` for Integer, `L1` for Logical, and `A` for Character. For example,

```
    Print '(7X,"start:",3G0.17,":end")', 123, .True., 'ok'
```

will print

```
       start:123Tok:end
```

- The new edit descriptors `EX`*w*`.`*d* and `EX`*w*`.`*d*`E`*e* can be used for output of floating-point numbers with a hexadecimal significand. The format is

    [s] `0X` x_0 `.` $x_1 x_2 \ldots$ *exponent*

where s is an optional plus or minus sign (+ or −), each x_i is a hexadecimal digit (`0`. . . `9` or `A`. . . `F`), and *exponent* is the binary exponent (power of two) expressed in decimal, with the format `P` s $z_1 \ldots z_n$, where if the optional `E`e appears, n is equal to e, and otherwise is the minimum number of digits needed to represent the exponent. If the exponent is equal to zero, the sign s is a plus sign.

If the number of digits d is zero, the number of mantissa digits x_i produced is the smallest number that represents the internal value exactly. Note that d must not be zero if the radix of the internal value is not a power of two.

For input, the effect of the `EX` edit descriptor is identical to that of the `F` edit descriptor.

Note that the value of the initial hexadecimal digit is not standardised, apart from being non-zero. Thus, depending on the compiler, writing the value 1.0 in `EX0.1` might produce `0X1.0P+0`, `0X2.0P-1`, `0X4.0P-2`, or `0X8.0P-3`. The NAG Fortran Compiler always shifts the mantissa so that the most significant bit is set, thus will output `0X8.0P-3` in this case.

- Input of floating-point numbers with list-directed, namelist, and explicit formatting (e.g. the `F` edit descriptor) accepts input values in hexadecimal-significand form. That is, the form produced by the `EX` edit descriptor. Note that a hexadecimal-significand value always begins with an optional sign followed by the digit zero and the letter `X`; that is, `+0X`, `-0X`, or `0X`, thus there is no ambiguity.

  For example, reading '`-0XA.P-3`' will produce the value `-1.25`.

  As usual for numeric input, lowercase input is treated the same as upper case; thus `-0xa.p-3` produces the same value as `-0XA.P-3`.

- The elemental intrinsic function `OUT_OF_RANGE` returns true if and only if a conversion would be out of range. It has the syntax:

  `OUT_OF_RANGE ( X, MOLD [ , ROUND ] )`

  `X` : type Real or Integer;

  `MOLD` : scalar of type Real or Integer;

  `ROUND` *(optional)* : scalar of type Logical;

  Result : Logical of default kind.

  The result is true if and only if the value of `X` is outside the range of values that can be converted to the type and kind of `MOLD` without error. If the `MOLD` argument is a variable, it need not have a defined value — only its type and kind are used. The `ROUND` argument is only allowed when `X` is type Real and `MOLD` is type Integer.

  For Real to Integer conversions, the default check is whether the value would be out of range for the intrinsic function `INT (X, KIND (MOLD))`; this is the same conversion that is used in intrinsic assignment. If the `ROUND` argument is present with the value `.TRUE.`, the check is instead whether the value would be out of range for the intrinsic function `NINT (X, KIND (MOLD))`.

  For example, `OUT_OF_RANGE (127.5, 0_int8)` is false, but `OUT_OF_RANGE (127.5, 0_int8, .TRUE.)` is true.

  If the value of `X` is an IEEE infinity, `OUT_OF_RANGE` will return `.TRUE.` if and only if the type and kind of `MOLD` does not support IEEE infinities. Similarly, if `X` is an IEEE NaN, the result is true if and only if `MOLD` does not support IEEE NaNs.

  Note that when checking conversions of type Real of one kind to type Real of another kind (for example, `REAL(real32)` to `REAL(real16)` or `REAL(real64)` to `REAL(real32)`), a finite value that is greater than `HUGE (KIND (MOLD))` will be considered out of range, but an infinite value will not be considered out of range. That is, `OUT_OF_RANGE (1.0E200_real64, 1.0_real32)` will return `.TRUE.`, but `OUT_OF_RANGE (IEEE_VALUE (1.0_real64, IEEE_POSITIVE_INF), 1.0_real32)` will return `.FALSE.`.

  Although this function is elemental, and can be used in constant expressions (if the value of `X` is constant and the `ROUND` argument is missing or constant), only the `X` argument is permitted to be an array. The result thus always has the rank and shape of `X`.

- The `VALUES` argument of the intrinsic subroutine `DATE_AND_TIME` can be any kind of integer with a decimal exponent range of at least four; that is, any kind except 8-bit integer. For example,

```
Program show_year
    Use Iso_Fortran_Env
    Integer(int16) v(8)
    Call Date_And_Time(Values=v)
    Print *,'The year is',v(1)
End Program
```

- The `WAIT` argument of the intrinsic subroutine `EXECUTE_COMMAND_LINE` can be any kind of logical. The `CMDSTAT` and `EXITSTAT` arguments can be any kind of integer with a decimal exponent range of at least four; that is, any kind except 8-bit integer. For example,

```
Program ok
    Use Iso_Fortran_Env
    Logical(logical8) :: w = .True._logical8
    Integer(int16) :: cstat
    Integer(int64) :: estat
    Call Execute_Command_Line('echo ok',w,estat,cstat)
    If (estat/=0 .Or. cstat/=0) Print *,'Bad STAT',estat,cstat
End Program
```

  will, assuming '`echo`' is the Unix echo command, display

```
ok
```

- The intrinsic subroutines `GET_COMMAND`, `GET_COMMAND_ARGUMENT` and `GET_ENVIRONMENT_VARIABLE` now have an optional `ERRMSG` argument at the end of the argument list. If an error occurs (i.e., a positive value would be assigned to the `STATUS` argument), the `ERRMSG` argument will be assigned an explanatory message. If no error occurs (zero or a negative value would be assigned to the `STATUS` argument), the `ERRMSG` argument remains unchanged. Note that the effect on the `ERRMSG` argument occurs (or does not occur) even if the `STATUS` argument is omitted. But as it is unchanged when no error occurs, it is not a substitute for the `STATUS` argument for detecting errors.

