NAG Fortran Compiler, Release 7.1

December 21, 2022

NAG® Fortran Compiler

11^{th} Edition – December 2021

NAG Ltd
30 St Giles'
OXFORD
OX1 3LE
United Kingdom
Tel: +44 (0)1865 511245 (general)
Tel: +44 (0)1865 311744 (support)
Web: www.nag.com

Nihon NAG KK
Hatchobori Frontier Building 2F
4-9-9
Hatchobori
Chuo-ku
Tokyo
104-0032
Japan
Tel: +81 (0)3 5542 6311
Fax: +81 (0)3 5542 6312
Web: www.nag-j.co.jp

NAG Inc
801 Warrenville Road, Suite 185
Lisle, IL 60532-4332
USA
Tel: +1 630 971 2337
Fax: +1 630 971 2706
Web: www.nag.com

NAG also has a number of distributors throughout the world. Please contact NAG for further details.

Introduction

1 Introduction to the Compiler

The NAG Fortran Compiler is based on the NAGWare f90 Compiler which was the world's first Fortran 90 compiler. The design goals of the development were to produce a compiler with the following characteristics:

- compiles standard Fortran to host-compatible C;
- good speed of compilation, reasonable efficiency of execution;
- good error checking, comprehensible error messages;
- full standard implementation, standard-conforming compiler (i.e. all constraints identified);
- modular construction;
- compiler written in C;
- maintainability, portability and re-usability.

The compiler is multi-pass; the passes have been kept distinct to improve maintainability and to allow re-use of the components.

Pass 1: Lexical and syntactic analysis, build symbol table and abstract syntax tree.

Pass 2: Semantic analysis, annotate parse tree and fill in symbol table; all major error and constraint detection takes place in this pass.

Pass 3: Code generation by parse tree transformation.

Pass 4: Code output, generate declarations and flatten transformed parse tree to C source code.

Pass 5: Compilation using the host operating system's C compiler.

Pass 6: Linking to executable code using the host system's linker, including linking in the Fortran run-time libraries.

1.1 Other Fortran-related Activities at NAG

NAG has released several Fortran-based numerical procedure libraries: the Fortran Library, SMP Library, Parallel Library and the Fortran 90 Library. A number of implementations of these libraries, using the NAG compiler and other compilers, are available.

NAG has supported the development and standardisation of Modern Fortran, participating both in ISO/IEC JTC1/SC22 Working Group 5, and the technical development committee INCITS/PL22.3. The head of the NAG compiler team, Malcolm Cohen, is the current Project Editor of the ISO/IEC Fortran standard.

From this it can be seen that NAG is committed to Fortran.

1.2 This Manual

This is the documentation for the NAG Fortran Compiler. This is not intended to be a language description or tutorial, but rather a guide to the use of the software and a quick reference for some of the features of the language.

The compiler is a full implementation of the Fortran 2008 programming language [ISO/IEC 1539-1:2010(E)] and many features from the latest Fortran 2018 standard [ISO/IEC 1539-1:2018(E)].

Using the Compiler

2 Usage

nagfor [mode] [option]... *file*...

3 Description

nagfor is the interface to the NAG Fortran Compiler system. The compiler translates programs written in Fortran into executable programs, relocatable binary modules, assembler source files or C source files.

The *mode* determines the action performed, and can be one of

=C Compile (and/or link) C source files, acting as the *companion processor*; this passes options to the C compiler that are suitable for the ABI and/or compatibility mode options specified, and differs from the *=compiler* mode in that it does not set NAG-specific macro definitions or alter the `#include` file search path to include the compiler library directory.

=compiler Compile (and/or link) the files; this is the default mode if none is specified.

=callgraph Produce a callgraph of the Fortran routines in the files (see the Producing a Call Graph section).

=depend Produce a dependency analysis of the Fortran files (see the Dependency Analysis section).

=epolish Pretty-print (polish) the Fortran files using the Enhanced Polisher (see the Enhanced Source File Polishing section).

=interfaces Produce a module or `INCLUDE` file containing procedure interfaces (see the Generating Interfaces section).

=polish Pretty-print (polish) the Fortran files (see the Source File Polishing section).

=unifyprecision
Unify the precision of floating-point and complex entities in the files (see the Unifying Precision section).

Options that do not apply to the current mode of operation (e.g. polish options when the mode is for compilation) are ignored.

The mode can also be specified as *−mode=***mode**; this may be useful if the user's command processor has a special meaning for equals signs (e.g. zsh).

4 File Types

A file ending in '`.f90`' or '`.f95`' is taken to be a Fortran free-form source file, a file ending in '`.f`', '`.for`' or '`.ftn`' is taken to be a Fortran fixed-form source file; these assumptions can be overridden with the *−fixed* or *−free* option. A file ending in '`.ff90`' or '`.ff95`' is taken to be a free-form file requiring preprocessing by fpp, and a file ending in '`.ff`' is taken to be a fixed-form file requiring preprocessing by fpp. On Unix, a file ending in '`.F90`' or '`.F95`' is taken to be a free-form file requiring preprocessing by fpp, and a file ending in '`.F`' is taken to be a fixed-form files requiring preprocessing by fpp. (Note that on MacOS and Windows, the file system is not case-sensitive so uppercase and lowercase letters are equivalent in filenames including in the suffixes.)

If a filename without a suffix is provided nagfor will look for a file with the suffix '`.f95`', and if that does not exist, the suffix '`.f90`'.

A file ending in '`.c`' is taken to be a C source file. In the *=compiler* mode, this is assumed to be the output from the compiler with the *−S* option, and the C compiler is passed *−D* and *−I* options suitable for compiling such a file. In the *=C* mode, it is assumed to be a file for the *companion processor*; no *−D* is passed, and only *−I* options specified by the user. In both cases, options are passed to the C compiler according to the ABI and compatibility mode options.

Non-intrinsic modules, `INCLUDE` files and `#include` files are expected to exist in the current working directory or in a directory named by an *−I* option.

5 Compiler Options

–132 Increase the length of each fixed source form input line from 72 characters to 132 characters. This has no effect on free source form input.

–abi=*abi*

(Linux x86-64 only) Specify the ABI to compile for, either **32** (the 32-bit ABI), or one of the 64-bit ABIs: **64c** or **64t**. The differences between the two 64-bit ABIs are as follows:

ABI	Object size represented in	Character length represented in
*–abi=***64t**	64 bits	32 bits
*–abi=***64c**	64 bits	64 bits

Programs compiled with *–abi=32* will run on any x86 Linux system; those compiled with any 64-bit ABI will only run on a 64-bit kernel.

The default is *–abi=64t*. The *–abi=64c* option is compatible with the ABI used by Intel Fortran.

–abi=*abi*

(Windows only) Specify the ABI to compile for, either **32** (the 32-bit ABI) or **64** (the 64-bit ABI). The default is *–abi=64* on Windows x64; on 32-bit Windows the default is *–abi=32* and the *–abi=64* option is not available.

–align=*alignment*

(MacOS only) Specify the alignment of variables and components, which must be one of:

natural (natural alignment for best performance; this can alter the interpretation of `COMMON` block or `SEQUENCE` type layout in a non-standard-conforming manner), or

standard (use standard-conforming alignment; this is the default).

The whole program should be compiled with the same alignment option.

–B*binding*

Specify **static** or **dynamic** binding. This only has effect if specified during the link phase. The default is dynamic binding. On SPARC and SG/Irix, these options are positional and can be used to selectively bind some libraries statically and some dynamically. This option is not available on IBM z9 Open Edition.

–c Compile only (produce .o file for each source file), do not link the .o files to produce an executable file. This option is equivalent to *–otype=obj*.

–C Compile with all but the most expensive runtime checks; this omits the *–C=alias*, *–C=dangling*, *–C=intovf* and *–C=undefined* options.

–C=*check*

Compile checking code according to the value of *check*, which must be one of:

alias	(check for assignments to aliased dummy arguments),
all	(perform all checks except for *–C=undefined*),
array	(check array bounds),
bits	(check bit intrinsic arguments),
calls	(check procedure references),
dangling	(check for dangling pointers),
do	(check DO loops for zero step values and illicit modification of the index variable via host association),
intovf	(check for integer overflow),
none	(do no checking: this is the default),
present	(check OPTIONAL references),
pointer	(check POINTER references),
recursion	(check for invalid recursion) or
undefined	(check for undefined variables).

The *–C=alias* option will produce a runtime error when it is detected that assignment to a dummy argument affects another dummy argument. At this release this is only detected for scalar dummy arguments.

The *–C=dangling* option will produce a runtime error when a dangling pointer is used; additionally, if the runtime option 'show_dangling' is set, a warning will be produced at the time the pointer becomes dangling (see Runtime Environment Variables for further information).

The *−C=undefined* option is subject to a number of limitations; in particular, it is not binary compatible with Fortran code compiled without that option, and is not compatible with calling C code via a `BIND(C)` interface. See the Undefined Variable Detection section for further details.

−coarray
This option is short for *−coarray=cosmp*.

−coarray=*mode*
Set the coarray operation mode to *mode*, which must be **single** for Single Image mode, or **cosmp** for Co-SMP mode; the option is not case-sensitive. The default is *−coarray=single*.

In Single Image mode (*−coarray=single*), all coarray syntax is accepted, but execution will not be in parallel: only a single image is supported.

In Co-SMP mode (*−coarray=cosmp*), parallel execution of multiple images on an SMP machine is supported. The maximum number of images in this mode is 1000. If the *−num_images=N* option is used, the default number of images to execute is *N*; with *−num_images=auto*, the default number of images is the number of hardware threads available on the processor. Note that the number of images may exceed the number of hardware threads, but doing so will only improve performance if images spend a lot of time waiting (e.g. for synchronisation or input/output). The *−num_images=* option may be overridden by the runtime environment variable `NAGFORTRAN_NUM_IMAGES`.

Code that uses any coarray features (coarray syntax or image control statements) or that has any common blocks or global (saved or initialised) variables, and that is compiled with *−coarray=single*, **must never** be executed in Co-SMP mode, as it will not work correctly. Code that avoids those features, and which is intended to work both in Co-SMP mode and single image mode, should be compiled with the *−thread_safe* option.

The *−coarray=cosmp* option cannot be used at the same time as *−gline* or *−openmp*. The *−coarray=cosmp* option may be specified with the *−C=undefined* option, but it will automatically disable the latter option.

−colour Colour the message output from the compiler using ANSI escape sequences and the default foreground colouring scheme which is: red for error messages (including fatal errors), blue for warning messages and green for information messages.

−colour=*scheme*
Colour the message output from the compiler according to the specified *scheme*. This is a comma-separated list of colour specifications, each consisting of a message category name ("error", "warn" or "info") followed by a colon and the foreground colour name, optionally followed by a plus sign and the background colour name. The colouring for unspecified categories will be the default.

Colours are: black, red, green, yellow, blue, magenta, cyan and white.

E.g. *−colour=error:red+blue,warn:cyan,info:magenta+yellow*
would be a rather garish colour scheme.

−compatible
Make external linkages compatible with other compilers where possible; on Windows this is Microsoft Fortran (32-bit mode) or Intel Fortran (64-bit mode), on MacOS and Linux this is g77, g95 and gfortran, and on other systems this is the operating system vendor's compiler. This affects the naming convention and procedure calling convention (for example, on Windows it causes use of the "STDCALL" calling convention that is commonly used for most DLLs, and the names are in upper case with no added trailing underscore). On Windows in 64-bit mode, *−compatible* is always in effect.

−convert=*format*
Set the default conversion mode for unformatted files to *format*. This format may be overridden by an explicit `CONVERT=` specifier in the `OPEN` statement, or by the environment variable `FORT_CONVERT`*n* (where *n* is the unit number). The value of *format* must be one of the following (not case-sensitive):

Format	Description
BIG_ENDIAN	synonym for BIG_IEEE
BIG_IEEE_DD	big-endian with IEEE floating-point, quad precision is double-double
BIG_IEEE	big-endian with IEEE floating-point, including quad precision
BIG_NATIVE	big-endian with native floating-point format
LITTLE_ENDIAN	synonym for LITTLE_IEEE
LITTLE_IEEE_DD	little-endian with IEEE floating-point, quad precision is double-double
LITTLE_IEEE	little-endian with IEEE floating-point, including quad precision
LITTLE_NATIVE	little-endian with native floating-point format
NATIVE	no conversion (the default)

–D*name*
Defines *name* to fpp as a preprocessor variable. This only affects files that are being preprocessed by fpp.

–d_lines In fixed form only, accept lines beginning with "D" as normal Fortran statements, replacing the D with a space. Without this option, such lines are treated as comments.

–dcfuns Enable recognition of non-standard double precision complex intrinsic functions. These act as specific versions of the standard generic intrinsics as follows:

Non-standard	Equivalent Standard Fortran Generic Intrinsic Function
CDABS(A)	ABS(A)
DCMPLX(X,Y)	CMPLX(X,Y,KIND=KIND(0d0))
DCONJG(Z)	CONJG(Z)
DIMAG(Z)	AIMAG(Z)
DREAL(Z)	REAL(Z) or DBLE(Z)

–double Double the size of default INTEGER, LOGICAL, REAL and COMPLEX. Entities specified with explicit kind numbers or byte lengths are unaffected. If quadruple precision REAL is available, the size of DOUBLE PRECISION is also doubled.

–dryrun Show but do not execute commands constructed by the compiler driver.

–dusty Allows the compilation and execution of "legacy" software by downgrading the category of common errors found in such software from "Error" to "Warning" (which may then be suppressed entirely with the *–w* option). This option disables *–C=calls*, and also enables Hollerith i/o (see the *–hollerith_io* option).

–encoding=*charset*
Specifies that the encoding system of the Fortran source files is *charset*, which must be one of **ISO_Latin_1**, **Shift_JIS** or **UTF_8**. If this option is not specified, the default encoding is UTF-8 for Fortran source files that begin with a UTF-8 Byte Order Mark, and ISO Latin-1 (if the language setting is English) or Shift-JIS (if the language setting is Japanese) for other Fortran source files.

–english Produce compiler messages in English (default).

–F Preprocess only, do not compile. Each file that is preprocessed will produce an output file of the same name with the suffix replaced by .f, .f90 or .f95 according to the suffix of the input file. This option is equivalent to *–otype=Fortran*.

–f90_sign
Use the Fortran 77/90 version of the SIGN intrinsic instead of the Fortran 95 one (they differ in the treatment of negative zero).

–f95 Specify that the base language is Fortran 95. This only affects extension message generation (Fortran 2003 and 2008 features will be reported as extensions).

–f2003 Specify that the base language is Fortran 2003. This only affects extension message generation (Fortran 2008 features will be reported as extensions).

–f2008 Specify that the base language is Fortran 2008. This is the default.

–f2018 Specify that the base language is Fortran 2018. This implies the *–recursive* option.

–fixed Interpret all Fortran source files according to fixed-form rules.

–float-store
(Gnu C based systems only) Do not store floating-point variables in registers on machines with floating-point registers wider than 64 bits. This can avoid problems with excess precision.

–fpp Preprocess the source files using fpp even if the suffix would normally indicate an ordinary Fortran file.

–framework *f*
(MacOS only) Use framework *f* during linking.

–free Interpret all Fortran source files according to free-form rules.

–g Produce information for interactive debugging by the host system debugger.

–g90 Produce debugging information for dbx90, a Fortran 90 aware front-end to the host system debugger. This produces a debug information (.g90) file for each Fortran source file. This option must be specified for both compilation and linking.

–gc Enables automatic garbage collection of the executable program. This option must be specified for both compilation and linking, and is unavailable on IBM z9 OpenEdition, MacOS, and Windows. It is incompatible with the *–thread_safe* and *–mtrace* options. For more details see the Automatic Garbage Collection section.

–gline Compile code to produce a traceback when a runtime error message is generated. Only routines compiled with this option will appear in such a traceback. This option increases both executable file size and execution time. It is incompatible with the *–thread_safe*, *–openmp* and *–coarray=cosmp* options.

For example:

```
Runtime Error: Invalid input for real editing
Program terminated by I/O error on unit 5 (Input_Unit,Formatted,Sequential)
main.f90, line 28: Error occurred in READ_DATA
main.f90, line 57: Called by READ_COORDS
main.f90, line 40: Called by INITIAL
main.f90, line 13: Called by $main$
```

–help Display a one-line summary of the options available for the current mode (*=compiler*, *=callgraph*, *=depend*, *=epolish*, *=interfaces*, *=polish* or *=unifyprecision*).

–hollerith_io
Enable Fortran-66 compatible input/output of character data stored in numeric variables using the `A` edit descriptor. This was superseded by the `CHARACTER` datatype in Fortran 77.

–I *pathname*
Add *pathname* to the list of directories which are to be searched for module information (`.mod`) files and `INCLUDE` files. The current working directory is always searched first, then any directories named in *–I* options, then the compiler's library directory (see the *–Qpath* option).

–i8 Set the size of default `INTEGER` and `LOGICAL` to 64 bits. This can be useful for switching between libraries that have 32-bit integer arguments (on one platform) and 64-bit integer arguments (on another platform), but which do not provide a named constant with the necessary `KIND` value.

This has no effect on default `REAL` and `COMPLEX` sizes, so the compiler is not standard-conforming in this mode.

–indirect *file*
Read the contents of *file* as additional arguments to the compiler driver. This option may also be given by "@*file*"; note in this case there is no space between the '@' and the file name.

In an indirect file, arguments may be given on separate lines; on a single line, multiple arguments may be separated by blanks. A blank can be included in an option or file name by putting the whole option or file name in quotes (`"`); this is the only quoting mechanism. An indirect file may reference other indirect files.

–ieee=*mode*
Set the mode of IEEE arithmetic operation according to *mode*, which must be one of **full**, **nonstd** or **stop**.

full enables all IEEE arithmetic facilities including non-stop arithmetic.

nonstd Disables non-stop arithmetic, terminating execution on floating overflow, division by zero or invalid operand. If the hardware supports it, this also disables IEEE gradual underflow, producing zero instead of a denormalised number; this can improve performance on some systems.

stop enables all IEEE arithmetic facilities except for non-stop arithmetic; execution will be terminated on floating overflow, division by zero or invalid operand.

The *−ieee* option must be specified when compiling the main program unit, and its effect is global. The default mode is *−ieee=stop*. For more details see the IEEE 754 Arithmetic Support section. This option is not available on IBM z9 Open Edition with hexadecimal floating point.

−info Request output of information messages, both "Info" and "Remark" (the least important). The default is to suppress these messages.

−kind=*option*

Specify the kind numbering system to be used; *option* must be one of **byte**, **sequential** or **unique**.

For *−kind=byte*, the kind numbers for `INTEGER`, `REAL` and `LOGICAL` will match the number of bytes of storage (e.g., default `REAL` is 4 and `DOUBLE PRECISION` is 8). Note that `COMPLEX` kind numbers are the same as its `REAL` components, and thus half of the total byte length in the entity.

For *−kind=sequential* (the default), the kind numbers for all datatypes are numbered sequentially from 1, increasing with precision (e.g., default `REAL` is 1 and `DOUBLE PRECISION` is 2).

For *−kind=unique*, the kind numbers are unique across all data types, so that a kind number for one data type cannot be accidentally used for another data type (except that `COMPLEX` and `REAL` are still the same). These kind numbers are all greater than 100 so do not match byte sizes either.

This option does not affect the interpretation of byte-length specifiers (an extension to Fortran 77).

−l*x* Link with library lib*x*.a. The linker will search for this library in the directories specified by −L*dir* options followed by the normal system directories (see the ld(1) command).

−L*dir* Add *dir* to the list of directories for library files (see the ld(1) command).

−M Produce module information files (`.mod` files) only. This option is equivalent to *−otype=mod*.

−max_internal_proc_instances=*N*

Set the maximum number of simultaneously active host instances of an internal procedure that is being passed as an actual argument, or assigned to a procedure pointer, to *N*. The default maximum is normally 30, and increased to 160 if either the *−openmp* or *−thread_safe* options are used.

−max_parameter_size=*N*

Set the maximum size of a `PARAMETER` to *N* MB (megabytes). *N* must be in the range 1 to 1048576 (1MB to 1TB); the default is 50 MB.

−maxcontin=*N*

Increase the limit on the number of continuation lines from 255 to *N*. This option will not decrease the limit below the standard number.

−mdir *dir*

Write any module information (`.mod`) files to directory *dir* instead of the current working directory.

−message_encoding=*charset*

Set the encoding scheme for compiler messages to *charset*, which must be one of **ISO_Latin_1**, **Shift_JIS** or **UTF_8** (not case-sensitive). The *−message_encoding=ISO_Latin_1* option is incompatible with the *−nihongo* option. The default message encoding is Shift_JIS on Windows and UTF_8 on other systems.

−mismatch

Downgrade consistency checking of procedure argument lists so that mismatches produce warning messages instead of error messages. This only affects calls to a routine which is not in the current file; calls to a routine in the file being compiled must still be correct. This option disables *−C=calls*.

−mismatch_all

Further downgrade consistency checking of procedure argument lists so that calls to routines in the same file which are incorrect will produce warnings instead of error messages. This option disables *−C=calls*.

−mtrace Trace memory allocation and deallocation. This option is a synonym for *−mtrace=on*.

–mtrace=*trace_opt_list*

Trace memory allocation and deallocation according to the value of *trace_opt_list*, which must be a comma separated list of one or more of:

address (display addresses),

all (all options except for **off**),

line (display file/line info if known),

off (disable tracing output),

on (enable tracing output),

paranoia (protect memory allocator data structures against the user program),

size (display size in bytes) or

verbose (all options except for **off** and **paranoia**).

This option should be specified during both compilation and linking, and is incompatible with the *–gc* option. For more details see the Memory Tracing section. The *–mtrace=paranoia* option is not available on IBM z9 Open Edition.

–nan Initialise `REAL` and `COMPLEX` variables to IEEE Signalling NaN, causing a runtime crash if the values are used before being set. This affects local variables, module variables, and `INTENT(OUT)` dummy arguments only; it does not affect variables in `COMMON` or `EQUIVALENCE`. This option is not available on IBM z9 Open Edition with hexadecimal floating point.

–nihongo

Produce compiler messages in Japanese (if necessary, the encoding can be changed by the *–message_encoding=* option). This option is not available on IBM z9 Open Edition.

–no_underflow_warning

Suppress the warning message that normally appears if a floating-point underflow occurred during execution. This option is only effective if specified when compiling the main program.

–nocheck_modtime

Do not check for `.mod` files being out of date.

–nomod Suppress module information (`.mod`) file production. Combining this with *–M* will produce no output (other than error and warning messages) at all, equivalent to *–otype=none*.

–noqueue

If no licence for the compiler is immediately available, exit with an error instead of queueing for it.

–num_images=*N*

Set the expected number of images the program will run with to *N*, which should be a number in the range 1 to 1000, 'auto', or 'unknown'.

In Single Image mode (*–coarray=single*), the only affect is on analysis of constant cosubscripts: if *N* is numeric, and they evaluate to an image index greater than *N*, an error will be produced. The effect of *–num_images=unknown* (or *–num_images=auto*) is to suppress such analysis.

In CoSMP mode (*–coarray=cosmp*), the effect is to specify the default number of images at execution time; this may be overridden by the runtime environment variable `NAGFORTRAN_NUM_IMAGES`. The effect of *–num_images=auto* (or *–num_images=unknown*) is to set the default number of images to the number of hardware threads on the processor. This option takes effect when compiling the main program.

The default in *–coarray=single* mode is *–num_images=1*, and the default in *–coarray=smp* mode is *–num_images=auto*.

–o *output*

Name the output file *output* instead of the default. If an executable is being produced the default is `a.out`; otherwise it is *file*.o with the *-c* option, *file*.c with the *-S* option, and *file*.f, *file*.f90 or *file*.f95 with the *-F* option, where *file* is the base part of the source file (i.e. with the suffix removed).

–O Normal optimisation, equivalent to *–O2*.

–O*N* Set the optimisation level to *N*. The optimisation levels are:

–O0 No optimisation. This is the default, and is recommended when debugging.

−O1 Minimal quick optimisation.

−O2 Normal optimisation.

−O3 Further optimisation.

−O4 Maximal optimisation.

−Oassumed
This is a synonym for *−Oassumed=contig*.

−Oassumed=*shape*
Optimises assumed-shape array dummy arguments according to the value of *shape*, which must be one of

always_contig
Optimised for contiguous actual arguments. If the actual argument is not contiguous a runtime error will occur (the compiler is not standard-conforming under this option).

contig Optimised for contiguous actual arguments; if the actual argument is not contiguous (i.e. it is an array section) a contiguous local copy is made. This may speed up array section accessing if a sufficiently large number of array element or array operations is performed (i.e. if the cost of making the local copy is less than the overhead of discontiguous array accesses), but usually makes such accesses slower. Note that this option does not affect dummy arguments with the `TARGET` attribute; these are always accessed via the dope vector.

section Optimised for low-moderate accesses to array section (discontiguous) actual arguments. This is the default.

Note that `CHARACTER` arrays are not affected by these options.

−Oblock=*N*
Specify the dimension of the blocks used for evaluating the `MATMUL` intrinsic. The default value (only for *−O1* and above) is system and datatype dependent.

−Onopropagate
Disable the optimisation of constant propagation. This is the default for *−O1* and lower.

−Onoteams
Generate coarray access code assuming that teams are not being used. This will produce incorrect results if executed while a `CHANGE TEAM` construct is active.

−Opropagate
Enable the optimisation of constant propagation. This is the default for *−O2* and higher.

−Orounding
Specify that the program does not alter the default rounding mode. This enables the use of faster code for the `ANINT` intrinsic.

−Ounroll=*N*
Specify the depth to which simple loops and array operations should be unrolled. The default is no unrolling (i.e. a depth of 1) for *−O0* and *−O1*, and a depth of 2 for *−O* and higher optimisation levels. It can be advantageous to disable the Fortran compiler's loop unrolling if the C compiler normally does a very good job itself — this can be accomplished with *−Ounroll=1*.

−Ounsafe
Perform possibly unsafe optimisations that may depend on the numerical stability of the program. On IBM z9 Open Edition this option, in conjunction with *−O4*, passes NOSTRICT to the C compiler.

−openmp
Enable OpenMP, producing extension messages for use of OpenMP features more recent than the currently fully-supported OpenMP version. In nagfor 7.1, this is equivalent to *−openmp=3.1*.

−openmp=*version*
Recognise OpenMP directives and link with the OpenMP support library. Produce extension messages for the use of OpenMP features more recent than OpenMP **version**, which must be equal to 3.0, 3.1, 4.0, 4.5, 5.0 or 5.1. For more details see the OpenMP Support section. This option is incompatible with the *−coarray=smp* option.

–otype=*filetype*
Specify the type of output file required to *filetype*, which must be one of

c	(C source file),
exe	(executable file),
fortran	(Fortran source file),
mod	(module information file),
none	(no output file),
obj	(object file).

The *−c*, *−F* and *−M* options are equivalent to *−otype=obj*, *−otype=Fortran* and *−otype=mod* respectively.

–pg Compile code to generate profiling information which is written at run-time to an implementation-dependent file (usually *gmon.out* or *mon.out*). An execution profile may then be generated using *gprof*. This option must be specified for compilation and linking and may be unavailable on some implementations.

–pic Produce position-independent code (small model), for use in a shared library. If the shared library is too big for the small model, use *−PIC*. This option is not available on IBM z9 Open Edition.

–PIC Produce position-independent code (large model), for use in a shared library. This option is not available on IBM z9 Open Edition.

–quiet Suppress the compiler banner and the summary line, so that only diagnostic messages will appear.

–Qpath *pathname*
Change the compiler library pathname from its default location to *pathname*. (The default location on Unix is usually '`/usr/local/lib/NAG_Fortran`' or '`/opt/NAG_Fortran/lib`') This option is unnecessary on Windows as the installed location is automatically detected.

–r8 Double the size of default `REAL` and `COMPLEX`, and on machines for which quadruple-precision floating-point arithmetic is available, double the size of `DOUBLE PRECISION` (and the non-standard `DOUBLE COMPLEX`). `REAL` or `COMPLEX` specified with explicit `KIND` numbers or byte lengths are unaffected — but since the `KIND` intrinsic returns the correct values, `COMPLEX(KIND(0d0))` on a machine with quad-precision floating-point will correctly select quad-precision `COMPLEX`.

This has no effect on `INTEGER` sizes, and so the compiler is not standard-conforming in this mode.

Note: This option has been superseded by the *−double* option which doubles the size of all numeric data types.

–recursive
Specifies that procedures are `RECURSIVE` by default. This option is implied by the *–f2018* option.

–round_hreal
Round all half precision operations to half precision. Without this option, half precision expressions are evaluated in single precision and only rounded to half precision when being assigned to a variable or passed as an actual argument to a non-intrinsic or non-mathematical procedure.

This option affects compile-time evaluation as well as runtime evaluation.

–s Strip symbol table information from the executable file. This option is only effective if specified during the link phase.

–S Produce assembler (actually C source code). The resulting .c file should be compiled with the NAG Fortran compiler, not with the C compiler directly. This option is equivalent to *−otype=c*.

–save This is equivalent to inserting the `SAVE` statement in all subprograms which are not pure, not declared `RECURSIVE`, and not `RECURSIVE` by default (see the *−recursive* option). It thus causes all non-automatic local variables in such subprograms to be statically allocated. It has no effect on variables in BLOCK constructs.

–strict95
Produce obsolescence warning messages for use of '`CHARACTER*`' syntax. This message is not produced by default since many programs contain this syntax.

–target=*machine*
Specify the machine for which code should be generated and optimised.

- For x86-32 (x86-compatible 32-bit mode compilation on Linux and Windows), *machine* may be one of

i486, i586, i686, pentium2, pentium3, pentium4, prescott
the specified Intel processor,
k6, k6-2, k6-3, k6-4, athlon, athlon-4, athlon-xp, athlon-mp
the specified AMD processor,
pentium (equivalent to *i586*) or
pentiumpro (equivalent to *i686*).

The default is to compile for pentium4 on Linux, and prescott on Windows.

- For x86-64 (x86-compatible 64-bit mode compilation on Linux, MacOS and Windows), *machine* may be **athlon64**, **nocona**, or **core2**.
- For Sun/SPARC, *machine* may be one of

V7 SPARCstation 1 et al,
V8 SPARCstation 2 et al,
super SuperSPARC,
ultra UltraSPARC or
native the current machine.

The default is to compile for SPARC V7.

Note that programs compiled for later versions of the architecture may not run, or may run much more slowly, on an earlier machine. The *−target=native* option is not available with gcc.

- For HP9000/700, *machine* may be one of

2.0 the specified revision of the PA-RISC architecture (default) or
native the current machine.

−tempdir ***directory***
Set the directory used for the compiler's temporary files to *directory*. The default is to use the directory named by the `TMPDIR` environment variable, or if that is not set, /tmp on Unix-like systems and the Windows temporary folder on Windows.

−thread_safe
Compile code for safe execution in a multi-threaded environment. This must be specified when compiling and also during the link phase. It is incompatible with the *−gc* and *−gline* options.

−time Report execution times for the various compilation phases.

−u Specify that `IMPLICIT NONE` is in effect by default, unless overridden by explicit `IMPLICIT` statements.

−u=sharing
Specify default sharing of `NONE` in OpenMP `PARALLEL` and `TASK` constructs (including in combined constructs such as `PARALLELDO`). This has the same effect as the `DEFAULT(NONE)` clause, unless overridden by an explicit `DEFAULT(...)` directive.

−unsharedrts
Bind with the unshared (static) version of the Fortran runtime system; this allows a dynamically linked executable to be run on systems where the NAG Fortran Compiler is not installed. This option is only effective if specified during the link phase.

−v Verbose. Print the name of each file as it is compiled.

−V Print version information about the compiler.

−w Suppress all warning messages. This option is a synonym for *−w=all*.

−w=*class*
Suppress the warning messages specified by *class*, which must be one of **all**, **alloctr**, **obs**, **ques**, **uda**, **uei**, **uep**, **uip**, **ulv**, **unreffed**, **unused**, **uparam**, **usf**, **usy**, **x77** or **x95**.

−w=all suppresses all warning messages;
−w=alloctr suppresses warning messages about the use of allocatable components, dummy arguments and functions;
−w=note Suppress informational Notes;
−w=obs suppresses warning messages about the use of obsolescent features;

−w=ques suppresses warning messages about questionable usage;

−w=uda suppresses warning messages about unused dummy arguments;

−w=uei suppresses warning messages about unused explicit imports;

−w=uep suppresses warning messages about unused external procedures;

−w=uip suppresses warning messages about unused intrinsic procedures;

−w=ulv suppresses warning messages about unused local variables;

−w=unreffed suppresses warning messages about variables set but never referenced;

−w=unused suppresses warning messages about unused entities — this is equivalent to '*−w=uda −w=uei −w=uep −w=uip −w=ulv −w=uparam −w=usf −w=usy*';

−w=uparam suppresses warning messages about unused `PARAMETER`s;

−w=usf suppresses warning messages about unused statement functions;

−w=usy suppresses warning messages about unused symbols;

−w=x77 suppresses extension warnings for obsolete but common extensions to Fortran 77 — these are TAB format, byte-length specifiers, Hollerith constants and D lines;

−w=x95 suppresses extension warnings for extensions to modern Fortran (not just Fortran 95) that are not part of any Fortran standard.

−W*options*

The *−W* option can be used to specify the path to use for a compilation component or to pass an option directly to such a component. The possible combinations are:

−W0=*path* Specify the path used for the Fortran Compiler front-end. Note that this does not affect the library directory; the *−Qpath* option should be used to specify that.

−Wc=*path* Specify the path to use for invoking the C compiler; this is used both for the final stage of compilation and for linking.

−Wc,*option* Pass *option* directly to the host C compiler when compiling (producing the .o file). Multiple options may be specified in a single *−Wc,* option by separating them with commas.

−Wl=*path* Specify the path to use for invoking the linker (producing the executable).

−Wl,*option* Pass *option* directly to the host C compiler when linking (producing the executable). Multiple options may be specified in a single *−Wl,* option by separating them with commas. A comma may be included in an option by repeating it, e.g. *−Wl,-filelist=file1,,file2,,file3* becomes the linker option *−filelist=file1,file2,file3*.

−Wp=*path* Specify the path to use for invoking the fpp preprocessor.

−Wp,*option* Pass *option* directly to fpp when preprocessing.

−Warn=*class*

Produce additional warning messages specified by *class*, which must be one of:

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

constant_coindexing warn if an image selector has constant cosubscripts;

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 operation or function 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*.

−wmismatch=*proc-name-list*
Specify a list of external procedures for which to suppress argument data type and arrayness consistency checking. The procedure names should be separated by commas, e.g. *−wmismatch=p_one,p2*. Unlike the *−mismatch* option, this only affects data type and arrayness checking, and no warning messages are produced.

−xlicinfo
Report on the availability of licences for the compiler instead of compiling anything. Also report the exact version of Kusari being used.

−xs (Sun/SPARC option only) Store the symbol tables in the executable (otherwise debugging is only possible if the object files are kept).

6 Files

***file*.a** Library of object files.

***file*.c** C source file.

***file*.f** Fortran source file in fixed format (obsolete).

***file*.f90** Fortran source file in free format.

***file*.f95** Fortran source file in free format.

***file*.ff** Preprocessor source file for fixed-form Fortran (obsolete).

***file*.F** (Unix) Preprocessor source file for fixed-form Fortran (obsolete).

***file*.ff90** Preprocessor source file for free-form Fortran.

***file*.F90** (Unix) Preprocessor source file for free-form Fortran.

***file*.ff95** Preprocessor source file for free-form Fortran.

***file*.F95** (Unix) Preprocessor source file for free-form Fortran.

***name*.mod** Compiled module information file; *name* is the name of the module in lower case.

***file*.o** Object file

/opt/NAG_Fortran/lib
Default NAG Fortran Compiler library directory on Sun Solaris (see *−Qpath*); referred to as *library* hereafter.

/usr/local/lib/NAG_Fortran
Default NAG Fortran Compiler library directory on other Unix-based operating systems.

C:\Program Files\NAG\EFBuilder 7.1\nagfor\lib
Default NAG Fortran Compiler library directory on 32-bit Windows.

C:\Program Files (x86)\NAG\EFBuilder 7.1\nagfor\lib
Default NAG Fortran Compiler library directory on 64-bit Windows.

***library*/f90_iostat.f90**
Source code for the `f90_iostat` module.

***library*/f90_kind.f90**
Source code for the `f90_kind` module.

***library*/f90_stat.f90**
Source code for the `f90_stat` module.

***library*/f90_util.f90**
A sample Fortran 90 program that displays implementation-specific information

***library*/iso_fortran_env.f90**
Source code for the `iso_fortran_env` module.

***library*/nagfmcheck.f90**
Source code for the *nagfmcheck* program, see the Memory Tracing section.

7 Compilation Messages

The messages produced by the NAG Fortran Compiler itself during compilation are intended to be self-explanatory. The linker, or more rarely the host C compiler, may produce occasional messages.

Messages produced by the compiler are classified by severity level; these levels are:

Remark a comment about the source code (this is the least important class of informational message).

Info informational message, noting an aspect of the source code in which the user may be interested.

Note an informational message of greater import than "Info".

Warning the source code appears likely to be in error.

Questionable
some questionable usage has been found in the source code which may indicate a programming error. This has the same severity as "warning".

Extension some non-standard-conforming source code has been detected but has successfully been compiled as an extension to the language. This has the same severity as "warning".

Obsolescent
some archaic source code has been detected which although standard-conforming was classified as obsolescent by the Fortran standard (selected according to the *–f95*, *–f2003* and *–f2008* options). This has the same severity as "warning".

Deleted feature used
a feature that was present in an older Fortran standard but deleted from the Fortran standard selected by a *–fN* option was used. This has the same severity as "warning".

Error the source code does not conform to the Fortran standard or does not make sense. Compilation continues after recovery.

Fatal a serious error in the user's program from which the compiler cannot recover, the compilation is immediately terminated.

Panic an internal inconsistency is found by one of the compiler's self-checks; this is a bug in the compiler itself and NAG should be notified.

8 Compiler Limits

Item	Limit
Maximum `INCLUDE` file nesting	20
Maximum number of `INCLUDE` file references per compilation	2047
Maximum `DATA`-implied-`DO` loop nesting	99
Maximum array-constructor-implied-`DO` loop nesting	99
Maximum number of dummy arguments	32767
Maximum number of arguments to `MIN` and `MAX`	100
Maximum character length (except as below)	2147483647
Maximum character length (64-bit Windows and -abi=64c Linux)	1099511627775 (2^{40}-1)
Maximum array size (32-bit systems)	2147483647 bytes
Maximum array size (64-bit systems)	1 TiB
Maximum unit number	2147483647
Maximum input/output record length	2147483647 bytes

9 Input/Output Information

Item	Value
Standard error (stderr) unit number	0
Standard input (stdin) unit number	5
Standard output (stdout) unit number	6
Default maximum record length for formatted output	1024 characters
Default maximum record length for unformatted output	2147483647 bytes

The default directory used for files opened with `STATUS='SCRATCH'` is '`/tmp`' on Unix and the Windows temporary directory on Windows. This default may be overridden with the `TMPDIR` environment variable.

10 OpenMP Support

OpenMP 3.1 is fully supported. Some features from more recent OpenMP specifications are also supported; check the Release Notes for details.

When using the IEEE arithmetic support modules, the IEEE modes (rounding, halting and underflow) are propagated into spawned OpenMP threads at the beginning of a `PARALLEL` construct, and any IEEE flag that are set by an OpenMP thread is passed back to the parent thread at the end of the `PARALLEL` construct.

The following table lists the OpenMP environment variables with their default values and, if applicable, their limits.

Environment Variable	**Default**	**Limits**
`OMP_NUM_THREADS`	*number of cores*	1-32768
`OMP_DYNAMIC`	False	true or false
`OMP_NESTED`	False	true or false
`OMP_STACKSIZE`	0	<1GB (32-bit) or 16GB (64-bit)
`OMP_WAIT_POLICY`	None	active or passive
`OMP_MAX_ACTIVE_LEVELS`	1	1-64
`OMP_THREAD_LIMIT`	32768	1-32768
`OMP_CANCELLATION`	False	true or false

Note that although the NAG runtime supports up to 32768 threads, operating system limits may prevent usage of so many.

OpenMP is not compatible with the *−coarray=cosmp* and *−gline* options.

11 Automatic File Preconnection

All logical unit numbers are automatically preconnected to specific files. These files need not exist and will only be opened or created if they are accessed with `READ` or `WRITE` without an explicit `OPEN`. By default the specific filename for unit n is `fort.`n; however if the environment variable `FORT`nn exists its value is used as the filename. Note that there are two digits in this variable name, e.g. the variable controlling unit 1 is `FORT01` whereas the default filename is '`fort.1`' (unless the prefix has been changed, see the description of module `F90_PRECONN_IO`).

A file preconnected in this manner is opened with `ACCESS='SEQUENTIAL'`. If the initial `READ` or `WRITE` is an unformatted i/o statement, it is opened with `FORM='UNFORMATTED'` otherwise it is opened with `FORM='FORMATTED'`. By default a formatted connection is opened with `BLANK='NULL'` and `POSITION='REWIND'` (see module `F90_PRECONN_IO`).

Automatic preconnection applies only to the initial use of a logical unit; once `CLOSE`d the unit will not be reconnected automatically but must be explicitly `OPEN`ed.

Note that this facility means that it is possible for a `READ` or `WRITE` statement with an `IOSTAT=` clause to receive an i/o error code associated with the implicit `OPEN`.