  For example, executing this program

```
Program test
  Character(200) msg,value
  Integer status
  Call Get_Environment_Variable('Does Not Exist',value,status,Errmsg=msg)
  If (status>0) Print *,Trim(msg)
End Program
```

  will, unless there is an environment variable called '`Does Not Exist`', print something like

```
Environment variable does not exist
```

- The intrinsic function `IMAGE_INDEX` has two additional forms:

  `IMAGE_INDEX( COARRAY, SUB, TEAM )`
  `IMAGE_INDEX( COARRAY, SUB, TEAM_NUMBER )`

  `TEAM` : scalar of type `TEAM_TYPE`, `Intent(In)`;
  `TEAM_NUMBER` : scalar of type Integer, `Intent(In)`.

  The meanings of the `COARRAY` and `SUB` arguments are unchanged, except that the subscripts are interpreted as being for the specified team. The return value is likewise the image index in the specified team.

- The random-number generator (intrinsic `RANDOM_NUMBER`) is now per-image. Fortran 2008 permitted a compiler to use a single random-number stream, shared between all images; Fortran 2018 does not permit that, instead requiring each image to have its own random-number state.

- The new intrinsic subroutine `RANDOM_INIT` initialises the random-number generator on the invoking image. It has the syntax:

  `CALL RANDOM_INIT ( REPEATABLE, IMAGE_DISTINCT )`

  `REPEATABLE` : scalar of type Logical, `Intent(In)`;
  `IMAGE_DISTINCT` : scalar of type Logical, `Intent(In)`.

  If `IMAGE_DISTINCT` is true, the initial state (seed) of the random-number generator will be different on every invoking image; otherwise, the initial state will not depend on which image it is. If `REPEATABLE` is true, each execution of the program will use the same initial seed (image-dependent if `IMAGE_DISTINCT` is also true); otherwise, each execution of the program will use a different initial seed.

  The default for the NAG Fortran Compiler, when no call to `RANDOM_INIT` is made, is `REPEATABLE`=false and `IMAGE_DISTINCT`=true.

- There are additional constants, types, and procedures in the standard intrinsic module `IEEE_ARITHMETIC`, providing additional support for IEEE (ISO/IEC 60559) arithmetic. Three of the pre-existing procedures now have additional, optional, arguments.

  - The named constant `IEEE_AWAY` is of type `IEEE_ROUND_TYPE`, and represents a rounding mode that rounds ties away from zero. The IEEE standard only requires this rounding mode for decimal, and binary hardware does not support this, so it cannot be used.
  - The elemental function `IEEE_FMA` performs a fused multiply-add operation. It has the syntax:

    `IEEE_FMA (A, B, C)`

    `A` : type Real;
    `B` : same type and kind as `A`;
    `C` : same type and kind as `A`.

    Result : Same type and kind as `A`.

    The result of the function is the value of (`A`×`B`)+`C` with only one rounding; that is, the whole operation is computed mathematically and only rounded to the format of `A` at the end. For example, `IEEE_OVERFLOW` is not signalled if `A`×`B` overflows, but only if the final result is out of range.

    Restriction: this function must not be invoked when the kind of `A` is not an IEEE format, i.e. if `IEEE_SUPPORT_DATATYPE (A)` returns false.

  - The pure subroutine `IEEE_GET_MODES` retrieves the halting modes, rounding modes, and underflow mode in a single object. Its syntax is:

    `IEEE_GET_MODES (MODES)`

    `MODES` : scalar of type `IEEE_MODES_TYPE`, `Intent(Out)`.

    The retrieved modes can be used by `IEEE_SET_MODES` to restore the modes.

  - The pure subroutine `IEEE_GET_ROUNDING_MODE`, which retrieves the current rounding mode, now has an optional argument to specify the radix. The syntax is thus now:

    `IEEE_GET_ROUNDING_MODE (ROUND_VALUE` *[*`, RADIX` *]*`)`

    `ROUND_VALUE` : type `IEEE_ROUND_TYPE`, `Intent(Out)`;
    `RADIX` (optional) : Integer, `Intent(In)`, must be equal to two or ten.

    The `ROUND_VALUE` argument is assigned the current rounding mode for the specified radix. If `RADIX` does not appear, the binary rounding mode is assigned.

  - The elemental function `IEEE_INT` converts an IEEE Real value to Integer with a specific rounding mode. It has the syntax:

    `IEEE_INT (A, ROUND` *[*`, KIND` *]*`)`

    `A` : type Real;
    `ROUND` : type `IEEE_ROUND_TYPE`;
    `KIND` (optional) : scalar Integer constant expression;
    Result : type Integer, with kind `KIND` if `KIND` appears, otherwise default kind.

    The value of `A` is rounded to an integer using the rounding mode specified by `ROUND`. If that value is representable in the result kind, the result has that value; otherwise, the result is processor-dependent and `IEEE_INVALID` is signalled.
    This operation is either the `convertToInteger{round}` or the `convertToIntegerExact{round}` operation specified by the IEEE standard. If it is the latter, and `IEEE_INVALID` was not signalled, but `A` was not already an integer, `IEEE_INEXACT` is signalled.

    Restriction: this function must not be invoked when the kind of `A` is not an IEEE format, i.e. if `IEEE_SUPPORT_DATATYPE (A)` returns false.