12 IEEE 754 Arithmetic Support

If no floating-point option is specified, any floating divide-by-zero, overflow or invalid operand exception will cause the execution of the program to be terminated (with an informative message and usually a core dump). Occurrence of floating underflow may be reported on normal termination of the program. On hardware supporting IEEE 754 standard arithmetic gradual underflow with denormalised numbers will be enabled. Note that this mode of operation is the only one available on hardware which does not support IEEE 754.

If the *−ieee=full* option is specified, non-stop arithmetic is enabled; thus `REAL` variables may take on the values +Infinity, −Infinity and NaN (Not-a-Number). If any of the floating exceptions listed above are detected by the hardware during execution, this fact will be reported on normal termination. The *−ieee=full* option must be specified when compiling the main program and has global effect; that is, it affects the entire executable program.

If the *−ieee=nonstd* option is specified, floating-point exceptions are handled in the default manner (i.e. execution is terminated). However, gradual underflow is **not** enabled, so results which would have produced a denormalised number produce zero instead. This option can only be used on hardware for which this mode of operation is faster. Like *−ieee=full*, the *−ieee=nonstd* option must be specified when compiling the main program and has global effect.

13 Half precision floating-point

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

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

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

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

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

14 Random Number Algorithm

The random number generator supplied as the intrinsic subroutine `RANDOM_NUMBER` is the "Mersenne Twister".

Note that this generator has a large state (630 32-bit integers) and an extremely long period (approx 10^{6000}), and therefore it is **strongly recommended** that the `RANDOM_SEED` routine only be used with a `PUT` argument that is the value returned by a previous call with `GET`; i.e., only to repeat a previous sequence. This is because if a user-specified seed has low entropy (likely since there are 630 values to be supplied), it is highly likely to set the generator to an apparently-low-entropy part of the sequence.

If you do want to provide your own seed (and thus entropy), you should store your values in the initial elements of the seed array and set all the remaining elements to zero — trailing zero elements will be ignored and not used to initialise the generator. Note that the seed is a random bitstream, and is therefore expected to have approximately half of its bits nonzero (thus providing many small integer values will likely result in a low-entropy part of the Mersenne Twister sequence being reached).

15 Automatic Garbage Collection

The *−gc* option enables use of the runtime garbage collector. It is necessary to use this option during the link phase for it to have effect; specifying it additionally during the compilation phase can result in improved performance.

The supplied Technical Information note (TECHINFO) lists whether garbage collection is available for your system. If it is available, there will be a file '`gc.o`' in the compiler's library directory.

The collector used is based on version 5.3 of the publicly available general purpose garbage collecting storage allocator of Hans-J Boehm, Alan J Demers and Xerox Corporation, described in "Garbage Collection in an Uncooperative Environment" (H Boehm and M Weiser, Software Practice and Experience, September 1988, pp 807-820).

The copyright notice attached to their latest version is as follows:

```
Copyright 1988, 1989 Hans-J. Boehm, Alan J. Demers
Copyright (c) 1991-1995 by Xerox Corporation.  All rights reserved.
Copyright 1996-1999 by Silicon Graphics.  All rights reserved.
Copyright 1999 by Hewlett-Packard Company.  All rights reserved.


THIS MATERIAL IS PROVIDED AS IS, WITH ABSOLUTELY NO WARRANTY EXPRESSED
OR IMPLIED.  ANY USE IS AT YOUR OWN RISK.

Permission is hereby granted to use or copy this program
for any purpose,  provided the above notices are retained on all copies.
Permission to modify the code and to distribute modified code is granted,
provided the above notices are retained, and a notice that the code was
modified is included with the above copyright notice.
```

Note that the "NO WARRANTY" disclaimer refers to the original copyright holders Boehm, Demers, Xerox Corporation, Silicon Graphics and Hewlett-Packard Company. The modified collector distributed in binary form with the NAG Fortran Compiler is subject to the same warranty and conditions as the rest of the NAG Fortran compilation system.

The module `F90_GC` is provided; it contains functions and variables that can control the behaviour of the garbage collector.

16 Memory Tracing

Tracing of memory allocation and deallocation is provided by the *−mtrace* option. Control is provided over whether the address, size, and line number of each allocation is displayed, or the tracing output can be suppressed entirely. A "paranoia" mode is provided where the memory allocator protects its data structures against inadvertent modification by the user program.

Runtime environment variables may be used to override the tracing options a program was built with, and to specify where to write the tracing output. These are only operative if the program was built with some tracing option; *−mtrace=off* will build a program with the tracing-capable memory allocator.

If *−mtrace=off* is not specified, use of any *−mtrace* option will implicitly do a *−mtrace=on*.

Basic tracing produces a message to the memory tracing file (normally standard error) for each allocation and deallocation, including those for automatic variables, i/o buffers and compiler-generated temporaries. Each allocation is numbered sequentially; the first three items are the i/o buffers for units 0, 5 and 6 (standard error, standard input and standard output).

All *−mtrace=* suboptions may be overridden at run time by the `NAGFORTRAN_MTRACE_OPTIONS` environment variable, which should be set to the required *trace_opt_list* (e.g. '`on,size`'). The memory tracing file may be specified at run time by the `NAGFORTRAN_MTRACE_FILE` environment variable.

The `NAGFORTRAN_MTRACE_OPTIONS` variable can also contain an option to limit the total amount of memory that may

be allocated. The '`limit=`*N*' option limits the maximum memory allocated to *N* MiB (mebibytes), but only if the program was built with a tracing option (minimally, *−mtrace=off*). Exceeding the memory limit will result in a normal "out of memory" condition, which if it occurs in an `ALLOCATE` statement, can be captured by a `STAT=` clause. Note that the memory limit applies to the overall memory usage including automatic variables and compiler-generated array temporaries.

The *−mtrace* option must be specified when linking, and is incompatible with *−gc*. Additionally, line number information is only available for those files compiled with *−mtrace=line*.

The `nagfmcheck` program can be used to check the output from the *−mtrace* option. It is designed to be used as a filter. Any lines that do not look like memory tracing output are ignored. It reports to standard output any errors it detects such as deallocating something twice, deallocating something that was never allocated, or deallocating something with a size different from that with which it was allocated. It also reports any apparent memory leaks, though this is less useful if the program terminated prematurely.

17 Undefined Variable Detection

Use of undefined variables can be detected with the *−C=undefined* option. Program units compiled with this option use a different ABI, which means that they are incompatible with program units compiled without this option, and not interoperable with C; thus the whole program must be Fortran code and compiled the same way. For this reason, *−C=undefined* is not part of *−C* or *−C=all*.

Currently, there are a number of other limitations on the use of *−C=undefined*.

1. It is incompatible with pointers in an initialised `COMMON`.
2. All intrinsic modules are available, but the `ISO_C_BINDING` module can only be used with all-Fortran programs as the option makes changes to the ABI.
3. Internal `READ` from a `CHARACTER` array requires the entire specified array subobject to be "defined", even those elements corresponding to records not actually read.
4. Internal `WRITE` to a `CHARACTER` array is considered to define the entire specified array subobject, even those elements corresponding to records not actually written.
5. Certain intrinsic functions require the entirety of their arguments to be defined, even if some portions are not actually required for the value of the function. For example, the `PAD` argument to `RESHAPE` when no padding is actually required, and elements of the `ARRAY` argument to `PACK` that correspond to false elements of the `MASK`.
6. It is incompatible with the use of coarrays.
7. It cannot be used on types with length type parameters.
8. It cannot be used when `CLASS(*)` variables are allocated using the `MOLD=` specifier.
9. It cannot be used with `ALLOCATE` when the `SOURCE=` expression is a `CLASS(*)` dummy and the actual argument is a constant.

18 Data Types

The table below lists the intrinsic data types provided by the NAG Fortran Compiler together with their kind numbers. There are three possible schemes for the intrinsic kind type parameters: the default mode of operation (which may be specified explicitly by the *−kind=sequential* option), the "byte" numbering scheme (specified by the *−kind=byte* option) and the "unique" numbering scheme (specified by the *−kind=unique*).

Type Name	KIND Number (sequential)	KIND Number (byte)	KIND Number (unique)	Name	Description
REAL	1	4	301	REAL32*	Single precision floating-point
REAL	2	8	302	REAL64*	Double precision floating-point
REAL	3	16	303	REAL128*	Quad precision floating-point
REAL	16	2	304	REAL16*	Half precision floating-point
COMPLEX	1	4	301	REAL32*	Single precision complex
COMPLEX	2	8	302	REAL64*	Double precision complex
COMPLEX	3	16	303	REAL128*	Quadruple precision complex
COMPLEX	16	2	304	REAL16*	Half precision complex
LOGICAL	1	1	201	BYTE	Single byte logical
LOGICAL	2	2	202	TWOBYTE	Double byte logical
LOGICAL	3	4	203	WORD	Default logical
LOGICAL	4	8	204	LOGICAL64	Eight byte logical
INTEGER	1	1	101	INT8*	8-bit integer
INTEGER	2	2	102	INT16*	16-bit integer
INTEGER	3	4	103	INT32*	32-bit (default) integer
INTEGER	4	8	104	INT64*	64-bit integer
CHARACTER	1	1	646	ASCII	ASCII or ISO 8859-1 character
CHARACTER	2	2	213	JIS	JIS X 0213 character
CHARACTER	3	3	5323	UCS2	Unicode (UCS-2) character
CHARACTER	4	4	10646	UCS4	ISO 10646 (UCS-4) character

The Name column of the table indicates the name provided by the intrinsic module F90_KIND; the ones marked * are also provided by the standard intrinsic module ISO_FORTRAN_ENV. Using these names avoids the portability problems that arise if the kind numbers are hard-coded.

Note that on all machines except Sun Solaris with the SunPro C compiler, quadruple precision is actually "double double" precision; this provides nearly twice the precision of Double precision but with a reduced exponent range.

19 Modules

To use a module it must be an intrinsic module, previously compiled, or defined in the file prior to its use. When separately compiling a module the $-c$ option should be specified.

Compiling a module creates a '.mod' file and a '.o' file. The '.mod' file is used by the compiler at compile time to provide information about module contents, the '.o' file (if generated) contains the code of any module procedures and must be specified when creating an executable file.

Note that the name of the '.mod' file will be the name of the module, the '.o' file will be named after the original source file.

When a previously compiled module is USEd the NAG Fortran Compiler attempts to find its source file and, if that is successful, checks the modification times producing a warning message if the '.mod' file is out of date.

20 Runtime Environment Variables

The following variables control the runtime environment for programs compiled with the NAG Fortran Compiler.

NAGFORTRAN_MTRACE_FILE
: Programs compiled using any *−mtrace=* option will write the memory trace to this file. The default is standard error.

NAGFORTRAN_MTRACE_OPTIONS
: Changes the memory tracing options for programs compiled using any *−mtrace=* option.

`NAGFORTRAN_NUM_IMAGES`
: Sets the number of images with which to execute a program in Co-SMP mode (it has no effect if the main program was compiled with *−coarray=single*). If the value of the variable is not an integer value, or is less than one or greater than 1000, it is ignored. In the absence of this variable, the number of images for a Co-SMP mode program is taken from the *−num_images=* option, or from the number of hardware threads.

`NAGFORTRAN_RUNTIME_ERROR_FILE`
: Runtime error messages will be written to this file. The default is standard error.

`NAGFORTRAN_RUNTIME_LANGUAGE`
: Controls the language used for runtime error messages. This may be 'English' or 'Japanese' (not case-sensitive); the default is English.

`NAGFORTRAN_RUNTIME_OPTIONS`
: Controls runtime optional behaviour excluding memory tracing. This is a comma-separated list of options from the following list.

Option	Effect
autoskip_namelist	Enables auto-skipping namelist input.
blank_common_size=*N*	Sets the default size of blank `COMMON` blocks when executing in Co-SMP mode.
log_autoskip_namelist	Enables auto-skipping namelist input, with logging.
show_dangling	Enables tracing of dangling pointers; this only affects code compiled with *−C=dangling*.
suppress_underflow_warning	Do not produce the usual warning on program termination when the floating-point underflow flag is set.
underflow_warning	Do produce the usual warning on program termination when the floating-point underflow flag is set.

The **autoskip_namelist** option enables autoskipping namelist input. In this mode, when the name after the ampersand in the input record does not match the namelist group name in the `READ` statement, instead of raising an i/o error condition it skips records until it finds one that begins with an ampersand and the correct name.

The **blank_common_size=***N* option sets the default size of blank `COMMON` blocks to *N* bytes when executing in Co-SMP mode with multiple images; it has no effect otherwise. If not specified, the default size is one mebibyte (1048576 bytes). This option is only needed if blank `COMMON` blocks in different program units have different sizes, and the largest one is not encountered first.

The **log_autoskip_namelist** option enables autoskipping namelist input (as above), with logging. In this mode, when an autoskip occurs, the location of the `READ` statement and the action being taken are logged to standard error, for example:

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

The **show_dangling** option causes messages to be produced on the runtime error file when a dangling pointer is created, reassociated with something else, nullified, or ceases to exist. For example,

```
[a.f90, line 20: Dangling pointer P detected (number 1), associated at b.f90, line 18]
[c.f90, line 7: Dangling pointer P (number 1) has been reassociated]
[c.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]
```

The **suppress_underflow_warning** runtime option has the same effect as the *−no_underflow_warning* compilation option; that is, it suppresses the usual warning message on program termination when the floating-point underflow flag is set.

The **underflow_warning** runtime option requests that if the floating-point underflow flag is set on program termination, a warning message should be produced. This is the default behaviour, but the runtime option will override the *−no_underflow_warning* compilation option.

`TMPDIR` Controls the directory used for scratch files (the default is system-dependent).

21 Debugging

On Windows debugging is built-in to the Fortran Builder. For operating systems other than Windows a Modern Fortran-aware debugger might be available as dbx90; see TECHINFO.txt for details.

In general, host system debuggers, such as dbx or gdb, may be used successfully on Fortran code as the names of the original source files, plus line numbers, are passed through to the intermediate C files. In using such debuggers it should be noted that most local variables have an underscore appended to their names. It may be useful to look at the intermediate C code when debugging; this is produced by the *−S* option.

22 Producing a Call Graph

The call graph generator takes a set of Fortran source files and produces a call graph with optional index and called-by tables. C files and fpp-processed files are not handled.

The call graph generator understands the following compiler options with the same meaning: *−132*, *−dcfuns*, *−double*, *−dryrun*, *−dusty*, *−encoding*, *−english*, *−f2003*, *−f2008*, *−f95*, *−fixed*, *−free*, *−help*, *−I*, *−i8*, *−indirect*, *−info*, *−kind*, *−max_parameter_size*, *−maxcontin*, *−mismatch*, *−mismatch_all*, *−nihongo*, *−nocheck_modtime*, *−nomod*, *−noqueue*, *−o*, *−openmp*, *−Qpath*, *−r8*, *−strict95*, *−thread_safe*, *−u*, *−u=sharing*, *−v*, *−V*, *−w* and *−xlicinfo*.

The "@*filename*" syntax may also be used, with the same effect as the "*−indirect filename*" option.

The call graph is written to the file specified by the *−o* option, or to standard output if no *−o* option is specified.

The following additional options control the output produced.

−calledby
: Produce a "called-by" table showing, for each routine, the routines that call it directly or indirectly. This is produced at the end of the output.

−indent=*N*
: Indent by *N* for each level in the graph, up to the maximum. The default is *−indent=4*.

−indent_max=*N*
: The maximum indentation is *N*. The default is *−indent_max=70*.

−index Produce an alphabetic index listing, for each routine, the line of the call graph where the routine first appears. This follows the call graph itself and precedes the called-by table (when the *−calledby* option is used).

−show_entry
: Show `ENTRY` point names in the call graph; without this option, calls to an `ENTRY` point are shown as calls to the containing subprogram.

−show_generic
: If a call is via a generic identifier, show the generic identifier in the call graph.

−show_host
: Show the host scope names for calls to internal and module procedures.

−show_pclass
: Show the class of each procedure (e.g. 'module', 'internal', ...).

−show_rename
: If a called procedure was renamed on a `USE` statement, show the renaming.

23 Dependency Analysis

The dependency analyser takes a set of Fortran source files and produces dependency information in the form specified. C files and fpp-processed files are not handled.

The dependency analyser understands the following compiler options with the same meaning: *−132*, *−dryrun*, *−english*, *−fixed*, *−free*, *−help*, *−I*, *−indirect*, *−maxcontin*, *−nihongo*, *−o*, *−Qpath*, *−tempdir*, *−v* and *−V*. The "@*filename*" syntax may also be used with the same effect as the "*−indirect filename*" option.

The following additional options control the operation of the dependency analyser:

−otype=*type*
: This option controls the output form, *type* must be one of:

  blist (the filenames as an ordered build list),
  dfile (the dependencies in Makefile format, written to separate *file*`.d` files),
  info (the dependencies as English descriptions) or
  make (the dependencies in Makefile format).

  The default is *−otype=info*. If *−otype=dfile* is specified, no *−o* option is permitted; otherwise, the result is written to the file specified by the *−o* option or to standard output if no *−o* option is specified.

−paths=*pathtype*
: Specifes the form to use for dependency paths; *pathtype* must be either **absolute** or **relative**. With *−paths=absolute*, paths for `INCLUDE` files that are relative specifications will be prefixed by the current working directory.

24 Generating Interfaces

The interface generator takes a set of Fortran source files and produces interfaces for the procedures therein. The output is either a module (in a new source file), or an `INCLUDE` file.

The interfaces are written either to the file specified by the *−o* option, or if module output is being produced to the file with the same name as the module and extension '.f90', or otherwise (an `INCLUDE` file is being produced) to 'interfaces.inc'. In each case the interfaces are all within a single `INTERFACE` block.

The interface generator understands the following compiler options with the same meaning: *−132*, *−dcfuns*, *−double*, *−dryrun*, *−dusty*, *−encoding*, *−english*, *−f2003*, *−f2008*, *−f95*, *−fixed*, *−free*, *−help*, *−I*, *−i8*, *−indirect*, *−info*, *−kind*, *−max_parameter_size*, *−maxcontin*, *−mismatch*, *−mismatch_all*, *−nihongo*, *−nocheck_modtime*, *−noqueue*, *−o*, *−openmp*, *−Qpath*, *−r8*, *−strict95*, *−tempdir*, *−thread_safe*, *−u*, *−u=sharing*, *−v*, *−V*, *−w* and *−xlicinfo*.

The interface generator understands all the enhanced polish options with the same meaning.

The following additional options control the operation of the interface generator:

−cmt_generation
: Add a comment before the `INTERFACE` statement, giving the date and time that the file was generated. This is the default.

−cmt_provenance
: Add a comment after each procedure heading (`SUBROUTINE` or `FUNCTION` statement) indicating the source of the procedure.

−module=*X*
: Specifies the name of the module to generate containing procedure interfaces. The default is '`interfaces`'.

−otype=*type*
: Specify the type of output file required to *type*, which must be one of

  include (INCLUDE file),
  module (Fortran module in a new source file).

  The default is *−otype=module*.

−nocmt_generation
: Do not add any comment before the `INTERFACE` statement.

–nocmt_provenance
Do not add any comment after each procedure heading. This is the default.

25 Source File Polishing

The polisher takes a set of Fortran source files, which may be in fixed or free form, and produces a free form "polished" version of each file. C files and fpp-processed files are not handled.

The polisher understands the following compiler options with the same meaning: *–132*, *–encoding*, *–english*, *–f2003*, *–f2008*, *–f95*, *–fixed*, *–free*, *–help*, *–I*, *–indirect*, *–info*, *–maxcontin*, *–nihongo*, *–noqueue*, *–o*, *–openmp*, *–Qpath*, *–tempdir*, *–v*, *–V*, *–w* and *–xlicinfo*.

The polished output is written to the file specified by the *–o* option, or to the same filename with the extension replaced by '.f90_pol' if no *–o* option is specified. The output file cannot have the same name as the input file.

The following additional options control the operation of the polisher:

–align_right_continuation
Align the continuation markers (ampersands) at the end of a continued line to column N+2, where N is the normal line width (specified by the *–width=* option). This only affects lines that do not end with an inline comment.

–alter_comments
Enable options to alter comments; without this option, any options that would otherwise alter the comments are ignored.

–array_constructor_brackets=*X*
Specify the form to use for array constructor delimiters; X must be one of **Asis** (same as the input file), **ParenSlash** (use parentheses+slash pairs, i.e. '`(/ ... /)`') or **Square** (use square brackets, i.e. '`[ ... ]`'). The default is *–array_constructor_brackets=Asis*.

–blank_cmt_to_blank_line
Turn comment lines that have no text (other than the comment-initiating character) into plain blank lines; this is the default if the *–alter_comments* option is set.

–blank_line_after_decls
Ensure that there is a blank line after the declarations and before the first executable statement; this is the default.

–bom=*X*
Specify whether to write a Unicode Byte-Order Mark at the beginning of the output file; X must be one of **Asis** (same as the input file), **Insert** (insert a byte-order mark) or **Remove** (remove any byte-order mark). This option only has effect if the input file is known to be in UTF-8 encoding, either because it begins with a byte-order mark or the *–encoding=UTF8* option was used. The default is *–bom=Asis*.

–break_long_comment_word
If a comment line will be split into two lines, the comment may be broken in the middle of a long word.

–character_decl=*style*
Specify the style to be used for `CHARACTER` type declaration statements; *style* must be one of the following (not case-sensitive):

Asis	(same as the input statement, but obey any *–kind_keyword=* option),
Keywords	(use `LEN=` and `KIND=`),
Kind_Keyword_Only	(use `KIND=` but not `LEN=`) or
No_Keywords	(use modern style with no keywords).

The default is **Asis**; with any other style, the obsolescent "`CHARACTER*`*length*" form will be changed to the modern "`CHARACTER(`*length*`)`" form. When both the length and kind appear in the input statement, the length will appear first in the output statement.

–commas_in_formats=*X*
Specify whether to add optional commas in `FORMAT` statements; X must be one of **Asis** (use the same comma scheme as the input), **Insert** or **Remove**. The default is *–commas_in_formats=Insert*.

–dcolon_column=*N*
Align double colon '::' in declarations at column N and align any subsequent continuation lines to match. The default is for no special alignment, which is equivalent to *–dcolon_column=0*.

–dcolon_in_decls=*X*
Specifies how to handle the optional double colon '::' in declarations; X must be one of **Asis** (preserve the input status), **Insert** (insert '::' if not present), or **Remove** (remove '::' if present and optional); the default is *–dcolon_in_decls=Asis*.

–delete_all_comments
Delete all comments (if the *–alter_comments* option is set).

–delete_blank_lines
Delete blank lines and comment lines that have no text (other than the comment-initiating character), if the *–alter_comments* option is set.

–delete_unused_labels
Delete labels that are never referenced; this is the default.

–format_start=*N*
If renumbering `FORMAT` statements in a separate sequence, the first `FORMAT` statement will be N; the default is *–format_start=90000*.

–format_step=*N*
If renumbering `FORMAT` statements in a separate sequence, the step from one label to the next will be N; the default is *–format_step=10*. Note that this may be negative (but not zero).

–idcase=*X*
Set the case to use for identifiers; X must be one of **Asis**, **Capitalised**, **lowercase**, **UPPERCASE**, **Camel_Case**, or an abbreviation thereof (both **C** and **Ca** are treated as **Capitalised**, not **Camel_Case**); the default is *–idcase=lowercase*. The interpretation of X is not case-sensitive (e.g. *–idcase=u* is the same as *–idcase=U*). Note that *–idcase=Asis* is only available for basic polishing (*=polish*), not in Enhanced Polish (*=epolish*) or any other tool (e.g. *=unifyprecision*).

–indent=*N*
Indent statements within a construct by N spaces from the current indentation level; the default is *–indent=2*.

–indent_comment_marker
When indenting comments, the comment-initiating character should be indented to the indentation level; this is the default.

–indent_comments
Indent comments; this is the default if the *–alter_comments* option is set. The result is also affected by the *–indent_comment_marker* option.

–indent_continuation=*N*
Indent continuation lines by an additional N spaces; the default is *–indent_continuation=2*.

–indent_max=*N*
Set the maximum indentation level to N spaces; the default is *–indent_max=60*. The value must be at least 10 less than the output line length (*–width=*).

–inline_comment_indent=*N*
Set the indentation level for inline comments to column N; the default is *–inline_comment_index=35*.

–keep_blank_lines
Do not delete blank lines or comment lines with no text; this is the opposite of *–delete_blank_lines* and is the default.

–keep_comments
Do not delete non-blank comment lines; this is the opposite of *–delete_comments* and is the default.

–keep_unused_labels
Do not delete unused (unreferenced) labels; this is the opposite of *–delete_unused_labels*.

–kind_keyword=*X*
Specifies how to handle the `KIND=` specifier in declarations; X must be one of **Asis** (take no action but preserve the input status), **Insert** (insert `KIND=` if not present), or **Remove** (remove `KIND=` if present); the default is *–kind_keyword=Asis*.

–kwcase=*X*
Set the case to use for language keywords; *X* must be one of **Capitalised**, **lowercase**, **UPPERCASE**, **Camel_Case**, or an abbreviation thereof (both **C** and **Ca** are treated as **Capitalised**, not **Camel_Case**); the default is *–kwcase=Capitalised*. The interpretation of *X* is not case-sensitive (e.g. *–kwcase=u* is the same as *–kwcase=U*). The *–kwcase=C* produces '`Double Precision`' and '`Non_recursive`'; with *–kwcase=Camel*, the latter is produced as '`Non_Recursive`'.

–label_after_indent
Indent labels; this is the opposite to *–label_before_indent*.

–label_before_indent
Output the statement label, if any, before indenting the statement; this is the default.

–leave_formats_in_place
Leave `FORMAT` statements in the same position as they are in the input file; this is the opposite of *–move_formats_to_end*, and is the default.

–margin=*N*
Set the left margin (initial indent) to *N*. The value must be at least 10 less than the output line length (*–width=*). The default value for the left margin is 4.

–move_formats_to_end
Move `FORMAT` statements to the end of the subprogram or program unit, immediately before the `CONTAINS` or `END` statement.

–name_scopes=*X*
Specify whether to add optional keywords and scope names to the `END` or `END TYPE` statement for a scope; *X* must be one of **Asis** (leave as is), **Insert** (insert keywords and/or names), **Keywords** (insert keywords but remove names) or **Remove** (remove optional keywords and names). This option also applies to the `END INTERFACE` statement. The default is *–name_scopes=Keywords*.

–noalign_right_continuation
Do not align the continuation markers (ampersands) at the end of continued lines; this is the default.

–noalter_comments
Do not alter comments in any way; this is the default.

–noblank_cmt_to_blank_line
Do not turn blank comments to blank lines.

–noblank_line_after_decls
Do not insert a blank line between the last declaration and the first executable statement.

–nobreak_long_comment_word
If a comment line will be split into two lines, do not break the comment in the middle of a long word; this is the default.

–nodcolon_column
Do not align double colon '`::`' in declarations. This is the default, and is equivalent to specifying alignment at column zero via *–dcolon_column=0*.

–noindent_comment_marker
Place the comment-initiating character for a comment line in column 1.

–noindent_comments
Do not indent the text of a comment line.

–norenumber
Do not renumber statement labels.

–noseparate_format_numbering
When renumbering statement labels, use a single sequence for both `FORMAT` and non-`FORMAT` statements; this is the default.

–noterminate_do_with_enddo
Do not change `DO` loop terminating statements.

–nowrap_comments
Do not wrap long comment lines (they will still be indented if comments are being indented).

–relational=*X*
Specifies the form to use for relational operators, *X* must be either **F77-** (use `.EQ.`, `.LE.`, etc.) or **F90+** (use `==`, `<=`, etc.); the default is *–relational=F90+*.

–renumber
Renumber statement labels; this is the default.

–renumber_start=*N*
When renumbering statement labels, the first label will be *N*; the default is *–renumber_start=100*.

–renumber_step=*N*
When renumbering statement labels, the step from one label to the next will be *N*; the default value is *–renumber_step=10*.

–separate_format_numbering
When renumbering statement labels, renumber `FORMAT` statements in a separate sequence from non-`FORMAT` statements.

–terminate_do_with_enddo
Change the terminating statements of all `DO` loops so that each loop ends with an `ENDDO` statement; this is the default.

–width=*N*
Set the maximum length of the text on each output line to *N*; the default is *–width=78*. Note that in the case of continuation lines, an additional two characters (‘ `&`’) will be produced after the last text on a line and this may take the line length over the limit. The width must be at least 10 more than the left margin (*–margin=*) and the maximum indent (*–indent_max=*). The maximum width setting is 1024, however values higher than 130 will produce output that does not conform to the Fortran standard.

–wrap_comments
Wrap long comment lines that would otherwise exceed the maximum line length. This is the default.

26 Enhanced Source File Polishing

The enhanced polisher takes a set of Fortran source files, which may be in fixed or free form, and produces a free form “polished” version of each file. C files and fpp-processed files are not handled. Unlike the simple polisher, the Fortran source files must be compilable without error; this is because the information needed for enhanced polishing requires successful semantic analysis of the files.

The enhanced polisher understands the following compiler options with the same meaning: *–132*, *–abi*, *–dcfuns*, *–double*, *–dryrun*, *–dusty*, *–encoding*, *–english*, *–f2003*, *–f2008*, *–f95*, *–fixed*, *–free*, *–help*, *–I*, *–i8*, *–indirect*, *–info*, *–kind*, *–max_parameter_size*, *–maxcontin*, *–mismatch*, *–mismatch_all*, *–nihongo*, *–nocheck_modtime*, *–nomod*, *–noqueue*, *–o*, *–openmp*, *–Qpath*, *–r8*, *–strict95*, *–tempdir*, *–thread_safe*, *–u*, *–u=sharing*, *–v*, *–V*, *–w* and *–xlicinfo*.

The enhanced polisher includes all the simple polish options, which are not repeated here, except for *–idcase=Asis*.

Note that unlike `nagfor =polish`, *–name_scopes=Asis* acts as if it were *–name_scopes=Keywords*, which is the default. Similarly, *–array_constructor_brackets=Asis* acts as if it were *–array_constructor_brackets=ParenSlash*, and is the default, and *–dcolon_in_decls=Asis* acts as if it were *–dcolon_in_decls=Insert*, and is the default.

The default filename extension for the output file is ‘.f90_epo’, used when no *–o* option is specified.

The following additional options control the operation of this tool.

–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.

This option is equivalent to *–add_arg_keywords=all2,intrinsic3*.

–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.

–case:*kind=case-list*

Specifies case rules for specific kinds of name; this option overrides other case options except for *–casex*. The colon is followed by a comma-separated list of "*kind=case*", where *case* is a case specification (`UPPERCASE`, `lowercase`, `Capitalised`, `Camel_Case`), and *kind* is one of the categories listed below:

`comp`	Component
`constr`	Construct name
`intr`	Intrinsic procedure
`param`	PARAMETER
`proc`	Procedure
`tbp`	Type-bound procedure
`tparam`	Derived type parameter
`type`	Derived type
`var`	Variable

For example, *–case:var=lower,proc=u* specifies lowercase for variables and UPPERCASE for procedures. If there is no setting for a particular kind of name, it will fall back to an appropriate category; `param`, `type`, `comp`, `tparam` and `proc` all fall back to `var`, `intr` will fall back to `proc`, and `tbp` will fall back to `comp` or `proc`. If there is no rule or fall-back rule, the *–idcase=* option setting (or default) is used.

–casex:*name-list*

Specifies exceptions to the case rules. The colon is followed by a comma-separated list of names in the exact case required. For example, *–casex:MaxVal,XYz* will result in every occurence of a name equivalent to `maxval` or `xyz` appearing as `MaxVal` or `XYz` respectively.

–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**). The default is *–intrinsic_case=as_names*.

–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.

27 Unifying Precision

The precision unifier standardises floating-point and complex variable declarations, floating-point and complex literal constants, and some specific (non-generic) intrinsic procedures in a set of Fortran source files in order to unify the precision of these entities.

Standardisation to quadruple precision is only available on machines for which quadruple-precision floating-point arithmetic is available.

The tool attempts to make a standardising precision parameter accessible in program units (and interface blocks) via a use statement. You can control the form of this statement: the *−pp_name=* option controls the name of the precision parameter, and the *−pp_module=* option supplies the name of its host module (which is known as the 'precision module'). The default form for the use statement (when no options are specified) is `USE WORKING_PRECISION, ONLY: WP`.

The *−precision=* option (whose default value is **Double**) can be supplied to set the desired unifying precision. The tool will use this setting when performing a number of checks of the validity of the standardisation process on the input files.

The precision module can be created by the tool, but otherwise does not itself undergo precision unification. A warning is issued if the tool encounters this module. A message is also emitted if no definition for the precision parameter is found in the module, or otherwise if the defined precision parameter specifies a different kind to the desired precision as provided or implied by the *−precision=* option.

The tool searches each program unit and interface block in the input source and determines whether the precision parameter is already accessible. If it is not, then a use statement, in the form given above, is inserted in the last allowable position for its statement type. For an internal or module procedure this statement is placed in the host. If the precision parameter is already declared in the form `INTEGER, PARAMETER :: wp = constant_expression`, then this declaration is deleted and a new use statement added, as previously described. (This `PARAMETER` form of statement is only recognised as declaring the precision parameter if it precedes all declarations of floating-point or complex entities in the scoping unit.) Any other form of definition or import of the precision parameter will not be modified, and the tool issues a warning that the standardised use statement could not be inserted.

Type declarations for floating-point and complex entities are standardised to include the precision parameter as kind parameter. Entities that are implicitly typed to be floating-point or complex are explicitly declared, in the same form. In the case when a function is defined with a floating-point or complex type specification on the function statement, this specification is deleted and a distinct type declaration statement for the function result is inserted into the function's declaration section.

Floating-point and complex literal constants are standardised to use the precision parameter as their kind.

The option *−pu_floats=* controls the extent of precision conversions that are applied. The default behaviour described above for floating-point and complex entities corresponds to *−pu_floats=On*. The value *−pu_floats=Default_Kinds* may be supplied in order to limit the precision unification only to entities having default kind; i.e., kind specifiers already given in type declarations or for literals will be preserved, even if they differ from the desired unifying precision. Modification of all floating-point and complex entities may be suppressed altogether via *−pu_floats=Off*.

The following specific procedure references are standardised to the generic replacement listed below:

Specific	Generic	Specific	Generic	Specific	Generic
ALOG10	LOG10	DATAN	ATAN	DSINH	SINH
ALOG	LOG	DBLE	REAL	DSIN	SIN
AMAX0(...)	REAL(MAX(...))	DCMPLX	CMPLX	DSQRT	SQRT
AMAX1	MAX	DCONJG	CONJG	DTANH	TANH
AMIN0(...)	REAL(MIN(...))	DCOS	COS	DTAN	TAN
AMIN1	MIN	DCOSH	COSH	FLOAT	REAL
AMOD	MOD	DDIM	DIM	IABS	ABS
CABS	ABS	DEXP	EXP	IDIM	DIM
CCOS	COS	DIMAG	AIMAG	IDINT	INT
CDABS	ABS	DINT	AINT	IDNINT	NINT
CEXP	EXP	DLOG10	LOG10	IFIX	INT
CLOG	LOG	DLOG	LOG	ISIGN	SIGN
CSIN	SIN	DMAX1	MAX	MAX0	MAX
CSQRT	SQRT	DMIN1	MIN	MAX1(...)	INT(MAX(...))
DABS	ABS	DMOD	MOD	MIN0	MIN
DACOS	ACOS	DNINT	ANINT	MIN1(...)	INT(MIN(...))
DASIN	ASIN	DREAL	REAL	SNGL	REAL
DATAN2	ATAN2	DSIGN	SIGN		

(See also the description of *−dcfuns*.)

Furthermore, `DBLE` is converted to `REAL`. Following that, the `KIND=` argument is added to calls to `REAL` and `CMPLX`, when appropriate.

In cases where unifying the precision of the input source may lead in the generated output to undesirable side effects, or even invalid Fortran, the tool will attempt to issue a warning alerting you to the possibility. Here is a non-exhaustive list of situations where it may be inappropriate to apply this tool.

1. Your source intentionally uses a mix of floating-point and complex precisions and you are running the tool in (the default) mode *−pu_floats=On*.
2. You are employing Fortran language features for generic programming (such as generic interface blocks or parameterised derived types).
3. You have floating-point or complex data in `EQUIVALENCE` statements or in references to the `TRANSFER` intrinsic.
4. You have explicitly-typed intrinsic functions, or are passing intrinsic functions as procedure arguments.
5. You are using the `DPROD` intrinsic (perhaps as a means of performing higher- (double-) precision computations in a single-precision program unit).
6. You are mixing Fortran and non-Fortran code.

For procedures spread across several files clearly it is desirable to make sure this tool is applied to all files consistently. This will ensure, for example, that procedure references and the corresponding procedure definitions do not become inconsistent with respect to the type standardisation.

The precision unifier understands the following compiler options with the same meaning: *−132*, *−abi*, *−dcfuns*, *−double*, *−dryrun*, *−dusty*, *−encoding*, *−english*, *−f2003*, *−f2008*, *−f95*, *−fixed*, *−free*, *−help*, *−I*, *−i8*, *−indirect*, *−info*, *−kind*, *−max_parameter_size*, *−maxcontin*, *−mismatch*, *−mismatch_all*, *−nihongo*, *−nocheck_modtime*, *−nomod*, *−noqueue*, *−o*, *−openmp*, *−Qpath*, *−r8*, *−strict95*, *−tempdir*, *−thread_safe*, *−u*, *−u=sharing*, *−v*, *−V*, *−w* and *−xlicinfo*.

Note that using the *−double* or *−r8* option affects the meaning of the *−precision=* option; see the description of the latter, below.

The standardised output is written to the file specified by the *−o* option, or to the same filename with the extension replaced by '.f90_prs' if no *−o* option is specified. The output file cannot have the same name as the input file.

The precision unifier understands all the enhanced polish options with the same meaning.

The following additional options control the operation of this tool:

−nocmt_generation
If creating the precision module, do not add a comment saying when it was generated.

−pp_create_module
Automatically create the precision module, in the file whose name is the name of the module, converted to lower case, with file type '.f90'; thus the default filename is `working_precision.f90`. If the file already exists it will not be overwritten by this option.

The created module will contain only the definition of the precision parameter, and unless the *−nocmt_generation* option is given, a comment identifying when the module was created.

−pp_name=*X*
Specifies the name of the precision parameter to use in the standardised output, which must be a legal identifier that does not conflict with existing names in the input source. The default is '`WP`'.

−pp_module=*X*
Specifies the name of the precision module from which the precision parameter is to be imported. This module name must be a legal identifier that does not conflict with existing names in the input source. The default is '`WORKING_PRECISION`'.

−pp_nocreate_module
Do not create the precision module. This is the default.

−precision=*X*
Specifies the desired target unifying precision in the output; *X* must be one of **Half**, **Single** (i.e., same precision as default `REAL`), **Double** (i.e., same precision as default `DOUBLE PRECISION`) or **Quadruple**. The default is *−precision=Double*.

Note that, since *−double* and *−r8* double the size of default `REAL` (and possibly default `DOUBLE PRECISION`), specifying *−double* or *−r8* will likewise modify the meaning of this *−precision=* option too.

−pu_floats=*X*
Controls the precision-unification mode for floating-point and complex entities; *X* must be one of **Off**, **On** or **Default_Kinds**. In the latter mode of operation already-kinded entities will not be modified. The default is *−pu_floats=On*.

Debugging with dbx90

28 dbx90 command line

dbx90 [option]... *executable-file*

29 Description of dbx90

dbx90 is a Fortran-oriented debugger for use with the NAG Fortran Compiler on Unix-like systems (e.g. Linux, Solaris). Its syntax is quite similar to that of dbx, which it invokes as a sub-process to carry out the actual machine-dependent debugging commands. (On gcc-based implementations, gdb is used.)

The program to be debugged should be compiled and linked with the *−g90* option. This creates a debug information (`.g90`) file for each Fortran source file.

If the environment variable `DBX90_DBXPATH` is defined, dbx90 will use it to locate the native debugger instead of the built-in path.

30 dbx90 options

−compatible
: This permits dbx90 to debug code compiled by the NAG Fortran Compiler using the *−compatible* option.

−I ***pathname***
: Add *pathname* to the list of directories which are to be searched for module information (`.mod`) files and debug information (`.g90`) files. The current working directory is always searched first, then any directories named in *−I* options, then the compiler's library directory (usually '`/usr/local/lib/NAG_Fortran`' or '`/opt/NAG_Fortran/lib`')

−Qpath ***path***
: Set the compiler's library directory to *path*. This is only used for finding the module information (`.mod`) files for intrinsic modules.

31 dbx90 Commands

alias List command aliases.

alias ***name text***
: Create a new alias *name* with the replacement value *text*. The replacement text is not limited to a single word but may contain spaces.

assign ***var = expr***
: Assign the value of *expr* to variable *var*. The variable can be scalar, an array (including an array section), a scalar component, or an element of an array component, but cannot be of derived type. The value must be a scalar expression (see "Expressions" for more details).

cont Continue execution from where it was stopped (either by a breakpoint or an interrupt).

delete ***n*** Delete breakpoint number *n*.

delete all Delete all breakpoints.

display List the expressions to be displayed after each breakpoint is reached.

display ***expr***
: Add *expr* to the list of expressions to display after each breakpoint. *Expr* may also be an array section.

dump Display all local variables and their values.

down [***n***] Move the focus down (i.e. to the procedure called by the current procedure). If *n* is present, move down *n* levels.

help [***topic***]
Display a brief help message on the specified *topic*, or on using dbx90 generally.

history List the history command buffer. Old commands can be executed with:

!! repeat last command,

! *n* repeat command *n* in the history buffer, and

! -*n* repeat the n^{th} last command.

history *n* Set the size of the history command buffer to *n* commands. The default size of the history command buffer is 20.

if *expr* This is actually a suffix to the breakpoint ('stop') commands not an independent command. It prevents the triggering of the breakpoint until the expression *expr* (a scalar expression) is `.TRUE.` (or non-zero).

list [***line1***[**,*line2***]]
Display the next 10 lines of the program, display line *line1* of the current file or display lines *line1* to *line2* of the current file.

next [***n***] Execute the next *n* lines (default 1) of the current (or parent) procedure. Any procedure reference in these lines will be executed in its entirety unless a breakpoint is contained therein.

print *expr* [**,*expr***]**...**
Display the value of *expr*, which can be a scalar expression, array section, derived type component,or a variable of any data type. Several expressions may be given separated by commas.

quit Exit from dbx90, immediately terminating any program being debugged.

raw *dbx-command*
Pass *dbx-command* directly to "dbx". This is not recommended for normal use.

rerun [***command-line***]
Begin a new execution of the program, passing *command-line* to it (if present) or the *command-line* from the previous run or rerun command if not.

run [***command-line***]
Begin a new execution of the program, passing *command-line* to it (if present) or blank command line if not.

scope [***name***]
Display the current procedure name or set the focus to the specified procedure *name*.

status List the breakpoints which are currently set.

step [***n***] Execute the next *n* lines (default 1) of the program, counting lines in referenced procedures (i.e. step into procedure references).

stop *name*
Set a breakpoint which triggers when variable *name* is accessed. Note that *name* cannot be '`at`' or '`in`'. This command is not available on Solaris or HP-UX.

stop at *lineno*
Set a breakpoint at line *lineno* of the file containing the current procedure.

stop in *name*
Set a breakpoint at the beginning of procedure *name*. Note that this breakpoint occurs at the beginning of procedure initialisation, not at the first executable statement. If there is no procedure called '`MAIN`', the main program can be specified using that name.

undisplay *expr*
Remove *expr* from the "display" list.

up [***n***] Move the focus up (i.e. to the caller of the current procedure). If *n* is present, move up *n* levels.

whatis ***name***
Describe how *name* would be explicitly declared.

where Display the stack of active procedures with their dummy argument names.

which ***name***
Display the fully qualified form of *name* which can be used for access from another scope.

32 dbx90 Expressions

32.1 Scalar expressions

Scalar expressions in dbx90 are composed of literal constants, scalar variable references including array elements, intrinsic operations and parentheses.

Literal constants can be of any intrinsic type, e.g.

```
INTEGER    42
REAL       1.2
           1.3e2
COMPLEX    (5.2,6.3)
CHARACTER  "string"
LOGICAL    .TRUE.
           .T.
```

Subscript expressions must be scalar and of type `INTEGER`.

All intrinsic operations are supported except for exponentiation and concatenation, that is:

`+`, `-`, `*`, `/`, `==`, `/=`, `<`, `<=`, `>`, `>=`, `.AND.`, `.OR.`, `.NOT.`, `.EQV.`, `.NEQV.`, `.EQ.`, `.NE.`, `.LT.`, `.LE.`, `.GT.`, `.GE.`

(Operator names are not case-sensitive).

Note: array operations and operations involving variables of complex, character or derived type are not supported.

32.2 Array sections

Assignment, printing and displaying of array sections follows the Fortran syntax, e.g.

```
ARRAY(:)
ARRAY(1:5)
ARRAY(1:10:2)
```

If the stride is supplied it must be a positive scalar expression – negative strides are not supported. All subscript expressions must be scalar – vector subscripts are not supported.

32.3 Derived type component specification

Individual components of a derived type scalar or array may be printed using normal Fortran syntax.

For example,

```
print var%a
```

will print the “a” component of derived type “var”.

Components of all data types are supported.

Components which are of derived type will be displayed recursively until either:

a. there are no further nested derived types, or

b. a derived type array component is reached.

Array components of intrinsic data types will be truncated to '`<array>`', and derived type array components will be truncated to '`<derived type array>`'.

Allocatable components of derived types are supported.

Derived type assignment is not supported; however, scalar non-derived-type components may be assigned values.

33 dbx90 Command aliases

The following set of command aliases are defined:

```
a   assign
b   stop at
bp  stop in
c   cont
h   history
l   list
n   next
p   print
q   quit
r   rerun
s   step
```

New aliases may be created using the `alias` command, e.g.

```
alias xp1 print x+1
```

34 dbx90 limitations

Breakpoints set at the beginning of a routine occur before procedure initialisation; at this point attempting to print or display an assumed-shape dummy argument, a variable with a complicated `EQUIVALENCE` or an automatic variable will produce dbx crashes, dbx90 crashes or nonsensical output. Execution must be stepped to the first executable statement (e.g. using the `next` command or by setting a second breakpoint) before any of these will work satisfactorily.

Strides in array sections must be positive.

35 Example of dbx90

This is an example of the use of dbx90 in debugging some Fortran code which contains both `COMMON` blocks and modules.

The file to be debugged is called '`fh4.f90`' and contains:

```
MODULE fh4
  REAL r
END MODULE fh4

PROGRAM fh4_prog
  USE fh4
  COMMON/fh4com/i
  i = 2
  CALL sub
```

```
  PRINT *,i,r
END PROGRAM fh4_prog

SUBROUTINE sub
  USE fh4
  COMMON/fh4com/i
  r = 0.5*i
  i = i*3
END SUBROUTINE sub
```

It is first compiled with the *−g90* option and then run under dbx90:

```
% nagfor -g90 -o fh4 fh4.f90
% dbx90 fh4
NAG dbx90 Version 5.2(22)
Copyright 1995-2008 The Numerical Algorithms Group Ltd., Oxford, U.K.
GNU gdb Red Hat Linux (6.5-15.fc6rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db lib
rary "/lib/libthread_db.so.1".

(dbx90)
```

Setting a breakpoint in routine `SUB` and running the program.

```
(dbx90) stop in sub
[1] stop in SUB in file "fh4.f90"
(dbx90) run
stopped in SUB at line 16 in file "fh4.f90"
   16     r = 0.5*i
(dbx90)
```

Printing the value of a variable, which may be local, in a `COMMON` block, or in a `USE`d module.

```
(dbx90) print i
I = 2
(dbx90) next
17        i = i*3
(dbx90) print r
R = 1
(dbx90) next
18      END SUBROUTINE sub
(dbx90) print i
I = 6
```

Variables can also be assigned values.

```
(dbx90) assign i = 7
I  = 7
(dbx90) cont
 7   1.0000000

Program exited normally.
(dbx90) quit
%
```

36 Troubleshooting `dbx90`

The diagnostic messages produced by dbx90 itself are intended to be self-explanatory.

If you receive the error message '`Cannot exec dbx`' when starting dbx90 then you must set the environment variable `DBX90_DBXPATH` to the pathname of dbx (or gdb, or xdb).

Preprocessing with fpp

37 Overview of fpp

The fpp preprocessor is automatically invoked by the compiler driver when the *–fpp* option is used or the source file has an option that implies preprocessing (e.g. '.ff90'), but may also be invoked from the compiler library directory (usually /usr/local/lib/NAG_Fortran).

38 fpp command line

fpp [option]... [*input-file* [*output-file*]]

39 Description of fpp

fpp is the preprocessor used by the NAG Fortran compiler. It optionally accepts two filenames as arguments: *input-file* and *output-file* are, respectively, the input file read and the output file written by the preprocessor. By default standard input and output are used.

When used via nagfor, either because the input source file type was automatically recognised as requiring preprocessing (e.g. .ff90 files), or the *–fpp* option was used, the macro __NAG_COMPILER_RELEASE is automatically defined to be the integer release number (major*10+minor, e.g. 61 for release 6.1), and the macro __NAG_COMPILER_BUILD is automatically defined to be the build number (for release 6.1 this will have a value greater than 6100).

40 fpp options

–c_com={*yes*|*no*}
By default, C style comments are recognized. Turn this off by specifying *–c_com=no*.

–D*name*
Define the preprocessor variable *name* to be 1 (one). This is the same as if a *–Dname=1* option appeared on the fpp command line, or as if a
#define *name* 1
line appeared in the input file.

–D*name=def*
Define *name* as if by a #define directive. This is the same as if a
#define *name def*
line appeared at the beginning of the input file. The *–D* option has lower precedence than the *–U* option. Thus, if the same name is used in both a *–U* option and a *–D* option, the name will be undefined regardless of the order of the options.

–e Accept extended source lines, up to 132 characters long.

–fixed Specifies fixed format input source.

–free Specifies free format input source.

–I*pathname*
Add *pathname* to the list of directories which are to be searched for #include files whose names do not begin with '/'. If the #include file name is enclosed in double-quotes ("..."), it is searched for first in the directory of the file with the #include line; if the file name was enclosed in angle brackets (<...>) this directory is not searched. Then, the file is searched for in directories named in *–I* options, and finally in directories from the standard list.

-M Generate a list of makefile dependencies and **write** them to the standard output. This list indicates that the object file which would be generated from the input file depends on the input file as well as the include files referenced.

–macro={*yes*|*no_com*|*no*}
By default, macros are expanded **everywhere**. Turn off macro expansion in comments by specifying *–macro=no_com* and turn off macro expansion all together by specifying *–macro=no*

–P Do not put line numbering directives to the output file. Line numbering directives appear as `#`*line-number file-name*

–U*name*
Remove any initial definition of *name*, where *name* is an fpp variable that is predefined by a particular preprocessor. Here is a partial list of variables that might be predefined, depending upon the architecture of the system:

Operating System: `unix`, `__unix` and `__SVR4`;
Hardware: `sun`, `__sun`, `sparc` and `__sparc`.

–undef Remove initial definitions for all predefined symbols.

–w Suppress warning messages.

–w0 Suppress warning messages.

–Xu Convert upper-case letters to lower-case, except within character-string constants. The default is not to convert upper-case letters to lower-case.

–Xw For fixed source form only, treat blanks as insignificant. The default for fpp is that blanks are significant in both source forms.

–Y*directory*
Use the specified *directory* instead of the standard list of directories when searching for `#include` files.

41 Using fpp

41.1 Source files

fpp operates on both fixed and free form source files. Files with the (non-case-sensitive) extension '`.f`', '`.ff`', '`.for`' or '`.ftn`' are assumed to be fixed form source files. All other files (e.g. those with the extension '`.ff90`') are assumed to be free form source files. These assumptions can be overridden by the *–fixed* and *–free* options. Tab format lines are recognised in fixed form.

A source file may contain **fpp** tokens. An fpp token is similar to a Fortran token, and is one of:

- an fpp directive name;
- a symbolic name or Fortran keyword;
- a literal constant;
- a Fortran comment;
- an fpp comment;
- a special character which may be a blank character, a control character, or a graphic character that is not part of one of the previously listed tokens.

41.2 Output

Output consists of a modified copy of the input plus line numbering directives (unless the *–P* option is used). A line numbering directive has the form
`#`*line-number file-name*
and these are inserted to indicate the original source line number and filename of the output line that follows.

41.3 Directives

All fpp directives start with the hash character (`#`) as the first character on a line. Blank and tab characters may appear after the initial '`#`' to indent the directive. The directives are divided into the following groups:

- macro definitions;
- inclusion of external files;
- line number control;
- conditional source code selection.

41.4 Macro definition

The `#define` directive is used to define both simple string variables and more complicated macros:

`#define` *name token-string*

This is the definition of an fpp variable. Wherever '*name*' appears in the source lines following the definition, '*token-string*' will be substituted for it.

`#define` *name([argname1[,argname2]...]) token-string*

This is the definition of a function-like macro. Occurrences of the macro '*name*' followed by a comma-separated list of arguments within parentheses are substituted by the token string produced from the macro definition. Every occurrence of an argument name from the macro definition's argument list is substituted by the token sequence of the corresponding macro actual argument.

Note that there must be no space or tab between the macro name and the left parenthesis of the argument list in this directive; otherwise, it will be interpreted as a simple macro definition with the left parenthesis treated as the first character of the replacement *token-string*.

`#undef` *name*

Remove any macro definition for *name*, whether such a definition was produced by a $-D$ option, a `#define` directive or by default. No additional tokens are permitted on the directive line after the name.

The macro NAGFOR is defined by default.

41.5 Including external files

There are two forms of file inclusion:
`#include "`*filename*`"`
and
`#include <`*filename*`>`

Read in the contents of *filename* at this location. The lines read in from the file are processed by fpp as if they were part of the current file.

When the `<`*filename*`>` notation is used, *filename* is only searched for in the standard "include" directories. See the $-I$ and $-Y$ options above for more detail. No additional tokens are permitted in the directive line after the final '`"`' or '`>`'.

41.6 Line number control

`#`*line-number [*`"`*filename*`"`*]*

Generate line control information for the next pass of the compiler. The *line-number* must be an unsigned integer literal constant, and specifies the line number of the following line. If `"`*filename*`"` does not appear, the current filename is unchanged.

41.7 Conditional selection of source text

There are three forms of conditional selection of source text:

1. ```
   #if condition_1
     block_1
   #elif condition_2
     block_2
   #else
     block_n
   #endif
   ```

2. ```
   #ifdef name
     block_1
   #elif condition
     block_2
   #else
     block_n
   #endif
   ```

3. ```
   #ifndef name
     block_1
   #elif condition
     block_2
   #else
     block_n
   #endif
   ```

The "#else" and "#elif" parts are optional. There may be more than one "#elif" part. Each condition is an expression consisting of fpp constants, macros and macro functions. Condition expressions are similar to cpp expressions, and may contain any cpp operations and operands with the exception of C long, octal and hexadecimal constants. Additionally, fpp will accept and evaluate the Fortran logical operations `.NOT.`, `.AND.`, `.OR.`, `.EQV.`, `.NEQV.`, the relational operators `.GT.`, `.LT.`, `.LE.`, `.GE.`, and the logical constants `.TRUE.` and `.FALSE.`.

42 Preprocessing details

42.1 Scope of macro or variable definitions

The scope of a definition begins from the place of its definition and encloses all the source lines (and source lines from `#include`d files) from that definition line to the end of the current file.

However, it does not affect:

- files included by Fortran `INCLUDE` lines;
- fpp and Fortran comments;
- `IMPLICIT` single letter specifications;
- `FORMAT` statements;
- numeric and character constants.

The scope of the macro effect can be limited by means of the `#undef` directive.
   ```

42.2 End of macro definition

A macro definition can be of any length but is only one logical line. These may be split across multiple physical lines by ending each line but the last with the macro continuation character '\' (backslash). The backslash and newline are not part of the replacement text. The macro definition is ended by a newline that is not preceded by a backslash.

For example:

```
#define long_macro_name(x,\
y) x*y
```

42.3 Function-like macro definition

The number of macro call arguments must be the same as the number of arguments in the corresponding macro definition. An error is produced if a macro is used with the wrong number of arguments.

42.4 Cancelling macro definitions

`#undef` *name*

After this directive, '*name*' will not be interpreted by fpp as a macro or variable name. This directive has no effect if '*name*' is not a macro name.

42.5 Conditional source code selection

`#if` *condition*

Condition is a constant expression, as specified below. Subsequent lines up to the first matching `#elif`, `#else` or `#endif` directive appear in the output only if the *condition* is true.

The lines following a `#elif` directive appear in the output only if

- the condition in the matching `#if` directive was false,
- the conditions in all previous matching `#elif` directives were false, and
- the condition in this `#elif` directive is true.

If the condition is true, all subsequent matching `#elif` and `#else` directives are ignored up to the matching `#endif`.

The lines following a `#else` directive appear in the output only if all previous conditions in the construct were false.

The macro function 'defined' can be used in a constant expression; it is true if and only if its argument is a defined macro name.

The following operations are allowed.

- C language operations: `<`, `>`, `==`, `!=`, `>=`, `<=`, `+`, `-`, `/`, `*`, `%`, `<<`, `>>`, `&`, `~`, `|`, `!`, `&&` and `||`. These are interpreted in accordance with C language semantics, for compatibility with cpp.
- Fortran language operations: `.AND.`, `.OR.`, `.NEQV.`, `.XOR.`, `.EQV.`, `.NOT.`, `.GT.`, `.LT.`, `.LE.`, `.GE.`, `.NE.`, `.EQ.` and `**`.
- Fortran logical constants: `.TRUE.` and `.FALSE.`.

Only these items, integer literal constants, and names can be used within a constant expression. Names that are not macro names are treated as if they were '0'. The C operation '`!=`' (not equal) can be used in `#if` or `#elif` directives, but cannot be used in a `#define` directive, where the character '`!`' is interpreted as the start of a Fortran comment.

`#ifdef` ***name***
: Subsequent lines up to the matching #else, #elif, or #endif appear in the output only if the name has been defined, either by a #define directive or by the $-D$ option, and in the absence of an intervening #undef directive. No additional tokens are permitted on the directive line after name.

`#ifndef` ***name***
: Subsequent lines up to the matching #else, #elif, or #endif appear in the output only if name has not been defined, or if its definition has been removed with an #undef directive. No additional tokens are permitted on the directive line after name.

`#elif` ***constant-expression***
: Any number of #elif directives may appear between an #if, #ifdef, or #ifndef directive and a matching #else or #endif directive.

`#else`
: This inverts the sense of the conditional directive otherwise in effect. If the preceding conditional would indicate that lines are to be included, then lines between the `#else` and the matching `#endif` are ignored. If the preceding conditional indicates that lines would be ignored, subsequent lines are included in the output.

#endif
: End a section of lines begun by one of the conditional directives #if, #ifdef, or #ifndef. Each such directive must have a matching #endif.

42.6 Including external files

Files are searched as follows:

for `#include "`*filename*`"`:

- in the directory, in which the processed file has been found;
- in the directories specified by the $-I$ option;
- in the default directory.

for `#include <`*filename*`>`:

- in the directories specified by the $-I$ option;
- in the default directory.

Fpp directives (lines beginning with the `#` character) can be placed anywhere in the source code, in particular immediately before a Fortran continuation line. The only exception is the prohibition of fpp directives within a macro call divided on several lines by means of continuation symbols.

42.7 Comments

Fpp permits comments of two kinds:

1. Fortran language comments. A source line containing one of the characters '`C`', '`c`', '`*`', '`d`' or '`D`' in the first column is considered to be a comment line. Within such lines macro expansions are not performed. The '`!`' character is interpreted as the beginning of a comment extending to the end of the line. The only exception is the case when this symbol occurs within a constant expression in a `#if` or `#elif` directive.
2. Fpp comments enclosed in the '`/*`' and '`*/`' character sequences. These are excluded from the output. Fpp comments can be nested so that for each opening sequence '`/*`' there must be a corresponding closing sequence '`*/`'. Fpp comments are suitable for excluding the compilation of large portions of source instead of commenting every line with Fortran comment symbols. Using "`#if 0 ... #endif`" achieves the same effect without being ridiculous.

42.8 Macro functions

The macro function

`defined`(*name*) or `defined` *name*

expands to `.TRUE.` if *name* is defined as a macro, and to `.FALSE.` otherwise.

42.9 Macro expression

If, during expansion of a macro, the column width of a line exceeds column 72 (for fixed form) or column 132 (for free form), fpp inserts appropriate continuation lines.

In fixed form there are limitations on macro expansion in the label part of the line (columns 1-5):

- a macro call (together with possible arguments) should not extend past column 5;
- a macro call whose name begins with one of the Fortran comment characters is treated as a comment;
- a macro expansion may produce text extending beyond the column 5 position. In such a case a warning will be issued.

In the fixed form when the $-Xw$ option has been specified an ambiguity may appear if a macro call occurs in a statement position and a macro name begins or coincides with a Fortran keyword. For example, in the following text:

```
#define call p(x)   call f(x)
        call p(0)
```

fpp can not determine with certainty how to interpret the 'call p' token sequence. It could be considered as a macro name. The current implementation does the following:

- the longer identifier is chosen (callp in this case);
- from this identifier the longest macro name or keyword is extracted;
- if a macro name has been extracted a macro expansion is performed. If the name begins with some keyword **fpp** outputs an appropriate warning;
- the rest of the identifier is considered as a whole identifier.

In the above example the macro expansion would be performed and the following warning would be output:

warning: possibly incorrect substitution of macro callp

It should be noted that this situation appears only when preprocessing fixed form source code and when the blank character is not being interpreted as a token delimiter. It should be said also that if a macro name coincides with a keyword beginning part, as in the following case:

```
#define INT    INTEGER*8
        INTEGER k
```

then in accordance with the described algorithm, the INTEGER keyword will be found earlier than the INT macro name. Thus, there will be no warning when preprocessing such a macro definition.

43 `fpp` diagnostics

There are three kinds of diagnostic messages:

- warnings. preprocessing of source code is continued and the return value remains to be 0.

- errors. Fpp continues preprocessing but sets the return code to a nonzero value, namely the number of errors.
- fatal error. Fpp cancels preprocessing and returns a nonzero return value.

The messages produced by fpp are intended to be self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

Extensions

44 Non-standard Extensions

The following extensions to the standard Fortran language are accepted by the NAG Fortran Compiler.

44.1 BOZ literal constants outside DATA statements

The NAG Fortran Compiler accepts hexadecimal, octal and binary literal constants in any context where an integer literal would be accepted. Such constants have a data type of default `INTEGER`, unless there are more digits in the BOZ constant than will fit in a default `INTEGER`; e.g., `Z'12345678'` would be of type default `INTEGER`, and `Z'123456789'` would be a 64-bit integer.

44.2 Longer Names

Names may be up to 199 characters long.

44.3 Dollar Sign in Names

The dollar sign '`$`' may appear in names as if it were a letter. On some systems this name is directly used in the link name, but on others, it is translated to `DOLLAR`.

44.4 Input/output endian/format conversion

Runtime conversion of unformatted input/output files is enabled by the `CONVERT=` specifier. This is an `OPEN` statement specifier, and takes a scalar default character argument; the accepted values are:

`BIG_ENDIAN`
: synonym for `BIG_IEEE`;

`BIG_IEEE_DD`
: big-endian with IEEE single and double precision floating point but with double-double for quad precision;

`BIG_IEEE` big-endian with IEEE floating point include 128-bit quad precision;

`BIG_NATIVE`
: big-endian with native floating-point formats;

`LITTLE_ENDIAN`
: synonym for `LITTLE_IEEE`;

`LITTLE_IEEE_DD`
: little-endian with IEEE single and double precision floating point but with double-double for quad precision;

`LITTLE_IEEE`
: little-endian with IEEE floating point include 128-bit quad precision;

`LITTLE_NATIVE`
: little-endian with native floating-point formats;

`NATIVE` native endianness and data format.

It is also possible to specify this with an environment variable at runtime; when unit n is opened, if the environment variable `FORT_CONVERT`n exists, its value is interpreted as a `CONVERT=` specifier value. If the environment variable exists it takes precedence over any `CONVERT=` specifier in the `OPEN` statement. If the environment variable does not exist, and there is no `CONVERT=` specifier in the `OPEN` statement, the conversion mode is taken from the `-convert=` option at compile time; the default is `NATIVE` (i.e. no conversion).

The `CONVERT=` specifier is also available in the `INQUIRE` statement, and sets its argument to the current conversion mode or to `'UNKNOWN'` if the file is not connected for unformatted input/output.

44.5 Elemental `BIND(C)` procedures

Interoperable (`BIND(C)`) procedures are permitted to be declared as `ELEMENTAL`. Such a procedure must satisfy the normal Fortran requirements for elemental procedures, in particular they must have scalar dummy arguments and must be pure (free from side-effects).

44.6 Maximum array rank is 31

The maximum rank of an array has been increased to 31 (the Fortran 2008 standard only requires 15, and previous Fortran standards only required 7). For example,

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

declares a 30-dimensional array (which will take 4GiB of memory to store).

45 Obsolete Extensions

The following extensions were common in the Fortran 77 era, and are still in frequent use (though they have been superseded and are thus unnecessary). Warning messages (marked 'Extension:') are produced for each occurrence of any extension; these particular ones may be suppressed with the *−w=x77* option.

45.1 Byte Sizes

Byte sizes for `REAL`, `INTEGER`, `LOGICAL` and `COMPLEX` are accepted and mapped to Modern Fortran KINDS.

Byte Size Specification	Standard Fortran	Using F90_KIND or ISO_FORTRAN_ENV
REAL*2	Real	Real(real16)
REAL*4	Real	Real(real32)
REAL*8	Double Precision	Real(real64)
REAL*16	Real(Selected_Real_Kind(30))	Real(real128)
COMPLEX*4	Complex(Selected_Real_Kind(3))	Complex(real16)
COMPLEX*8	Complex	Complex(real32)
COMPLEX*16	Complex(Kind(0d0))	Complex(real64)
COMPLEX*32	Complex(Selected_Real_Kind(30))	Complex(real128)
INTEGER*1	Integer(Selected_Int_Kind(2))	Integer(int8)
INTEGER*2	Integer(Selected_Int_Kind(4))	Integer(int16)
INTEGER*4	Integer	Integer(int32)
INTEGER*8	Integer(Selected_Int_Kind(18))	Integer(int64)
Byte Size Specification	Standard Fortran	Using F90_KIND
LOGICAL*1		Logical(byte)
LOGICAL*2		Logical(twobyte)
LOGICAL*4	Logical	Logical(word)
LOGICAL*8		Logical(logical64)

The byte length may also be overridden in the type declaration, similar to overriding the character length. For example,

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

45.2 TAB Format

The occurrence of a TAB character in fixed-form source is treated as follows:

a. an initial TAB followed by a digit is expanded to 5 spaces (putting the digit in the continuation column),

b. an initial TAB followed by any other character is expanded to 6 spaces (putting the character after the TAB in column 7, making the line the initial line of a statement), and

c. other TAB characters are treated as single blanks except in character context where they remain TABs (but they are still treated as taking one column for the purposes of line length).

45.3 Hollerith Constants

Hollerith constants may be used as actual arguments or to initialise objects in DATA. In either case they must be associated with an object of intrinsic numeric data type, not with a `CHARACTER` or derived type object.

Hollerith i/o (i.e., use of the A edit descriptor with non-`CHARACTER` data) is only enabled if the using subprogram was compiled with the *−hollerith_io* or *−dusty* option.

45.4 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.

45.5 Increased Line Length in Fixed Source Form

The *−132* option increases the effective length of each fixed source form input line from 72 characters to 132 characters.

Note that when this option is used the NAG Fortran Compiler no longer conforms to the Fortran language standard. The meaning of a program will change if it contains a character constant which is continued across a line boundary. A standard-conforming program containing an H-edit descriptor which is continued across a line boundary will very likely be rejected.

For new Fortran programs we recommend the use of free source form instead of this option. Free source form provides superior detection of typographical errors and is also part of the Fortran standard and thus fully portable to all standard-conforming compilers.

45.6 Increased Maximum Number of Continuation Lines

The *−maxcontin=N* option increases the limit on the maximum number of continuation lines to *N*. Note that this option can affect both fixed source form and free source form, but never decreases the continuation line limit below the standard.

The Fortran 90 and 95 standards specified that the maximum number of continuation lines in fixed source form was 19, and that the maximum number of continuation lines in free source form was 39. The Fortran 2003 standard increased this to 255 lines regardless of source form.

45.7 Intrinsic functions with mixed-kind arguments

The `ATAN`, `ATAN2`, `DIM`, `MAX`, `MIN`, `MOD`, `MODULO` and `SIGN` intrinsic functions will accept integer and real arguments that differ in kind; note that integer and real arguments still cannot be mixed in a single intrinsic function reference.

For `SIGN`, the kind of the result is the same as the kind of the first argument (which supplies the magnitude of the result), ignoring the kind of the second argument (which only supplies the sign of the result). For all the others, the kind of the result is the same as for arithmetic operations, i.e. for integers the kind with the largest number of digits, and for reals the kind with the greatest precision.

For example, if `X` is `REAL(real32)` and `Y` is `REAL(real64)`:
`MAX(X,Y)` has kind `real64` and its value is equal to `MAX(REAL(X,real64),Y)`;
`SIGN(X,Y)` has kind `real32` and its value is equal to `SIGN(X,REAL(SIGN(1.0_real64,Y),real32))`.

45.8 `ACCESS='APPEND'` specifier on `OPEN` statement

The VAX FORTRAN specifier `ACCESS='APPEND'` can be used on the `OPEN` statement. It is equivalent to specifying both `ACCESS='SEQUENTIAL'` and `POSITION='APPEND'`. For example,

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

has the same effect as

```
OPEN(17,FILE='my.log',POSITION='APPEND') ! ACCESS='SEQUENTIAL' is the default.
```

This is supported only as an aid to porting old programs; it should be changed to the `POSITION='APPEND'` specifier.

45.9 VAX FORTRAN `TYPE` statement

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

For example,

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

is equivalent to

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

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

45.10 Auto-skipping `NAMELIST` input

The Fortran standard requires that when namelist input is performed, the name after the ampersand in the input record must match the namelist group name (in the `READ` statement). However, a common extension is for the i/o library to "skip forwards" in the file, looking for an input record that matches namelist initial record required.

Normally, the NAG Fortran system raises an i/o error condition if the names do not match, but when auto-skipping namelist input is in effect, instead it skips records until it reaches the end of the file or finds a record that begins with an ampersand and the correct name.

For example, given the program

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

and the input data

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

with auto-skipping namelist it will print

```
 Result 123 456
```

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

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

45.11 Legacy Application Support

The *−dusty* option downgrades the category of common errors found in "legacy" software from "Error" to "Warning", allowing such programs to be compiled and run. (The messages may be suppressed altogether by additionally specifying the *−w* option.)

This option also effectively provides the extensions of allowing `COMMON` blocks to be initialised outside of `BLOCK DATA`, and of accepting VAX format octal and hexadecimal constants (these have the forms `'...'O` and `'...'X` respectively).

45.12 Mismatched Argument Lists

The *−mismatch* option downgrades checking of procedure argument lists so that inconsistencies between calls to routines which are not in the current file being processed produce warning messages rather than error messages.

The *−mismatch_all* option further downgrades argument list checking so that incorrect calls to routines present in the current file being processed produce warning messages instead of error messages.

45.13 Double Precision Complex Extensions

Double precision complex entities may be declared with the `DOUBLE COMPLEX` keywords instead of the standard Fortran '`Complex(Kind(0d0))`' specification.

If the *−dcfuns* option has been used, additional intrinsic functions are available (see the documentation of the option for full details). These functions have all been redundant since Fortran 90.

Intrinsic Modules

46 Intrinsic Module Overview

A number of intrinsic modules are provided that are available for use in programs. An intrinsic module is one that is pre-compiled or built in to the compiler system; several of these are part of Fortran 2003, others are specific to NAG.

The standard intrinsic modules from Fortran 2003 that are available are:

ieee_arithmetic	Special facilities for IEEE floating-point arithmetic
ieee_exceptions	IEEE arithmetic exception handling
ieee_features	IEEE feature control
iso_c_binding	Facilities for interface to C functions
iso_fortran_env	Fortran-specific environmental facilities

The non-standard intrinsic modules supplied by NAG are:

f90_gc	Garbage collector control
f90_iostat	Input/output error codes (source form provided)
f90_kind	Kind number parameters (source form provided)
f90_preconn_io	File preconnection control
f90_stat	STAT= error codes (source form provided)
f90_unix_dir	Unix system functions — directories and files
f90_unix_dirent	Unix system functions — directory reading
f90_unix_env	Unix system functions — environment
f90_unix_errno	Unix system functions — error codes
f90_unix_io	Unix system functions — input/output (incomplete)
f90_unix_proc	Unix system functions — processes

Note that although the above f90_unix_* modules contain Unix-specific functions, in many cases these are also usable on Windows and Mac OS X; a function that is not available can still be called but will return the `ENOSYS` error code.

47 f90_gc

A module is provided for controlling the garbage collector more precisely, called "`F90_GC`". It contains the following procedures described below.

In the description of each procedure, an argument whose `KIND` is denoted by '*' can accept any kind of that type. Other `KIND` indications use the named parameters from the `F90_KIND` module; these named parameters are not, however, exported from `F90_GC`.

```
SUBROUTINE ENABLE_INCREMENTAL_GC()
```

Enables incremental garbage collection; once enabled, it cannot be disabled. Full collections will still be performed with a frequency determined by the "Full GC frequency" setting (see `GET_FULL_GC_FREQUENCY` and `SET_FULL_GC_FREQUENCY`.

```
LOGICAL(word) FUNCTION EXPAND_HEAP(N)
INTEGER(*),INTENT(IN) :: N
```

This function attempts to expand the heap by N blocks (of 4K bytes), returning `.TRUE.` if and only if it is successful. This is unaffected by the heap expansion setting (see `GET_HEAP_EXPANSION` and `SET_HEAP_EXPANSION`).

Note that on some systems this may return a false indication of success as the operating system delays the actual allocation of memory until an attempt is made to use it — at which point the program may be aborted. Therefore this function should *not* be used to attempt to allocate all the virtual memory on a system.

```
SUBROUTINE GCOLLECT()
```

Manually initiates a garbage collection; this is a full collection even if incremental collection has been enabled (see ENABLE_INCREMENTAL_GC). This is not affected by the "GC allowed" setting (see GET_GC_ALLOWED and SET_GC_ALLOWED).

```
SUBROUTINE GET_BYTES_SINCE_GC(NBYTES)
INTEGER(int32 or int64),INTENT(OUT) :: NBYTES
```

Returns the number of bytes allocated since the last collection.

```
SUBROUTINE GET_FREE_BYTES(NBYTES)
INTEGER(int32 or int64),INTENT(OUT) :: NBYTES
```

Returns a conservative estimate of the number of free bytes in the heap.

```
SUBROUTINE GET_FULL_GC_FREQUENCY(FREQUENCY)
INTEGER(int16 or int32 or int64),INTENT(OUT) :: FREQUENCY
```

Returns the number of partial collections that are done between each full collection when incremental collection is enabled (see ENABLE_INCREMENTAL_GC). This has no effect on manual collection (see GCOLLECT) or when incremental collection has not been enabled. The default value in Release 5.2 is 4.

```
SUBROUTINE GET_GC_ALLOWED(ALLOWED)
LOGICAL(*),INTENT(OUT) :: ALLOWED
```

Returns the "GC allowed" setting; the default setting is .TRUE.

```
SUBROUTINE GET_GC_VERBOSITY(VERBOSITY)
INTEGER(*),INTENT(OUT) :: VERBOSITY
```

Returns the "GC verbosity" level; this has a range of 0-100, and the default level is zero. Increasing the verbosity level increases the number of informational messages producing during garbage collection. In Release 5.2, the only significant verbosity levels are 0 and 10.

```
SUBROUTINE GET_HEAP_EXPANSION(ALLOWED)
LOGICAL(*),INTENT(OUT) :: ALLOWED
```

Returns the automatic heap expansion setting; the default setting is .TRUE.. Even when automatic heap expansion is disabled, the heap will still be expanded if that is necessary to satisfy an allocation request.

```
SUBROUTINE GET_HEAP_SIZE(NBYTES)
INTEGER(int32 or int64),INTENT(OUT) :: NBYTES
```

Returns the current heap size in bytes, or -1 if the heap size cannot be represented in the integer variable (i.e. if the current heap size is more than 2G bytes and the integer variable is only a 32-bit one).

```
SUBROUTINE GET_MAX_GC_RETRIES(N_RETRIES)
INTEGER(*),INTENT(OUT) :: N_RETRIES
```

Returns the maximum number of garbage collections attempted before reporting failure to the caller; if the allocation was not initiated by an ALLOCATE statement with a STAT= clause, this will result in program termination. The default value in Release 5.2 is zero.

```
SUBROUTINE GET_MAX_HEAP_SIZE(NBYTES)
INTEGER(int32 or int64),INTENT(OUT) :: NBYTES
```

Returns the maximum heap size setting, or zero if no maximum heap size has been set. The default setting is zero.

```
INTEGER(int32) FUNCTION NCOLLECTIONS()
```

Returns the number of garbage collections that have been performed.

```
SUBROUTINE SET_FULL_GC_FREQUENCY(FREQUENCY)
INTEGER(int16 or int32 or int64),INTENT(IN) :: FREQUENCY
```

Sets the number of partial garbage collections to be done between each full collection; this has no effect unless incremental collection has been enabled (see `ENABLE_INCREMENTAL_GC`). A full collection may still be done if there is a substantial increase in the number of in-use blocks.

```
SUBROUTINE SET_GC_ALLOWED(ALLOWED)
LOGICAL(*),INTENT(IN) :: ALLOWED
```

Controls the "GC allowed" setting; when this setting is `.FALSE.`, automatic garbage collection is inhibited. This does not affect manual garbage collection (see *GCOLLECT*).

```
SUBROUTINE SET_GC_VERBOSITY(VERBOSITY)
INTEGER(*),INTENT(IN) :: VERBOSITY
```

Sets the verbosity level; the default value is zero, and the range is limited to 0-100. In Release 5.2, the only significant levels are 0 and 10.

```
SUBROUTINE SET_HEAP_EXPANSION(ALLOWED)
LOGICAL(*),INTENT(IN) :: ALLOWED
```

Controls whether automatic heap expansion is allowed; if this is `.FALSE.`, automatic heap expansion will only occur if necessary to satisfy an allocation request (normally the heap is expanded when an heuristic determines that it would be advantageous). This does not affect manual heap expansion (see `EXPAND_HEAP`).

```
SUBROUTINE SET_MAX_GC_RETRIES(N_RETRIES)
INTEGER(*),INTENT(IN) :: N_RETRIES
```

Sets the maximum number of garbage collections attempted before reporting out of memory after heap expansion fails (i.e. is refused by the operating system). The default setting in Release 5.2 is zero.

```
SUBROUTINE SET_MAX_HEAP_SIZE(N)
INTEGER(int32 or int64),INTENT(IN) :: N
```

Sets the maximum size of the heap to N bytes. This will prevent the heap from automatic expansion beyond the specified limit, and prevent it from automatic expansion entirely if it is already beyond the limit.

48 f90_iostat

This module contains definitions of integer parameters for all the `IOSTAT` values that can be returned as a result of use of an input/output or data transfer statement.

For example:

```
USE f90_iostat
INTEGER ios
OPEN (10, FILE='a.b', IOSTAT=iost, STATUS='NEW')
IF (iost==IOERR_NEW_FILE_EXISTS) PRINT *, "File a.b existed already"
```

49 f90_kind

This module contains definitions of integer parameters that can be used as kind numbers. Users wishing to write portable software making use of non-default kinds should `USE` this module and use the parameters instead of numeric values. For example, users should use `LOGICAL(KIND=BYTE)` instead of `LOGICAL(KIND=1)`. The available `KIND` parameters are shown below; their exact meanings (i.e. the values they represent) are implementation dependent.

```
INTEGER,PARAMETER :: SINGLE
```

For `REAL` and `COMPLEX`, selects the default real or default complex kind; this is equivalent to leaving the `KIND` selector off entirely.

```
INTEGER,PARAMETER :: DOUBLE
```

Selects the double precision real kind; this is equivalent to declaring `REAL` entities using the `DOUBLE PRECISION` type specifier, to declaring `COMPLEX` entities using `COMPLEX(KIND(0d0))`, and to using the exponent letter `D` on literal constants.

```
INTEGER,PARAMETER :: QUAD
```

`REAL/COMPLEX` kind selector for real and complex types with approximately twice the precision of `DOUBLE`. This might not be available on some systems; on a system without this type, the value of this parameter will be negative.

```
INTEGER,PARAMETER :: REAL16
```

`REAL/COMPLEX` kind selector for real and complex types that are represented using 16-bit floating-point numbers.

```
INTEGER,PARAMETER :: REAL32
```

`REAL/COMPLEX` kind selector for real and complex types that are represented using 32-bit floating-point numbers.

```
INTEGER,PARAMETER :: REAL64
```

`REAL/COMPLEX` kind selector for real and complex types that are represented using 64-bit floating-point numbers.

```
INTEGER,PARAMETER :: REAL64x2
```

`REAL/COMPLEX` kind selector for real and complex types that are represented using "double-double" floating-point numbers. A double-double floating-point number consists of two 64-bit values, one of which is at least `DIGITS(1._REAL64)` smaller than the other; this has almost twice the precision of `REAL64` (except when near zero), but a smaller exponent range.

This type is not available on all systems; on a system without this type, the value of this parameter is -1.

```
INTEGER,PARAMETER :: REAL128
```

`REAL/COMPLEX` kind selector for real and complex types that are represented using 128-bit floating-point numbers. This will select a "true 128-bit" floating-point type if one is available, and if not it will select a "double-double" floating-point type if that is available; if no 128-bit floating-point type is available the value of this parameter is -1.

```
INTEGER,PARAMETER :: INT8
```

`INTEGER` kind selector for integer types with at least 8 bits of precision.

```
  INTEGER,PARAMETER :: INT16
```

`INTEGER` kind selector for integer types with at least 16 bits of precision.

```
  INTEGER,PARAMETER :: INT32
```

`INTEGER` kind selector for integer types with at least 32 bits of precision.

```
  INTEGER,PARAMETER :: INT64
```

`INTEGER` kind selector for integer types with at least 64 bits of precision.

```
  INTEGER,PARAMETER :: BYTE
```

`LOGICAL` kind selector for logical types occupying only one byte of memory.

```
  INTEGER,PARAMETER :: TWOBYTE
```

`LOGICAL` kind selector for logical types occupying the same space as `INTEGER(INT16)` entities.

```
  INTEGER,PARAMETER :: WORD
```

`LOGICAL` kind selector for a 32-bit logical type.

```
  INTEGER,PARAMETER :: LOGICAL64
```

`LOGICAL` kind selector for a 64-bit logical type.

```
  INTEGER,PARAMETER :: ASCII
```

`CHARACTER` kind selector for the ASCII character set.

```
  INTEGER,PARAMETER :: JIS
```

`CHARACTER` kind selector for the JIS X 0213:2004 character set.

```
  INTEGER,PARAMETER :: UCS2
```

`CHARACTER` kind selector for the UCS-2 (Unicode) character set.

```
  INTEGER,PARAMETER :: UCS4
```

`CHARACTER` kind selector for the UCS-4 (ISO 10646) character set.