  - The elemental functions `IEEE_MAX_NUM`, `IEEE_MAX_NUM_MAG`, `IEEE_MIN_NUM`, `IEEE_MIN_NUM_MAG` perform maximum/minimum operations ignoring NaN values. If an argument is a signalling NaN, `IEEE_INVALID` is raised, if only one argument is a NaN, the result is the other argument; only if both arguments are NaNs is the result a NaN. The syntax for `IEEE_MAX_NUM` is:

    `IEEE_MAX_NUM (X, Y)`

`X` : type Real;
`Y` : same type and kind as `X`;
Result : same type and kind as `X`.

The value of the result is the maximum value of `X` and `Y`, ignoring NaN.

Restriction: this function must not be invoked when the kind of `X` is not an IEEE format, i.e. if `IEEE_SUPPORT_DATATYPE (X)` returns false.

The `IEEE_MAX_NUM_MAG` has the same syntax (apart from the name), and the result is whichever of `X` and `Y` has the greater magnitude. The same restriction applies.

The `IEEE_MIN_NUM` has the same syntax (apart from the name), and the result is the minimum value of `X` and `Y`, ignoring NaN. The same restriction applies.

The `IEEE_MIN_NUM_MAG` has the same syntax (apart from the name), and the result is whichever of `X` and `Y` has the smaller magnitude. The same restriction applies.

- The derived type `IEEE_MODES_TYPE` contains the all the floating-point modes: the halting modes, the rounding modes, and the underflow mode. It is used by the `IEEE_GET_MODES` and `IEEE_SET_MODES` subroutines.
- The elemental functions `IEEE_QUIET_{EQ|NE|LT|LE|GT|GE}` compare two IEEE format numbers, without raising any signal if an operand is a quiet NaN. If an operand is a signalling NaN, the `IEEE_INVALID` exception is raised. `IEEE_QUIET_EQ` and `IEEE_QUIET_NE` are exactly the same as `==` and `/=`, apart from only being usable on IEEE format numbers. The `IEEE_QUIET_EQ` function has the syntax:

  `IEEE_QUIET_EQ (A, B)`

  `A` : type Real (any IEEE kind);
  `B` : same type and kind as `A`;

  Result : Logical of default kind.

  The syntax of `IEEE_QUIET_NE` et al is the same, except for the name of the function.

  Restriction: these functions must not be invoked when the kind of X is not an IEEE format, i.e. if `IEEE_SUPPORT_DATATYPE (X)` returns false.
- The elemental function `IEEE_REAL` converts an Integer or IEEE format Real value to the specified IEEE format Real value. Its syntax is:

  `IEEE_REAL (A, [, KIND ])`

  `A` : type Real or Integer;
  `KIND` (optional) : scalar Integer constant expression;
  Result : type Real, with kind `KIND` if `KIND` appears, otherwise default kind.

  If the value of `A` is representable in the kind of the result, that value is the result. Otherwise, the value of `A` is rounded to the kind of the result using the current rounding mode.

  Restriction: this function must not be invoked when the kind of `A` or the result is not an IEEE format, i.e. if `IEEE_SUPPORT_DATATYPE (A)` or if `IEEE_SUPPORT_DATATYPE(REAL(0,KIND))` returns false.
- The pure subroutine `IEEE_SET_MODES` sets the halting modes, rounding modes, and underflow mode to the state when a previous call to `IEEE_GET_MODES` was made. Its syntax is:

  `IEEE_SET_MODES (MODES)`

  `MODES` : scalar of type `IEEE_MODES_TYPE`, `Intent(In)`.

  The value of `MODES` must be one that was obtained via `IEEE_GET_MODES`.
- The pure subroutine `IEEE_SET_ROUNDING_MODE`, which sets the rounding mode, now has an optional argument to specify the radix. The syntax is thus now:

  `IEEE_SET_ROUNDING_MODE (ROUND_VALUE [, RADIX ])`

  `ROUND_VALUE` : type `IEEE_ROUND_TYPE`, `Intent(In)`;
  `RADIX` (optional) : Integer, `Intent(In)`, must be equal to two or ten.

  The rounding mode for the specified radix is set to `ROUND_VALUE`. If `RADIX` does not appear, the binary rounding mode is set.

  Restriction: This subroutine must not be invoked unless there is some `X` (with radix `RADIX` if it is present) for which both `IEEE_SUPPORT_DATATYPE(X)` and `IEEE_SUPPORT_ROUNDING(ROUND_VALUE,X)` are true.

- The elemental functions `IEEE_SIGNALING_{EQ|NE|LT|LE|GT|GE}` compare two IEEE format numbers, raising the `IEEE_INVALID` exception if an operand is a NaN (whether quiet or signalling). `IEEE_SIGNALING_LT`, `IEEE_SIGNALING_LE`, `IEEE_SIGNALING_GT` and `IEEE_SIGNALING_GE` are exactly the same as `<`, `<=`, `>` and `>=`, apart from only being usable on IEEE format numbers. The `IEEE_SIGNALING_EQ` function has the syntax:

  `IEEE_SIGNALING_EQ (A, B)`

  `A` : type Real (any IEEE kind);
  `B` : same type and kind as `A`;

  Result : Logical of default kind.

  The syntax of `IEEE_SIGNALING_NE` et al is the same, except for the name of the function.

  Restriction: these functions must not be invoked when the kind of X is not an IEEE format, i.e. if `IEEE_SUPPORT_DATATYPE (X)` returns false.

- The elemental function `IEEE_SIGNBIT` queries the sign bit of an IEEE format number. It has the syntax:

  `IEEE_SIGNBIT (X)`

  `X` : type Real (any IEEE kind);

  Result : Logical of default kind.

  The result is true if and only if the sign bit is set (indicating negative for any value that is not a NaN).