50 f90_preconn_io

This module enables alteration of the default values used by automatically preconnected files.

50.1 Procedures

Note that in the descriptions below, `LOGICAL(*)` means any kind of `LOGICAL`. Also, note that although the `VERBOSE` argument is an optional argument, the actual argument is not permitted to be a non-present optional dummy argument.

```
SUBROUTINE GET_PCIO_OPTIONS(BLANK,POSITION,PREFIX,VERBOSE)
CHARACTER(*),OPTIONAL,INTENT(OUT) :: BLANK,POSITION,PREFIX
LOGICAL(*),OPTIONAL,INTENT(OUT) :: VERBOSE
```

Returns the current settings for file preconnection; the meanings for these settings are described under the `IOINIT` routine. Both the `BLANK` and `POSITION` values are returned in upper case, even if lower case was used in a call to `IOINIT` to set them.

```
SUBROUTINE IOINIT(BLANK,POSITION,PREFIX,VERBOSE)
CHARACTER(*),OPTIONAL,INTENT(IN) :: BLANK,POSITION,PREFIX
LOGICAL(*),OPTIONAL,INTENT(IN) :: VERBOSE
```

This procedure controls automatic file preconnection. Files can be automatically preconnected when a logical unit is initially referenced without an OPEN statement; preconnection does not occur again after file CLOSE.

The name of the file to be preconnected is determined first by searching for an environment variable of the form `FORT`*nn* where *nn* is the two-digit logical unit number; e.g. `FORT03`. If this environment variable is found its value is used as the filename; otherwise the name "`fort.`*n*" is used; e.g. '`fort.3`'. The file is opened either with `FORM='FORMATTED'` or `FORM='UNFORMATTED'` depending on whether the initial reference is with a formatted or unformatted i/o statement.

The `BLANK` and `POSITION` arguments control the `BLANK=` and `POSITION=` keywords in the implied `OPEN` statement when the file is preconnected; initially these are set to `BLANK='NULL'` and `POSITION='REWIND'`. Note that these have no meaning (and no effect) for unformatted files.

The `PREFIX` argument changes the *prefix* used to find the environment variable containing the preconnected file name. Initially this is '`FORT`'. Only the first 30 characters of `PREFIX` are used.

The `VERBOSE` argument controls activity reporting; if it is `.TRUE.`, subsequent preconnection activity will produce informative messages on the standard error unit.

50.2 Example

```
   USE F90_PRECONN_IO
   CALL IOINIT(BLANK='ZERO',PREFIX='MYFILE')
   READ (99,10) I,J,K,L,M
10 FORMAT(I3,I1,I3,I1,I3)
```

When the READ statement is executed, the environment variable `MYFILE99` is interrogated to discover which file is to be automatically preconnected to unit 99. The unit will be connected with `BLANK='ZERO'`. If no environment variable is found the file '`fort.99`' will be opened.

51 f90_stat

This module contains definitions of integer parameters for all the `STAT=` values that can be returned as a result of use of an `ALLOCATE` or `DEALLOCATE` statement.

51.1 Parameters

```
  INTEGER,PARAMETER :: STAT_ALREADY_ALLOCATED
```

An allocatable variable in an ALLOCATE statement is already currently allocated.

```
INTEGER,PARAMETER :: STAT_MEMORY_LIMIT_EXCEEDED
```

An allocation in an ALLOCATE statement requested more memory than the limit in this version of the NAG Fortran compiler.

```
INTEGER,PARAMETER :: STAT_NO_MEMORY
```

Insufficient free memory available to satisfy the requested allocation.

```
INTEGER,PARAMETER :: STAT_NOT_ALLOCATED
```

An allocatable variable in a DEALLOCATE statement is not currently allocated.

```
INTEGER,PARAMETER :: STAT_NOT_ASSOCIATED
```

A pointer in a DEALLOCATE statement is disassociated.

```
INTEGER,PARAMETER :: STAT_PART_OF_A_LARGER_OBJECT
```

A pointer in a DEALLOCATE statement refers to part of a larger object.

```
INTEGER,PARAMETER :: STAT_POINTER_UNDEFINED
```

A pointer in a DEALLOCATE statement is undefined. (This value is never returned to the user program, which is always immediately terminated if the use of an undefined pointer is detected.)

```
INTEGER,PARAMETER :: STAT_WRONG_COLOUR
```

A pointer in a DEALLOCATE statement is associated with a target that was not created by pointer allocation.

51.2 Example

```
USE f90_stat
REAL,ALLOCATABLE :: big(:,:,:)
INTEGER :: status
ALLOCATE(big(100,1024,1024),STAT=status)
IF (status==STAT_NO_MEMORY) PRINT *,'Out of memory'
```

52 f90_unix_*

These modules are described in the next part, *Modern Fortran API to Posix*.

53 ieee_*, iso_c_binding, iso_fortran_env

These modules are described in the *Fortran 2003 Extensions* part, under *10.6 IEEE arithmetic support*, *10.5.1 The ISO_C_BINDING module* and *10.8.5 The ISO_FORTRAN_ENV module* respectively.

Modern Fortran API to Posix

54 Posix Module Overview

The following modules are provided by NAG as a partial interface to the operating system facilities defined by ISO/IEC 9945-1:1990 Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language].

f90_unix_dir	Directories and Files
f90_unix_dirent	Directory Reading
f90_unix_env	Environment
f90_unix_errno	Error Codes
f90_unix_file	File Characteristics
f90_unix_io	Input/Output (incomplete)
f90_unix_proc	Processes

55 f90_unix_dir

This module contains part of a Fortran API to functions detailed in ISO/IEC 9945-1:1990 Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language].

The procedures in this module are from sections 5.2: Working Directory, 5.3.3 Set File Creation Mask, 5.3.4 Link to a File, 5.4 Special File Creation and 5.5 File Removal.

Error handling is described in `F90_UNIX_ERRNO`. Note that for procedures with an optional `ERRNO` argument, if an error occurs and `ERRNO` is not present, the program will be terminated.

All the procedures in this module are both generic and specific.

55.1 Parameters

```
INTEGER,PARAMETER :: MODE_KIND
```

The integer kind used to represent file permissions (see ISO/IEC 9945-1). Parameters for specific permissions are contained in `F90_UNIX_FILE`.

55.2 Procedures

```
SUBROUTINE CHDIR(PATH,ERRNO)
CHARACTER(*),INTENT(IN) :: PATH
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Sets the current working directory to `PATH`. Note that any trailing blanks in `PATH` may be significant. If `ERRNO` is present it receives the error status.

Possible error conditions include `EACCES`, `ENAMETOOLONG`, `ENOTDIR` and `ENOENT` (see `F90_UNIX_ERRNO`).

```
SUBROUTINE GETCWD(PATH,LENPATH,ERRNO)
CHARACTER(*),OPTIONAL,INTENT(OUT) :: PATH
INTEGER(int32),OPTIONAL,INTENT(OUT) :: LENPATH
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Accesses the current working directory information. If `PATH` is present, it receives the name of the current working directory, blank-padded or truncated as appropriate if the length of the current working directory name differs from

that of PATH. If LENPATH is present, it receives the length of the current working directory name. If ERRNO is present it receives the error status.

If neither PATH nor LENPATH is present, error EINVAL is raised. If the path to current working directory cannot be searched, error EACCES is raised. If PATH is present and LENPATH is not present, and PATH is shorter than the current working directory name, error ERANGE is raised. (See F90_UNIX_ERRNO).

```
SUBROUTINE LINK(EXISTING,NEW,ERRNO)
CHARACTER(*),INTENT(IN) :: EXISTING,NEW
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Creates a new link (with name given by NEW) for an existing file (named by EXISTING).

Possible errors include EACCES, EEXIST, EMLINK, ENAMETOOLONG, ENOENT, ENOSPC, ENOTDIR, EPERM, EROFS, EXDEV (see F90_UNIX_ERRNO).

```
SUBROUTINE MKDIR(PATH,MODE,ERRNO)
CHARACTER(*),INTENT(IN) :: PATH
INTEGER(mode_kind),INTENT(IN) :: MODE
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Creates a new directory with name given by PATH and mode MODE (see F90_UNIX_FILE for mode values). Note that any trailing blanks in PATH may be significant.

Possible errors include EACCES, EEXIST, EMLINK, ENAMETOOLONG, ENOENT, ENOSPC, ENOTDIR and EROFS (see F90_UNIX_ERRNO).

```
SUBROUTINE MKFIFO(PATH,MODE,ERRNO)
CHARACTER(*),INTENT(IN) :: PATH
INTEGER(mode_kind),INTENT(IN) :: MODE
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Creates a new FIFO special file with name given by PATH and mode MODE. Note that any trailing blanks in PATH may be significant.

Possible errors include EACCES, EEXIST, ENAMETOOLONG, ENOENT, ENOSPC, ENOTDIR and EROFS (see F90_UNIX_ERRNO).

```
SUBROUTINE RENAME(OLD,NEW,ERRNO)
CHARACTER(*),INTENT(IN) :: OLD
CHARACTER(*),INTENT(IN) :: NEW
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Changes the name of the file OLD to NEW. Any existing file NEW is first removed. Note that any trailing blanks in OLD or NEW may be significant.

Possible errors include EACCES, EBUSY, EEXIST, ENOTEMPTY, EINVAL, EISDIR, ENAMETOOLONG, EMLINK, ENOENT, ENOSPC, ENOTDIR, EROFS and EXDEV (see F90_UNIX_ERRNO).

```
SUBROUTINE RMDIR(PATH,ERRNO)
CHARACTER(*),INTENT(IN) :: PATH
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Removes the directory PATH. Note that any trailing blanks in PATH may be significant.

Possible errors include EACCES, EBUSY, EEXIST, ENOTEMPTY, ENAMETOOLONG, ENOENT, ENOTDIR and EROFS (see F90_UNIX_ERRNO).

```
SUBROUTINE UMASK(CMASK,PMASK)
INTEGER(mode_kind),INTENT(IN) :: CMASK
INTEGER(mode_kind),OPTIONAL,INTENT(OUT) :: PMASK
```

Sets the file mode creation mask of the calling process to CMASK. If PMASK is present it receives the previous value of the mask.

```
SUBROUTINE UNLINK(PATH,ERRNO)
CHARACTER(*),INTENT(IN) :: PATH
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Deletes the file PATH. Note that any trailing blanks in PATH may be significant.

Possible errors include EACCES, EBUSY, ENAMETOOLONG, ENOENT, ENOTDIR, EPERM and EROFS (see F90_UNIX_ERRNO).

56 f90_unix_dirent

This module contains part of a Fortran API to functions detailed in ISO/IEC 9945-1:1990 Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language].

The functions in this module are from Section 5.1.2: Directory Operations

Error handling is described in F90_UNIX_ERRNO. Note that for procedures with an optional ERRNO argument, if an error occurs and ERRNO is not present, the program will be terminated.

All the procedures in this module are specific and not generic.

56.1 Procedures

In the description of each procedure, an argument whose KIND is denoted by '*' can accept any kind of that type. Other KIND indications use the named parameters from the F90_KIND or F90_UNIX_ERRNO modules; these named parameters are not, however, exported from F90_UNIX_DIRENT.

```
SUBROUTINE CLOSEDIR(DIRUNIT,ERRNO)
INTEGER(*),INTENT(IN) :: DIRUNIT
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Close a directory stream that was opened by OPENDIR.

If DIRUNIT does not refer to an open directory stream, error EBADF (see F90_UNIX_ERRNO) is raised.

```
SUBROUTINE OPENDIR(DIRNAME,DIRUNIT,ERRNO)
CHARACTER(*),INTENT(IN) :: DIRNAME
INTEGER(*),INTENT(OUT) :: DIRUNIT
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Opens a directory stream, returning a handle to it in DIRUNIT.

Possible errors include EACCES, ENAMETOOLONG, ENOENT, ENOTDIR, EMFILE and ENFILE (see F90_UNIX_ERRNO).

```
SUBROUTINE READDIR(DIRUNIT,NAME,LENNAME,ERRNO)
INTEGER(*),INTENT(IN) :: DIRUNIT
CHARACTER(*),INTENT(OUT) :: NAME
INTEGER(int32 or int64),INTENT(OUT) :: LENNAME
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Reads the first/next directory entry. The name of the file is placed into NAME, blank-padded or truncated as appropriate if the length of the file name differs from LEN(NAME). The length of the file name is placed in LENNAME.

If there are no more directory entries, NAME is unchanged and LENNAME is negative.

If DIRUNIT is not a directory stream handle produced by OPENDIR, or has been closed by CLOSEDIR, error EBADF (see F90_UNIX_ERRNO) is raised.

```
SUBROUTINE REWINDDIR(DIRUNIT,ERRNO)
INTEGER(*),INTENT(IN) :: DIRUNIT
INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Rewinds the directory stream so that the next call to READDIR on that stream will return the name of the first file in the directory.

57 f90_unix_env

This module contains part of a Fortran API to functions detailed in ISO/IEC 9945-1:1990 Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language].

The functions in this module are from Section 4: Process Environment, plus gethostname from 4.3BSD.

Error handling is described in F90_UNIX_ERRNO. Note that for procedures with an optional ERRNO argument, if an error occurs and ERRNO is not present, the program will be terminated.

All the procedures in this module are generic; some are also specific (this may change in a future release).

57.1 Parameters

```
INTEGER(int32),PARAMETER :: clock_tick_kind
```

The integer kind used for clock ticks (see TIMES).

```
INTEGER(int32),PARAMETER :: id_kind
```

Integer kind for representing IDs; that is, users (uids), groups (gids), process groups (pgids) and processes (pids).

```
INTEGER(int32),PARAMETER :: long_kind
```

The integer kind corresponding to the C type 'long'. This is not actually used in this module, but is suitable for use when declaring a variable to receive a value from SYSCONF.

```
INTEGER(int32),PARAMETER :: sc_stdin_unit, sc_stdout_unit, sc_stderr_unit,
    sc_arg_max, sc_child_max, sc_clk_tck, sc_job_control, sc_open_max,
    sc_ngroups_max, sc_saved_ids, sc_stream_max, sc_tzname_max, sc_version
```

Values used as arguments to SYSCONF. The following table describes the returned information from SYSCONF, this is not the value of the SC_* constant.

SC_STDIN_UNIT
: The logical unit number for standard input (READ *,...); this is the same value as INPUT_UNIT in the standard intrinsic module ISO_FORTRAN_ENV.

SC_STDOUT_UNIT
: The logical unit number for standard output (PRINT, WRITE(*,...)); this is the same value as OUTPUT_UNIT in the standard intrinsic module ISO_FORTRAN_ENV.

SC_STDERR_UNIT
: The logical unit number on which errors are reported; this is the same value as ERROR_UNIT in the standard intrinsic module ISO_FORTRAN_ENV.

SC_ARG_MAX
: Maximum length of arguments for the EXEC functions, in bytes, including environment data.

SC_CHILD_MAX
: Maximum number of simultaneous processes for a single user.

SC_CLK_TCK
: Number of clock ticks per second. (This is the same value returned by the CLK_TCK function).

SC_JOB_CONTROL
: Value available only if job control is supported by the operating system.

SC_NGROUPS_MAX
: Maximum number of simultaneous supplementary group IDs per process.

SC_OPEN_MAX
: Maximum number of files open simultaneously by a single process.

SC_SAVED_IDS
: Value available only if each process has a saved set-uid and set-gid.

SC_STREAM_MAX
: Maximum number of logical units that can be simultaneously open. Not always available.

SC_TZNAME_MAX
: Maximum number of characters for the name of a time zone.

SC_VERSION
: Posix version number. This will be 199009 if the underlying operating system's C interface conforms to ISO/IEC 9945-1:1990.

```
INTEGER(int32),PARAMETER :: time_kind
```

The integer kind used for holding date/time values (see TIME).

57.2 Types

```
TYPE tms
    INTEGER(clock_tick_kind) utime, stime, cutime, cstime
END TYPE
```

Derived type holding CPU usage time in clock ticks. UTIME and STIME contain CPU time information for a process, CUTIME and CSTIME contain CPU time information for its terminated child processes. In each case this is divided into user time (UTIME, CUTIME) and system time (STIME, CSTIME).

```
TYPE utsname
    CHARACTER(...) sysname, nodename, release, version, machine
END TYPE
```

Derived type holding data returned by UNAME. Note that the character length of each component is fixed, but may be different on different systems. The values in these components are blank-padded (if short) or truncated (if long). For further information see ISO/IEC 9945-1:1990.

57.3 Procedures

```
PURE INTEGER(KIND=clock_tick_kind) FUNCTION clk_tck()
```

Returns the number of clock ticks in one second of CPU time (see TIMES).

```
PURE SUBROUTINE ctermid(s,lens)
    CHARACTER(*),OPTIONAL,INTENT(OUT) :: s
    INTEGER(int32),OPTIONAL,INTENT(OUT) :: lens
```

If present, S is set to the filename of the controlling terminal. If present, LENS is set to the length of the filename of the controlling terminal. If S is longer than the filename of the controlling terminal it is padded with blanks; if S is shorter it is truncated — it is the user's responsibility to check the value of LENS to detect such truncation.

If the filename of the controlling terminal cannot be determined for any reason LENS (if present) will be set to zero and S (if present) will be blank.

```
SUBROUTINE getarg(k,arg,lenarg,errno)
    INTEGER(*),INTENT(IN) :: k
    CHARACTER(*),OPTIONAL,INTENT(OUT) :: arg
    INTEGER(int32),OPTIONAL,INTENT(OUT) :: lenarg
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Accesses command-line argument number K, where argument zero is the program name. If ARG is present, it receives the argument text (blank-padded or truncated as appropriate if the length of the argument differs from that of ARG). If LENARG is present, it receives the length of the argument. If ERRNO is present, it receives the error status.

Note that if K is less than zero or greater than the number of arguments (returned by IARGC) error EINVAL (see F90_UNIX_ERRNO) is raised.

This procedure is very similar to the standard intrinsic procedure GET_COMMAND_ARGUMENT.

```
  PURE INTEGER(id_kind) FUNCTION getegid()
```

Returns the effective group number of the calling process.

```
SUBROUTINE getenv(name,value,lenvalue,errno)
    CHARACTER(*),INTENT(IN) :: name
    CHARACTER(*),OPTIONAL,INTENT(OUT) :: value
    INTEGER(int32),OPTIONAL,INTENT(OUT) :: lenvalue
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Accesses the environment variable named by NAME. If VALUE is present, it receives the text value of the variable (blank-padded or truncated as appropriate if the length of the value differs from that of VALUE). If LENVALUE is present, it receives the length of the value. If ERRNO is present, it receives the error status.

If there is no such variable, error EINVAL (see F90_UNIX_ERRNO) is raised. Other possible errors include ENOMEM.

```
PURE INTEGER(id_kind) FUNCTION geteuid()
```

Returns the effective user number of the calling process.

```
PURE INTEGER(id_kind) FUNCTION getgid()
```

Returns the group number of the calling process.

```
SUBROUTINE getgroups(grouplist,ngroups,errno)
    INTEGER(id_kind),OPTIONAL,INTENT(OUT) :: grouplist(:)
    INTEGER(int32),OPTIONAL,INTENT(OUT) :: ngroups
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Retrieves supplementary group number information for the calling process. If `GROUPLIST` is present, it is filled with the supplementary group numbers. If `NGROUPS` is present, it receives the number of supplementary group numbers. If `ERRNO` is present, it receives the error status.

If `GROUPLIST` is present but too small to contain the complete list of supplementary group numbers, error `EINVAL` (see `F90_UNIX_ERRNO`) is raised. The maximum number of supplementary group numbers can be found using `SYSCONF` (enquiry `SC_NGROUPS_MAX`); alternatively, '`CALL GETGROUPS(NGROUPS=N)`' will reliably return the actual number in use.

```
PURE SUBROUTINE gethostname(name,lenname)
    CHARACTER(*),OPTIONAL,INTENT(OUT) :: name
    INTEGER(int32),OPTIONAL,INTENT(OUT) :: lenname
```

This provides the functionality of 4.3BSD's *gethostname*. If `NAME` is present it receives the text of the standard host name for the current processor, blank-padded or truncated if appropriate. If `LENNAME` is present it receives the length of the host name. If no host name is available `LENNAME` will be zero.

```
PURE SUBROUTINE getlogin(s,lens)
    CHARACTER(*),OPTIONAL,INTENT(OUT) :: s
    INTEGER(int32),OPTIONAL,INTENT(OUT) :: lens
```

Accesses the user name (login name) associated with the calling process. If `S` is present, it receives the text of the name (blank-padded or truncated as appropriate if the length of the login name differs from that of `S`). If `LENS` is present, it receives the length of the login name.

```
PURE INTEGER(id_kind) FUNCTION getpgrp()
```

Returns the process group number of the calling process.

```
PURE INTEGER(id_kind) FUNCTION getpid()
```

Returns the process number of the calling process.

```
PURE INTEGER(id_kind) FUNCTION getppid()
```

Returns the process number of the parent of the calling process.

```
PURE INTEGER(id_kind) FUNCTION getuid()
```

Returns the user number of the calling process.

```
PURE INTEGER(int32) FUNCTION iargc()
```

Returns the number of command-line arguments; this is the same value as the intrinsic function `COMMAND_ARGUMENT_COUNT`, except that it returns -1 if even the program name is unavailable (the intrinsic function erroneously returns the same value, 0, whether the program name is available or not).

```
SUBROUTINE isatty(lunit,answer,errno)
    INTEGER(*),INTENT(IN) :: lunit
    LOGICAL(*),INTENT(OUT) :: answer
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

ANSWER receives the value .TRUE. if and only if the logical unit identified by LUNIT is connected to a terminal.

If LUNIT is not a valid unit number or is not connected to any file, error EBADF (see F90_UNIX_ERRNO) is raised.

```
SUBROUTINE setgid(gid,errno)
    INTEGER(*),INTENT(IN) :: gid
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Sets the group number of the calling process to GID. For full details refer to section 4.2.2 of ISO/IEC 9945-1:1990.

If GID is not a valid group number, error EINVAL (see F90_UNIX_ERRNO) is raised. If the process is not allowed to set the group number to GID, error EPERM is raised.

```
SUBROUTINE setpgid(pid,pgid,errno)
    INTEGER(*),INTENT(IN) :: pid, pgid
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Sets the process group number of process PID (or, if PID is zero, the calling process) to PGID. For full details refer to section 4.3.3 of ISO/IEC 9945-1:1990.

Possible errors include EACCES, EINVAL, ENOSYS, EPERM, ESRCH (see F90_UNIX_ERRNO).

```
SUBROUTINE setsid(sid,errno)
    INTEGER(id_kind),OPTIONAL,INTENT(OUT) :: sid
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Creates a session and sets the process group number of the calling process. For full details refer to section 4.3.2 of ISO/IEC 9945-1:1990. If SID is present it receives the new session id (equal to the process id); if an error occurs it receives -1.

Possible errors include EPERM (see F90_UNIX_ERRNO).

```
SUBROUTINE setuid(uid,errno)
    INTEGER(*),INTENT(IN) :: uid
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Sets the user number of the calling process to UID. For full details refer to section 4.2.2 of ISO/IEC 9945-1:1990.

If UID is not a valid group number, error EINVAL (see F90_UNIX_ERRNO) is raised. If the process is not allowed to set the user number to UID, error EPERM is raised.

```
SUBROUTINE sysconf(name,val,errno)
    INTEGER(*),INTENT(IN) :: name
    INTEGER(*),INTENT(OUT) :: val
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Returns the value of a system configuration variable. The variables are named by integer PARAMETERs defined in this module, and are described in the **Parameters** section.

If NAME is not a valid configuration variable name, error EINVAL (see F90_UNIX_ERRNO) is raised. If VAL is too small an integer kind to contain the result, error ERANGE is raised; kind LONG_KIND is guaranteed to be big enough for all values.

```
SUBROUTINE time(itime,errno)
    INTEGER(time_kind),INTENT(OUT) :: itime
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

ITIME receives the operating system date/time in seconds since the Epoch.

```
INTEGER(KIND=clock_tick_kind) FUNCTION times(buffer)
    TYPE(tms),INTENT(OUT) :: buffer
```

This function returns the elapsed real time in clock ticks since an arbitrary point in the past, or -1 if the function is unavailable. `BUFFER` is filled in with CPU time information for the calling process and any terminated child processes.

If this function returns zero the values in `BUFFER` will still be correct but the elapsed-time timer was not available.

```
SUBROUTINE ttyname(lunit,s,lens,errno)
    INTEGER(*),INTENT(IN) :: lunit
    CHARACTER(*),OPTIONAL,INTENT(OUT) :: s
    INTEGER(int32),OPTIONAL,INTENT(OUT) :: lens
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Accesses the name of the terminal connected to the logical unit identified by `LUNIT`. If `S` is present, it receives the text of the terminal name (blank-padded or truncated as appropriate, if the length of the terminal name differs from that of `S`). If `LENS` is present, it receives the length of the terminal name. If `ERRNO` is present, it receives the error status.

If `LUNIT` is not a valid logical unit number, or is not connected, error `EBADF` (see `F90_UNIX_ERRNO`) is raised; otherwise, if the function is not available, `ENOSYS` is raised, or if `LUNIT` is not connected to a terminal, error `ENOTTY` is raised.

```
SUBROUTINE uname(name,errno)
    TYPE(UTSNAME),INTENT(OUT) :: name
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Returns information about the operating system in `NAME`.

58 f90_unix_errno

This module contains part of a Fortran API to functions detailed in ISO/IEC 9945-1:1990 Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language].

The facilities in this module are from Section 2.4 Error Numbers.

58.1 Error Handling

Many procedures provided by the F90_UNIX_* modules have an optional `ERRNO` argument, declared as

```
INTEGER(error_kind),OPTIONAL,INTENT(OUT)
```

If this argument is provided it receives the error status from the procedure; zero indicates successful completion, otherwise it will be a non-zero error code, usually one of the ones listed in this module.

If the `ERRNO` argument is omitted and an error condition is raised, the program will be terminated with an informative error message.

If a procedure has no `ERRNO` argument it indicates that no error condition is possible - the procedure always succeeds.

58.2 Parameters

All parameters are of type `INTEGER` with kind `ERROR_KIND`. The following table lists the error message typically associated with each error code; for full details see ISO/IEC 9945-1:1990, either section 2.4 or the appropriate section for the function raising the error.

E2BIG	Arg list too long
EACCES	Permission denied
EAGAIN	Resource temporarily unavailable
EBADF	Bad file descriptor
EBUSY	Resource busy
ECHILD	No child process
EDEADLK	Resource deadlock avoided
EDOM	Domain error
EEXIST	File exists
EFAULT	Bad address
EFBIG	File too large
EINTR	Interrupted function call
EINVAL	Invalid argument
EIO	Input/Output error
EISDIR	Is a directory
EMFILE	Too many open files
EMLINK	Too many links
ENAMETOOLONG	Filename too long
ENFILE	Too many open files in system
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOLCK	No locks available
ENOMEM	Not enough space
ENOSPC	No space left on device
ENOSYS	Function not implemented
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTTY	Inappropriate I/O control operation
ENXIO	No such device or address
EPERM	Operation not permitted
EPIPE	Broken pipe
ERANGE	Result too large
EROFS	Read-only file system
ESPIPE	Invalid seek
ESRCH	No such process
EXDEV	Improper link

59 f90_unix_file

This module contains part of a Fortran API to functions detailed in ISO/IEC 9945-1:1990 Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language].

The functions in this module are from Section 5.6 File Characteristics.

Error handling is described in F90_UNIX_ERRNO. Note that for procedures with an optional ERRNO argument, if an error occurs and ERRNO is not present, the program will be terminated.

All the procedures in this module are generic; some may be specific but this is subject to change in a future release.

59.1 Parameters

```
INTEGER(int32),PARAMETER :: F_OK
```

Flag for requesting file existence check (see ACCESS).

```
USE F90_UNIX_ENV,ONLY:ID_KIND
```

See F90_UNIX_ENV for a description of this parameter.

```
USE F90_UNIX_DIR,ONLY:MODE_KIND
```

See F90_UNIX_DIR for a description of this parameter.

```
INTEGER(int32),PARAMETER :: R_OK
```

Flag for requesting file readability check (see ACCESS).

```
INTEGER(MODE_KIND),PARAMETER :: S_IRGRP
```

File mode bit indicating group read permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IROTH
```

File mode bit indicating other read permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IRUSR
```

File mode bit indicating user read permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IRWXG
```

Mask to select the group accessibility bits from a file mode (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IRWXO
```

Mask to select the other accessibility bits from a file mode (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IRWXU
```

Mask to select the user accessibility bits from a file mode (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_ISGID
```

File mode bit indicating that the file is set-group-ID (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_ISUID
```

File mode bit indicating that the file is set-user-ID (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IWGRP
```

File mode bit indicating group write permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IWOTH
```

File mode bit indicating other write permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IWUSR
```

File mode bit indicating user write permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IXGRP
```

File mode bit indicating group execute permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IXOTH
```

File mode bit indicating other execute permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IXUSR
```

File mode bit indicating user execute permission (see STAT_T).

```
USE F90_UNIX_ENV,ONLY :: TIME_KIND
```

See F90_UNIX_ENV for a description of this parameter.

```
INTEGER(int32),PARAMETER :: W_OK
```

Flag for requesting file writability check (see ACCESS).

```
INTEGER(int32),PARAMETER :: X_OK
```

Flag for requesting file executability check (see ACCESS).

59.2 Types

```
TYPE stat_t
    INTEGER(MODE_KIND) st_mode
    INTEGER(...) st_ino
    INTEGER(...) st_dev
    INTEGER(...) st_nlink
    INTEGER(id_kind) st_uid
    INTEGER(id_kind) st_gid
    INTEGER(...) st_size
    INTEGER(TIME_KIND) st_atime, st_mtime, st_ctime
END TYPE
```

Derived type holding file characteristics.

ST_MODE File mode (read/write/execute permission for user/group/other, plus set-group-ID and set-user-ID bits).

ST_INO File serial number.

ST_DEV ID for the device on which the file resides.

ST_NLINK The number of links (see F90_UNIX_DIR, LINK operation) to the file.

ST_UID User number of the file's owner.

ST_GID Group number of the file.

ST_SIZE File size in bytes (regular files only).

ST_ATIME Time of last access.

ST_MTIME Time of last modification.

ST_CTIME Time of last file status change.

```
TYPE UTIMBUF
    INTEGER(time_kind) actime, modtime
END TYPE
```

Data type holding time values for communication to UTIME. ACTIME is the new value for ST_ATIME, MODTIME is the new value for ST_MTIME.

59.3 Procedures

```
PURE SUBROUTINE access(path,amode,errno)
    CHARACTER(*),INTENT(IN) :: path
    INTEGER(*),INTENT(IN) :: amode
    INTEGER(error_kind),INTENT(OUT) :: errno
```

Checks file accessibility according to the value of AMODE; this should be F_OK or a combination of R_OK, W_OK and X_OK. In the latter case the values may be combined by addition or the intrinsic function IOR.

The result of the accessibility check is returned in ERRNO, which receives zero for success (i.e. the file exists for F_OK, or all the accesses requested by the R_OK et al combination are allowed) or an error code indicating the reason for access rejection. Possible rejection codes include EACCES, ENAMETOOLONG, ENOENT, ENOTDIR and EROFS (see F90_UNIX_ERRNO).

If the value of AMODE is invalid, error EINVAL is returned.

Note that most ACCESS enquiries are equivalent to an INQUIRE statement, in particular:

```
CALL ACCESS(PATH,F_OK,ERRNO)
```
returns success (ERRNO==0) if and only if
INQUIRE(FILE=PATH,EXIST=LVAR) would set LVAR to .TRUE.;

```
CALL ACCESS(PATH,R_OK,ERRNO)
```
returns success (ERRNO==0) if and only if
INQUIRE(FILE=PATH,READ=CHVAR) would set CHVAR to 'YES';

```
CALL ACCESS(PATH,W_OK,ERRNO)
```
returns success (ERRNO==0) if and only if
INQUIRE(FILE=PATH,WRITE=CHVAR) would set CHVAR to 'YES';

```
CALL ACCESS(PATH,IOR(W_OK,R_OK),ERRNO)
```
returns success (ERRNO==0) if and only if
INQUIRE(FILE=PATH,READWRITE=CHVAR) would set CHVAR to 'YES'.

The only differences being that ACCESS returns a reason for rejection, and can test file executability.

```
SUBROUTINE CHMOD(PATH,MODE,ERRNO)
    CHARACTER(*),INTENT(IN) :: PATH
    INTEGER(*),INTENT(IN) :: MODE
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Sets the file mode (ST_MODE) to MODE.

Possible errors include EACCES, ENAMETOOLONG, ENOTDIR, EPERM and EROFS (see F90_UNIX_ERRNO).

```
SUBROUTINE CHOWN(PATH,OWNER,GROUP,ERRNO)
    CHARACTER(*),INTENT(IN) :: PATH
    INTEGER(id_kind),INTENT(IN) :: OWNER, GROUP
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

Changes the owner (ST_UID) of file PATH to OWNER, and the group number (ST_GID) of the file to GROUP.

Possible errors include EACCES, EINVAL, ENAMETOOLONG, ENOTDIR, ENOENT, EPERM and EROFS (see F90_UNIX_ERRNO).

```
SUBROUTINE FSTAT(LUNIT,BUF,ERRNO)
    INTEGER(*),INTENT(IN) :: LUNIT
    TYPE(stat_t),INTENT(OUT) :: BUF
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

BUF receives the characteristics of the file connected to logical unit LUNIT.

If LUNIT is not a valid logical unit number or is not connected to a file, error EBADF is raised (see F90_UNIX_ERRNO).

```
PURE LOGICAL(word) FUNCTION isblk(mode)
    INTEGER(mode_kind),INTENT(IN) :: mode
```

Returns .TRUE. if and only if the MODE value indicates that the file is a "block device".

```
PURE LOGICAL(word) FUNCTION ischr(mode)
    INTEGER(mode_kind),INTENT(IN) :: mode
```

Returns .TRUE. if and only if the MODE value indicates that the file is a "character device".

```
PURE LOGICAL(word) FUNCTION isdir(mode)
    INTEGER(mode_kind),INTENT(IN) :: mode
```

Returns .TRUE. if and only if the MODE value indicates that the file is a directory (or folder).

```
PURE LOGICAL(word) FUNCTION isfifo(mode)
    INTEGER(mode_kind),INTENT(IN) :: mode
```

Returns .TRUE. if and only if the MODE value indicates that the file is a "FIFO" (named or unnamed pipe).

```
PURE LOGICAL(word) FUNCTION isreg(mode)
    INTEGER(mode_kind),INTENT(IN) :: mode
```

Returns .TRUE. if and only if the MODE value indicates that the file is a "regular" (i.e. normal) file.

```
SUBROUTINE STAT(PATH,BUF,ERRNO)
    CHARACTER(*),INTENT(IN) :: PATH
    TYPE(stat_t),INTENT(OUT) :: BUF
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: ERRNO
```

BUF receives the characteristics of the file PATH.

Possible errors include EACCES, ENAMETOOLONG, ENOENT and ENOTDIR (see F90_UNIX_ERRNO).

```
SUBROUTINE utime(path,times,errno)
    CHARACTER(*),INTENT(IN) :: path
    TYPE(utimbuf),OPTIONAL,INTENT(IN) :: times
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Set the access and modification times of the file named by `PATH` to those specified by the `ACTIME` and `MODTIME` components of `TIMES` respectively.

Possible errors include `EACCES`, `ENAMETOOLONG`, `ENOENT`, `ENOTDIR`, `EPERM` and `EROFS` (see `F90_UNIX_ERRNO`).

60 f90_unix_io

This module will contain part of a Fortran API to functions detailed in ISO/IEC 9945-1:1990 Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language].

In this release only the `FLUSH` procedure is supported. Users are advised to use the `ONLY` clause when using this module, as it will have further names added in later releases.

Error handling is described in `F90_UNIX_ERRNO`. Note that for procedures with an optional `ERRNO` argument, if an error occurs and `ERRNO` is not present, the program will be terminated.

All procedures in this module are generic and may also be specific (this may change in a future release).

60.1 Procedures

```
SUBROUTINE FLUSH(LUNIT,ERRNO)
INTEGER(*),INTENT(IN) :: LUNIT
INTEGER(ERROR_KIND),OPTIONAL,INTENT(OUT) :: ERRNO
```

Flushes the output buffer of logical unit `LUNIT`. The effect is similar to the Fortran 2003 `FLUSH` statement, except that the statement allows flushing a unit which is not connected but the procedure does not.

If `LUNIT` is not a valid unit number or is not connected to a file, error `EBADF` is raised (see `F90_UNIX_ERRNO`).

61 f90_unix_proc

This module contains part of a Fortran API to functions detailed in ISO/IEC 9945-1:1990 Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language].

The functions in this module are from Section 3: Process Primitives, excluding 3.3 Signals. The C language functions `abort`, `atexit`, `exit` and `system` are also provided by this module.

Error handling is described in `F90_UNIX_ERRNO`. Note that for procedures with an optional `ERRNO` argument, if an error occurs and `ERRNO` is not present, the program will be terminated.

All the procedures in this module are generic; some may be specific but this is subject to change in a future release.

61.1 Parameters

```
INTEGER(int32),PARAMETER :: atomic_int
```

Integer kind for “atomic” operations in signal handlers (see `ALARM`).

```
INTEGER(int32),PARAMETER :: atomic_log
```

Logical kind for “atomic” operations in signal handlers (see ALARM).

```
USE f90_unix_env, ONLY: pid_kind=>id_kind
```

Integer kind for representing process IDs; this has been superseded by ID_KIND from F90_UNIX_ENV.

```
USE f90_unix_env, ONLY: time_kind
```

Integer kind for representing times in seconds.

```
INTEGER(int32),PARAMETER :: wnohang
```

Option bit for WAITPID indicating that the calling process should not wait for the child process to stop or exit.

```
INTEGER(int32),PARAMETER :: wuntraced
```

Option bit for WAITPID indicating that status should be returned for stopped processes as well as terminated ones.

61.2 Procedures

```
SUBROUTINE abort(message)
    CHARACTER(*),OPTIONAL :: message
```

ABORT cleans up the i/o buffers and then terminates execution, producing a core dump on Unix systems. If MESSAGE is given it is written to logical unit 0 (zero) preceded by ‘ abort:’.

```
SUBROUTINE alarm(seconds,subroutine,secleft,errno)
    INTEGER(*),INTENT(IN) :: seconds
    INTERFACE
        SUBROUTINE subroutine()
        END
    END INTERFACE
    OPTIONAL subroutine
    INTEGER(time_kind),OPTIONAL,INTENT(OUT) :: secleft
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Establishes an “alarm” call to the procedure SUBROUTINE to occur after SECONDS seconds have passed, or cancels an existing alarm if SECONDS==0. If SUBROUTINE is not present, any previous association of a subroutine with the alarm signal is left unchanged. If SECLEFT is present, it receives the number of seconds that were left on the preceding alarm or zero if there were no existing alarm.

The subroutine invoked by the alarm call is only permitted to define VOLATILE SAVEd variables that have the type INTEGER(atomic_int) or LOGICAL(atomic_log); defining or referencing any other kind of variable may result in unpredictable behaviour, even program termination. Furthermore, it shall not perform any array or CHARACTER operations, input/output, or invoke any intrinsic function or module procedure.

If an alarm call is established with no handler (i.e. SUBROUTINE was not present on the first call) the process may be terminated when the alarm goes off.

Possible errors include ENOSYS.

```
SUBROUTINE atexit(subroutine,errno)
    INTERFACE
        SUBROUTINE subroutine()
        END
    END INTERFACE
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Registers an argumentless subroutine for execution on normal termination of the program.

If the program terminates normally, all subroutines registered with ATEXIT will be invoked in reverse order of their registration. Normal termination includes using the F90_UNIX_PROC procedure EXIT, executing a Fortran STOP statement or executing a main program END statement. ATEXIT subroutines are invoked before automatic file closure.

If the program terminates due to an error or by using the F90_UNIX_PROC procedure FASTEXIT, these subroutines will not be invoked.

Possible errors include ENOMEM.

Note: The list of ATEXIT procedures registered via Fortran is separate from those registered via C; the latter will be invoked after all files have been closed.

```
SUBROUTINE execl(path,arg0...,errno)
    CHARACTER(*),INTENT(IN) :: path
    CHARACTER(*),INTENT(IN) :: arg0...
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Executes a file instead of the current image, like EXECV. The arguments to the new program are specified by the dummy arguments which are named ARG0, ARG1, etc. up to ARG20 (additional arguments may be provided in later releases). Note that these are not optional arguments, any actual argument that is itself an optional dummy argument must be present. This function is the same as EXECV except that the arguments are provided individually instead of via an array; and because they are provided individually, there is no need to provide the lengths (the lengths being taken from each argument itself).

Errors are the same as for EXECV.

```
SUBROUTINE execlp(file,arguments,,errno)
    CHARACTER(*),INTENT(IN) :: file
    CHARACTER(*),INTENT(IN) :: arguments
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Executes a file instead of the current image, like EXECV. The arguments to the new program are specified by the dummy arguments which are named ARG0, ARG1, etc. up to ARG20 (additional arguments may be provided in later releases). Note that these are not optional arguments, any actual argument that is itself an optional dummy argument must be present. This function is the same as EXECL except that determination of the program to be executed follows the same rules as EXECVP.

Errors are the same as for EXECV.

```
SUBROUTINE execv(path,argv,lenargv,errno)
    CHARACTER(*),INTENT(IN) :: path
    CHARACTER(*),INTENT(IN) :: argv(:)
    INTEGER(*),INTENT(IN) :: lenargv(:)
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Executes the file PATH in place of the current process image; for full details see ISO/IEC 9945-1:1990 section 3.1.2. ARGV is the array of argument strings, LENARGV containing the desired length of each argument. If ARGV is not zero-sized, ARGV(1)(:LENARGV(1)) is passed as argument zero (i.e. the program name).

If LENARGV is not the same shape as ARGV, error EINVAL is raised (see F90_UNIX_ERRNO). Other possible errors include E2BIG, EACCES, ENAMETOOLONG, ENOENT, ENOTDIR, ENOEXEC and ENOMEM.

```
SUBROUTINE EXECVE(path,argv,lenargv,env,lenenv,errno)
    CHARACTER(*),INTENT(IN) :: path
    CHARACTER(*),INTENT(IN) :: argv(:)
    INTEGER(*),INTENT(IN) :: lenargv(:)
    CHARACTER(*),INTENT(IN) :: env(:)
    INTEGER(*),INTENT(IN) :: lenenv(:)
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Similar to EXECV, with the environment strings specified by ENV and LENENV being passed to the new program; for full details see ISO/IEC 9945-1:1990 section 3.1.2.

If LENARGV is not the same shape as ARGV or LENENV is not the same shape as ENV, error EINVAL is raised (see F90_UNIX_ERRNO). Other errors are the same as for EXECV.

```
SUBROUTINE execvp(file,argv,lenargv,errno)
    CHARACTER(*),INTENT(IN) :: file
    CHARACTER(*),INTENT(IN) :: argv(:)
    INTEGER(*),INTENT(IN) :: lenargv(:)
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

The same as EXECV except that the program to be executed, FILE, is searched for using the PATH environment variable (unless it contains a slash character, in which case EXECVP is identical in effect to EXECV).

Errors are the same as for EXECV.

```
SUBROUTINE exit(status)
    INTEGER(int32),OPTIONAL,INTENT(IN) :: status
```

Terminate execution as if executing the END statement of the main program (or an unadorned STOP statement). If STATUS is given it is returned to the operating system (where applicable) as the execution status code.

```
SUBROUTINE fastexit(status)
    INTEGER,OPTIONAL,INTENT(IN) :: status
```

This provides the functionality of ISO/IEC 9945-1:1990 function _exit (section 3.2.2). There are two main differences between FASTEXIT and EXIT:

1. When EXIT is called all open logical units are closed (as if by a CLOSE statement). With FASTEXIT this is not done, nor are any file buffers flushed, thus the contents and status of any file connected at the time of calling FASTEXIT are undefined.
2. Subroutines registered with ATEXIT are not executed.

```
SUBROUTINE fork(pid,errno)
    INTEGER(id_kind),INTENT(OUT) :: pid
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Creates a new process which is an exact copy of the calling process. In the new process, the value returned in PID is zero; in the calling process the value returned in PID is the process ID of the new (child) process.

Possible errors include EAGAIN, ENOMEM and ENOSYS (see F90_UNIX_ERRNO).

```
SUBROUTINE pause(errno)
    INTEGER(error_kind),INTENT(OUT) :: errno
```

Suspends process execution until a signal is raised. If the action of the signal was to terminate the process, the process is terminated without returning from PAUSE. If the action of the signal was to invoke a signal handler (e.g. via ALARM), process execution continues after return from the signal handler.

If process execution is continued after a signal, ERRNO is set to EINTR.

If this functionality is not available, ERRNO is set to ENOSYS.

```
PURE SUBROUTINE sleep(seconds,secleft)
    INTEGER(*),INTENT(IN) :: seconds
    INTEGER(time_kind),OPTIONAL,INTENT(OUT) :: secleft
```

Suspends process execution for SECONDS seconds, or until a signal has been delivered. If SECLEFT is present, it receives the number of seconds remaining in the sleep time (zero unless the sleep was interrupted by a signal).

```
SUBROUTINE system(string,status,errno)
    CHARACTER(*),INTENT(IN) :: string
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: status,errno
```

Passes STRING to the command processor for execution. If STATUS is present it receives the completion status - this is the same status returned by WAIT and can be decoded with WIFEXITED etc. If ERRNO is present it receives the error status from the SYSTEM call itself.

Possible errors are those from FORK or EXECV.

```
SUBROUTINE wait(status,retpid,errno)
    INTEGER(int32),OPTIONAL,INTENT(OUT) :: status
    INTEGER(id_kind),OPTIONAL,INTENT(OUT) :: retpid
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Wait for any child process to terminate (returns immediately if one has already terminated). See ISO/IEC 9945-1:1990 section 3.2.1 for full details.

If STATUS is present it receives the termination status of the child process. If RETPID is present it receives the process number of the child process.

Possible errors include ECHILD and EINTR (see F90_UNIX_ERRNO).

```
SUBROUTINE waitpid(pid,status,options,retpid,errno)
    INTEGER(id_kind),INTENT(IN) :: pid
    INTEGER(int32),OPTIONAL,INTENT(OUT) :: status
    INTEGER(int32),OPTIONAL,INTENT(IN) :: options
    INTEGER(id_kind),OPTIONAL,INTENT(OUT) :: retpid
    INTEGER(error_kind),OPTIONAL,INTENT(OUT) :: errno
```

Wait for a particular child process to terminate (or for any one if PID==(-1)). If OPTIONS is not present it is as if it were present with a value of 0. See ISO/IEC 9945-1:1990 section 3.2.1 for full details.

Possible errors include ECHILD, EINTR and EINVAL (see F90_UNIX_ERRNO).

```
PURE INTEGER(int32) FUNCTION wexitstatus(stat_val)
    INTEGER(int32),INTENT(IN) :: stat_val
```

If WIFEXITED(STAT_VAL) is .TRUE., this function returns the low-order 8 bits of the status value supplied to EXIT or FASTEXIT by the child process. If the child process executed a STOP statement or main program END statement, the value will be zero. If WIFEXITED(STAT_VAL) is .FALSE., the function value is undefined.

```
PURE LOGICAL(word) FUNCTION wifexited(stat_val)
    INTEGER(error_kind),INTENT(IN) :: stat_val
```

Returns .TRUE. if and only if the child process terminated by calling FASTEXIT, EXIT, by executing a STOP statement or main program END statement, or possibly by some means other than Fortran.

```
PURE LOGICAL(word) FUNCTION wifsignaled(stat_val)
    INTEGER(int32),INTENT(IN) :: stat_val
```

Returns .TRUE. if and only if the child process terminated by receiving a signal that was not caught.

```
PURE LOGICAL(word) FUNCTION wifstopped(stat_val)
    INTEGER(int32),INTENT(IN) :: stat_val
```

Returns `.TRUE.` if and only if the child process is stopped (and not terminated). Note that `WAITPID` must have been used with the `WUNTRACED` option to receive such a status value.

```
PURE INTEGER(int32) FUNCTION wstopsig(stat_val)
    INTEGER(int32),INTENT(IN) :: stat_val
```

If `WIFSTOPPED(STAT_VAL)` is `.TRUE.`, this function returns the signal number that caused the child process to stop. If `WIFSTOPPED(STAT_VAL)` is `.FALSE.`, the function value is undefined.

```
PURE INTEGER(int32) FUNCTION wtermsig(stat_val)
    INTEGER(int32),INTENT(IN) :: stat_val
```

If `WIFSIGNALED(STAT_VAL)` is `.TRUE.`, this function returns the signal number that caused the child process to terminate. If `WIFSIGNALED(STAT_VAL)` is `.FALSE.`, the function value is undefined.

Standard Fortran 95

62 Fortran 95 Program Structure

This section contains a quick reference guide to the structure of a Fortran 95 program.

Expressions are described in the next section.

- Syntax element definition is represented by '::='.
- Names in italics (e.g., '*expr*') are syntax element names.
- Items in '[ ]' are optional.
- Ellipsis '...' indicates optional repetition of the preceding item.
- '{' and '}' are for grouping only.
- Items separated by '|' are a "one-of-many" selection.
- Statements end in '*-stmt*' and are described in the section "Fortran 95 Statements".

INCLUDE Line

The `INCLUDE` line has the form:

```
INCLUDE char-literal-constant
```

The *char-literal-constant* must not have a *kind-param* and must be the pathname of an accessible file; relative pathnames will be searched for in the current working directory and all directories named by $-I$ options. The `INCLUDE` line is effectively replaced by the contents of the named file.

Program Unit Structure

program-unit ::= *main-program* | *module* | *procedure* | *block-data-subprogram*

main-program ::= [*program-stmt*]
body
end-program-stmt

body ::= *declaration-section*
[*executable-section*]
[*contains-section*]

Note: A *contains-section* cannot appear in an internal procedure (an internal procedure is a procedure in the *contains-section* of another procedure).

contains-section ::= *contains-stmt*
procedure...

module ::= *module-stmt*
[*declaration-section*]
[*contains-section*]
end-module-stmt

procedure ::= *function-subprogram* | *subroutine-subprogram*

function-subprogram ::= *function-stmt*
body
end-function-stmt

subroutine-subprogram ::= *subroutine-stmt*
body
end-subroutine-stmt

block-data-subprogram ::= *block-data-stmt*
declaration-section
end-block-data-stmt

Declaration Section

declaration-section ::= [*use-stmt...*]
[*implicit-part...*]
[*declaration...*]

implicit-part ::= *implicit-stmt* | *parameter-stmt* | *format-stmt* | *entry-stmt*

declaration ::= *derived-type-definition* | *interface-block* | *declarative* |
format-stmt | *entry-stmt*

declarative ::= *allocatable-stmt* | *common-stmt* | *data-stmt* | *dimension-stmt* |
equivalence-stmt | *external-stmt* | *intent-stmt* | *intrinsic-stmt* | *namelist-stmt* |
optional-stmt | *parameter-stmt* | *pointer-stmt* | *private-stmt* | *public-stmt* |
save-stmt | *statement-function-stmt* | *target-stmt* | *type-declaration-stmt*

derived-type-definition ::= *type-stmt*
component-def-stmt...
end-type-stmt

interface-block ::= *interface-stmt*
{ *interface-body* | *module-procedure-stmt* } ...
end-interface-stmt

interface-body ::= *function-stmt declaration-section end-function-stmt* |
subroutine-stmt declaration-section end-subroutine-stmt

Executable Section

executable-section ::= [*executable* | *executable-construct* | *data-stmt*]...

executable ::= *allocate-stmt* | *arithmetic-if-stmt* | *assignment-stmt* | *backspace-stmt* | *call-stmt* |
close-stmt | *computed-goto-stmt* | *continue-stmt* | *cycle-stmt* | *deallocate-stmt*
| *endfile-stmt* | *exit-stmt* | *forall-stmt* | *goto-stmt* | *if-stmt* | *inquire-stmt* |
nullify-stmt | *open-stmt* | *pause-stmt* | *pointer-assignment-stmt* | *print-stmt* |
read-stmt | *return-stmt* | *rewind-stmt* | *stop-stmt* | *where-stmt* | *write-stmt*

executable-construct ::= *select-case-construct* | *do-loop* | *forall-construct* | *if-construct* |
where-construct

select-case-construct ::= *select-stmt*
[*case-stmt executable-section*] ...
end-select-stmt

if-construct ::= *if-then-stmt*
[*executable-section*]
[*elseif-stmt executable-section*] ...
[*else-stmt executable-section*]
endif-stmt

where-construct ::= *where-construct-stmt*
where-body
[*elsewhere-stmt* *where-body*]
endwhere-stmt
where-body ::= [*where-assignment-stmt* | *where-stmt* | *where-construct* |
elsewhere-mask-stmt] ...

forall-construct ::= *forall-construct-stmt*
forall-body
end-forall-stmt
forall-body ::= [*forall-assignment-stmt* | *where-stmt* | *where-construct* | *forall-stmt* |
forall-construct] ...

do-loop ::= *do-stmt*
executable-section
do-ending
do-ending ::= *enddo-stmt* | *executable*

Note: If the *do-stmt* specifies a terminating label, the *do-ending* is the statement with that label and shall not be a *goto-stmt*, *return-stmt*, *stop-stmt*, *exit-stmt*, *cycle-stmt*, *arithmetic-if-stmt* or any form of `END` statement. If the *do-stmt* does not specify a terminating label, the *do-ending* shall be an *enddo-stmt*.

63 Fortran 95 Expressions

This section contains a quick reference guide to expression syntax and semantics in Fortran 95. For more details see Metcalf, Reid and Cohen "Fortran 95/2003 Explained" or the Fortran 95 standard "IS 1539-1:1997".

- Syntax element definition is represented by '::='.
- Names in italics (e.g., '*expr*') are syntax element names.
- Items in '[ ]' are optional.
- Ellipsis '...' indicates optional repetition of the preceding item.
- '{' and '}' are for grouping only.
- Items separated by '|' are a "one-of-many" selection.

General Form

The general form of an expression *expr* is as follows:

expr ::= [*unary-operator*] *operand* [*binary-operator operand*]...

operand ::= *literal-constant* | *object* | *array-constructor* |
structure-constructor | *function-reference* | (*expr*)

binary-operator ::= `+` | `-` | `*` | `/` | `**` | `==` | `/=` | `<` | `>` | `<=` | `>=` | *//* |
`.AND.` | `.OR.` | `.EQV.` | `.NEQV.` | *user-operator*

unary-operator ::= `+` | `-` | `.NOT.` | *user-operator*

user-operator ::= `.`*letter*[*letter...*]`.`

The operators `==`, `/=`, `<`, `>`, `<=` and `>=` may also be represented by `.EQ.`, `.NE.`, `.LT.`, `.GT.`, `.LE.` and `.GE.` respectively.

Operators

Binary operators are associated left-to-right except for exponentiation, which associates right-to-left. The precedence of each operator is shown by the following table, from highest to lowest:

Operators	Binary/Unary
user-operator	Unary
`**`	Binary
`* /`	Binary
`+ -`	Unary
`+ -`	Binary
`//`	Binary
`== /= < > <= >=`	Binary
`.NOT.`	Unary
`.AND.`	Binary
`.OR.`	Binary
`.EQV. .NEQV.`	Binary
user-operator	Binary

Literal Constants

literal-constant ::= *integer-literal* | *real-literal* | *double-precision-literal* |
logical-literal | *complex-literal* | *character-literal*

integer-literal ::= *digit-string* [_ *kind-specifier*]
digit-string ::= { *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* }...
kind-specifier ::= *digit-string* | *name*

Note: *name* must be an `INTEGER PARAMETER` name.

Examples:

```
42
17_1
32767_int16
```

real-literal ::= *mantissa* [`E` *exponent*] [`_` *kind-specifier*] `|`
digit-string `E` *exponent* [`_` *kind-specifier*]

mantissa ::= *digit-string* `.` [*digit-string*] `|`
`.` *digit-string*

exponent ::= [`+` | `-`] *digit-string*

Examples:

```
4.2
.7_1
327E+67_real64
```

double-precision-literal ::= { *mantissa* `|` *digit-string* } `D` *exponent*

Examples:

```
4d2
1.7D-10
```

logical-literal ::= { `.TRUE.` | `.FALSE.` } [`_` *kind-specifier*]

complex-literal ::= `(` *int-or-real-literal* `,` *int-or-real-literal* `)`
int-or-real-literal ::= *integer-literal* `|` *real-literal*

Examples:

```
(42,17_1)
(0,1d0)
```

character-literal ::= [*kind-specifier* `_`] { `'` [*char...*] `'` | `"` [*char...*] `"` }

Note: *char* is any character other than the quoting character, or the quoting character occurring twice (e.g., `''''` is the same as `"'"`).

Examples:

```
'42'
"Say 'Hello'."
```

Constants

constant ::= *literal-constant* `|` *name*

Named constants are declared by the `PARAMETER` statement or by a type-declaration statement with the `PARAMETER` attribute

Objects

object ::= *object-ref* [`%` *object-ref*]... [`(` *substring-range* `)`]
object-ref ::= *name* [*section-subscript-list*]
section-subscript-list ::= `(` *section-subscript* [`,` *section-subscript*]... `)`
section-subscript ::= *expr* | *triplet*
triplet ::= [*expr*] `:` [*expr*] [`:` *expr*]
substring-range ::= [*expr*] `:` [*expr*]

name ::= *letter* [*letter* | *digit* | `_`] ...

letter ::= `A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M` | `N` | `O` | `P` | `Q` | `R` | `S` | `T` | `U` | `V` | `W` | `X` | `Y` | `Z`

variable ::= *object*

constant-subobject ::= *object* | *character-literal* `(` *substring-range* `)`

> Note: For a *variable*, the name in the initial *object-ref* must be that of a variable.
> For a *constant-subobject*, the name in the initial *object-ref* must be that of a constant (i.e., a `PARAMETER`).

An *object* is scalar if and only if all its *object-ref*'s are scalar. An *object-ref* with non-zero rank (i.e., an array) can only be followed by scalar non-pointer *object-ref*s.

An **array element** is a scalar object whose final *object-ref* contains a *section-subscript-list* where each *section-subscript* is a scalar expression. For example

```
ARRAY(I,J)
SCALAR%ARRAY(I,J)
```

A **structure component** is an object with more than one *object-ref* whose final *object-ref* has no *section-subscript-list* and all preceding *object-ref*s are scalar. For example

```
SCALAR%SCALAR
ARRAY(I,J)%SCALAR
```

A **substring** is a scalar object with a *substring-range*. For example

```
SCALAR(I:J)
ARRAY(I,J)(K:L)
```

A **whole array** is an array object with only one *object-ref*.

An **array section** is an array object where a non-final *object-ref* has non-zero rank, or the final *object-ref* has a *section-subscript-list* and one or more of the *section-subscripts* has non-zero rank or is a *triplet*. For example

```
ARRAY(2:M)
ARRAY%SCALAR
```

Array Constructor

array-constructor ::= `(/` *ac-item* [`,` *ac-item*] ... `/)`
ac-item ::= *expr* | *ac-implied-do*
ac-implied-do ::= `(` *expr* [`,` *expr*] ... `,` *name* `=` *expr* `,` *expr* [`,` *expr*] `)`

> Note: *name* is an integer variable name, but its value is not affected by the execution of the array constructor. All expressions in an array constructor must have the same type and type parameters; in particular, for characters, they must have the same length.

Structure Constructor

structure-constructor ::= *name* `(` *expr* [`,` *expr*] ... `)`

Note: *name* must be the name of a derived type. Each expression must be assignment-compatible with the corresponding component of the derived type.

Function Reference

function-reference ::= *name* `(` [*expr* [`,` *expr*] ...] `)`

Note: *name* must be the name of a function.

64 Fortran 95 Statements

This section contains a quick reference guide to the statements in Fortran 95.

- Syntax element definition is represented by '::='.
- Names in italics (e.g., '*expr*') are syntax element names.
- Items in '[ ]' are optional.
- Ellipsis '...' indicates optional repetition of the preceding item.
- '{' and '}' are for grouping only.
- Items separated by '|' are a "one-of-many" selection.

Type Specification

The following syntax is used by several declaration statements.

type-spec ::= *numeric-type* [*kind-specifier*] |
numeric-type `*` *digit-string* |
`DOUBLE PRECISION` |
`CHARACTER` [*char-specifier*]

numeric-type ::= `COMPLEX` | `INTEGER` | `LOGICAL` | `REAL`
kind-specifier ::= `(` [`KIND =`] *expr* `)`

char-specifier ::= `*` *digit-string* |
`*` `(` *char-length* `)` |
`(` *char-length* [`,` [`KIND =`] *expr*] `)` |
`(` `LEN =` *char-length* [`,` `KIND =` *expr*] `)` |
`(` `KIND =` *expr* [`,` `LEN =` *char-length*] `)`

char-length ::= `*` | *expr*

Labels and Construct Names

label ::= *digit-string*

Note that the *digit-string* in a label must contain at most 5 digits, and at least one of them must be non-zero. Leading zeroes are not significant, but do count towards the limit of 5.

Although it is not shown in the syntax definitions, all statements may be labelled and the `FORMAT` statement must be labelled.

construct-name ::= *name*

Construct names are "class 1" names, and must not be the same as any other class 1 name in a subprogram; class 1 names includes variables, procedures, program unit names, et cetera.

Entity Declaration List

The following syntax is used by both Component Declaration and Type Declaration statements.

entity-decl-list ::= *entity-decl* [`,` *entity-decl*] ...
entity-decl ::= *name* [`*` *char-length*] [*array-spec*] [*initial-value*]

array-spec ::= *explicit-shape* | *assumed-shape* | *deferred-shape* | *assumed-size*

explicit-shape ::= `(` *explicit-bound* [`,` *explicit-bound*] ... `)`
explicit-bound ::= [*expr* `:`] *expr*

assumed-shape ::= `(` *assumed-bound* [`,` *assumed-bound*] ... `)`
assumed-bound ::= [*expr*] `:`

deferred-shape ::= `(` *deferred-bound* [`,` *deferred-bound*] ... `)`
deferred-bound ::= `:`

assumed-size ::= `(` [*explicit-bound* `,`]... *assumed-size-bound* `)`
assumed-size-bound ::= [*expr* `:`] `*`

initial-value ::= `=` *expression* | `=>` `NULL()`

Allocatable Statement

`ALLOCATABLE` [`::`] *name* [*deferred-shape*] [`,` *name* [*deferred-shape*]] ...

Declares the listed entities to be allocatable arrays. If array bounds are present, they must be deferred.

Allocate Statement

`ALLOCATE` `(` *allocate-item* [`,` *allocate-item*] ... [`,` `STAT` `=` *variable*] `)`
allocate-item ::= *variable* [*explicit-shape*]

Allocates a pointer or allocatable array. If the allocation fails and the `STAT=` clause is present, the `STAT=` variable will be assigned a non-zero value.

Arithmetic If Statement *(obsolescent)*

`IF` `(` *expr* `)` *label* `,` *label* `,` *label*

Branches to one of three labels depending on whether *expr* is negative, zero or positive respectively. The expression must be scalar and of type integer or real.

Assignment Statement

variable `=` *expr*

The expression is evaluated and assigned to the variable. For intrinsic assignment, it must be assignment-compatible with the variable, that is:

1. If the variable is scalar, the expression must be scalar. If the variable is an array, the expression may be scalar or an array of the same shape.
2. If the variable is of type `LOGICAL`, the expression must be `LOGICAL` but may be any `KIND`.
3. If the variable is of type `CHARACTER`, the expression must be of type `CHARACTER` and of the same `KIND`.
4. If the variable is of type `INTEGER`, `REAL` or `COMPLEX`, the expression must be of type `INTEGER`, `REAL` or `COMPLEX`.
5. If the variable is of derived type, the expression must be of that type.

For defined assignment, there must be a generic interface for `ASSIGNMENT(=)` which matches the types, kinds and ranks of the variable and expression; the user-specified assignment routine is called. Note that in the case of derived types, defined assignment is permitted to override the intrinsic assignment.

Backspace Statement

`BACKSPACE` *expr*
`BACKSPACE` `(` *position-spec-list* `)`

position-spec-list ::= *position-spec* [`,` *position-spec*]
position-spec ::= { [`UNIT=`] *expr* } | { `IOSTAT=` *variable* } | { `ERR=` *label* }

Note: A *position-spec-list* is required to have a `UNIT=` *position-spec*; the `UNIT=` keyword and equals sign may be omitted only if it is the first in the list.

Positions the file connected to the specified unit to the record preceding the current one. An error condition is raised if the file is not connected, or the unit does not support backspacing.

The effect of each *position-spec* is as below:

`UNIT=` Specifies the i/o unit; it must be a scalar integer expression identifying an external file.

`ERR=` Transfers control to the specified label if an error condition occurs during the i/o statement.

`IOSTAT=` Sets the *variable* to a positive number if an error occurs and to zero otherwise.

Block Data Statement

`BLOCK DATA` [*name*]

This is the first statement of a block data subprogram. All but one block data subprogram must be named.

Call Statement

`CALL` *name* [`(` [*actual-arg-list*] `)`]

actual-arg-list ::= *actual-arg* [`,` *actual-arg*]...
actual-arg ::= *expr* `|` `*` *label*

Calls the named subroutine.

Case Statement

`CASE DEFAULT` [*construct-name*]
`CASE (` *case-value-range* [`,` *case-value-range*]... `)` [*construct-name*]

case-value-range ::= *expr* [`:` [*expr*]] `|`
`:` *expr*

Marks the beginning of a `CASE` part (and the end of any preceding `CASE` part). Statements in this part are executed if the corresponding `SELECT` expression value satisfies the appropriate `CASE` condition:

`.EQ.` A *case-value-range* of the form `(`*expr*`)`.

`.LE.` A *case-value-range* of the form `(:`*expr*`)`.

`.GE.` A *case-value-range* of the form `(`*expr*`:)`.

range A *case-value-range* of the form `(`*expr*`:`*expr*`)` is satisfied for values greater than or equal to the first expression, and less than or equal to the second expression.

`DEFAULT` The `CASE DEFAULT` clause is selected if the value does not satisfy any other `CASE` statements in that `SELECT` construct.

Note that within a `SELECT` construct, each `CASE` statement must have distinct conditions so that only one can be satisfied.

Close Statement

`CLOSE` *expr*
`CLOSE (` *position-spec-list* `)`

(See the `BACKSPACE` statement for the *position-spec-list* definition.)

Closes the specified unit.

Common Statement

`COMMON` [/ [*common-block-name*] /] *common-object-list*
[[,] / [*common-block-name*] / *common-object-list*]...

common-object-list ::= *common-object* [, *common-object*]...
common-object ::= *name* [*array-spec*]

Declares a common block. If no *common-block-name* is specified, "blank common" is the common block declared. Multiple `COMMON` statements for the same common block act as if the *common-object-list*s were concatenated in a single statement.

Component Definition Statement

type-spec [[, *component-attribute-list*] ::] *entity-decl-list*

component-attribute-list ::= *component-attr* [, *component-attr*]...
component-attr ::= `DIMENSION` *array-spec* |
`POINTER`

Declares one or more components of a derived type. Any *array-spec* in a component definition must be *deferred-shape* if the `POINTER` attribute is present, and must be *explicit-shape* otherwise. Any *initial-value* that is present defines the default value for that component of any new entities of the type.

Computed Goto Statement *(obsolescent)*

`GOTO (` *label* [, *label*] ...) *expr*

The (integer scalar) expression is evaluated; if it is less than one or greater than the number of labels in the list, control is transferred to the next statement. Otherwise control is transferred to the corresponding label.

Contains Statement

`CONTAINS`

This statement separates the declarations of a module from its contained procedures, and the declarations and executables of a main program or procedure from its contained procedure.

Continue Statement

`CONTINUE`

This is an executable statement that has no effect. If it has a label it may be used as the terminating statement of a `DO` construct or as the target of a `GOTO`, computed-`GOTO` or assigned-`GOTO`.

Cycle Statement

`CYCLE` [*construct-name*]

Begins the next iteration of either the specified `DO` construct, or if *construct-name* is omitted, the innermost enclosing `DO` construct.

Data Statement

`DATA` *data-set* [, *data-set*]...

data-set ::= *data-object-list* / *data-value-list* /

data-object-list ::= *data-object* [, *data-object*]...
data-object ::= *variable* | *data-implied-do*
data-implied-do ::= (*data-object* [, *data-object*]... *do-spec*)

data-value-list ::= *data-value* [, *data-value*]...
data-value ::= [*data-repeat* *] *data-constant*
data-repeat ::= *constant* | *constant-subobject*

data-constant ::= *literal-constant* | NULL() | *structure-constructor* | *object* |
{ + | - } { *real-literal* | *integer-literal* }

Declares the initial value of the specified objects. This implicitly declares those objects to have the SAVE attribute.

Deallocate Statement

DEALLOCATE (*expr* [, *expr*] [, STAT = *variable*])

Deallocates the storage occupied by an allocatable array or pointer. An error is raised if an allocatable array to be deallocated is not allocated, or if a pointer to be deallocated is dissociated or is associated with an object that was not allocated with ALLOCATE.

Dimension Statement

DIMENSION [::] *name array-spec* [, *name array-spec*]...

Declares the name(s) to be arrays with the specified bounds.

Do Statement

[*construct-name* :] DO [*label*] [,] [*loop-control*]

loop-control ::= *do-spec* |
WHILE (logical-expr)
do-spec ::= *name* = *expr* , *expr* [, *expr*]

The initial statement of a DO loop. If *label* is present, the loop ends on the statement with that label, which cannot be a GOTO, RETURN, STOP, EXIT, CYCLE, END or arithmetic IF statement. Nested DO loops can share the same ending statement, provided it is not an ENDDO statement. If the *loop-control* is missing, the DO loop terminates only if control is explicitly transferred outside the loop (e.g., by an EXIT, GOTO or RETURN statement).

If *construct-name* is present, the DO loop must end with an ENDDO statement identified with the same construct-name.

Else Statement

ELSE [*construct-name*]

Begins the ELSE part of an IF-THEN construct. Statements in this part are executed only if the IF condition is false and all ELSEIF conditions at the same level are false. If the IF-THEN statement had a construct-name, the ELSE statement may specify the same construct-name.

Elseif Statement

ELSE IF (*expr*) THEN [*construct-name*]

Begins a (new) ELSEIF part of an IF-THEN construct. Statements in this part are executed only if the IF condition is false, all preceding ELSEIF conditions at the same level are false, and this ELSEIF condition is true. If the IF-THEN statement had a construct-name, the ELSEIF statement may specify the same construct-name.

Elsewhere Statement

`ELSEWHERE` [*construct-name*]

Begins the `ELSEWHERE` part of a `WHERE` construct. The statements in this part are executed only for those elements for which the `WHERE` mask are false, and all `ELSEWHERE` masks at the same level are also false.

If the `WHERE` statement had a construct-name, the `ELSEWHERE` statement may specify the same construct-name.

Elsewhere Mask Statement

`ELSEWHERE ( ` *expr* ` )` [*construct-name*]

Begins a masked `ELSEWHERE` part of a `WHERE` construct. The statements in this part are executed only for those elements for which the previous masks are false and this `ELSEWHERE` mask are true. (The previous masks are the `WHERE` mask and all preceding `ELSEWHERE` masks at the same level in this `WHERE` construct.) Note that the elements of the `ELSEWHERE` mask that do not correspond to false elements of the previous masks are not evaluated.

If the `WHERE` statement had a construct-name, the `ELSEWHERE`-mask statement may specify the same construct-name.

End Block Data Statement

`END` [`BLOCK DATA` [*name*]]

The last statement of a block data subprogram. If *name* is present, the `BLOCK DATA` statement at the beginning of the subprogram must have specified the same name.

Enddo Statement

`END DO` [*construct-name*]

Marks the end of a `DO` construct. The *construct-name* shall be present if and only if it were present on the `DO` statement, and must be the same construct-name if so. If the `DO` statement specifies an ending label, the `ENDDO` statement must be labelled with that label.

Endfile Statement

`ENDFILE` *expr*
`ENDFILE ( ` *position-spec-list* ` )`

(See the `BACKSPACE` statement for the *position-spec-list* definition).

Writes an endfile record to the specified external file, truncating it at the current point.

End Function Statement

`END` [`FUNCTION` [*name*]]

The last statement of a function subprogram. If the function subprogram is a contained subprogram, the keyword `FUNCTION` must be present. If *name* is present, it must be the name of the function.

Endif Statement

`END IF` [*construct-name*]

Marks the end of an `IF-THEN` construct. The *construct-name* shall be present if and only if it were present on the `IF-THEN` statement, and must be the same construct-name if so.

End Interface Statement

`END INTERFACE` [*generic-spec*]

Marks the end of an interface block. If the `INTERFACE` statement had a *generic-spec*, it may appear on the `ENDINTERFACE` statement.

End Module Statement

`END` [`MODULE` [*name*]]

The final statement of a module subprogram. If *name* is present, it must match the name on the `MODULE` statement.

End Program Statement

`END` [`PROGRAM` [*name*]]

The final statement of a main program unit. If *name* is present, the main program must have a `PROGRAM` statement and the names must be the same.

End Select Statement

`END SELECT` [*construct-name*]

Marks the end of a `SELECT` construct. The *construct-name* shall be present if and only if it were present on the `SELECT` statement, and must be the same construct-name if so.

End Type Statement

`END TYPE` [*name*]

Marks the end of a derived type definition. If *name* is present it must be the name of the derived type.

Endwhere Statement

`ENDWHERE` [*construct-name*]

Marks the end of a `WHERE` construct. The *construct-name* shall be present if and only if it were present on the `WHERE` statement, and must be the same construct-name if so.

Equivalence Statement

`EQUIVALENCE` *equivalence-set* [`,` *equivalence-set*]...
equivalence-set ::= `(` *variable* { `,` *variable* }... `)`

Declares each object in an equivalence-set to occupy the same storage.

Entry Statement

`ENTRY` *name* [`(` [*arg-list*] `)`]

Declares an additional entry point to the enclosing subprogram (entry points are not allowed in block data, main program, module and internal subprograms).

External Statement

`EXTERNAL` *name* [`,` *name*]...

Declares the listed names to be external subprograms or block data subprograms.

Exit Statement

`EXIT` [*construct-name*]

Transfers control to the statement following named `DO` loop or, if *construct-name* is omitted, the innermost enclosing `DO` loop.

Forall Statement

`FORALL ( ` *triplet-spec* [`,` *triplet-spec*]... [`,` *expr*] `)` *forall-assignment-stmt*
triplet-spec ::= *name* `=` *expr* `:` *expr* [`:` *expr*]

The iteration space of a `FORALL` statement or construct is the cross-product of the sets of possible index values defined by each *triplet-spec* masked by the final *expr* (if present). Note that the scope of the index names is limited to the `FORALL` statement – a variable with the same name outside the `FORALL` statement is unaffected.

The `FORALL` statement executes the *forall-assignment* statement for each index value set in the iteration space.

Forall Assignment Statement

variable `=` *expr*
variable `=>` *expr*

This is exactly like a normal assignment statement except that the *expr* is evaluated for each element of the iteration space before assignment or pointer assignment to each *variable*. Note that an assignment must not assign to the same element of an array more than once in the iteration space, and if the variable is scalar then the iteration space must be exactly one element.

Forall Construct Statement

`FORALL ( ` *triplet-spec* [`,` *triplet-spec*]... [`,` *expr*] `)`

(See the `FORALL` statement for the *triplet-spec* definition and the explanation of the iteration space.)

Begins a `FORALL` construct.

Format Statement

`FORMAT ( ` [*format-list*] `)`

format-list ::= *format-item* [`,` *format-item*]...

format-item ::= [*digit-string*] { *data-edit* `|` `(` *format-list* `)` } `|` *other-edit*

data-edit ::= { `I` | `B` | `O` | `Z` } *digit-string* [`.` *digit-string*] `|`
{ `F` | `D` } *digit-string* `.` *digit-string* `|`
{ `E` | `EN` | `ES` | `G` }
digit-string `.` *digit-string* [`E` *digit-string*] `|`
`L` *digit-string* `|`
`A` [*digit-string*]

other-edit ::= *digit-string* { `/` | `P` | `X` } `|`
{ `T` | `TR` | `TL` } *digit-string* `|`
character-literal `|`
digit-string H *char*... `|`
`/` | `:` | `BN` | `BZ` | `S` | `SP` | `SS`

> Note: The *character-literal* must not have a *kind-specifier*. The `H` edit descriptor is followed by *digit-string chars*, which may be any character except end-of-line; this edit descriptor is *obsolescent* and the *character-literal* one should be used instead.

Declares an i/o format.

Note: The comma between *format-items* may be omitted as follows:

1. Between a '`P`' descriptor and a following '`D`', '`E`', '`EN`', '`ES`', '`F`' or '`G`' descriptor,
2. Before a '`/`' descriptor with no preceding *digit-string*,
3. After a '`/`' descriptor and
4. Before or after a '`:`' descriptor.

Function Statement

[*prefix*] `FUNCTION` *name* `(` [*name* [`,` *name*] ...] `)` [`RESULT(`*name*`)`]

prefix :: = { *type-spec* | `RECURSIVE` | `PURE` | `ELEMENTAL` }...

Note: At most one occurrence of each prefix item is allowed.

This is the first statement of a function subprogram. If no `RESULT` variable is specified the result variable has the same name as the function name (thus for direct recursion, a `RESULT` clause is necessary as well as the `RECURSIVE` keyword).

Goto Statement

`GOTO` *label*

Branches to the specified label, which must be on a branch target statement (i.e., the subprogram `END` statement, an executable statement, the first statement of an executable construct or the last statement of an enclosing executable construct).

If Statement

`IF (` *expr* `)` *executable*

Executes the sub-statement if and only if the condition is true. The sub-statement cannot itself be an `IF` statement.

If Then Statement

[*construct-name* `:`] `IF (` *expr* `) THEN`

Begins an `IF-THEN` construct and the `THEN` part thereof. Statements in this part are executed if and only if the condition is true. This statement may have a construct-name; if it does, the corresponding `ENDIF` statement shall have the same construct-name and intervening `ELSE` and `ELSEIF` statements at the same level may have the same construct-name.

Implicit Statement

`IMPLICIT` *implicit-spec* [`,` *implicit-spec*]...

implicit-spec ::= *type-spec* `(` *letter-spec* [`,` *letter-spec*] ... `)`
letter-spec ::= *letter* [`-` *letter*]

Alters the implicit type mapping from the default. The default map is

```
IMPLICIT REAL(A-H,O-Z),INTEGER(I-N)
```

in an external subprogram or interface body, and the same as the containing subprogram in a contained subprogram.

Implicit None Statement

```
IMPLICIT NONE
```

This statement sets the implicit type mapping for each letter to null, i.e., there are no implicit types. It must occur before any `PARAMETER` statements or other declarations (but after any `USE` statements).

Inquire Statement

`INQUIRE ( IOLENGTH=`*object*` )` *output-item* [`,` *output-item*]...
`INQUIRE (` *inquire-spec* [`,` *inquire-spec*]... `)`

inquire-spec ::= [`UNIT=`] *expr* | `ACCESS=` *variable* | `ACTION=` *variable* | `BLANK=` *variable* | `CONVERT=` *variable* | `DELIM=` *variable* | `DIRECT=` *variable* | `ERR=` *label* | `EXIST=` *variable* | `FILE=` *expr* | `FORM=` *variable* | `FORMATTED=` *variable* | `IOSTAT=` *variable* | `NAME=` *variable* | `NAMED=` *variable* | `NEXTREC=` *variable* | `NUMBER=` *variable* | `OPENED=` *variable* | `PAD=` *variable* | `POSITION=` *variable* | `READ=` *variable* | `READWRITE=` *variable* | `RECL=` *variable* | `SEQUENTIAL=` *variable* | `UNFORMATTED=` *variable*

output-item ::= *expr* | `(` { *output-item* `,` } ... *do-spec* `)`

The first form enquires as to the length needed to be specified for `RECL=` in the `OPEN` statement for an unformatted sequential file to be able to write records as large as the *output-item* list.

The second form enquires either by unit or by file; exactly one `UNIT=` or `FILE=` clause must be present (the `UNIT=` keyword can be omitted if it is the first *inquire-spec*). If the `FILE=` clause is used and that file is currently connected to a unit, the effect is as if that unit were specified.

The effect of each clause is as below:

`ACCESS=` Sets the scalar character object to `'SEQUENTIAL'` if the unit is connected for sequential access, to `'DIRECT'` if the unit is connected for direct access, and to `'UNDEFINED'` if there is no connection.

`ACTION=` Sets the scalar character object to `'READ'` if the unit is connected for input only, to `'WRITE'` if the unit is connected for output only, to `'READWRITE'` if the unit is connected for both input and output, and to `'UNDEFINED'` if there is no connection.

`BLANK=` If the unit is connected for formatted i/o, sets the scalar character object to `'NULL'` if blanks are treated as nulls on input and to `'ZERO'` if they are treated as zeroes on input. Otherwise the object is set to `'UNDEFINED'`.

`CONVERT=` Sets the scalar character object to `'UNKNOWN'` if the file is not connected for unformatted input/output, and otherwise to the value of the `CONVERT=` specifier in the `OPEN` statement or as determined by the `FORT_CONVERT`*n* environment variable. Possible values are `'NATIVE'`, `'BIG_NATIVE'`, `'LITTLE_NATIVE'`, `'BIG_IEEE'`, `'LITTLE_IEEE'`, `'BIG_IEEE_DD'` and `'LITTLE_IEEE_DD'`.

`DELIM=` Sets the scalar character object to `'APOSTROPHE'` if the apostrophe is used to delimit character data in list-directed or namelist output for the unit, to `'QUOTE'` if the quotation mark is to be so used, to `'NONE'` if no delimiter is to be used, and to `'UNDEFINED'` if there is no connection.