  Restriction: this function must not be invoked when the kind of `X` is not an IEEE format, i.e. if `IEEE_SUPPORT_DATATYPE (X)` returns false.

- The elemental function `IEEE_RINT` now has an optional argument to specify the rounding required. The syntax is thus now:

  `IEEE_RINT (X` *[*`, ROUND` *]*`)`

  `X` : type Real;
  `ROUND` (optional) : type `IEEE_ROUND_TYPE`.

  Result : Same type and kind as `X`.

  When `ROUND` is present, the result is the value of `X` rounded to an integer according to the mode specified by `ROUND`; this is the operation the IEEE standard calls `roundToIntegral{rounding}`. When `ROUND` is absent, the result is the value of `X` rounded to an integer according to the current rounding mode; this is the operation the IEEE standard calls `roundToIntegralExact`.

  Restriction: this function must not be invoked when the kind of `X` is not an IEEE format, i.e. if `IEEE_SUPPORT_DATATYPE (X)` returns false.

- The named constant `C_PTRDIFF_T` has been added to the intrinsic module `ISO_C_BINDING`. This is the integer kind that corresponds to the C type `ptrdiff_t`, which is an integer large enough to hold the difference between two pointers. For example, the interface

```
Interface
  Function diff_cptr(a,b) Bind(C)
    Use Iso_C_Binding
    Type(C_ptr),Value :: a, b
    Integer(C_ptrdiff_t) diff_cptr
  End Function
End Interface
```

  interoperates with the C function

```
ptrdiff_t diff_cptr(void *a,void *b) {
    return a - b;
}
```

  In the intrinsic module `ISO_C_BINDING`, the procedures `C_LOC` and `C_FUNLOC` are considered to be pure procedures, and `C_F_POINTER` and `C_F_PROCPOINTER` are considered to be impure. When it is used within a pure procedure, the argument of `C_FUNLOC` must also be a pure procedure.

- The default accessibility in a module of entities accessed from another module (via the `USE` statement) can be controlled by specifying that module name in a `PUBLIC` or `PRIVATE` statement, overriding the default accessibility of other entities in the importing module. For example, in

  ```
  Module mymod
    Use Iso_Fortran_Env
    Real(real32) x
    Integer(int64) y
    Private Iso_Fortran_Env
  End Module
  ```

  all the entities in `ISO_FORTRAN_ENV` are by default `PRIVATE` in module `mymod`, without needing to list them individually.

  This new default accessibility can be overridden by an explicit `PUBLIC` or `PRIVATE` declaration. Also, if an entity in a remote module (two or more `USE` statements away) is accessed by more than one intervening module, it is default `PRIVATE` only if every route to the entity is default `PRIVATE`, and default `PUBLIC` if any route is default `PUBLIC`. For example, in

  ```
  Module remote
      Real a,b
  End Module
  Module route_one
      Use remote
      Private remote
  End Module
  Module route_two
      Use remote
  End Module
  Module my_module
      Use route_one
      Use route_two
      Private route_one
  End Module
  ```

  the variables `A` and `B` in module `REMOTE` are `PUBLIC` in module `MY_MODULE`, because they are accessible via module `ROUTE_TWO` which is default `PUBLIC`.

- The `IMPLICIT NONE` statement can now have `TYPE` and `EXTERNAL` specifiers. Its full syntax is now:

  `IMPLICIT NONE` *[* `(` *[implicit-none-specifier-list]* `)` *]*

  where *implicit-none-specifier-list* is a comma-separated list of the keywords `EXTERNAL` and `TYPE`. No keyword may appear more than once in the list. If the list does not appear, or if `TYPE` appears in the list, no other `IMPLICIT` statement may appear in the scoping unit.

  The semantics of:

  ```
  IMPLICIT NONE ()
  ```

  and

  ```
  IMPLICIT NONE (TYPE)
  ```

  are identical to that of

  ```
  IMPLICIT NONE
  ```

  If the keyword `EXTERNAL` appears, a procedure with an implicit interface that is referenced in the scoping unit must be given the `EXTERNAL` attribute explicitly: that is, it must be declared in an `EXTERNAL` statement, in a type declaration statement that has the `EXTERNAL` attribute, or in a procedure declaration statement. For example,

  ```
  Subroutine sub(x)
      Implicit None (External)
      Integer f
      Print *,f(x)
  End Subroutine
  ```

  will produce an error for the reference to the function `F`, because it does not have the `EXTERNAL` attribute.

  If the keyword `EXTERNAL` appears and the keyword `TYPE` does not appear, implicit typing is **not** disabled, and other `IMPLICIT` statements may appear in the scoping unit. If both the keywords `TYPE` and `EXTERNAL` appear, both implicit typing is disabled, and the `EXTERNAL` attribute is required for implicit-interface procedures.

- The `IMPORT` statement can appear in `BLOCK` constructs and nested subprograms. By default, such scoping units have access to all entities in the host scope by host association, so by itself this is only useful as (compiler-checked) documentation. For example,

```
Subroutine outer(x,y)
    Real,Intent(InOut) :: x, y(:)
    ...
Contains
    Subroutine inner
        Import :: x, y
        ...
```

- Control over host association is provided by the `IMPORT,NONE`, `IMPORT,ALL`, and `IMPORT,ONLY` statements. Like other `IMPORT` statements, they can appear only in interface bodies, `BLOCK` constructs, and contained subprograms, and appear in between `USE` statements and other specification statements.