`DIRECT=` Sets the scalar character object to `'YES'` if direct access is allowed for the unit, to `'NO'` if direct access is not allowed, and to `'UNKNOWN'` if the answer cannot be determined.

`ERR=` Transfers control to the specified label if an error condition occurs during execution of the enquiry.

`EXIST=` Sets the scalar default logical object to `.TRUE.` if the file or unit exists, and to `.FALSE.` otherwise.

`FORM=` Sets the scalar character object to `'FORMATTED'` if the unit is connected for formatted i/o, to `'UNFORMATTED'` if the unit is connected for unformatted i/o, and to `'UNDEFINED'` if there is no connection.

`FORMATTED=`
Sets the scalar character object to `'YES'` if formatted i/o is allowed for the unit, to `'NO'` if it is not allowed, and to `'UNKNOWN'` if the answer cannot be determined.

`IOSTAT=` Sets the scalar default integer object to a non-zero value to indicate an error condition occurring during the enquiry, and to zero otherwise.

`NAME=` Sets the scalar character object to the name of the file connected to the unit (undefined if there is no name).

`NAMED=` Sets the scalar default logical object to `.TRUE.` if the unit is connected to a named file, and to `.FALSE.` otherwise.

`NEXTREC=` If the unit is connected for direct access, sets the scalar default integer object to 1 greater than the number of the record last read or written (or to 1 if no record has been read or written); otherwise the object becomes undefined.

`NUMBER=` Sets the scalar default integer object to the number of the unit to which the file is connected, or to `-1` if the file is not connected to a unit.

`OPENED=` Sets the scalar default logical object to `.TRUE.` if the unit is connected to a file (or the file is connected to a unit), and to `.FALSE.` otherwise.

`PAD=` Sets the scalar character object to `'NO'` if the connection of the file to the unit included the `PAD='NO'` specifier; otherwise it is set to `'YES'`.

`POSITION=`
Sets the scalar character object to `'REWIND'` if the unit is positioned at the beginning of the file, to `'APPEND'` if it is positioned at the end of the file, to `'ASIS'` if the unit was connected with that specification (and no i/o or positioning has occurred since connection), to `'UNDEFINED'` if the unit is connected for direct access or there is no connection, and to a processor-dependent value otherwise.

`READ=` Sets the scalar character object to `'YES'` if input is allowed for the file or unit, to `'NO'` if input is not allowed for the file or unit, and to `'UNKNOWN'` if the answer cannot be determined.

`READWRITE=`
Sets the scalar character object to `'YES'` if both input and output are allowed for the file or unit, to `'NO'` if at least one of input or output is not allowed for the file or unit, and to `'UNKNOWN'` if the answer cannot be determined.

`RECL=` Sets the scalar default integer object to the record length of a file connected for direct access or to the maximum record length of a file connected for sequential access. If there is no connection the object is undefined.

`SEQUENTIAL=`
Sets the scalar character object to `'YES'` if sequential access is allowed for the unit, to `'NO'` if sequential access is not allowed, and to `'UNKNOWN'` if the answer cannot be determined.

`UNFORMATTED=`
Sets the scalar character object to `'YES'` if unformatted i/o is allowed for the unit, to `'NO'` if it is not allowed, and to `'UNKNOWN'` if the answer cannot be determined.

`WRITE=` Sets the scalar character object to `'YES'` if output is allowed for the file or unit, to `'NO'` if output is not allowed for the file or unit, and to `'UNKNOWN'` if the answer cannot be determined.

Intent Statement

`INTENT ( { IN | OUT | INOUT } ) [ :: ]` *name* `[ , ` *name* `]...`

Declares the specified names, which must be the names of dummy arguments, to have the specified intent. `INTENT(IN)` arguments cannot appear in any context where they will be modified, `INTENT(OUT)` arguments are undefined on entry to the procedure, and `INTENT(INOUT)` and `INTENT(OUT)` arguments can only be associated with modifiable actual arguments (e.g., not expressions).

Interface Statement

`INTERFACE`
`INTERFACE {` *name* `| ASSIGNMENT(=) | OPERATOR(`*operator*`) }`

The first form introduces an interface block, containing interface bodies which specify the interfaces to external or dummy procedures. The second form additionally defines a generic name or operator by which these procedures may be referenced, and its interface block may also contain `MODULE PROCEDURE` statements.

Intrinsic Statement

`INTRINSIC` *name* [`,` *name*]...

Declares the listed names to be intrinsic procedures.

Module Statement

`MODULE` *name*

This is the first statement of a module subprogram.

Module Procedure Statement

`MODULE PROCEDURE` *name* [`,` *name*]...

This statement is only allowed within generic interface blocks, where it declares the listed names as module procedures to be included in the generic.

Namelist Statement

`NAMELIST` *namelist-group* [[`,`] *namelist-group*]...

namelist-group ::= `/`*name*`/` *name* [`,` *name*]...

Declares one or more i/o namelists. Multiple `NAMELIST` specifications for the namelist group `/`*name*`/` are treated as if they were concatenated. The names in a namelist group must all be variables and not automatic, adjustable, allocatable, pointer, or contain a pointer.

Nullify Statement

`NULLIFY (` *object* [`,` *object*] ... `)`

Sets the pointer-association status of the listed objects, which must be pointers, to dissociated.

Open Statement

`OPEN (` *open-spec* [`,` *open-spec*]... `)`

open-spec ::= [`UNIT=`] *expr* | `ACCESS=` *expr* | `ACTION=` *expr* | `BLANK=` *expr* | `CONVERT=` *expr* | `DELIM=` *expr* | `ERR=` *label* | `FILE=` *expr* | `FORM=` *expr* | `IOSTAT=` *object* | `PAD=` *expr* | `POSITION=` *expr* | `RECL=` *expr* | `STATUS=` *expr*

Connects a file to a unit with the specified properties.

Optional Statement

`OPTIONAL` [`::`] *name* [`,` *name*]...

Declares the specified names, which must be the names of dummy arguments, to be optional dummy arguments.

Parameter Statement

`PARAMETER (` *name* `=` *expr* [`,` *name* `=` *expr*]... `)`

Declares the names to be named constants with the specified values. The expressions must be initialisation expressions and must be assignment compatible with the names.

Pause Statement *(deleted)*

`PAUSE` [*constant*]

Pauses program execution. If present, the constant must be a scalar character literal with no *kind-param* or a *digit-string* with at most 5 digits.

Pointer Statement

`POINTER` [`::`] *name* [*deferred-shape*] [`,` *name* [*deferred-shape*]]...

Declares the names to be pointers.

Pointer Assignment Statement

variable `=>` *expr*

Associates the pointer *variable* with *expr*, which must be another pointer, a variable with the `TARGET` attribute, or a reference to a function that returns a pointer result.

Print Statement

`PRINT` *format* [`,` *output-item*]...

format ::= `*` | *label* | *expr*

Synonymous with a `WRITE` statement with '`UNIT=*`' and a `FMT=`*format* clause.

The possibilities for *format* are:

'`*`' indicates list-directed formatting.

label must be the label of a `FORMAT` statement.

expr A character expression may be supplied; its value is interpreted as the text following the keyword `FORMAT` of a `FORMAT` statement. If the expression is array-valued the concatenation of all elements is interpreted in this way.

expr ***(obsolescent)*** A default scalar integer variable name may be supplied, in which case it must have been `ASSIGN`ed the label of a `FORMAT` statement.

Private Statement

`PRIVATE` [[`::`] *access-id* [`,` *access-id*]...]

access-id ::= *name* | `ASSIGNMENT(=)` | `OPERATOR(`*operator*`)`

This statement can only occur in the declaration section of a module or before the component definitions in a type definition.

When this statement appears in a type definition, there can be no *access-id*s; it causes the components of the type to be inaccessible from outside the module in which the type is defined.

In a module's declaration section, this statement either sets the default accessibility of entities within the module to be `PRIVATE`, i.e., not accessible, or the accessibility of each *access-id* is set to be `PRIVATE`.

Program Statement

`PROGRAM` *name*

This is the first statement of a main program. It is optional.

Public Statement

`PUBLIC` [[`::`] *access-id* [`,` *access-id*]...]

This statement can only occur in the declaration section of a module. With no *access-id* list, it confirms that the default accessibility of entities in the module is `PUBLIC`. With an *access-id* list, it explicitly sets the accessibility of those *access-id*s to `PUBLIC`.

Read Statement

`READ` *format* [`,` *input-item*]...
`READ (` *control-spec* [`,` *control-spec*]... `)` [*input-item* [`,` *input-item*]...]

input-item ::= *variable* | `(` { *input-item* `,` }... *do-spec* `)`

control-spec ::= [`UNIT=`] { `*` | *expr* } |
[`FMT=`] *format* | [`NML=`] *name* | `ADVANCE=` *expr* | `END=` *label* |
`EOR=` *label* | `ERR=` *label* | `IOSTAT=` *expr* | `REC=` *expr* | `SIZE=` *expr*

(See the `PRINT` statement for *format* details.)

Reads one or more records (or partial records with `ADVANCE='NO'`) from the specified unit.

The effect of each control-specifier is as below:

`UNIT=` Identifies the i/o unit; '*' indicates the default unit, a scalar integer expression indicates an external unit, and a character expression indicates an internal unit.

`FMT=` Establishes the format; this is absent for namelist formatting or for unformatted i/o.

`NML=` Specifies a namelist group name for namelist formatted i/o.

`ADVANCE=` Indicates whether non-advancing (expression evaluates to `'NO'`) or the usual advancing (expression evaluates to `'YES'`) i/o is performed. This control-specifier is only allowed for formatted sequential i/o with an explicit format (i.e., not namelist or list-directed).

`END=` Transfers control to the specified label if an end-of-file condition occurs during input (not allowed in `WRITE`).

`EOR=` Transfers control to the specified label if an end-of-record condition occurs during input (not allowed in `WRITE`). `ADVANCE='NO'` must be specified.

`ERR=` Transfers control to the specified label if an error condition occurs during i/o.

`IOSTAT=` Sets the object to -1 if an end-of-file occurs, to -2 if an end-of-record occurs (non-advancing only), to a positive number if an error occurs, and to zero otherwise. Note that the negative values may vary on other compilers.

`REC=` Specifies the record number for direct-access i/o.

`SIZE=` Sets the object to the number of characters transferred by data edit descriptors (not allowed in `WRITE`). `ADVANCE='NO'` must be specified.

Return Statement

`RETURN` [*expr*]

Return immediately from the procedure. If the procedure is a subroutine with alternate return arguments ***(obsolescent)***, the scalar integer expression indicates to which label control is to be transferred on return (if the expression is less than one or greater than the number of alternate return arguments, execution continues with the statement following the subroutine reference).

Rewind Statement

`REWIND` *expr*
`REWIND (` *position-spec-list* `)`

(See the `BACKSPACE` statement for the *position-spec-list* definition).

Positions an i/o unit, which must be connected to a rewindable file, to the beginning of the file.

Save Statement

`SAVE` [[`::`] *save-item* [`,` *save-item*]...]

save-item ::= *variable-name* | `/`*common-block-name*`/`

Specifies the `SAVE` attribute for the listed variables or common blocks, or, with no *save-item* list, specifies that all possible variables and common blocks in the current scoping unit should implicitly have the `SAVE` attribute by default.

Select Statement

[*construct-name* `:`] `SELECT CASE (` *expr* `)`

The initial statement of a `SELECT CASE` construct. Control is transferred to the `CASE` statement satisfied by the expression's value, or to the `END SELECT` statement if no `CASE` is satisfied by the value.

Statement Function Statement *(obsolescent)*

name `(` [*name* [`,` *name*] ...] `)` `=` *expr*

Defines a statement function.

Stop Statement

`STOP` [*constant*]

Halts program execution. If present, the constant must be a scalar character literal with no *kind-param* or a *digit-string* with at most 5 digits.

Subroutine Statement

[`RECURSIVE` | `PURE` | `ELEMENTAL`]... `SUBROUTINE` *name* [`(` [*arg-list*] `)`]

(Note that at most one occurrence of each keyword is allowed).

arg-list ::= *arg* [`,` *arg*]...
arg ::= *name* | `*`

This is the first statement of a subroutine subprogram. `RECURSIVE` must be specified if the subroutine calls itself, either directly or indirectly. If `PURE` is specified, the subroutine must satisfy the pure subroutine constraints and can then be called from a pure function. An *arg* that is '`*`' signfies an alternate return label; this is ***obsolescent***.

Target Statement

`TARGET` [`::`] *name* [*array-spec*] [`,` *name* [*array-spec*]]...

Declares that the specified entities have the `TARGET attribute`.

Type Statement

`TYPE` *name*

This statement marks the beginning of the definition of the derived type *name*.

Type Declaration Statement

type-spec [[, *attr-spec*] ... ::] *entity-decl-list*

attr-spec ::= ALLOCATABLE | DIMENSION *array-spec* | EXTERNAL | INTENT ({ IN | OUT | INOUT }) | INTRINSIC | OPTIONAL | PARAMETER | POINTER | PRIVATE | PUBLIC | SAVE | TARGET

Declares the listed entities to be of the specified type with the specified attributes.

Use Statement

USE *name* [, *rename-list*]
USE *name*, ONLY: *only-list*

rename-list ::= *rename* [, *rename*]...
rename ::= *local-name* => *remote-name*

only-list ::= *only-item* [, *only-item*]...
only-item ::= *name* | *rename*

The USE statement accesses the named module. Multiple USE statements for the same module act as if all the *rename-list*s and *only-list*s were concatenated.

If all the USE statements in a scoping unit for a particular module have the ONLY clause, only those items listed in a *rename-list* or *only-list* are accessible.

A *rename* causes item *remote-name* in the referenced module to be accessible in the local scoping unit by *local-name*. An *only-item* that is not a *rename* causes the *name* in the referenced module to be accessible in the local scoping unit by the same name.

Where Assignment Statement

variable = *expr*

The expression is evaluated (and the object updated) only for those elements for which the current control mask is true.

Where Statement

WHERE (*expr*) *where-assignment-stmt*

Executes the Where Assignment statement with the provided expression as the control mask.

Where Construct Statement

[*construct-name* :] WHERE (*expr*)

Begins a Where Construct with the provided expression as the control mask.

Write Statement

WRITE (*control-spec* [, *control-spec*] ...) *output-item* [, *output-item*]...

(See the READ statement for *control-spec* details.)

Writes one or more records (or partial records with ADVANCE='NO') to the specified unit.

65 Fortran 95 Intrinsic Procedures

This section provides a quick reference guide to the Fortran 90/95 intrinsic functions and subroutines.

Procedures marked with '*' are non-generic versions of other intrinsics. Procedures marked with '*E*' are elemental.

Procedures marked with '*I*' are inquiry functions; these query characteristics of a variable other than its value.

Procedures marked with '*P*' may be supplied as actual procedure arguments; when it is used as a procedure argument all arguments are scalar, no optional arguments are allowed, and it is not generic: if the name is normally generic the procedure argument version is default `INTEGER` if it begins with the letter I to N, and default `REAL` otherwise.

Arguments named '`KIND`' must be initialisation expressions.

Arguments printed in italics (e.g., '*DIM*') are optional.

Intrinsic Functions

Function Name	Flags	Description
`ABS(A)`	*EP*	Absolute value.
`ACHAR(I)`	*E*	Produce character from ASCII value.
`ACOS(X)`	*EP*	Arccosine.
`ADJUSTL(STRING)`	*E*	Adjust string to the left by moving leading blanks to the end.
`ADJUSTR(STRING)`	*E*	Adjust string to the right.
`AIMAG(Z)`	*EP*	The imaginary part of a complex number.
`AINT(A,`*`KIND`*` )`	*EP*	Truncate to a whole number.
`ALL(MASK,`*`DIM`*`)`		Reduce `MASK` with the `.AND.` operation.
`ALLOCATED(ARRAY)`	*I*	Whether an allocatable array is allocated.
`ALOG(X)`	**P*	`LOG` function restricted to scalar default real.
`ALOG10(X)`	**P*	`LOG10` function restricted to scalar default real.
`AMAX0(A1,A2,`*`A3,...`*`)`	*	`REAL(MAX(A1,A2,`*`A3,...`*`))` with scalar default integer args.
`AMAX1(A1,A2,`*`A3,...`*`)`	*	`MAX` function restricted to scalar default real.
`AMIN0(A1,A2,`*`A3,...`*`)`	*	`REAL(MIN(A1,A2,`*`A3,...`*`))` with scalar default integer args.
`AMIN1(A1,A2,`*`A3,...`*`)`	*	`MIN` function restricted to scalar default integer.
`AMOD(A,P)`	**P*	`MOD` function restricted to scalar default real.
`ANINT(A,`*`KIND`*`)`	*EP*	Round to a whole number.
`ANY(MASK,`*`DIM`*`)`		Reduce `MASK` with the .OR. operation.
`ASIN(X)`	*EP*	Arcsine.
`ASSOCIATED(POINTER,`*`TARGET`*`)`	*I*	Whether a pointer is associated with a target.
`ATAN(X)`	*EP*	Arctangent.
`ATAN2(Y,X)`	*EP*	Arctangent.
`BIT_SIZE(I)`		Number of bits in an integer type.
`BTEST(I,POS)`	*E*	Whether a particular bit is 1.
`CABS(A)`	**P*	`ABS` function restricted to scalar default complex.
`CCOS(X)`	**P*	`COS` function restricted to scalar default complex.
`CEILING(A,`*`KIND`*`)`	*E*	Smallest integer greater than or equal to `A`.
`CEXP(X)`	**P*	`EXP` function restricted to scalar default complex.
`CHAR(I,`*`KIND`*`)`	*E*	Produce character from native coded character set value.
`CLOG(X)`	**P*	`LOG` function restricted to scalar default complex.
`CMPLX(X,`*`Y`*`,`*`KIND`*`)`	*E*	Convert to complex.
`CONJG(Z)`	*EP*	Complex conjugate.

Function Name	Flags	Description
COS(X)	*EP*	Cosine.
COSH(X)	*EP*	Hyperbolic cosine.
COUNT(MASK,*DIM*)		Reduce MASK by counting .TRUE. elements.
CSHIFT(ARRAY,SHIFT,*DIM*)		Circular shift of an array.
CSIN(X)	**P*	SIN function restricted to scalar default complex.
CSQRT(X)	**P*	SQRT function restricted to scalar default complex.
DABS(A)	**P*	ABS function restricted to scalar double precision.
DACOS(X)	**P*	ACOS function restricted to scalar double precision.
DASIN(X)	**P*	ASIN function restricted to scalar double precision.
DATAN(X)	**P*	ATAN function restricted to scalar double precision.
DATAN2(Y,X)	**P*	ATAN2 function restricted to scalar double precision.
DBLE(A)	*E*	Convert to double precision.
DCOS(X)	**P*	COS function restricted to scalar double precision.
DCOSH(X)	**P*	COSH function restricted to scalar double precision.
DDIM(X,Y)	**P*	DIM function restricted to scalar double precision.
DEXP(X)	**P*	EXP function restricted to scalar double precision.
DIGITS(X)	*I*	Number of mantissa digits in the model for X.
DIM(X,Y)	*EP*	Non-negative difference, MAX(0,X-Y).
DINT(A)	**P*	AINT function restricted to scalar double precision.
DLOG(X)	**P*	LOG function restricted to scalar double precision.
DLOG10(X)	**P*	LOG10 function restricted to scalar double precision.
DMAX1(A1,A2,*A3*,...)	*	MAX function restricted to scalar double precision.
DMIN1(A1,A2,*A3*,...)	*	MIN function restricted to scalar double precision.
DMOD(A,P)	**P*	MOD function restricted to scalar double precision.
DNINT(A)	**P*	ANINT function restricted to scalar double precision.
DOT_PRODUCT(VECTOR_A,VECTOR_B)		Dot product.
DPROD(X,Y)	*EP*	Double precision result of X*Y.
DSIGN(A,B)	**P*	SIGN function restricted to scalar double precision.
DSIN(X)	**P*	SIN function restricted to scalar double precision.
DSINH(X)	**P*	SINH function restricted to scalar double precision.
DSQRT(X)	**P*	SQRT function restricted to scalar double precision.
DTAN(X)	**P*	TAN function restricted to scalar double precision.
DTANH(X)	**P*	TANH function restricted to scalar double precision.
EOSHIFT(ARRAY,SHIFT,*BOUNDARY*,*DIM*)		End-off array shift.
EPSILON(X)	*I*	Number almost neglible compared to 1.
EXP(X)	*EP*	Exponential ($e^{\mathtt{X}}$).
EXPONENT(X)	*E*	The exponent part of a floating-point number.
FLOAT(A)	*	REAL function restricted to scalar default integer.
FLOOR(A,*KIND*)	*E*	Largest integer less than or equal to A.
FRACTION(X)	*E*	The mantissa of a floating-point number.
HUGE(X)	*I*	The largest number in the model for a real type.
IABS(A)	**P*	ABS function restricted to scalar default integer.
IACHAR(C)	*E*	ASCII value for a character.
IAND(I,J)	*E*	Bitwise and.
IBCLR(I,POS)	*E*	Clear specified bit.
IBITS(I,POS,LEN)	*E*	Extract a group of bits.
IBSET(I,POS)	*E*	Set a bit.
ICHAR(C)	*E*	Native coded character set value for a character.
IDIM(X,Y)	**P*	DIM function restricted to scalar default integer.
IDINT(A)	*	INT function restricted to scalar double precision.
IDNINT(A)	**P*	NINT function restricted to scalar double precision.

Function Name	Flags	Description
IEOR(I,J)	*E*	Bitwise exclusive or.
IFIX(A)	*	INT function restricted to scalar default real.
INDEX(STRING,SUBSTRING,*BACK*)	*EP*	Search for substring.
INT(A,*KIND*)	*E*	Convert to integer.
IOR(I,J)	*E*	Bitwise inclusive or.
ISHFT(I,SHIFT)	*E*	Shift bits.
ISHFTC(I,SHIFT,*SIZE*)	*E*	Shift bits circularly.
ISIGN(A,B)	**P*	SIGN function restricted to scalar default integer.
KIND(X)	*I*	Kind type parameter of X.
LBOUND(ARRAY,*DIM*)	*I*	Lower bound(s) of an array.
LEN(STRING)	*IP*	Length of a character string.
LEN_TRIM(STRING)	*E*	Length of a character string ignoring trailing blanks.
LGE(STRING_A,STRING_B)	*E*	Comparison (>=) using ASCII collating sequence.
LGT(STRING_A,STRING_B)	*E*	Comparison (>) using ASCII collating sequence.
LLE(STRING_A,STRING_B)	*E*	Comparison (<=) using ASCII collating sequence.
LLT(STRING_A,STRING_B)	*E*	Comparison (<) using ASCII collating sequence.
LOG(X)	*EP*	Natural logarithm.
LOG10(X)	*EP*	Common logarithm.
LOGICAL(L,*KIND*)	*E*	Convert to a specific logical kind.
MATMUL(MATRIX_A,MATRIX_B)		Matrix multiplication.
MAX(A1,A2,*A3,...*)	*E*	Maximum value.
MAX0(A1,A2,*A3,...*)	*	MAX function restricted to scalar default integer.
MAX1(A1,A2,*A3,...*)	*	INT(MAX(A1,A2,*A3,...*)) with scalar default real args.
MAXEXPONENT(X)	*I*	Maximum model exponent value for X.
MAXLOC(ARRAY,*MASK*)		Position of maximum value in an array.
MAXLOC(ARRAY,DIM,*MASK*)		Dimensional reduction of maximum value positions.
MAXVAL(ARRAY,*MASK*)		Reduce a (masked) array with the MAX intrinsic.
MAXVAL(ARRAY,DIM,*MASK*)		Dimensional reduction by the MAX intrinsic.
MERGE(TSOURCE,FSOURCE,MASK)	*E*	Choose value depending on logical value.
MIN(A1,A2,*A3,...*)	*E*	Minimum value.
MIN0(A1,A2,*A3,...*)	*	MIN function restricted to scalar default integer.
MIN1(A1,A2,*A3,...*)	*	INT(MIN(A1,A2,*A3,...*)) with scalar default real args.
MINEXPONENT(X)	*I*	Minimum model exponent value for X.
MINLOC(ARRAY,*MASK*)		Position of minimum value in an array.
MINLOC(ARRAY,DIM,*MASK*)		Dimensional reduction of minimum value positions.
MINVAL(ARRAY,*MASK*)		Reduce a (masked) array with the MIN intrinsic.
MINVAL(ARRAY,DIM,*MASK*)		Dimensional reduction by the MIN intrinsic.
MOD(A,P)	*EP*	Remainder; sign(result) = sign(A).
MODULO(A,P)	*E*	Modulo; sign(result) = sign(P).
NEAREST(X,S)	*E*	Nearest machine-representable number.
NINT(A,*KIND*)	*EP*	Round and convert to integer.
NOT(I)	*E*	Bitwise complement.
NULL(*MOLD*)		Null (disassociated) pointer.
PACK(ARRAY,MASK,*VECTOR*)		Pack an array into a vector.
PRECISION(X)	*I*	Decimal model precision for X.
PRESENT(A)	*I*	Whether an optional argument is present.
PRODUCT(ARRAY,*MASK*)		Reduce a (masked) array by multiplication.
PRODUCT(ARRAY,DIM,*MASK*)		Dimensional reduction by multiplication.
RADIX(X)	*I*	Model radix for X.
RANGE(X)	*I*	Decimal model exponent range for X.
REAL(A,*KIND*)	*E*	Convert to real.

Function Name	Flags	Description
REPEAT(STRING,NCOPIES)		Concatenate a string with itself.
RESHAPE(SOURCE,SHAPE,*PAD*,*ORDER*)		Reshape an array.
RRSPACING(X)	*E*	Reciprocal relative model spacing near X.
SCALE(X,I)	*E*	X*RADIX(X)**I.
SCAN(STRING,SET,*BACK*)	*E*	Look for characters in a set.
SELECTED_INT_KIND(R)		Integer kind with at least R decimal digits.
SELECTED_REAL_KIND(*P*,*R*)		Real kind with at least P decimal precision and/or at least R decimal exponent range.
SET_EXPONENT(X,I)	*E*	X*RADIX(X)**(I-EXPONENT(X)).
SHAPE(SOURCE)	*I*	Shape of an array.
SIGN(A,B)	*EP*	A with the sign of B.
SIN(X)	*EP*	Sine.
SINH(X)	*EP*	Hyperbolic sine.
SIZE(ARRAY,*DIM*)	*I*	Size of an array or dimension.
SNGL(A)	*	REAL function restricted to scalar double precision.
SPACING(X)	*E*	Spacing of model numbers near X.
SPREAD(SOURCE,DIM,NCOPIES)		Replicate an array in a given dimension by copying.
SQRT(X)	*EP*	Square root.
SUM(ARRAY,*MASK*)		Reduce a (masked) array by addition.
SUM(ARRAY,DIM,*MASK*)		Dimensional reduction of a (masked) array by addition.
TAN(X)	*EP*	Tangent.
TANH(X)	*EP*	Hyperbolic tangent.
TINY(X)	*I*	Smallest model number for the kind of X.
TRANSFER(SOURCE,MOLD,*SIZE*)		Copy internal representation.
TRANSPOSE(MATRIX)		Transpose an array.
TRIM(STRING)		Character string with trailing blanks removed.
UBOUND(ARRAY,*DIM*)	*I*	Upper bound(s) of an array.
UNPACK(VECTOR,MASK,FIELD)		Unpack a vector into an array.
VERIFY(STRING,SET,*BACK*)	*E*	Look for characters not in set.

Intrinsic Subroutines

Subroutine Name	Flags	Description
CPU_TIME(TIME)		CPU execution time.
DATE_AND_TIME(*DATE*,*TIME*,*ZONE*,*VALUES*)		Date and time information.
MVBITS(FROM,FROMPOS,LEN,TO,TOPOS)	*E*	Move or copy bitfield.
RANDOM_NUMBER(HARVEST)		Return pseudo-random number(s).
RANDOM_SEED(*SIZE*,*PUT*,*GET*)		Control pseudo-random number generator.
SYSTEM_CLOCK(*COUNT*,*COUNT_RATE*,*COUNT_MAX*)		Real-time clock information.

Fortran 2003 Extensions

66 Fortran 2003 Overview

This part of the manual describes those parts of the Fortran 2003 language which are not in Fortran 95, and indicates which features are currently supported by the NAG Fortran Compiler.

Features marked in the section heading as '[5.3.1]' are newly available in release 5.3.1, those marked '[5.3]' were available in release 5.3, those marked '[5.2]' were available in release 5.2, those marked '[5.1]' were available in release 5.1 (and in some cases earlier), and those marked '[n/a]' are not yet available.

Fortran 2003 is a major advance over Fortran 95: the new language features can be grouped as follows:

- object-oriented programming features,
- allocatable attribute extensions,
- other data-oriented enhancements,
- interoperability with C,
- IEEE arithmetic support,
- input/output enhancements and
- miscellaneous enhancements.

The basic object-oriented features are type extension, polymorphic variables, and type selection; these provide inheritance and the ability to program ad-hoc polymorphism in a type-safe manner. The advanced features are typed allocation, cloning, type-bound procedures, type-bound generics, and object-bound procedures. Type-bound procedures provide the mechanism for dynamic dispatch (methods).

The `ALLOCATABLE` attribute is extended to allow it to be used for dummy arguments, function results, structure components, and scalars (not just arrays). An intrinsic procedure has been added to transfer an allocation from one variable to another. Finally, in intrinsic assignment, allocatable variables or components are automatically reallocated with the correct size if they have a different shape or type parameter value from that of the expression. This last feature, together with deferred character length, provides the user with true varying-length character variables.

There are two other major data enhancements: the addition of type parameters to derived types, and finalisation (by final subroutines). Other significant data enhancements are the `PROTECTED` attribute, pointer bounds specification and rank remapping, procedure pointers, and individual accessibility control for structure components.

Interoperability with the C programming language consists of allowing C procedures to be called from Fortran, Fortran procedures to be called from C, and for the sharing of global variables between C and Fortran. This can only happen where C and Fortran facilities are equivalent: an intrinsic module provides derived types and named constants for mapping Fortran and C types, and the `BIND(C)` syntax is added for declaring Fortran entities that are to be shared with C. Additionally, C style enumerations have been added.

Support for IEEE arithmetic is provided by three intrinsic modules. Use of the `IEEE_FEATURES` module requests IEEE compliance for specific Fortran features, the `IEEE_EXCEPTIONS` module provides access to IEEE modes and exception handling, and the `IEEE_ARITHMETIC` module provides enquiry functions and utility functions for determining the extent of IEEE conformance and access to IEEE-conformant facilities.

The input/output facilities have had three major new features: asynchronous input/output, stream input/output, and user-defined procedures for derived-type input/output (referred to as "defined input/output"). Additionally, the input/output specifiers have been regularised so that where they make sense: all specifiers that can be used on an `OPEN` statement can also be used on a `READ` or `WRITE` statement, and vice versa. Access to input/output error messages is provided by the new `IOMSG=` specifier, and processor-dependent constants for input/output (e.g. the unit number for the standard input file) are provided in a new intrinsic module.

Finally, there are a large number of miscellaneous improvements in almost every aspect of the language. Some of the more significant of these are the `IMPORT` statement (provides host association into interface blocks), the `VALUE` and

VOLATILE attributes, the ability to use all intrinsic functions in constant expressions, and extensions to the syntax of array and structure constructors.

67 Object-oriented programming

67.1 Type Extension

Type extension provides the first phase of object orientation: inheritance and polymorphic objects.

67.1.1 Extending Types [5.0]

Any derived type can be extended using the EXTENDS keyword, except for SEQUENCE types and BIND(C) types. (The latter types are "non-extensible", as are intrinsic types, whereas all other derived types are "extensible".) The extended type inherits all the components of the parent type and may add extra components.

For example:

```
TYPE point
  REAL x,y
END TYPE
TYPE,EXTENDS(point) :: point_3d
  REAL z
END TYPE
```

The type point_3d has x, y and z components. Additionally, it has a point component which refers to the inherited part; this "parent component" is "inheritance-associated" with the inherited components, so that the point%x component is identical to the x component et cetera.

However, when extending a type it is not required to add any new components; for example,

```
TYPE,EXTENDS(point) :: newpoint
END TYPE
```

defines a new type newpoint which has exactly the same components as point (plus the associated parent component). Similarly, it is no longer necessary for a type to contain any components:

```
TYPE empty_type
END TYPE
```

declares the extensible (but not extended) type empty_type which has no components at all.

67.1.2 Polymorphic Variables [5.0]

A polymorphic variable is a pointer, allocatable array or dummy argument that is declared using the CLASS keyword instead of the TYPE keyword. A CLASS(*typename*) variable can assume any type in the class of types consisting of TYPE(*typename*) and all extensions of *typename*.

For example:

```
REAL FUNCTION bearing(a)
  CLASS(point) a
  bearing = atan2(a%y,a%x)
END
```

The function bearing may be applied to a TYPE(point) object or to a TYPE(point_3d) object, or indeed to an object of any type that is an extension of TYPE(point).

67.1.3 Type Selection [5.0]

The `SELECT TYPE` construct provides both a means of testing the dynamic type of a polymorphic variable and access to the extended components of that variable.

For example:

```
CLASS(t) x
...
SELECT TYPE(p=>x)
TYPE IS (t1)
  !
  ! This section is executed only if X is exactly of TYPE(t1), not an
  ! extension thereof.  P is TYPE(t1).
  !
TYPE IS (t2)
  !
  ! This section is executed only if X is exactly of TYPE(t2), not an
  ! extension thereof.  P is TYPE(t2).
  !
CLASS IS (t3)
  !
  ! This section is executed if X is of TYPE(t3), or of some extension
  ! thereof, and if it is not caught by a more specific case.  P is CLASS(t3).
  !
END SELECT
```

Note that '`SELECT TYPE(x)`' is short for '`SELECT TYPE(x=>x)`'.

67.1.4 Unlimited polymorphism [5.2]

A variable that is '`CLASS(*)`' is an unlimited polymorphic variable. It has no type, but can assume any type including non-extensible types and intrinsic types (and kinds). Apart from allocation, deallocation and pointer assignment, to perform any operation on an unlimited polymorphic you first have to discover its type using `SELECT TYPE`. For example:

```
CLASS(*),POINTER :: x
CHARACTER(17),TARGET :: ch
x => ch
SELECT TYPE(x)
TYPE IS (COMPLEX(KIND=KIND(0d0)))
  PRINT *,x+1
TYPE IS (CHARACTER(LEN=*))
  PRINT *,LEN(x)
END SELECT
```

Note that in the case of `CHARACTER` the length must be specified as '`*`' and is automatically assumed from whatever the polymorphic is associated with.

In the case of a non-extensible (i.e. `BIND(C)` or `SEQUENCE`) type, `SELECT TYPE` cannot be used to discover the type; instead, an unsafe pointer assignment is allowed, for example:

```
TYPE t
  SEQUENCE
  REAL x
END TYPE
CLASS(*),POINTER :: x
TYPE(t),POINTER :: y
```

```
...
y => x ! Unsafe - the compiler cannot tell whether X is TYPE(t).
```

67.1.5 Ad hoc type comparison [5.3]

Two new intrinsic functions are provided for comparing the dynamic types of polymorphic objects. These are

```
EXTENDS_TYPE_OF(A,MOLD)
SAME_TYPE_AS(A,B)
```

The arguments must be objects of extensible types (though they need not be polymorphic). `SAME_TYPE_AS` returns `.TRUE.` if and only if both `A` and `B` have the same dynamic type. `EXTENDS_TYPE_OF` returns `.TRUE.` if and only if the dynamic type of `A` is the same as, or an extension of, the dynamic type of `MOLD`. Note that if `MOLD` is an unallocated unlimited polymorphic (`CLASS(*)`), the result will be true regardless of the state of `A`.

The arguments are permitted to be unallocated or disassociated, but they are not permitted to be pointers with an undefined association status.

It is recommended that where possible these intrinsic functions be avoided, and that `SELECT TYPE` be used for type checking instead.

67.2 Typed allocation [5.1]

The `ALLOCATE` statement now accepts a *type-spec*; this can be used to specify the dynamic type (and type parameters, if any) of an allocation. The *type-spec* appears before the allocation list, and is separated from it by a double colon.

For example, if `T` is an extensible type and `ET` is an extension of `T`,

```
CLASS(t),POINTER :: a(:)
ALLOCATE(et::a(100))
```

allocates `A` to have dynamic type `ET`. Note that the *type-spec* in an `ALLOCATE` statement omits the `TYPE` keyword for derived types, similarly to the `TYPE IS` and `CLASS IS` statements.

An unlimited polymorphic object can be allocated to be any type including intrinsic types: for example

```
CLASS(*),POINTER :: c,d
ALLOCATE(DOUBLE PRECISION::c)
READ *,n
ALLOCATE(CHARACTER(LEN=n)::d)
```

allocates `C` to be double precision real, and `D` to be of type `CHARACTER` with length `N`.

Typed allocation is only useful for allocating polymorphic variables and `CHARACTER` variables with deferred length (`LEN=:`). For a non-polymorphic variable, the *type-spec* must specify the declared type and, if it is type `CHARACTER` but not deferred-length, to have the same character length. The character length must not be specifed as an asterisk (`CHARACTER(LEN=*)`) unless the allocate-object is a dummy argument with an asterisk character length (and vice versa).

Finally, since there is only one *type-spec* it must be compatible with all the items in the allocation list.

67.3 Sourced allocation (cloning) [5.1]

The `ALLOCATE` statement now accepts the `SOURCE=` specifier. The dynamic type and value of the allocated entity is taken from the expression in the specifier. If the derived type has type parameters (q.v.), the value for any deferred type parameter is taken from the source expression, and the values for other type parameters must agree. This is not just applicable to derived types: if the entity being allocated is type `CHARACTER` with deferred length (`LEN=:`), the character length is taken from the source expression.

Only one entity can be allocated when the `SOURCE=` specifier is used. Note that when allocating an array the array shape is not taken from the source expression but must be specified in the usual way. If the source expression is an array, it must have the same shape as the array being allocated.

For example,

```
CLASS(*),POINTER :: a,b
...
ALLOCATE(a,SOURCE=b)
```

The allocated variable `A` will be a "clone" of `B`, whatever the current type of `B` happens to be.

67.4 Type-bound procedures [5.1]

Type-bound procedures provide a means of packaging operations on a type with the type itself, and also for dynamic dispatch to a procedure depending on the dynamic type of a polymorphic variable.

67.4.1 The type-bound procedure part

The type-bound procedure part of a type definition is separated from the components by the `CONTAINS` statement. The default accessibility of type-bound procedures is public even if the components are private; this may be changed by using the `PRIVATE` statement after the `CONTAINS`.

67.4.2 Specific type-bound procedures

The syntax of a specific, non-deferred, type-bound procedure declaration is:

`PROCEDURE` [[,*binding-attr-list*]::] *binding-name* [`=>`*procedure-name*]

The name of the type-bound procedure is *binding-name*, and the name of the actual procedure which implements it is *procedure-name*. If the optional `=>`*procedure-name* is omitted, the actual procedure has the same name as the binding.

A type-bound procedure is invoked via an object of the type, e.g.

```
CALL variable(i)%tbp(arguments)
```

Normally, the invoking variable is passed as an extra argument, the "passed-object dummy argument"; by default this is the first dummy argument of the actual procedure and so the first argument in the argument list becomes the second argument, etc. The passed-object dummy argument may be changed by declaring the type-bound procedure with the `PASS(`*argument-name*`)` attribute, in which case the variable is passed as the named argument. The `PASS` attribute may also be used to confirm the default (as the first argument), and the `NOPASS` attribute prevents passing the object as an argument at all. The passed-object dummy argument must be a polymorphic scalar variable of that type, e.g. `CLASS(t) self`.

When a type is extended, the new type either inherits or **overrides** each type-bound procedure of the old type. An overriding procedure must be compatible with the old procedure; in particular, each dummy argument must have the same type except for the passed-object dummy argument which must have the new type. A type-bound procedure that is declared to be `NON_OVERRIDABLE` cannot be overridden during type extension.

When a type-bound procedure is invoked, it is the dynamic type of the variable which determines which actual procedure to call.

The other attributes that a type-bound procedure may have are `PUBLIC`, `PRIVATE`, and `DEFERRED` (the latter only for abstract types, which are described later).

67.4.3 Generic type-bound procedures

A generic type-bound procedure is a set of specific type-bound procedures, in the same way that an ordinary generic procedure is a set of specific ordinary procedures. It is declared with the `GENERIC` statement, e.g.

```
GENERIC :: generic_name => specific_name_1, specific_name_2, specific_name_3
```

Generic type-bound procedures may also be operators or assignment, e.g.

```
GENERIC :: OPERATOR(+) => add_t_t, add_t_r, add_r_t
```

Such type-bound generic operators cannot have the `NOPASS` attribute; the dynamic type of the passed-object dummy argument determines which actual procedure is called.

When a type is extended, the new type inherits all the generic type-bound procedures without exception, and the new type may extend the generic with additional specific procedures. To override procedures in the generic, simply override the specific type-bound procedure. For example, in

```
TYPE mycomplex
  ...
CONTAINS
  PROCEDURE :: myc_plus_r => myc1_plus_r
  PROCEDURE,PASS(B) :: r_plus_myc => r_plus_myc1
  GENERIC :: OPERATOR(+) => myc_plus_r, r_plus_myc
END TYPE
...
TYPE,EXTENDS(mycomplex) :: mycomplex_2
  ...
CONTAINS
  PROCEDURE :: myc_plus_r => myc2_plus_r
  PROCEDURE,PASS(B) :: r_plus_myc => r_plus_myc2
END TYPE
```

the type `mycomplex_2` inherits the generic operator '`+`'; invoking the generic (`+`) invokes the specific type-bound procedure, which for entities of type `mycomplex_2` will invoke the overriding actual procedure (`myc2_plus_r` or `r_plus_myc2`).