  The `IMPORT,NONE` statement specifies that no entities in the host scope are accessible by host association. That is the default for interface bodies other than separate module procedure interfaces. If an `IMPORT,NONE` statement appears in a scoping unit, no other `IMPORT` statement may appear. For example, in

```
Subroutine outer(x,y)
    Real,Intent(InOut) :: x, y(:)
    ...
Contains
    Subroutine inner
        Import,None
        Implicit Integer (a-z)
        Read *,x
        Print *,x
    End Subroutine
End Subroutine
```

  the `X` in subroutine `INNER` is not a reference to the `X` in its host `OUTER`, but is an implicitly typed (`Integer`) variable that is local to `INNER`.

  The `IMPORT,ALL` statement specifies that all host entities are accessed. That means that a declaration which would otherwise make a host entity inaccessible (so-called “shadowing”), is invalid. For example, in

```
Subroutine outer(x,y)
    Real,Intent(InOut) :: x, y(:)
    ...
Contains
    Subroutine inner
        Import,All
        Integer,External :: y
        ...
```

  the declaration of `Y` inside `INNER` is invalid, and will produce a compilation error. If an `IMPORT,ALL` statement appears in a scoping unit, no other `IMPORT` statement may appear.

  The `IMPORT,ONLY` statement specifies that only host entities named in `IMPORT,ONLY` statements are accessible by host association. If an `IMPORT,ONLY` statement appears in a scoping unit, all other `IMPORT` statements must have the `ONLY` keyword. For example, in

```
Subroutine outer(x,y,z)
    Real,Intent(InOut) :: x, y(:),z
    ...
Contains
    Subroutine inner
        Import,Only:x,y
        z = x + y
```

  the references to `X` and `Y` in `INNER` are references to the host (`OUTER`) entities, but the reference to `Z` in `INNER` is to an implicitly-typed local variable.

- An implied `DO` loop in an array constructor, or in a `DATA` statement, may now have an Integer type-spec preceding the loop control. The syntax for such a loop in the array constructor is

  (*ac-value-list* , *integer-type-spec* :: *ac-do-variable* = *from* , *to* *[* , *step* *]*) and for such a loop in a `DATA` statement is

(*data-i-do-object-list* , *integer-type-spec* :: *data-i-do-variable* = *from* , *to* [, *step*])

where the *integer-type-spec* is any type-spec beginning with the keyword `INTEGER`.

The effect is that inside the loop, the implied-`DO` variable (*ac-do-variable* or *data-i-do-variable*) has the type and kind specified by the *integer-type-spec*, regardless of the type and kind it might have outside the loop.

Note that if there is an entity with the same name in the scope containing the loop, it must be a scalar variable; for example, it cannot be a procedure or type name.

For example,

```
Real x
Print *, [ (x, Integer :: x = 1, 10) ]
```

is valid, and results in the output of the integer values 1 to 10. Similarly,

```
Real x, y(10)
Data (y(x),Integer::x=1,10,2) / 5*0 /
```

is valid, and initialises the odd-numbered elements of `Y` to zero.

This is mostly useful when subscripts may exceed the range of default integer kind, for example,

```
Real,Pointer :: a(:)
...
Print *, [ (f(a(i)), Integer(int64) :: i = Lbound(a,1,int64), Ubound(a,1,int64)) ]
```

will apply function `F` to each element of `A`, even if the bounds are, for example 20000000000:20000000005.

- The `DO CONCURRENT` construct can have locality specifiers `LOCAL`, `LOCAL_INIT`, and `SHARED`. These locality specifiers determine how a variable can be used within and without the loop, and cannot be applied to the loop index variables, which are always effectively `LOCAL`. There is also a `DEFAULT(NONE)` locality specifier, which requires all variables used in `DO CONCURRENT` to be given an explicit locality. The revised syntax of the `DO CONCURRENT` statement, ignoring labels and construct names, is:

  `DO CONCURRENT` *concurrent-header [locality-spec]...*

  where *concurrent-header* is the same as before, and each *locality-spec* is one of:

  `LOCAL (` *variable-name-list* `)`
  `LOCAL_INIT (` *variable-name-list* `)`
  `SHARED (` *variable-name-list* `)`
  `DEFAULT (NONE)`

  A variable that appears in a `LOCAL` or `LOCAL_INIT` specifier must be a rather ordinary variable: it must not have the `ALLOCATABLE` or `OPTIONAL` attribute, must not have an allocatable ultimate component, and must not be a coarray or an assumed-size array. If it is polymorphic, it must have the `POINTER` attribute. Finally, it must be permitted to appear in a variable definition context: for example, it cannot be `INTENT(IN)`. The effect of `LOCAL` and `LOCAL_INIT` is that the variable inside the construct is completely separate from the one outside the construct; if `LOCAL`, it begins each iteration undefined, and if `LOCAL_INIT`, it begins each iteration with the value of the outside variable. This ensures that `LOCAL` and `LOCAL_INIT` variables cannot cause any dependency between iterations.

  A variable that is `SHARED` is the same variable inside the construct as outside. If it is given a value by any iteration, it must not be referenced or given a value by any other iteration. If it is allocatable or a pointer, it similarly must only be allocated, deallocated, or pointer-assigned by a single iteration. If a discontiguous array is `SHARED`, it must not be passed as an actual argument to a contiguous dummy argument (i.e. the dummy must be assumed-shape or a pointer, and must not have the `CONTIGUOUS` attribute).

  Providing locality specifiers for variables in a `DO CONCURRENT` construct not only improves the readability of the code, but makes it easier for the compiler to parallelise the loop, perhaps with a suitable compiler option. (At Release 7.2, the NAG Fortran Compiler has no such option.)

- The arithmetic `IF` statement is treated as a Deleted feature. (This is because the behaviour when the expression is an IEEE NaN is undefined, and can have no good definition.) For example, if the file `del.f90` contains

```
Subroutine sub(x)
    Real,Intent(In) :: x
    If (x) 1,2,3
1   Stop 1
2   Stop 2
3   Stop 3
End Subroutine
```

this warning message will be produced:

```
Deleted feature used: del.f90, line 3: Arithmetic IF statement
```

If the *−Error=Deleted* option is used, this will be treated as an error.