67.5 Abstract derived types [5.1]

An extensible derived type can be declared to be `ABSTRACT`, e.g.

```
TYPE, ABSTRACT :: mytype
```

An abstract type cannot be instantiated; i.e. it is not allowed to declare a non-polymorphic variable of abstract type, and a polymorphic variable of abstract type must be allocated to be a non-abstract extension of the type.

Abstract type may contain `DEFERRED` type-bound procedures, e.g.

```
...
CONTAINS
  PROCEDURE(interface_name),DEFERRED :: tbpname
```

No binding ("`=>` *name*") is allowed or implied by a deferred procedure binding. The `interface_name` must be the name of an abstract interface or a procedure with an explicit interface, and defines the interface of the deferred type-bound procedure.

When extending an abstract type, the extended type must also be abstract unless it overrides all of the deferred type-bound procedures with normal bindings.

67.6 Object-bound procedures [5.2]

These are procedure pointer components, and act similarly to type-bound procedures except that the binding is per-object not per-type. The syntax of a procedure pointer component declaration is:

`PROCEDURE(` [*proc-interface*] `)` **,** *proc-component-attr-spec-list* :: *proc-decl-list*

where

- each *proc-component-attr-spec* is one of `NOPASS`, `PASS`, `PASS(`*arg-name*`)`, `POINTER`, `PRIVATE` or `PUBLIC`, and
- each *proc-decl* is a component name optionally followed by default initialisation to null ('`=> NULL()`').

The `POINTER` attribute is required.

Note that object-bound procedures have a passed-object dummy argument just like type-bound procedures; if this is not wanted, the `NOPASS` attribute must be used (and this is required if the interface is implicit, i.e. when `proc-interface` is missing or is a type specification).

The following example demonstrates using a list of subroutines with no arguments.

```
TYPE action_list
  PROCEDURE(),NOPASS,POINTER :: action => NULL()
  TYPE(action_list),POINTER :: next
END TYPE
TYPE(t),TARGET :: top
TYPE(t),POINTER :: p
EXTERNAL sub1,sub2
top%action = sub1
ALLOCATE(top%next)
top%next%action = sub2
...
p => top
DO WHILE (ASSOCIATED(p))
  IF (ASSOCIATED(p%action)) CALL p%action
  p => p%next
END DO
```

68 ALLOCATABLE extensions

In Fortran 2003 the `ALLOCATABLE` attribute is permitted not just on local variables but also on components, dummy variables, and function results. These are the same as described in the ISO Technical Report ISO/IEC TR 15581:1999.

Also, the `MOVE_ALLOC` intrinsic subroutine has been added, as well as automatic reallocation on assignment.

68.1 Allocatable Dummy Arrays [4.x]

A dummy argument can be declared to be an allocatable array, e.g.

```
SUBROUTINE s(dum)
  REAL,ALLOCATABLE :: dum(:,:)
  ...
END SUBROUTINE
```

Having an allocatable dummy argument means that there must be an explicit interface for any reference: i.e. if the procedure is not an internal or module procedure there must be an accessible interface block in any routine which references that procedure.

Any actual argument that is passed to an allocatable dummy array must itself be an allocatable array; it must also have the same type, kind type parameters, and rank. For example:

```
REAL,ALLOCATABLE :: x(:,:)
CALL s(x)
```

The actual argument need not be allocated before calling the procedure, which may itself allocate or deallocate the argument. For example:

```
PROGRAM example2
  REAL,ALLOCATABLE :: x(:,:)
  OPEN(88,FILE='myfile',FORM='unformatted')
  CALL read_matrix(x,88)
  !
  ... process x in some way
  !
  REWIND(88)
  CALL write_and_delete_matrix(x,88)
END
!
MODULE module
CONTAINS
  !
  ! This procedure reads the size and contents of an array from an
  ! unformatted unit.
  !
  SUBROUTINE read_matrix(variable,unit)
    REAL,ALLOCATABLE,INTENT(OUT) :: variable(:,:)
    INTEGER,INTENT(IN) :: unit
    INTEGER dim1,dim2
    READ(unit) dim1,dim2
    ALLOCATE(variable(dim1,dim2))
    READ(unit) variable
    CLOSE(unit)
  END SUBROUTINE
  !
  ! This procedures writes the size and contents of an array to an
  ! unformatted unit, and then deallocates the array.
  !
  SUBROUTINE write_and_delete_matrix(variable,unit)
    REAL,ALLOCATABLE,INTENT(INOUT) :: variable(:,:)
    INTEGER,INTENT(IN) :: unit
    WRITE(unit) SIZE(variable,1),SIZE(variable,2)
    WRITE(unit) variable
    DEALLOCATE(variable)
  END SUBROUTINE
END
```

68.2 Allocatable Function Results [4.x]

The result of a function can be declared to be an allocatable array, e.g.

```
 FUNCTION af() RESULT(res)
   REAL,ALLOCATABLE :: res
```

On invoking the function, the result variable will be unallocated. It must be allocated before returning from the function. For example:

```
  !
  ! The result of this function is the original argument with adjacent
  ! duplicate entries deleted (so if it was sorted, each element is unique).
  !
  FUNCTION compress(array)
    INTEGER,ALLOCATABLE :: compress(:)
    INTEGER,INTENT(IN) :: array(:)
    IF (SIZE(array,1)==0) THEN
      ALLOCATE(compress(0))
    ELSE
      N = 1
      DO I=2,SIZE(array,1)
        IF (array(I)/=array(I-1)) N = N + 1
      END DO
      ALLOCATE(compress(N))
      N = 1
      compress(1) = array(1)
      DO I=2,SIZE(array,1)
        IF (array(I)/=compress(N)) THEN
          N = N + 1
          compress(N) = array(I)
        END IF
      END DO
    END IF
  END
```

The result of an allocatable array is automatically deallocated after it has been used.

68.3 Allocatable Structure Components [4.x]

A structure component can be declared to be allocatable, e.g.

```
MODULE matrix_example
  TYPE MATRIX
    REAL,ALLOCATABLE :: value(:,:)
  END TYPE
END MODULE
```

An allocatable array component is initially not allocated, just like allocatable array variables. On exit from a procedure containing variables with allocatable components, all the allocatable components are automatically deallocated. This is in contradistinction to pointer components, which are not automatically deallocated. For example:

```
SUBROUTINE sub(n,m)
  USE matrix_example
  TYPE(matrix) a,b,c
  !
  ! a%value, b%value and c%value are all unallocated at this point.
  !
  ALLOCATE(a%value(n,m),b%value(n,m))
  !
  ... do some computations, then
  !
  RETURN
  !
  ! Returning from the procedure automatically deallocates a%value, b%value,
  ! and c%value (if they are allocated).
  !
END
```

Deallocating a variable that has an allocatable array component deallocates the component first; this happens recursively so that all ALLOCATABLE subobjects are deallocated with no memory leaks.

Any allocated allocatable components of a function result are automatically deallocated after the result has been used.

```
PROGRAM deallocation_example
  TYPE inner
    REAL,ALLOCATABLE :: ival(:)
  END TYPE
  TYPE outer
    TYPE(inner),ALLOCATABLE :: ovalue(:)
  END TYPE
  TYPE(outer) x
  !
  ! At this point, x%ovalue is unallocated
  !
  ALLOCATE(x%ovalue(10))
  !
  ! At this point, x%ovalue(i)%ival are unallocated, i=1,10
  !
  ALLOCATE(x%ovalue(2)%ival(1000),x%ovalue(5)%ival(9999))
  !
  ! Only x%ovalue(2)%ival and x%ovalue(5)%ival are allocated
  !
  DEALLOCATE(x%ovalue)
  !
  ! This has automatically deallocated x%ovalue(2)%ival and x%ovalue(5)%ival
  !
END
```

In a structure constructor for such a type, the expression corresponding to an allocatable array component can be

- the NULL() intrinsic, indicating an unallocated array,
- an allocatable array which may be allocated or unallocated, or
- any other array expression, indicating an allocated array.

```
SUBROUTINE constructor_example
  USE matrix_example
  TYPE(matrix) a,b,c
  REAL :: array(10,10) = 1
  REAL,ALLOCATABLE :: alloc_array(:,:)
  a = matrix(NULL())
  !
  ! At this point, a%value is unallocated
  !
  b = matrix(array*2)
  !
  ! Now, b%value is a (10,10) array with each element equal to 2.
  !
  c = matrix(alloc_array)
  !
  ! Now, c%value is unallocated (because alloc_array was unallocated).
  !
END
```

Intrinsic assignment of such types does a “deep copy” of the allocatable array components; it is as if the allocatable array component were deallocated (if necessary), then if the component in the expression was allocated, the variable’s component is allocated to the right size and the value copied.

```
  SUBROUTINE assignment_example
    USE matrix_example
    TYPE(matrix) a,b
    !
    ! First we establish a value for a
    !
    ALLOCATE(a%value(10,20))
    a%value(3,:) = 30
    !
    ! And a value for b
    !
    ALLOCATE(b%value(1,1))
    b%value = 0
    !
    ! Now the assignment
    !
    b = a
    !
    ! The old contents of b%value have been deallocated, and b%value now has
    ! the same size and contents as a%value.
    !
  END
```

68.4 Allocatable Component Example

This example shows the definition and use of a simple module that provides polynomial arithmetic. To do this it makes use of intrinsic assignment for allocatable components, the automatically provided structure constructors and defines the addition (+) operator. A more complete version of this module would provide other operators such as multiplication.

```
!
! Module providing a single-precision polynomial arithmetic facility
!
MODULE real_poly_module
  !
  ! Define the polynomial type with its constructor.
  ! We will use the convention of storing the coefficients in the normal
  ! order of highest degree first, thus in an N-degree polynomial, COEFF(1)
  ! is the coefficient of X**N, COEFF(N) is the coefficient of X**1, and
  ! COEFF(N+1) is the scalar.
  !
  TYPE,PUBLIC :: real_poly
    REAL,ALLOCATABLE :: coeff(:)
  END TYPE
  !
  PUBLIC OPERATOR(+)
  INTERFACE OPERATOR(+)
    MODULE PROCEDURE rp_add_rp,rp_add_r,r_add_rp
  END INTERFACE
  !
CONTAINS
  TYPE(real_poly) FUNCTION rp_add_r(poly,real)
    TYPE(real_poly),INTENT(IN) :: poly
    REAL,INTENT(IN) :: real
    INTEGER isize
    IF (.NOT.ALLOCATED(poly%coeff)) STOP 'Undefined polynomial value in +'
    isize = SIZE(poly%coeff,1)
    rp_add_r%coeff(isize) = poly%coeff(isize) + real
```

```
  END FUNCTION
  TYPE(real_poly) FUNCTION r_add_rp(real,poly)
    TYPE(real_poly),INTENT(IN) :: poly
    REAL,INTENT(IN) :: real
    r_add_rp = rp_add_r(poly,real)
  END FUNCTION
  TYPE(real_poly) FUNCTION rp_add_rp(poly1,poly2)
    TYPE(real_poly),INTENT(IN) :: poly1,poly2
    INTEGER I,N,N1,N2
    IF (.NOT.ALLOCATED(poly1%coeff).OR..NOT.ALLOCATED(poly2%coeff)) &
      STOP 'Undefined polynomial value in +'
    ! Set N1 and N2 to the degrees of the input polynomials
    N1 = SIZE(poly1%coeff) - 1
    N2 = SIZE(poly2%coeff) - 1
    ! The result polynomial is of degree N
    N = MAX(N1,N2)
    ALLOCATE(rp_add_rp%coeff(N+1))
    DO I=0,MIN(N1,N2)
      rp_add_rp%coeff(N-I+1) = poly1%coeff(N1-I+1) + poly2%coeff(N2-I+1)
    END DO
    ! At most one of the next two DO loops is ever executed
    DO I=N1+1,N
      rp_add_rp%coeff(N-I+1) = poly2%coeff(N2-I+1)
    END DO
    DO I=N2+1,N
      rp_add_rp%coeff(N-I+1) = poly1%coeff(N1-I+1)
    END DO
  END FUNCTION
END MODULE
!
! Sample program
!
PROGRAM example
  USE real_poly_module
  TYPE(real_poly) p,q,r
  p = real_poly((/1.0,2.0,4.0/))   ! x**2 + 2x + 4
  q = real_poly((/1.0,-5.5/))      ! x - 5.5
  r = p + q                        ! x**2 + 3x - 1.5
  print 1,'The coefficients of the answer are:',r%coeff
1 format(1x,A,3F8.2)
END
```

When executed, the above program prints:

```
 The coefficients of the answer are:    1.00    3.00   -1.50
```

68.5 The MOVE_ALLOC intrinsic subroutine [5.2]

This subroutine moves an allocation from one allocatable variable to another. This can be used to expand an allocatable array with only one copy operation, and allows full control over where in the new array the values should go. For example:

```
  REAL,ALLOCATABLE :: a(:),tmp(:)
  ...
  ALLOCATE(a(n))
  ...
  ! Here we want to double the size of A, without losing any of the values
  ! that are already stored in it.
```

```
ALLOCATE(tmp(size(a)*2))
tmp(1:size(a)) = a
CALL move_alloc(from=tmp,to=a)
! TMP is now deallocated, and A has the new size and values.
```

To have the values end up somewhere different, just change the assignment statement, for example to move them all to the end:

```
tmp(size(a)+1:size(a)*2) = a
```

68.6 Allocatable scalars [5.2]

The ALLOCATABLE attribute may now be applied to scalar variables and components, not just arrays. This is most useful in conjunction with polymorphism (`CLASS`) and/or deferred type parameters (e.g. `CHARACTER(:)`); for more details see the "Typed allocation", "Sourced allocation" and "Automatic reallocation" sections.

68.7 Automatic reallocation [5.2]

If, in an assignment to a whole allocatable array, the expression being assigned is an array of a different size or shape, the allocatable array is reallocated to have the correct shape (in Fortran 95 this assignment would have been an error). For example:

```
ALLOCATE(a(10))
...
a = (/ (i,i=1,100) /)
! A is now size 100
```

Similarly, if an allocatable variable has a deferred type parameter (these are described in a later section), and is either unallocated or has a value different from that of the expression, the allocatable variable is reallocated to have the same value for that type parameter. This allows for true varying-length character variables:

```
CHARACTER(:),ALLOCATABLE :: name
...
name = 'John Smith'
! LEN(name) is now 10, whatever it was before.
name = '?'
! LEN(name) is now 1.
```

Note that since a subobject of an allocatable object is not itself allocatable, this automatic reallocation can be suppressed by using substrings (for characters) or array sections (for arrays), e.g.

```
name(:) = '?'             ! Normal assignment with truncation/padding.
a(:) = (/ (i,i=1,100) /)  ! Asserts that A is already of size 100.
```

69 Other data-oriented enhancements

69.1 Parameterised derived types [6.0 for kind type parameters, 6.1 for length]

Derived types may now have type parameters. Like those of the intrinsic types, they come in two flavours: "kind"-like ones which must be known at compile time (called "kind" type parameters), and ones like character length which may vary at runtime (called "length" type parameters).

69.1.1 Basic Syntax and Semantics

A derived type which has type parameters must list them in the type definition, give them a type, and specify whether they are "kind" or "length" parameters. For example,

```
TYPE real_matrix(kind,n,m)
  INTEGER,KIND :: kind
  INTEGER(int64),LEN :: n,m
```

All type parameters must be explicitly specified to be of type `INTEGER`, but the kind of integer may vary. Type parameters are always scalar, never arrays. Within the type definition, "kind" type parameters may be used in constant expressions, and any type parameter may be used in a specification expression (viz array bound, character length, or "length" type parameter value). For example, the rest of the above type definition might look like:

```
  REAL(kind) value(n,m)
END TYPE real_matrix
```

When declaring entities of such a derived type, the type parameters must be given after the name. For example,

```
TYPE(real_matrix(KIND(0d0),100,200)) :: my_real_matrix_variable
```

Similarly, the type parameters must be given when constructing values of such a type; for example,

```
my_real_matrix_variable = &
  real_matrix(kind(0d0),100,200)((/ (i*1.0d0,i=1,20000) /))
```

To examine the value of a derived type parameter from outside the type definition, the same notation is used as for component accesses, e.g.

```
print *,'Columns =',my_real_matrix_variable%m
```

Thus type parameter names are in the same class as component names and type-bound procedure names. However, a type parameter cannot be changed by using its specifier on the left-hand-side of an assignment. Furthermore, the intrinsic type parameters may also be examined using this technique, for example:

```
REAL :: array(:,:)
CHARACTER(*),INTENT(IN) :: ch
PRINT *,array%kind,ch%len
```

prints the same values as for `KIND(array)` and `LEN(ch)`. Note that a kind parameter enquiry is always scalar, even if the object is an array.

A derived type parameter does not actually have to be used at all within the type definition, and a kind type parameter might only be used within specification expressions. For example,

```
TYPE fixed_byte(n)
  INTEGER,KIND :: n
  INTEGER(1) :: value(n)
END TYPE
TYPE numbered_object(object_number)
  INTEGER,LEN :: object_number
END TYPE
```

Even though the `fixed_byte` parameter `n` is not used in a constant expression, a constant value must always be specified for it because it has been declared to be a "kind" type parameter. Similarly, even though `object_number` has not been used at all, a value must always be specified for it. This is not quite as useless as it might seem: each `numbered_object` has a single value for `object_number` even if the `numbered_object` is an array.

69.1.2 More Semantics

A derived type with type parameters can have default values for one or more of them; in this case the parameters with default values may be omitted from the type specifiers. For example,

```
TYPE char_with_maxlen(maxlen,kind)
  INTEGER,LEN :: maxlen = 254
  INTEGER,KIND   :: kind = SELECTED_CHAR_KIND('ascii')
  INTEGER        :: len = 0
  CHARACTER(len=maxlen,kind=kind) :: value
END TYPE
...
TYPE(char_with_maxlen) temp
TYPE(char_with_maxlen(80)) card(1000)
TYPE(char_with_maxlen(kind=SELECTED_CHAR_KIND('iso 10646'))) ucs4_temp
```

Note that although kind type parameters can be used in constant expressions and thus in default initialisation, components that are variable-sized (because they depend on length type parameters) cannot be default-initialised at all. Thus `value` in the example above cannot be default-initialised.

Further note that unlike intrinsic types, there are no automatic conversions for derived type assignment with different type parameter values, thus given the above declarations,

```
card(1) = card(2) ! This is ok, maxlen==80 for both sides.
temp = card       ! This is not allowed - maxlen 254 vs. maxlen 80.
```

69.1.3 Assumed type parameters

Assumed type parameters for derived types work similarly to character length, except that they are only allowed for dummy arguments (not for named constants). For example, the following subroutine works on any `char_with_maxlen` variable.

```
SUBROUTINE stars(x)
  TYPE(char_with_maxlen(*)) x
  x%value = REPEAT('*',x%maxlen)
END SUBROUTINE
```

69.1.4 Deferred type parameters

Deferred type parameters are completely new to Fortran 2003; these are available both for `CHARACTER` and for parameterised derived types, and work similarly to deferred array bounds. A variable with a deferred type parameter must have the `ALLOCATABLE` or `POINTER` attribute. The value of a deferred type parameter for an allocatable variable is that determined by allocation (either by a typed allocation, or by an intrinsic assignment with automatic reallocation). For a pointer, the value of a deferred type parameter is the value of the type parameter of its target. For example, using the type `real_matrix` defined above,

```
TYPE(real_matrix(KIND(0.0),100,200)),TARGET :: x
TYPE(real_matrix(KIND(0.0),:,:)),POINTER :: y, z
ALLOCATE(real_matrix(KIND(0.0),33,44) :: y)  ! Typed allocation.
z => x                                       ! Assumes from the target.
PRINT *,y%n,z%n                              ! Prints 33 and 100.
```

Note that it is not allowed to reference the value of a deferred type parameter of an unallocated allocatable or of a pointer that is not associated with a target.

If a dummy argument is allocatable or a pointer, the actual argument must have deferred exactly the same type parameters as the dummy. For example,

```
SUBROUTINE sub(rm_dble_ptr)
  TYPE(real_matrix(KIND(0d0),*,:)),POINTER :: rm_dble_ptr
...
TYPE(real_matrix(KIND(0d0),100,200)),POINTER :: x
TYPE(real_matrix(KIND(0d0),100,:)),POINTER :: y
TYPE(real_matrix(KIND(0d0),:,:)),POINTER :: z
CALL sub(x)  ! Invalid - X%M is not deferred (but must be).
CALL sub(y)  ! This is ok.
CALL sub(z)  ! Invalid - X%N is deferred (but must not be).
```

69.2 Finalisation [5.3]

An extensible derived type can have "final subroutines" associated with it; these subroutines are automatically called whenever an object of the type is about to be destroyed, whether by deallocation, procedure return, being on the left-hand-side of an intrinsic assignment, or being passed to an `INTENT(OUT)` dummy argument.

A final subroutine of a type must be a subroutine with exactly one argument, which must be an ordinary dummy variable of that type (and must not be `INTENT(OUT)`). It may be scalar or an array, and when an object of that type is destroyed the final subroutine whose argument has the same rank as the object is called. The final subroutine may be elemental, in which case it will handle any rank of object that has no other subroutine handling it. Note that if there is no final subroutine for the rank of an object, no subroutine will be called.

Final subroutines are declared in the type definition after the `CONTAINS` statement, like type-bound procedures. They are declared by a `FINAL` statement, which has the syntax

`FINAL` [`::`] *name* [`,` *name*]...

where each *name* is a subroutine that satisfies the above rules.

A simple type with a final subroutine is as follows.

```
TYPE flexible_real_vector
  LOGICAL :: value_was_allocated = .FALSE.
  REAL,POINTER :: value(:) => NULL()
CONTAINS
  FINAL destroy_frv
END TYPE
...
ELEMENTAL SUBROUTINE destroy_frv(x)
  TYPE(flexible_real_vector),INTENT(INOUT) :: x
  IF (x%value_was_allocated) DEALLOCATE(x%value)
END SUBROUTINE
```

If an object being destroyed has finalisable components, any final subroutine for the object-as-a-whole will be called before finalising any components. If the object is an array, each component will be finalised separately (and any final subroutine called will be the one for the rank of the component, not the rank of the object).

For example, in

```
TYPE many_vectors
  TYPE(flexible_real_vector) scalar
  TYPE(flexible_real_vector) array(2,3)
CONTAINS
  FINAL :: destroy_many_vectors_1
END TYPE
...
SUBROUTINE destroy_many_vectors_1(array1)
  TYPE(many_vectors) array1(:)
  PRINT *,'Destroying a',SIZE(array1),'element array of many vectors'
```

```
  END SUBROUTINE
  ...
  TYPE(many_vector) mv_object(3)
```

when `mv_object` is destroyed, firstly '`destroy_many_vectors_1`' will be called with `mv_object` as its argument; this will print

```
 Destroying a 3 element array of many vectors
```

Secondly, for each element of the array, both the `scalar` and `array` components will be finalised by calling `destroy_frv` on each of them. These may be done in any order (or, since they are elemental, potentially in parallel).

Note that final subroutines are not inherited through type extension; instead, when an object of extended type is destroyed, first any final subroutine it has will be called, then any final subroutine of the parent type will be called on the parent component, and so on.

69.3 The PROTECTED attribute [5.0]

The `PROTECTED` attribute may be specified by the `PROTECTED` statement or with the `PROTECTED` keyword in a type declaration statement. It protects a module variable against modification from outside the module.

69.3.1 Syntax

The syntax of the `PROTECTED` statement is:

`PROTECTED` [`::`] *name* [`,` *name*] ...

The `PROTECTED` attribute may only be specified for a variable in a module.

69.3.2 Semantics

Variables with the `PROTECTED` attribute may only be modified within the defining module. Outside of that module they are not allowed to appear in a variable definition context (e.g. on the left-hand-side of an assignment statement), similar to `INTENT(IN)` dummy arguments.

This allows the module writer to make the values of some variables generally available without relinquishing control over their modification.

69.3.3 Example

```
MODULE temperature_module
  REAL,PROTECTED :: temperature_c = 0, temperature_f = 32
CONTAINS
  SUBROUTINE set_temperature_c(new_value_c)
    REAL,INTENT(IN) :: new_value_c
    temperature_c = new_value_c
    temperature_f = temperature_c*(9.0/5.0) + 32
  END SUBROUTINE
  SUBROUTINE set_temperature_f(new_value_f)
    REAL,INTENT(IN) :: new_value_f
    temperature_f = new_value_f
    temperature_c = (temperature_f - 32)*(5.0/9.0)
  END SUBROUTINE
END
```

The `PROTECTED` attribute allows users of `temperature_module` to read the temperature in either Farenheit or Celsius, but the variables can only be changed via the provided subroutines which ensure that both values agree.

69.4 Pointer enhancements

69.4.1 INTENT for pointers [5.1]

A `POINTER` dummy argument may now have the `INTENT` attribute. This attribute applies to the pointer association status, not to the target of the pointer.

An `INTENT(IN)` pointer can be assigned to, but cannot be pointer-assigned, nullified, allocated or deallocated. An `INTENT(OUT)` pointer receives an undefined association status on entry to the procedure. An `INTENT(INOUT)` pointer has no restrictions on its use, but the actual argument must be a pointer variable, not a pointer function reference.

69.4.2 Pointer bounds specification [5.2]

The bounds of a pointer can be changed (from the default) in a pointer assignment by including them on the left-hand-side. For example,

```
REAL,TARGET :: x(-100:100,-10:10)
REAL,POINTER :: p(:,:)
p(1:,1:) => x
```

The upper bound is formed by adding the extent (minus 1) to the lower bound, so in the above example, the bounds of `P` will be `1:201,1:21`. Note that when setting the lower bound of any rank in a pointer assignment, the values must be explicitly specified (there is no default of 1 like there is in array declarators) and they must be specified for all dimensions of the pointer.

69.4.3 Rank-remapping Pointer Assignment [5.0]

This feature allows a multi-dimensional pointer to point to a single-dimensional object. For example:

```
REAL,POINTER :: diagonal(:),matrix(:,:),base(:)
...
ALLOCATE(base(n*n))
matrix(1:n,1:n) => base
diagonal => base(::n+1)
!
! DIAGONAL now points to the diagonal elements of MATRIX.
!
```

Note that when rank-remapping, the values for both the lower and upper bounds must be explicitly specified for all dimensions, there are no defaults.

69.5 Individual component accessibility [5.1]

It is now possible to set the accessibility of individual components in a derived type. For example,

```
TYPE t
  LOGICAL, PUBLIC :: flag
  INTEGER, PRIVATE :: state
END TYPE
```

The structure constructor for the type is not usable from outside the defining module if there is any private component that is not inherited, allocatable or default-initialised (see Structure constructor syntax enhancements).

69.6 Public entities of private type [5.1]

It is now possible to export entities (named constants, variables, procedures) from a module even if they have private type or (for procedures) have arguments of private type. For example,

```
MODULE m
  TYPE, PRIVATE :: hidden_type
    CHARACTER(6) :: code
  END TYPE
  TYPE(hidden_type), PUBLIC, PARAMETER :: code_green = hidden_type('green')
  TYPE(hidden_type), PUBLIC, PARAMETER :: code_yellow = hidden_type('yellow')
  TYPE(hidden_type), PUBLIC, PARAMETER :: code_red = hidden_type('red')
END
```

70 C interoperability [mostly 5.1]

70.1 The ISO_C_BINDING module

The intrinsic module ISO_C_BINDING contains

- for each C type (e.g. `float`), a named constant for use as a KIND parameter for the corresponding Fortran type,
- types `C_PTR` and `C_FUNPTR` for interoperating with C object pointers and function pointers,
- procedures for manipulating Fortran and C pointers.

70.1.1 The kind parameters

The kind parameter names are for using with the corresponding Fortran types; for example, `INTEGER` for integral types and `REAL` for floating-point types. This is shown in the table below. Note that only `c_int` is guaranteed to be available; if there is no compatible type the value will be negative.

C type	Fortran type and kind
_Bool	LOGICAL(c_bool)
char	CHARACTER(c_char) — For characters as text.
double	REAL(c_double)
double _Complex	COMPLEX(c_double_complex) or COMPLEX(c_double)
float	REAL(c_float)
float _Complex	COMPLEX(c_float_complex) or COMPLEX(c_float)
int	INTEGER(c_int)
int16_t	INTEGER(c_int16_t)
int32_t	INTEGER(c_int32_t)
int64_t	INTEGER(c_int64_t)
int8_t	INTEGER(c_int8_t)
int_fast16_t	INTEGER(c_int_fast16_t)
int_fast32_t	INTEGER(c_int_fast32_t)
int_fast64_t	INTEGER(c_int_fast64_t)
int_fast8_t	INTEGER(c_int_fast8_t)
int_least16_t	INTEGER(c_int_least16_t)
int_least32_t	INTEGER(c_int_least32_t)
int_least64_t	INTEGER(c_int_least64_t)
int_least8_t	INTEGER(c_int_least8_t)
intmax_t	INTEGER(c_intmax_t)
intptr_t	INTEGER(c_intptr_t)
long	INTEGER(c_long)
long double	REAL(c_long_double)
long double _Complex	COMPLEX(c_long_double_complex) or COMPLEX(c_long_double)
long long	INTEGER(c_long_long)
short	INTEGER(c_short)
signed char	INTEGER(c_signed_char) — For characters as integers.
size_t	INTEGER(c_size_t)

70.1.2 Using C_PTR and C_FUNPTR

These are derived type names, so you use them as Type(c_ptr) and Type(c_funptr). Type(c_ptr) is essentially equivalent to the C void *; i.e. it can contain any object pointer. Type(c_funptr) does the same thing for function pointers.

For C arguments like 'int *', you don't need to use Type(c_ptr), you can just use a normal dummy argument (in this case of type Integer(c_int)) without the VALUE attribute. However, for more complicated pointer arguments such as pointer to pointer, or for variables or components that are pointers, you need to use Type(c_ptr).

Null pointer constants of both Type(c_ptr) and Type(c_funptr) are provided: these are named C_NULL_PTR and C_NULL_FUNPTR respectively.

To create a Type(c_ptr) value, the function C_LOC(X) is used on a Fortran object X (and X must have the TARGET attribute). Furthermore, the Fortran object cannot be polymorphic, a zero-sized array, an assumed-size array, or an array pointer. To create a Type(c_funptr) value, the function C_FUNLOC is used on a procedure; this procedure must have the BIND(C) attribute.

To test whether a Type(c_ptr) or Type(c_funptr) is null, the C_ASSOCIATED(C_PTR_1) function can be used; it returns .TRUE. if and only if C_PTR_1 is not null. Two Type(c_ptr) or two Type(c_funptr) values can be compared using C_ASSOCIATED(C_PTR_1,C_PTR_2) function; it returns .TRUE. if and only if C_PTR_1 contains the same C address as C_PTR_2.

The subroutine C_F_POINTER(CPTR,FPTR) converts the TYPE(C_PTR) value CPTR to the scalar Fortran pointer FPTR; the latter can have any type (including non-interoperable types) but must not be polymorphic. The subroutine C_F_POINTER(CPTR,FPTR,SHAPE) converts a TYPE(C_PTR) value into the Fortran array pointer FPTR, where SHAPE is an integer array of rank 1, with the same number of elements as the rank of FPTR; the lower bounds of the resultant FPTR will all be 1.

The subroutine C_F_PROCPOINTER(CPTR,FPTR) is provided. This converts the TYPE(C_FUNPTR) CPTR to the Fortran procedure pointer FPTR.

Note that in all the conversion cases it is up to the programmer to use the correct type and other information.

70.2 BIND(C) types

Derived types corresponding to C `struct` types can be created by giving the type the `BIND(C)` attribute, e.g.

```
TYPE,BIND(C) :: mytype
```

The components of a BIND(C) type must have types corresponding to C types, and cannot be pointers or allocatables. Furthermore, a BIND(C) type cannot be a `SEQUENCE` type (it already acts like a `SEQUENCE` type), cannot have type-bound procedures, cannot have final procedures, and cannot be extended.

70.3 BIND(C) variables

Access to C global variables is provided by giving the Fortran variable the `BIND(C)` attribute. Such a variable can only be declared in a module, and cannot be in a `COMMON` block. By default, the C name of the variable is the Fortran name converted to all lowercase characters; a different name may be specified with the `NAME=` clause, e.g.

```
INTEGER,BIND(C,NAME="StrangelyCapiTalisedCName") :: x
```

Within Fortran code, the variable is referred to by its Fortran name, not its C name.

70.4 BIND(C) procedures

A Fortran procedure that can be called from C can be defined using the BIND(C) attribute on the procedure heading. By default its C name is the Fortran name converted to lowercase; a different name may be specified with the `NAME=` clause. For example

```
SUBROUTINE sub() BIND(C,NAME='Sub')
...
```

Again, the C name is for use only from C, the Fortran name is used from Fortran. If the C name is all blanks (or a zero-length string), there is no C name. Such a procedure can still be called from C via a procedure pointer (i.e. by assigning it to a `TYPE(C_FUNPTR)` variable).

A `BIND(C)` procedure must be a module procedure or external procedure with an explicit interface; it cannot be an internal procedure or statement function.

A `BIND(C)` procedure may be a subroutine or a scalar function with a type corresponding to a C type. Each dummy argument must be a variable whose type corresponds to a C type, and cannot be allocatable, assumed-shape, optional or a pointer. If the dummy argument does not have the `VALUE` attribute, it corresponds to a C dummy argument that is a pointer.

Here is an example of a Fortran procedure together with its reference from C:

```
SUBROUTINE find_minmax(x,n,max,min) BIND(C,NAME='FindMinMax')
  USE iso_c_binding
  REAL(c_double) x(*),max,min
  INTEGER(c_int),VALUE :: n
  INTRINSIC maxval,minval
  max = MAXVAL(x(:n))
  min = MINVAL(x(:n))
END

extern void FindMinMax(double *x,int n,double *maxval,double *minval);
double x[100],xmax,xmin;
```

```
int n;
...
FindMinMax(x,n,&xmax,&xmin);
```

This also allows C procedures to be called from Fortran, by describing the C procedure to be called in an interface block. Here is an example:

```
/* This is the prototype for a C library function from 4.3BSD. */
int getloadavg(double loadavg[],int nelem);

PROGRAM show_loadavg
  USE iso_c_binding
  INTERFACE
    FUNCTION getloadavg(loadavg,nelem) BIND(C)
      IMPORT c_double,c_int
      REAL(c_double) loadavg(*)
      INTEGER(c_int),VALUE :: nelem
      INTEGER(c_int) getloadavg
    END FUNCTION
  END INTERFACE
  REAL(c_double) averages(3)
  IF (getloadavg(averages,3)/=3) THEN
    PRINT *,'Unexpected error'
  ELSE
    PRINT *,'Load averages:',averages
  END IF
END
```

70.5 Enumerations

An enumeration defines a set of integer constants of the same kind, and is equivalent to the C `enum` declaration. For example,

```
  ENUM,BIND(C)
    ENUMERATOR :: open_door=4, close_door=17
    ENUMERATOR :: lock_door
  END ENUM
```

is equivalent to

```
  enum {
    open_door=4, close_door=17, lock_door
  };
```

If a value is not given for one of the enumerators, it will be one greater than the previous value (or zero if it is the first enumerator in the list). The kind used for a particular set of enumerators can be discovered by using the `KIND` intrinsic on one of the enumerators.

Note that the `BIND(C)` clause is required; the standard only defines enumerations for interoperating with C.

71 IEEE arithmetic support [4.x except as otherwise noted]

71.1 Introduction

Three intrinsic modules are provided to support use of IEEE arithmetic, these are: `IEEE_ARITHMETIC`, `IEEE_EXCEPTIONS` and `IEEE_FEATURES`. This extension is small superset of the one described by the ISO Technical Report ISO/IEC TR 15580:1999.

71.2 Exception flags, modes and information flow

The model of IEEE arithmetic used by Fortran 2003 is that there is a global set of flags that indicate whether particular floating-point exceptions (such as overflow) have occurred, several operation modes which affect floating-point operations and exception handling, and a set of rules for propagating the flags and modes to obtain good performance and reliability.

The propagation rule for the exception flags is that the information "flows upwards". Thus each procedure starts with the flags clear, and when it returns any flag that is set will cause the corresponding flag in the caller to become set. This ensures that procedures that can be executed in parallel will not interfere with each other via the IEEE exception flags. When the computer hardware only supports a single global set of flags, this model needs enforcement on in procedures that themselves examine the flags (by `IEEE_GET_FLAG` or `IEEE_GET_STATUS`).

The propagation rule for modes, is that the mode settings "flow downwards". This enables code motion optimisations within all routines, and the only cost is that procedures which modify the modes must restore them (to the state on entry) when they return.

The modes that are available are:

- separately, whether each floating point exception terminates the program or allows execution to continue (providing a default result and raising an exception flag);
- rounding mode for floating-point operations;
- underflow mode.

71.3 Procedures in the modules

All the procedures provided by these modules are generic procedures, and not specific: that means that they cannot be passed as actual arguments.

The function `IEEE_SELECTED_REAL_KIND`, and all the functions whose names begin with `IEEE_SUPPORT_`, are permitted to appear in specification and constant expressions (as long as the arguments are appropriate for the context). The elemental functions are also permitted to appear in specification expressions, but not constant expressions.

In the descriptions of the procedures, where it says `REAL(*)` it means any kind of `REAL` (this is not standard Fortran syntax). Conversely, where it says `LOGICAL` it means default `LOGICAL` only, not any other kind of `LOGICAL`.

The functions whose names begin '`IEEE_SUPPORT_`' are all enquiry functions. Many of these take a `REAL(*)` argument `X`; only the kind of `X` is used by the enquiry function, so `X` is permitted to be undefined, unallocated, disassociated, or an undefined pointer.

Note that a procedure must not be invoked on a data type that does not support the feature the procedure uses; the "support" enquiry functions can be used to detect this.

71.4 The IEEE_FEATURES module

This module defines the derived type `IEEE_FEATURES_TYPE`, and up to 11 constants of that type representing IEEE features: these are as follows.

`IEEE_DATATYPE`	whether any IEEE datatypes are available
`IEEE_DENORMAL`	whether IEEE subnormal values are available*
`IEEE_DIVIDE`	whether division has the accuracy required by IEEE*
`IEEE_HALTING`	whether control of halting is supported
`IEEE_INEXACT_FLAG`	whether the inexact exception is supported*
`IEEE_INF`	whether IEEE positive and negative infinities are available*
`IEEE_INVALID_FLAG`	whether the invalid exception is supported*
`IEEE_NAN`	whether IEEE NaNs are available*
`IEEE_ROUNDING`	whether all IEEE rounding modes are available*
`IEEE_SQRT`	whether SQRT conforms to the IEEE standard*
`IEEE_UNDERFLOW_FLAG`	whether the underflow flag is supported*

(*) for at least one kind of REAL.

Those feature types which are required by the user procedure should be explicitly referenced by the USE statement with an ONLY clause, e.g.

```
USE,INTRINSIC :: IEEE_FEATURES,ONLY:IEEE_SQRT
```

This ensures that if the feature specified is not available the compilation will fail.

The type IEEE_FEATURES_TYPE is not in itself useful.

71.5 IEEE_EXCEPTIONS

Provides data types, constants and generic procedures for handling IEEE floating-point exceptions.

71.5.1 Types and constants

```
TYPE IEEE_STATUS_TYPE
```

Variables of this type can hold a floating-point status value; it combines all the mode settings and flags.

```
TYPE IEEE_FLAG_TYPE
```

Values of this type specify individual IEEE exception flags; constants for these are available as follows.