- The `DO` construct with a label is considered to be Obsolescent (it is effectively replaced by the `END DO` statement and construct labels). Furthermore, the non-block `DO` construct is treated as a Deleted feature. A non-block `DO` is either two or more nested `DO` loops with a shared `DO` termination label, or a `DO` loop with a terminating statement other than `END DO` or `CONTINUE`. (This is because these are hard to understand, error-prone, and better functionality has been available via the block `DO` construct since Fortran 90.) For example, if the file `obsdel.f90` contains

```
Subroutine sub(w,x,y)
    Real,Intent(InOut) :: w(:),x(:,:), y(:)
    Integer i,j
    Do 100 i=1,Size(w)
        w(i) = w(i)**2 + 4*w(i) - 4
100 Continue
    Do 200 j=1,Size(x,2)
        Do 200 i=1,Size(x,1)
            If (x(i,j)<0) Go To 200
            x(i,j) = Sqrt(x(i,j)+1)
200 Continue
    Do 300 i=1,Size(y)
        If (y(i)<0) Go To 300
        y(i) = Log(y(i))
    300 Print *,y(i)
End Subroutine
```

  these warning messages will be produced:

```
Obsolescent: obsdel.f90, line 4: DO statement with label (100)
Obsolescent: obsdel.f90, line 7: DO statement with label (200)
Obsolescent: obsdel.f90, line 8: DO statement with label (200)
Deleted feature used: obsdel.f90, line 11: 200 is a shared DO termination label
Obsolescent: obsdel.f90, line 12: DO statement with label (300)
Deleted feature used: obsdel.f90, line 15: DO 300 ends neither with CONTINUE nor ENDDO
```

- The `FORALL` statement and construct are considered to be Obsolescent. This is because it usually has worse performance than ordinary `DO` or `DO CONCURRENT`. For example, if the file `obs.f90` contains

```
Subroutine sub(a,b,c)
    Real,Intent(InOut) :: a(:)
    Real,Intent(In) :: b(:),c
    Integer i
    Forall(i=1:Size(a))
        a(i) = b(i)**2 - c
    End Forall
End Subroutine
```

  this warning message will be produced:

```
Obsolescent: obs.f90, line 5: FORALL construct
```

- If the Fortran language level is 2018 or higher (the default), extension messages are produced for the use of non-standard intrinsic modules such as `F90_KIND` or `OMP_LIB`.

5 Fortran 2023 support

The NAG Fortran compiler supports the features from the recently revised and published Fortran 2023 standard that are listed below.

- A line in free source form can be up to 10000 (ten thousand) characters long. If the *−f2023* option was not used, an extension message will be produced for any line longer than 132 characters. (The NAG Fortran compiler continues to accept lines of any length, but lines longer than 10000 characters are reported as a NAG extension, not a Fortran 2023 extension.)

- The intrinsic inquiry function `SELECTED_LOGICAL_KIND` returns the kind value for a specified size of Logical. It has the following syntax:

  `SELECTED_LOGICAL_KIND ( BITS )`

  `BITS` : scalar Integer;

  Result : scalar Integer of default kind.

  The result is the kind type parameter value for type Logical that specifies a kind whose size is at least `BITS` bits; if `BITS` is greater than the storage size of the largest kind of Logical, the result is −1. Thus any value of `BITS` less than or equal to eight will return the kind of a single-byte Logical (as all known compilers support single-byte Logical). With the NAG Fortran compiler, the result will be −1 if `BITS` is greater than 64, as its largest supported Logical kind has 64 bits.

- The standard intrinsic module `ISO_FORTRAN_ENV` contains additional named constants, supplying the kind type parameter values for Logical kinds with the indicated bit sizes: `LOGICAL8`, `LOGICAL16`, `LOGICAL32` and `LOGICAL64`. If there is no kind of Logical whose storage size is exactly the size indicated, the constant has the value −1.

- The `AT` edit descriptor can be used for output of character data. The character data are output with trailing blanks omitted, as if each element of the output item were surrounded with a call to the intrinsic function `TRIM`.

  For example,

  ```
  Character(100) :: a(3)
  a(1) = 'o'
  a(2) = 'ka'
  a(3) = 'y'
  Print '(8X,3AT)', a
  ```

  will display the line

  ```
  okay
  ```

  with no trailing blanks.

  The `AT` edit descriptor is not allowed for input (`READ` statements).

6 Additional OpenMP support

OpenMP 4.0 and 4.5 are partially supported at this time. This includes the `SIMD` and `TARGET` constructs, including `DO SIMD` and `TARGET DATA`, and clauses such as the `LINEAR` clause. A forthcoming update will complete the support of OpenMP 4.0 and 4.5.

7 Additional error checking

- A warning message is issued if the field width in an edit descriptor, either in a `FORMAT` statement or in a constant character string used as a format, might be too small for output of some values. For example, the program

  ```
  Program na
      Read *,x
      Print 100,x
  100 Format('X has the value ',E9.3)
  End Program
  ```

  will produce the warning message

  ```
  Warning: na.f90, line 4: In E9.3 the width 9 may be too small for output of some numbers
  ```

  If the field width is too small for output of any finite number (an IEEE NaN only needs a width of three), the message will say that, e.g. if we change `E9.3` in the example to `E9.7`, it will produce the warning

  ```
  Warning: na.f90, line 4: Field width too small for output of finite numbers - the E9.7
      edit descriptor will produce all asterisks
  ```

  and if the field width is less than three, the message is again changed, e.g. for `E2.1`, the warning is

```
Warning: na.f90, line 4: Field width of 2 for the E edit descriptor will inevitably
    produce all asterisks as output
```

- Appearance of the `REC=` or `POS=` specifiers with the default unit, e.g. with `UNIT=*`, is detected as an error at compile time. That unit is a sequential formatted unit, and those specifiers cannot be used with sequential input/output. For example, compiling

  ```
  Program badio
      Write(*,'(A)',Rec=999) 'Oops'
  End Program
  ```

  produces the error

  ```
  Error: badio.f90, line 2: REC= and UNIT=* are not compatible
  ```

- Calling a procedure with an implicit interface, when that procedure appears elsewhere in the file and is elemental, is detected as an error at compile time. For example, compiling

  ```
  Program c
      Real a(10)
      Call d(a,1.0)
      Print *,a
  End Program
  Elemental Subroutine d(x,y)
      Real,Intent(Out) :: x
      Real,Intent(In) :: y
      x = y
  End Subroutine
  ```

  produces the error

  ```
  Error: c.f90: Explicit interface required for ELEMENTAL procedure D referenced from C
  ```

- The most obvious cases of modifying an active team variable in a `CHANGE TEAM` construct are detected as an error at compile time. The Fortran standard prohibits such changes, to allow a compiler to use the active team variable throughout the construct without any need to make a copy of its data. For example, compiling

  ```
  Subroutine teamswap(team1,team2)
    Use Iso_Fortran_Env
    Type(team_type),Intent(InOut) :: team1,team2
    Type(team_type) tmp
    Change Team (team2)
      tmp = team1     ! Okay.
      team1 = team2   ! Okay.
      team2 = tmp     ! BAD
    End Team
  End Subroutine
  ```

  produces the error message

  ```
  Error: teamswap.f90, line 8: Variable TEAM2 on left-hand side of assignment statement
         is the active team value for the CHANGE TEAM statement at line 5 of teamswap.f90
  ```

8 Other enhancements

- The low-order bits in double precision and quad precision `RANDOM_NUMBER` now have complete entropy. In previous releases, although the number sequence had complete entropy, the low-order bits (21 bits in double precision, and 53 bits in quad precision) were not random. (This would only affect a simulation if it needed more than 32 bits of randomness in each individual double precision value.)

  The new method does take significantly longer than before to produce the pseudo-random numbers, but the performance is still competitive with other compilers. The older, faster method may be selected by the *−random=5.3* option, and the new method may be confirmed with the *−random=7.2* option. These options affect use of `RANDOM_NUMBER` in the file(s) being compiled only; separate files may be compiled with different options and combined in the resulting program.

- The *−gline* option, which causes a traceback to be produced on error termination, can now be used with either the *−coarray=cosmp* option or the *−openmp* option. In CoSMP mode, the number of the image (in the initial team) that caused error termination will be reported, e.g.

  ```
  ERROR STOP: failatend
  pco391.f90, line 20: Error occurred in PCO391:SUB
  pco391.f90, line 18: Called by PCO391:SUB
  pco391.f90, line 11: Called by PCO391:TEST
  pco391.f90, line 7: Called by PCO391
  Error termination initiated by image 2
  ```

  and in OpenMP mode, the thread number created by a `PARALLEL` construct is reported, e.g.

  ```
  ERROR STOP: failatend
  suy008.f90, line 19: Error occurred in SUY008:SUB
  suy008.f90, line 17: Called by SUY008:SUB
  suy008.f90, line 10: Thread 8 created by OpenMP PARALLEL construct
  suy008.f90, line 8: Called by SUY008:TEST
  suy008.f90, line 2: Called by SUY008
  ```

- Comments extending past the standard line length are reported separately from statement text that extends past the standard line length.

- The *−xldarg* option passes the next argument on the command line to the linker phase in the position where it occurs, and without any translation (unlike the *−Wl,* option, which takes a comma-separated option list).

  For example,

  ```
  nagfor a.o -xldarg -pathlist=abc,xyz b.lib
  ```

  will typically invoke the linker phase as

  ```
  gcc a.o -pathlist=abc,xyz b.lib NAG-rts-specifications
  ```

  whereas

  ```
  nagfor a.o -Wl,-pathlist=abc,xyz b.lib
  ```

  would typically invoke the linker phase as

  ```
  gcc a.o b.lib NAG-rts-specifications -pathlist=abc xyz
  ```

- The *−fpplonglines* option directs the `fpp` preprocessor not to break output lines that are longer than 132 characters (in free source form). This may be useful if the output of `fpp` is to be passed to another tool, perhaps the `fpp` preprocessor itself, that does not expect `#line` directives in the middle of a token that is continued across lines.

- The *−w=longlines* option suppresses warnings about lines in free source form that are longer than permitted by the Fortran standard; this is 132 characters up to Fortran 2018, and 10000 characters for Fortran 2023.

  The NAG Fortran Compiler accepts free source form lines of any length.

- The *−u=all* option specifies that all *−u=* sub-options are active. In this release, this is *−u=external*, *−u=locality*, *−u=sharing*, and *−u=type*.

- The *−u=external* option specifies that `IMPLICIT NONE (EXTERNAL)` is in effect. This requires that external and dummy procedures must either have an explicit interface, or be explicitly given the `EXTERNAL` attribute.

  For example, if the file `bad.f90` contains

  ```
  Program bad
      Call oops
  End Program
  ```

  compiling it with the *−u=external* option will produce the error

  ```
  Error: bad.f90, line 3: External procedure OOPS does not have the EXTERNAL attribute
  ```

- The *−u=locality* specifies that all `DO CONCURRENT` constructs are treated as if they had specified `DEFAULT (NONE)`; that locality specifier requires all variables referenced in the `DO CONCURRENT` construct to have explicit locality.

  For example, if the file `b.f90` contains

  ```
  Program b
      Real x(100)
      Do Concurrent(i=1:100)
          x(i) = i**2.0
      End Do
      Print *,x
  End Program
  ```

  compiling it with the *−u=locality* option will produce the error

  ```
  Error: b.f90, line 4: Locality not specified for X in DO CONCURRENT with DEFAULT(NONE)
  ```

- The *−Warn=double_real_literal* option reports the presence of default double precision literal constants in the program. These are real literal constants with a D exponent letter (and thus, no kind specifier). The report is as a Note, unless inexactness of the conversion of the decimal to default double precision is detected, in which case it is a warning. For example, with the program

  ```
  Program dlits
      Print *,123d0
      Print *,0.123d0
  End Program
  ```

  the reported warnings are as follows:

  ```
  Note: dlits.f90, line 2: Default double precision literal constant 123D0
  Warning: dlits.f90, line 3: Inexact default double precision literal constant 0.123D0
  ```

  This option may be useful to detect the presence of default double precision literal constants in a program that is intended to have a specified kind type parameter that might not be double precision.

- The *−Warn=unkind_real_literal* option reports default real literal constants in the program that have no kind specifier. The report is as a Note, unless inexactness of the conversion of the decimal to default real is detected as being inexact, in which case it is a warning. For example, with the program

  ```
  Program lits
      Print *,123.0
      Print *,0.123
  End Program
  ```

  the reported warnings are as follows:

  ```
  Note: lits.f90, line 2: Real literal constant 123.0 has no kind specifier
  Warning: lits.f90, line 3: Inexact real literal constant 0.123 has no kind specifier
  ```

  This option may be useful to detect the presence of default (single precision) real literal constants in a program that is intended to be in double precision, or to have a specified kind type parameter that might not be single precision.

- The *−Error=**class*** option specifies treating warning messages in *class* as errors. The value of *class* must be one of the following (not case-sensitive):

Ancient	use of an obsolete and non-standard FORTRAN 77 extension;
Deleted	use of a feature that has been deleted from the Fortran standard;
Extension	use of a feature that is an extension to the Fortran standard;
Obsolescent	use of a feature that the Fortran standard says is obsolescent;
Questionable	valid but questionable usage that may indicate a programming error;
Warning	any warning-class message other than the above;
Any_Warning	any of the above.

  Note that *−Error=Obsolescent* implies *−Error=Deleted*, as all deleted features were previously obsolescent. Also, *−Error=Extension* implies *−Error=Ancient*, as ancient obsolete extensions are extensions.

  Also note that messages do not have their prefix changed, e.g. a warning message will still begin ‘`Warning:`’, but if the *−colour* option is used, the message colour will be coloured as an error.

- The *−strict95* option, which produced obsolescence (warning) messages for the use of ‘`CHARACTER*`’ syntax, is now enabled by default, and is thus ignored. Instead, there is a new option *−nonstrict*, which suppresses those messages, and also suppresses extension messages when the declared type of a value for a polymorphic allocatable component is an extension of the declared type of the component (this was always intended to be allowed, but there was a wording error in Fortran 2003 which was not corrected until Fortran 2018).

- Additional *−target=machine* options are available on Linux and Windows; *machine* may be **nehalem**, **westmere**, **sandybridge**, **ivybridge**, **haswell**, **broadwell**, **skylake**, **cannonlake**, **icelake**, **zen**, or **native** (which targets the current machine).

- The polish option *−elcase=**X*** sets the case to use for exponent letters (`E` or `D`): X must be one of **Asis**, **lowercase**, **UPPERCASE**, or an abbreviation thereof; the default is *−elcase=UPPERCASE*. The interpretation of X is not case-sensitive (e.g. *−elcase=u* is the same as *−elcase=U*). Note that *−elcase=Asis* is only available for basic polishing (*=polish*), not in Enhanced Polish (*=epolish*) or any other tool (e.g. *=unifyprecision*).

- The polish option *−canonicalise_floating_literals* specifies canonicalisation of floating-point literal constants. The canonical form of a floating-point literal always has a decimal point, with at least one digit before it and after it. If it has an exponent letter of `E`, the exponent is always non-zero. If there is an exponent, the exponent letter is always followed by a sign, and the exponent value has no leading zero. For example,

  ```
  Print *, .1, 1e0, 3d0, 2d-04
  ```

  would be reformatted (with *−elcase=U*) as:

  ```
  Print *, 0.1, 1.0, 3.0D+0, 2.0D-4
  ```

- When the *−openmp* option is used, Polish now auto-terminates OpenMP `DO` constructs, and indents the body thereof. For example, polishing

  ```
  !$OMP do
  do i=1,10
  a(i) = i
  end do
  ```

  previously produced

  ```
  !$Omp Do
  Do i=1,10
    a(i) = i
  End Do
  ```

  but now produces

  ```
  !$Omp Do
    Do i=1,10
      a(i) = i
    End Do
  !$Omp End Do
  ```

- Enhanced polish can add parentheses to argumentless `SUBROUTINE` statements and `CALL` statements. By default, the parentheses are omitted (except for a `SUBROUTINE` statement with a `BIND(C)` clause). The *−subroutine_parens* option will cause the parentheses to be produced.

  For example, using enhanced polish with *−subroutine_parens* (and no other option) on

  ```
  CALL mysub
  ```

  will produce the output

  ```
  Call mysub()
  ```

- When generating Makefile dependencies, the *−odir* `dir` option specifies that the object files will be located in a different directory (not the current working directory). For example, the object file that depends on “`file.f90`” will be considered to be “*dir*`/file.o`” instead of simply “`file.o`”.