IEEE_DIVIDE_BY_ZERO	division by zero flag
IEEE_INEXACT	inexact result flag
IEEE_INVALID	invalid operation flag
IEEE_OVERFLOW	overflow flag
IEEE_UNDERFLOW	underflow flag

In addition, two array constants are available for indicating common combinations of flags:

```
TYPE(IEEE_FLAG_TYPE),PARAMETER :: &
   IEEE_USUAL(3) = (/ IEEE_DIVIDE_BY_ZERO,IEEE_INVALID,IEEE_OVERFLOW /), &
   IEEE_ALL(5) = (/ IEEE_DIVIDE_BY_ZERO,IEEE_INVALID,IEEE_OVERFLOW, &
                    IEEE_UNDERFLOW,IEEE_INEXACT /)
```

71.5.2 Procedures

The procedures provided by IEEE_EXCEPTIONS are as follows.

```
ELEMENTAL SUBROUTINE IEEE_GET_FLAG(FLAG,FLAG_VALUE)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
  LOGICAL,INTENT(OUT) :: FLAG_VALUE
```

Sets FLAG_VALUE to .TRUE. if the exception flag indicated by FLAG is currently set, and to .FALSE. otherwise.

```
ELEMENTAL SUBROUTINE IEEE_GET_HALTING_MODE(FLAG,HALTING)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
  LOGICAL,INTENT(OUT) :: HALTING
```

Sets HALTING to .TRUE. if the program will be terminated on the occurrence of the floating-point exception designated by FLAG, and to .FALSE. otherwise.

```
PURE SUBROUTINE IEEE_GET_STATUS(STATUS_VALUE)
  TYPE(IEEE_STATUS_TYPE),INTENT(OUT) :: STATUS_VALUE
```

Sets `STATUS_VALUE` to the current floating-point status; this contains all the current exception flag and mode settings.

```
PURE SUBROUTINE IEEE_SET_FLAG(FLAG,FLAG_VALUE)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
  LOGICAL,INTENT(IN) :: FLAG_VALUE
```

Sets the exception flag designated by `FLAG` to `FLAG_VALUE`. `FLAG` may be an array of any rank, as long as it has no duplicate values, in which case `FLAG_VALUE` may be scalar or an array with the same shape.

```
PURE SUBROUTINE IEEE_SET_HALTING_MODE(FLAG,HALTING)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
  LOGICAL,INTENT(IN) :: HALTING
```

Sets the halting mode for the exception designated by `FLAG` to `HALTING`. `FLAG` may be an array of any rank, as long as it has no duplicate values, in which case `HALTING` may be scalar or an array with the same shape.

```
PURE SUBROUTINE IEEE_SET_STATUS(STATUS_VALUE)
  TYPE(IEEE_STATUS_TYPE),INTENT(IN) :: STATUS_VALUE
```

Sets the floating-point status to that stored in `STATUS_VALUE`. This must have been previously obtained by calling `IEEE_GET_STATUS`.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_FLAG(FLAG)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
```

Returns whether the exception flag designated by `FLAG` is supported for all kinds of `REAL`.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_FLAG(FLAG,X)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
  REAL(*),INTENT(IN) :: X
```

Returns whether the exception flag designated by `FLAG` is supported for `REAL` with the kind of `X`.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_HALTING(FLAG)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
```

Returns whether control of the "halting mode" for the exception designated by `FLAG` is supported.

71.6 IEEE_ARITHMETIC module

Provides additional functions supporting IEEE arithmetic: it includes the entire contents of `IEEE_EXCEPTIONS`.

71.6.1 IEEE datatype selection

The `IEEE_SELECTED_REAL_KIND` function is similar to the `SELECTED_REAL_KIND` intrinsic function, but selects among IEEE-compliant `REAL` types ignoring any that are not compliant.

71.6.2 Enquiry functions

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_DATATYPE()
```

Returns whether all real variables `X` satisfy the conditions for `IEEE_SUPPORT_DATATYPE(X)`.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_DATATYPE(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether variables with the kind of `X` satisfy the following conditions:

- the numbers with absolute value between `TINY(X)` and `HUGE(X)` are exactly those of an IEEE floating-point format;
- the `+`, `-` and `*` operations conform to IEEE for at least one rounding mode;
- the functions `IEEE_COPY_SIGN`, `IEEE_LOGB`, `IEEE_NEXT_AFTER`, `IEEE_NEXT_DOWN`, `IEEE_NEXT_UP`, `IEEE_REM`, `IEEE_SCALB` and `IEEE_UNORDERED` may be used.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_DENORMAL()

PURE LOGICAL FUNCTION IEEE_SUPPORT_DENORMAL(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether for all real kinds, or variables with the kind of `X`, subnormal values (with absolute value between zero and `TINY`) exist as required by IEEE and operations on them conform to IEEE.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_DIVIDE()

PURE LOGICAL FUNCTION IEEE_SUPPORT_DIVIDE(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether intrinsic division (`/`) conforms to IEEE, for all real kinds or variables with the kind of `X` respectively.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_INF()

PURE LOGICAL FUNCTION IEEE_SUPPORT_INF(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether for all real kinds, or variables with the kind of `X`, positive and negative infinity values exist and behave in conformance with IEEE.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_IO()

PURE LOGICAL FUNCTION IEEE_SUPPORT_IO(X)
  REAL(*),INTENT(IN) :: X
```

[5.2] Returns whether for all real kinds, or variables with the kind of `X`, conversion to and from text during formatted input/output conforms to IEEE, for the input/output rounding modes `ROUND='DOWN'`, `'NEAREST'`, `'UP'` and `'ZERO'` (and the corresponding edit descriptors `RD`, `RN`, `RU` and `RZ`).

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_NAN()

PURE LOGICAL FUNCTION IEEE_SUPPORT_NAN(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether for all real kinds, or variables with the kind of X, positive and negative "Not-a-Number" values exist and behave in conformance with IEEE.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_ROUNDING(ROUND_VALUE)
  TYPE(IEEE_ROUND_TYPE),INTENT(IN) :: ROUND_VALUE

PURE LOGICAL FUNCTION IEEE_SUPPORT_ROUNDING(ROUND_VALUE,X)
  TYPE(IEEE_ROUND_TYPE),INTENT(IN) :: ROUND_VALUE
  REAL(*),INTENT(IN) :: X
```

Returns whether for all real kinds, or variables with the kind of X, the rounding mode designated by ROUND_VALUE may be set using IEEE_SET_ROUNDING_MODE and conforms to IEEE.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_SQRT()

PURE LOGICAL FUNCTION IEEE_SUPPORT_SQRT(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether the intrinsic function SQRT conforms to IEEE, for all real kinds or variables with the kind of X respectively.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_SUBNORMAL()

PURE LOGICAL FUNCTION IEEE_SUPPORT_SUBNORMAL(X)
  REAL(*),INTENT(IN) :: X
```

[7.0] Returns whether for all real kinds, or variables with the kind of X, subnormal values (with absolute value between zero and TINY) exist as required by IEEE and operations on them conform to IEEE. This function is from Fortran 2018.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_STANDARD()

PURE LOGICAL FUNCTION IEEE_SUPPORT_STANDARD(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether for all real kinds, or variables with the kind of X, all the facilities described by the IEEE modules except for input/output conversions (see IEEE_SUPPORT_IO) are supported and conform to IEEE.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_UNDERFLOW_CONTROL()

PURE LOGICAL FUNCTION IEEE_SUPPORT_UNDERFLOW_CONTROL(X)
  REAL(*),INTENT(IN) :: X
```

[5.2] Returns whether for all real kinds, or variables with the kind of X, the underflow mode can be controlled with IEEE_SET_UNDERFLOW_MODE.

71.6.3 Rounding mode

```
TYPE IEEE_ROUND_TYPE
```

Values of this type specify the IEEE rounding mode. The following predefined constants are provided.

IEEE_DOWN	round down (towards minus infinity)
IEEE_NEAREST	round to nearest (ties to even)
IEEE_TO_ZERO	round positive numbers down, negative numbers up
IEEE_UP	round up (towards positive infinity)
IEEE_OTHER	any other rounding mode

```
PURE SUBROUTINE IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
  TYPE(IEEE_ROUND_TYPE),INTENT(OUT) :: ROUND_VALUE
```

Set ROUND_VALUE to the current rounding mode.

```
PURE SUBROUTINE IEEE_SET_ROUNDING_MODE(ROUND_VALUE)
  TYPE(IEEE_ROUND_TYPE),INTENT(IN) :: ROUND_VALUE
```

Set the rounding mode to that designated by ROUND_VALUE.

71.6.4 Underflow mode

The underflow mode is either "gradual underflow" as specified by the IEEE standard, or "abrupt underflow".

With gradual underflow, the space between -TINY(X) and TINY(X) is filled with equally-spaced "subnormal" numbers; the spacing of these numbers is equal to the spacing of model numbers above TINY(X) (and equal to the smallest subnormal number). This gradually reduces the precision of the numbers as they get closer to zero: the smallest number has only one bit of precision, so any calculation with such a number will have a very large relative error.

With abrupt underflow, the only value between -TINY(X) and TINY(X) is zero. This kind of underflow is nearly universal in non-IEEE arithmetics and is widely provided by hardware even for IEEE arithmetic. Its main advantage is that it can be much faster.

```
SUBROUTINE IEEE_GET_UNDERFLOW_MODE(GRADUAL)
  LOGICAL,INTENT(OUT) :: GRADUAL
```

Sets GRADUAL to .TRUE. if the current underflow mode is gradual underflow, and to .FALSE. if it is abrupt underflow.

```
SUBROUTINE IEEE_SET_UNDERFLOW_MODE(GRADUAL)
  LOGICAL,INTENT(IN) :: GRADUAL
```

Sets the underflow mode to gradual underflow if GRADUAL is .TRUE., and to abrupt underflow if it is .FALSE..

71.6.5 Number Classification

```
TYPE IEEE_CLASS_TYPE
```

Values of this type indicate the IEEE class of a number; this is equal to one of the provided constants:

IEEE_NEGATIVE_DENORMAL	negative subnormal number x, in the range $-$TINY(x)$<x<0$
IEEE_NEGATIVE_INF	$-\infty$ (negative infinity)
IEEE_NEGATIVE_NORMAL	negative normal number x, in the range $-$HUGE(x)$\leq x\leq -$TINY(x)
IEEE_NEGATIVE_ZERO	-0 (zero with the sign bit set)
IEEE_POSITIVE_DENORMAL	positive subnormal number x, in the range $0<x<$TINY(x)
IEEE_POSITIVE_INF	$+\infty$ (positive infinity)
IEEE_POSITIVE_NORMAL	positive normal number x, in the range TINY(x)$\leq x\leq$HUGE(x)
IEEE_POSITIVE_ZERO	$+0$ (zero with the sign bit clear)
IEEE_QUIET_NAN	Not-a-Number (usually the result of an invalid operation)
IEEE_SIGNALING_NAN	Not-a-Number which raises the invalid signal on reference
[5.2] IEEE_OTHER_VALUE	any value that does not fit one of the above categories

[7.0] The constants IEEE_POSITIVE_SUBNORMAL and IEEE_NEGATIVE_SUBNORMAL, from Fortran 2018, are also provided; they have the same values as IEEE_POSITIVE_DENORMAL and IEEE_NEGATIVE_DENORMAL respectively.

The comparison operators .EQ. (=) and .NE. (/=) are provided for comparing values of this type.

```
ELEMENTAL TYPE(IEEE_CLASS_TYPE) FUNCTION IEEE_CLASS(X)
  REAL(*),INTENT(IN) :: X
```

returns the classification of the value of `X`.

```
ELEMENTAL REAL(*) FUNCTION IEEE_VALUE(X,CLASS)
  REAL(*),INTENT(IN) :: X
  TYPE(IEEE_CLASS_TYPE),INTENT(IN) :: CLASS
```

Returns a "sample" value with the kind of `X` and the classification designated by `CLASS`.

71.6.6 Test functions

The following procedures are provided for testing IEEE values.

```
ELEMENTAL LOGICAL FUNCTION IEEE_IS_FINITE(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether X is "finite", i.e. not an infinity, NaN, or `IEEE_OTHER_VALUE`.

```
ELEMENTAL LOGICAL FUNCTION IEEE_IS_NAN(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether X is a NaN.

```
ELEMENTAL LOGICAL FUNCTION IEEE_IS_NEGATIVE(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether X is negative; it differs the comparison `X<0` only in the case of negative zero, where it returns `.TRUE.`.

```
ELEMENTAL LOGICAL FUNCTION IEEE_UNORDERED(X,Y)
  REAL(*),INTENT(IN) :: X,Y
```

Returns the value of '`IEEE_IS_NAN(X) .OR. IEEE_IS_NAN(Y)`'.

71.6.7 Arithmetic functions

```
ELEMENTAL REAL(*) FUNCTION IEEE_COPY_SIGN(X,Y)
  REAL(*),INTENT(IN) :: X,Y
```

Returns `X` with the sign bit of `Y`.

```
ELEMENTAL REAL(*) FUNCTION IEEE_LOGB(X)
  REAL(*),INTENT(IN) :: X
```

If `X` is zero, returns $-\infty$ if infinity is supported and `-HUGE(X)` otherwise. For nonzero `X`, returns `EXPONENT(X)-1`

```
ELEMENTAL REAL(*) FUNCTION IEEE_NEXT_AFTER(X,Y)
  REAL(*),INTENT(IN) :: X,Y
```

Returns the nearest number to `X` that is closer to `Y`, or `X` if `X` and `Y` are equal. If the result is subnormal, `IEEE_UNDERFLOW` is signalled. If the result is infinite but `X` was finite, `IEEE_OVERFLOW` is signalled.

```
ELEMENTAL REAL(*) FUNCTION IEEE_NEXT_DOWN(X)
  REAL(*),INTENT(IN) :: X
```

[7.0] Returns the nearest number to `X` that is less than it, unless `X` is $-\infty$ or NaN, in which case if X is a signalling NaN a quiet NaN is returned, otherwise X is returned. No exception is signalled unless X is a signalling NaN.

```
ELEMENTAL REAL(*) FUNCTION IEEE_NEXT_UP(X)
  REAL(*),INTENT(IN) :: X
```

[7.0] Returns the nearest number to `X` that is greater than it, unless `X` is $+\infty$ or NaN, in which case if X is a signalling NaN a quiet NaN is returned, otherwise X is returned. No exception is signalled unless X is a signalling NaN.

```
ELEMENTAL REAL(*) FUNCTION IEEE_REM(X,Y)
  REAL(*),INTENT(IN) :: X,Y
```

Returns the exact remainder resulting from the division of `X` by `Y`.

```
ELEMENTAL REAL(*) FUNCTION IEEE_RINT(X)
  REAL(*),INTENT(IN) :: X
```

Returns `X` rounded to an integer according to the current rounding mode.

```
ELEMENTAL REAL(*) FUNCTION IEEE_SCALB(X,I)
  REAL(*),INTENT(IN) :: X
  INTEGER(*),INTENT(IN) :: I
```

Returns `SCALE(X,I)`, i.e. X*2^I, without computing 2^I separately.

72 Input/output Features

72.1 Stream input/output [5.1]

A stream file is a file that is opened with the `ACCESS='STREAM'` specifier. A stream file is either a formatted stream or an unformatted stream.

A formatted stream file is equivalent to a C text stream; this acts much like an ordinary sequential file, except that there is no limit on the length of a record. Just as in C, when writing to a formatted stream, an embedded newline character in the data causes a new record to be created. The new intrinsic enquiry function `NEW_LINE(A)` returns this character for the kind (character set) of `A`; if the character set is ASCII, this is equal to `IACHAR(10)`. For example,

```
OPEN(17,FORM='formatted',ACCESS='stream',STATUS='new')
WRITE(17,'(A)'), 'This is record 1.'//NEW_LINE('A')//'This is record 2.'
```

An unformatted stream file is equivalent to a C binary stream, and has no record boundaries. This makes it impossible to `BACKSPACE` an unformatted stream. Data written to an unformatted stream is transferred to the file with no formatting, and data read from an unformatted stream is transferred directly to the variable as it appears in the file.

When reading or writing a stream file, the `POS=` specifier may be used to specify where in the file the data is to be written. The first character of the file is at position 1. The `POS=` specifier may also be used in an `INQUIRE` statement, in which case it returns the current position in the file. When reading or writing a formatted stream, the `POS=` in a `READ` or `WRITE` shall be equal to 1 (i.e. the beginning of the file) or to a value previously discovered through `INQUIRE`.

Note that unlike unformatted sequential files, writing to an unformatted stream file at a position earlier than the end of the file does not truncate the file. (However, this truncation does happen for formatted streams.)

Finally, the `STREAM=` specifier has been added to the `INQUIRE` statement. This specifier takes a scalar default character variable, and assigns it the value `'YES'` if the file may be opened for stream input/output (i.e. with `ACCESS='STREAM'`), the value `'NO'` if the file cannot be opened for stream input/output, and the value `'UNKNOWN'` if it is not known whether the file may be opened for stream input/output.

72.2 The BLANK= and PAD= specifiers [5.1]

The BLANK= and PAD= specifiers, which previously were only allowed on an OPEN statement (and INQUIRE), are now allowed on a READ statement. These change the BLANK= or PAD= mode for the duration of that READ statement only.

72.3 Decimal Comma [5.1]

It is possible to read and write numbers with a decimal comma instead of a decimal point. Support for this is provied by the DECIMAL= specifier and the DC and DP edit descriptors. The DECIMAL= specifier may appear in OPEN, READ, WRITE and INQUIRE statements; possible values are 'POINT' (the default) and 'COMMA'. For an unconnected or unformatted unit, INQUIRE returns 'UNDEFINED'. The DC edit descriptor temporarily sets the mode to DECIMAL='COMMA', and the DP edit descriptor temporarily sets the mode to DECIMAL='POINT'.

When the mode is DECIMAL='COMMA', all floating-point output will produce a decimal comma instead of a decimal point, and all floating-point input will expect a decimal comma. For example,

```
  PRINT '(1X,"Value cest ",DC,F0.2)',1.25
```

will produce the output

```
 Value cest 1,25
```

Additionally, in this mode, a comma cannot be used in list-directed input to separate items; instead, a semi-colon may be used.

72.4 The DELIM= specifier [5.1]

The DELIM= specifier, which previously was only allowed on an OPEN statement (and INQUIRE), is now allowed on a WRITE statement. It changes the DELIM= mode for the duration of that WRITE statement; note that this only has any effect if the WRITE statement uses list-directed or namelist output. For example,

```
  WRITE(*,*,DELIM='QUOTE') "That's all folks!"
```

will produce the output

```
 'That''s all folks!'
```

72.5 The ENCODING= specifier [5.1]

The ENCODING= specifier is permitted on OPEN and INQUIRE statements. Standard values for this specifier are 'UTF-8' and the default value of 'DEFAULT'; the 'UTF-8' value is only allowed on compilers that support a Unicode character kind (see SELECTED_CHAR_KIND). This release of the NAG Fortran Compiler supports 'DEFAULT' and 'ASCII'.

72.6 The IOMSG= specifier [5.1]

The IOMSG= specifier has been added to all input/output statements. This takes a scalar default character variable, which in the event of an error is assigned an explanatory message. (Note that this is only useful if the statement contains an IOSTAT= or ERR= specifier, otherwise the program will be terminated on error anyway.) If no error occurs, the value of the variable remains unchanged.

72.7 The IOSTAT= specifier [5.1]

This now accepts any kind of integer variable (previously this was required to be default integer).

72.8 The SIGN= specifier [5.1]

The SIGN= specifier has been added to the OPEN, WRITE and INQUIRE statements; possible values are 'PLUS', 'SUPPRESS' and 'PROCESSOR_DEFINED' (the default). For the NAG Fortran Compiler, SIGN='PROCESSOR_DEFINED' has the same effect as SIGN='SUPPRESS'.

The effect of SIGN='PLUS' is the same as the SP edit descriptor, the effect of SIGN='SUPPRESS' is the same as the SS edit descriptor, and the effect of SIGN='PROCESSOR_DEFINED' is the same as the S edit descriptor.

72.9 Intrinsic functions for testing IOSTAT= values [5.1]

The intrinsic functions IS_IOSTAT_END and IS_IOSTAT_EOR test IOSTAT= return values, determining whether a value indicates an end-of-file condition or an end-of-record condition. These are equivalent to testing the IOSTAT= return value against the named constants IOSTAT_END and IOSTAT_EOR respectively; these constants are available through the ISO_FORTRAN_ENV module.

72.10 Input/output of IEEE infinities and NaNs [5.1]

Input and output of IEEE infinities and NaNs is possible: the output format is

- -Infinity (or -Inf if it will not fit) for negative infinity;
- Infinity (or Inf if it will not fit) for positive infinity, or +Infinity (or +Inf) with SP or SIGN='PLUS' mode;
- NaN for a NaN.

Furthermore, the output is right-justified within the output field. For list-directed output the output field is the minimum size to hold the result.

Input of IEEE infinities and NaNs is now possible; these take the same form as the output described above, except that:

- case is not significant,
- a NaN may be preceded by a sign, which is ignored, and
- a NaN may be followed by alphanumeric characters enclosed in parentheses (the NAG Fortran Compiler also ignores these).

The result of reading a NaN value in NAG Fortran is always a quiet NaN, never a signalling one.

72.11 Output of floating-point zero [5.1]

List-directed and namelist output of floating-point zero is now done using F format instead of E format. (The Fortran 90 and 95 standards both specified E format.)

72.12 NAMELIST and internal files [5.1]

Namelist input/output is now permitted to/from internal files.

72.13 Variables permitted in NAMELIST

All variables except for assumed-size arrays are now permitted to appear in a namelist group [6.0 for allocatable and pointer, 5.3.1 for the rest]. Note that an allocatable variable that appears in a namelist group must be allocated, and a pointer variable that appears in a namelist group must be associated, when a READ or WRITE statement with that namelist is executed. Also, if a variable is polymorphic or has an ultimate component that is allocatable or a pointer, it is only permitted in a namelist when it will be processed by defined input/output (see below).

72.14 Recursive input/output [5.2]

Input/output to internal files is now permitted while input/output to another internal file or an external file is in progress. This occurs when a function in an input/output list executes an internal file `READ` or `WRITE` statement.

Input/output to an external file while external file input/output is already in progress remains prohibited, except for the case of nested data transfer (see "Defined input/output").

72.15 Asynchronous input/output

72.15.1 Basic syntax [5.1]

Asynchronous input/output syntax is accepted; this consists of the `ASYNCHRONOUS=` specifier on `OPEN`, `READ`, `WRITE` and `INQUIRE`, the `ID=` specifier on `READ`, `WRITE` and `INQUIRE`, and the `PENDING=` specifier on `INQUIRE`.

Except for the `INQUIRE` statement, the `ASYNCHRONOUS=` specifier takes a scalar default character expression; this must evaluate either to `'YES'` or `'NO'`, treating lowercase the same as uppercase. In the `READ` and `WRITE` statements, this character expression must be a constant expression, and the statement must refer to an external file whose connection allows asynchronous input/output; if `ASYNCHRONOUS='YES'` is specified, the data transfer may occur asynchronously. In the `OPEN` statement, this specifier determines whether asynchronous data transfer is allowed for that file: the default setting is `'NO'`. For the `INQUIRE` statement, the `ASYNCHRONOUS=` specifier takes a scalar default character variable, and sets it to `'YES'` if the file is currently connected for asynchronous input/output, `'NO'` if the current connection does not allow asynchronous input/output and `'UNKNOWN'` if the file is not connected.

For the `READ` and `WRITE` statements, the `ID=` specifier takes a scalar integer variable. This specifier is only permitted if `ASYNCHRONOUS='YES'` also appears. The integer variable is assigned the "identifier" of the asynchronous data transfer that the `READ` or `WRITE` initiates; this value can be used in `INQUIRE` and `WAIT` statements to track the progress of the asynchronous data transfer.

For the `INQUIRE` statement, the `ID=` specifier takes a scalar integer expression whose value must be that returned from `ID=` on a `READ` or `WRITE` statement for that file, and is only permitted in conjunction with the `PENDING=` specifier. The `PENDING=` specifier takes a scalar default logical variable and sets it to `.TRUE.` if the specified asynchronous data transfer is still underway and to `.FALSE.` if it has completed. If `PENDING=` is used without `ID=`, the enquiry is about all outstanding asynchronous data transfer on that file.

After initiating an asynchronous data transfer, the variables affected must not be referenced or defined until after the transfer is known to have finished. For an asynchronous `WRITE` of a local variable, this means not returning from the procedure until after ensuring the transfer is complete. An asynchronous data transfer is known to have been finished if there is a subsequent synchronous data transfer, an `INQUIRE` statement which returns `.FALSE.` for `PENDING=`, or a `WAIT` statement has been executed; in each case, for that file.

72.15.2 Basic Example

The following example uses two buffers, `buf1` and `buf2`, alternately reading into one while processing the other, while there is still data in the file to process (each dataset being followed by a single logical value which indicates whether more is to come).

```
REAL :: buf1(n,m),buf2(n,m)
LOGICAL more,stillwaiting
READ (unit) buf1
DO
  READ (unit) more ! Note: synchronous
  IF (more) READ (unit,ASYNCHRONOUS='YES') buf2
  CALL process(buf1)
  IF (.NOT.more) EXIT
  READ (unit) more ! Note: synchronous
  IF (more) READ (unit,ASYNCHRONOUS='YES') buf1
  CALL process(buf2)
  IF (.NOT.more) EXIT
```

```
END DO
```

Note that the synchronous READ statements automatically "wait" for any outstanding asynchronous data transfer to complete before reading the logical value; this ensures that the dataset will have finished being read into its buffer and is safe to process.

72.15.3 The ASYNCHRONOUS attribute [5.2]

A READ or WRITE statement with ASYNCHRONOUS='YES' automatically gives the ASYNCHRONOUS attribute to any variable that appears in its input/output list, in a SIZE= specifier, or which is part of a namelist specified by NML=. This is adequate for asynchronous data transfers that initiate and complete within a single procedure. However, it is inadequate for transfers to/from module variables or dummy arguments if the procedure returns while the transfer is still underway.

The ASYNCHRONOUS attribute may be explicitly specified in a type declaration statement or in an ASYNCHRONOUS statement. The latter has the syntax

ASYNCHRONOUS [::] *variable-name* [, *variable-name*]...

If a variable with the ASYNCHRONOUS attribute is a dummy array and is not an assumed-shape array or array pointer, any associated actual argument cannot be an array section, an assumed-shape array or array pointer. Furthermore, if a dummy argument has the ASYNCHRONOUS attribute the procedure must have an explicit interface. Both of these restrictions apply whether the attribute was given explicitly or implicitly.

72.15.4 The WAIT statement [5.2]

The WAIT statement provides the ability to wait for an asynchronous data transfer to finish without performing any other input/output operation. It has the form

WAIT (*wait-spec* [, *wait-spec*]...)

where *wait-spec* is one of the following:

UNIT = *file-unit-number*
END = *label*
EOR = *label*
ERR = *label*
ID = *scalar-integer-variable*
IOMSG = *scalar-default-character-variable*
IOSTAT = *scalar-integer-variable*

The UNIT= specifier must appear, but the 'UNIT =' may be omitted if it is the first specifier in the list. The ID= specifier takes a scalar integer expression whose value must be that returned from ID= on a READ or WRITE statement for the file; if the ID= specifier does not appear the WAIT statement refers to all pending asynchronous data transfer for that file.

On completion of execution of the WAIT statement the specified asynchronous data transfers have been completed. If the specified file is not open for asynchronous input/output or is not connected, the WAIT statement has no effect (it is not an error unless the ID= specifier was used with an invalid value).

Here is an example of using the WAIT statement.

```
REAL array(1000,1000,10),xferid(10)
! Start reading each segment of the array
DO i=1,10
  READ (unit,id=xfer(i)) array(:,:,i)
END DO
```

```
...
! Now process each segment of the array
DO i=1,10
  WAIT (unit,id=xfer(i))
  CALL process(array(:,:,i)
END DO
```

72.15.5 Execution Semantics

At this time all actual input/output operations remain synchronous, as allowed by the standard.

72.16 Scale factor followed by repeat count [5.1]

The comma that was previously required between a scale factor (*n*`P`) and a repeat count (e.g. the '`3`' in `3E12.2`), is now optional. This trivial extension was part of Fortran 66 that was removed in Fortran 77, and reinstated in Fortran 2003.

72.17 FLUSH statement [5.2]

Execution of a `FLUSH` statement causes the data written to a specified file to be made available to other processes, or causes data placed in that file by another process to become available to the program. The syntax of the `FLUSH` statement is similar to the `BACKSPACE`, `ENDFILE` and `REWIND` statements, being one of the two possibilities

`FLUSH` *file-unit-number*
`FLUSH (` *flush-spec* [, *flush-spec*]... `)`

where *file-unit-number* is the logical unit number (a scalar integer expression), and *flush-spec* is one of the following:

`UNIT =` *file-unit-number*
`IOSTAT =` *scalar-integer-variable*
`IOMSG =` *scalar-default-character-variable*
`ERR =` *label*

The `UNIT=` specifier must appear, but the '`UNIT =`' may be omitted if it is the first *flush-spec* in the list.

Here is an example of the use of a `FLUSH` statement.

```
WRITE (pipe) my_data
FLUSH (pipe)
```

72.18 Defined input/output [6.2]

The generic identifiers `READ(FORMATTED)`, `READ(UNFORMATTED)`, `WRITE(FORMATTED)` and `WRITE(UNFORMATTED)` provide the ability to replace normal input/output processing for an item of derived type. In the case of formatted input/output, the replacement will occur for list-directed formatting, namelist formatting, and for an explicit format with the `DT` edit descriptor: it does not affect any other edit descriptors in an explicit format. Using defined input/output, it is possible to perform input/output on derived types containing pointer components and allocatable components, since the user-defined procedure will be handling it.

Here is a type definition with defined input/output procedures.

```
TYPE tree
  TYPE(tree_node),POINTER :: first
CONTAINS
  PROCEDURE :: fmtread=>tree_fmtread
```

```
    PROCEDURE :: fmtwrite=>tree_fmtwrite
    GENERIC,PUBLIC :: READ(formatted)=>fmtread, WRITE(formatted)=>fmtwrite
  END TYPE
```

Given the above type definition, whenever a `TYPE(tree)` object is an effective item in a formatted input/output list, the module procedure `tree_fmtread` will be called (in a `READ` statement) or the module procedure `tree_fmtwrite` will be called (in a `WRITE` statement) to perform the input or output of that object. Note that a generic interface block may also be used to declare procedures for defined input/output; this is the only option for sequence or `BIND(C)` types but is not recommended for extensible types.

The procedures associated with each input/output generic identifier must have the same characteristics as the ones listed below, where *type-declaration* is `CLASS(`*derived-type-spec*`)` for an extensible type and `TYPE(`*derived-type-spec*`)` for a sequence or `BIND(C)` type. Note that if the derived type has any length type parameters, they **must** be "assumed" (specified as '`*`').

```
SUBROUTINE formatted_read(var,unit,iotype,vlist,iostat,iomsg)
type-declaration,INTENT(INOUT) ::  var
INTEGER,INTENT(IN) ::  unit
CHARACTER(*),INTENT(IN) ::  iotype
INTEGER,INTENT(IN) ::  vlist(:)
INTEGER,INTENT(OUT) ::  iostat
CHARACTER(*),INTENT(INOUT) ::  iomsg

SUBROUTINE unformatted_read(var,unit,iostat,iomsg)
type-declaration,INTENT(INOUT) ::  var
INTEGER,INTENT(IN) ::  unit
INTEGER,INTENT(OUT) ::  iostat
CHARACTER(*),INTENT(INOUT) ::  iomsg

SUBROUTINE formatted_write(var,unit,iotype,vlist,iostat,iomsg)
type-declaration,INTENT(IN) ::  var
INTEGER,INTENT(IN) ::  unit
CHARACTER(*),INTENT(IN) ::  iotype
INTEGER,INTENT(IN) ::  vlist(:)
INTEGER,INTENT(OUT) ::  iostat
CHARACTER(*),INTENT(INOUT) ::  iomsg

SUBROUTINE unformatted_write(var,unit,iostat,iomsg)
type-declaration,INTENT(IN) ::  var
INTEGER,INTENT(IN) ::  unit
INTEGER,INTENT(OUT) ::  iostat
CHARACTER(*),INTENT(INOUT) ::  iomsg
```

In each procedure, `unit` is either a normal unit number if the parent input/output statement used a normal unit number, a negative number if the parent input/output statement is for an internal file, or a processor-dependent number (which might be negative) for the '`*`' unit. The `iostat` argument **must** be assigned a value before returning from the defined input/output procedure: either zero to indicate success, the negative number `IOSTAT_EOR` (from the intrinsic module `ISO_FORTRAN_ENV`) to signal an end-of-record condition, the negative number `IOSTAT_END` to signal an end-of-file condition, or a positive number to indicate an error condition. The `iomsg` argument **must** be left alone if no error occurred, and must be assigned an explanatory message if `iostat` is set to a nonzero value.

For the formatted input/output procedures, the `iotype` argument will be set to '`LISTDIRECTED`' if list-directed formatting is being done, '`NAMELIST`' if namelist formatting is being done, and '`DT`' concatenated with the *character-literal* if the `DT` edit descriptor is being processed. The `vlist` argument contains the list of values in the `DT` edit descriptor if present, and is otherwise a zero-sized array. Note that the syntax of the `DT` edit descriptor is:

`DT` [*character-literal*] [`(` *value* [`,` *value*]... `)`]

where blanks are insignificant, *character-literal* is a default character literal constant with no kind parameter, and each *value* is an optionally signed integer literal constant with no kind parameter. For example, 'DT', 'DT"z8,i4,e10.2"', 'DT(100,-3,+4,666)' and 'DT"silly example"(0)' are all syntactically correct `DT` edit descriptors: it is up to the user-defined procedure to interpret what they might mean.

During execution of a defined input/output procedure, there must be no input/output for an external unit (other than for the `unit` argument), but input/output for internal files is permitted. No file positioning commands are permitted. For unformatted input/output, all input/output occurs within the current record, no matter how many separate data transfer statements are executed by the procedure; that is, file positioning both before and after "nested" data transfer is suppressed. For formatted input/output, this effect is approximately equivalent to the nested data transfer statements being considered to be nonadvancing; explicit record termination (using the slash (`/`) edit descriptor, or transmission of newline characters to a stream file) is effective, and record termination may be performed by a nested list-directed or namelist input/output statement.

If `unit` is associated with an external file (i.e. non-negative, or equal to one of the constants `ERROR_UNIT`, `INPUT_UNIT` or `OUTPUT_UNIT` from the intrinsic module `ISO_FORTRAN_ENV`), the current settings for the pad mode, sign mode, etc., can be discovered by using `INQUIRE` with `PAD=`, `SIGN=`, etc. on the `unit` argument. If `unit` is negative (associated with an internal file), `INQUIRE` will raise the error condition `IOSTAT_INQUIRE_INTERNAL_UNIT`.

Finally, defined input/output is not compatible with asynchronous input/output; all input/output statements involved with defined input/output must be synchronous.

73 Miscellaneous Fortran 2003 Features

73.1 Abstract interfaces and the PROCEDURE statement [5.1]

Abstract interfaces have been added, together with the procedure declaration statement. An abstract interface is defined in an interface block that has the `ABSTRACT` keyword, i.e.

```
ABSTRACT INTERFACE
```

Each interface body in an abstract interface block defines an abstract interface instead of declaring a procedure. The name of an abstract interface can be used in the procedure declaration statement to declare a specifc procedure with that interface, e.g.

```
PROCEDURE(aname) :: spec1, spec2
```

declares `SPEC1` and `SPEC2` to be procedures with the interface (i.e. type, arguments, etc.) defined by the abstract interface `ANAME`.

The procedure declaration statement can also be used with the name of any procedure that has an explicit interface, e.g.

```
PROCEDURE(x) y
```

declares `Y` to have the same interface as `X`. Also, procedures with implicit interfaces can be declared by using `PROCEDURE` with a type specification instead of a name, or by omitting the name altogether.

The following attributes can be declared at the same time on the procedure declaration statement: `BIND(C...)`, `INTENT(`*intent*`)`, `OPTIONAL`, `POINTER`, `PRIVATE`, `PUBLIC`, `SAVE`. For example,

```
PROCEDURE(aname),PRIVATE :: spec3
```

Note that `POINTER` declares a procedure pointer (see next section), and that `INTENT` and `SAVE` are only allowed for procedure pointers not for ordinary procedures. The NAG Fortran Compiler also allows the `PROTECTED` attribute to be specified on the procedure declaration statement: this is an extension to the published Fortran 2003 standard.

73.2 Named procedure pointers [5.2]

A procedure pointer is a procedure with the `POINTER` attribute; it may be a named pointer or a structure component (the latter are described elsewhere). The usual way of declaring a procedure pointer is with the procedure declaration statement, by including the `POINTER` clause in that statement: for example,

```
PROCEDURE(aname),POINTER :: p => NULL()
```

declares P to be a procedure pointer with the interface ANAME, and initialises it to be a disassociated pointer.

A named procedure pointer may also be declared by specifying the POINTER attribute in addition to its normal procedure declaration: for example, a function declared by a type declaration statement will be a function pointer if the POINTER attribute is included in the type declaration:

```
REAL, EXTERNAL, POINTER :: funptr
```

The POINTER statement can also be used to declare a procedure pointer, either in conjunction with an interface block, an EXTERNAL statement, or a type declaration statement, for example:

```
INTERFACE
  SUBROUTINE sub(a,b)
    REAL,INTENT(INOUT) :: a,b
  END SUBROUTINE
END INTERFACE
POINTER sub
```

Procedure pointers may also be stored in derived types as procedure pointer components. The syntax and effects are slightly different, making them act like "object-bound procedures", and as such are described in the object-oriented programming section.

73.3 Intrinsic modules [4.x]

The Fortran 2003 standard classifies modules as either intrinsic or non-intrinsic. A non-intrinsic module is the normal kind of module (i.e. user-defined); an intrinsic module is one that is provided as an intrinsic part of the Fortran compiler.

There are five **standard** modules in Fortran 2003: IEEE_ARITHMETIC, IEEE_EXCEPTIONS, IEEE_FEATURES, ISO_C_BINDING and ISO_FORTRAN_ENV.

A program is permitted to have a non-intrinsic module with the same name as that of an intrinsic module: to this end, the USE statement has been extended: 'USE,INTRINSIC ::' specifies that an intrinsic module is required, whereas 'USE,NON_INTRINSIC ::' specifies that a non-intrinsic module is required. If these are not used, the compiler will select an intrinsic module only if no user-defined module is found. For example,

```
USE,INTRINSIC :: iso_fortran_env
```

uses the standard intrinsic module ISO_FORTRAN_ENV, whereas

```
USE,NON_INTRINSIC :: iso_fortran_env
```

uses a user-defined module with that name. Note that the double-colon '::' is required if either specifier is used.

73.4 Renaming user-defined operators on the USE statement [5.2]

It is now possible to rename a user-defined operator on the USE statement, similarly to how named entities can be renamed. For example,

```
USE my_module, OPERATOR(.localid.)=>OPERATOR(.remotename.)
```

would import everything from MY_MODULE, but the .REMOTENAME. operator would have its name changed to .LOCALID..

Note that this is only available for user-defined operator names; the intrinsic operators .AND. et al cannot have their names changed in this way, nor can ASSIGNMENT(=) be renamed. The local name must be an operator if and only if the remote (module entity) name is an operator: that is, both of

```
USE my_module, something=>OPERATOR(.anything.)
USE my_module, OPERATOR(.something.)=>anything
```

are invalid (a syntax error will be produced).

73.5 The ISO_FORTRAN_ENV module [5.1]

The standard intrinsic module `ISO_FORTRAN_ENV` is now available. It contains the following default `INTEGER` named constants.

`CHARACTER_STORAGE_SIZE`
: size of a character storage unit in bits.

`ERROR_UNIT`
: logical unit number for error reporting ("stderr").

`FILE_STORAGE_SIZE`
: size of the file storage unit used by `RECL=` in bits.

`INPUT_UNIT`
: default ('`*`') unit number for `READ`.

`IOSTAT_END`
: `IOSTAT=` return value for end-of-file.

`IOSTAT_EOR`
: `IOSTAT=` return value for end-of-record.

`NUMERIC_STORAGE_SIZE`
: size of a numeric storage unit in bits.

`OUTPUT_UNIT`
: unit used by `PRINT`, the same as the '`*`' unit for `WRITE`.

73.6 The IMPORT statement [5.1]

The `IMPORT` statement has been added. This has the syntax

`IMPORT` [[`::`] *name* [`,` *name*]...]

and is only allowed in an interface body, where it imports the named entities from the host scoping unit (normally, these entities cannot be accessed from an interface body). If no names are specified, normal host association rules are in effect for this interface body.

The `IMPORT` statement must follow any `USE` statements and precede all other declarations, in particular, `IMPLICIT` and `PARAMETER` statements. Anything imported with `IMPORT` must have been declared prior to the interface body.

73.7 Length of names and statements

Names are now ([4.x]) permitted to be 63 characters long (instead of 31), and statements are now ([5.2]) permitted to have 255 continuation lines (instead of 39).

73.8 Array constructor syntax enhancements

Square brackets (`[ ]`) can now ([5.1]) be used in place of the parenthesis-slash pairs (`(/ /)`) for array constructors. This allows expressions to be more readable when array constructors are being mixed with ordinary parentheses.

```
RESHAPE((/(i/2.0,i=1,100)/),(/2,3/))  ! Old way
RESHAPE([(i/2.0,i=1,100)],[2,3])      ! New way
```

Array constructors may now ([5.2]) begin with a type specification followed by a double colon (::); this makes zero-sized constructors easy (and eliminates potential ambiguity with character length), and also provides assignment conversions thus eliminating the need to pad all character strings to the same length.

```
[ Logical :: ]                              ! Zero-sized logical array
[ Double Precision :: 17.5, 0, 0.1d0 ]     ! Conversions
[ Character(200) :: 'Alf', 'Bernadette' ]  ! Padded to length 200
```

73.9 Structure constructor syntax enhancements [5.3]

There are three enhancements that have been made to structure constructors in Fortran 2003:

1. component names can be used as keywords, the same way that dummy argument names can be used as argument keywords;
2. values can be omitted for components that have default initialisation; and
3. type names can be the same as generic function names, and references are resolved by choosing a suitable function (if the syntax matches the function's argument list) and treating as a structure constructor only if no function matches the actual arguments.

A fourth enhancement is made in the Fortran 2008 standard: a value can be omitted for a component that is allocatable.

This makes structure constructors more like built-in generic functions that can be overridden when necessary. Here is an example showing all three enhancements.

```
TYPE quaternion
  REAL x=0,ix=0,jx=0,kx=0
END TYPE
...
INTERFACE quaternion
  MODULE PROCEDURE quat_from_complex
END INTERFACE
...
TYPE(quaternion) FUNCTION quat_from_complex(c) RESULT(r)
  COMPLEX c
  r%x = REAL(c)
  r%y = AIMAG(c)
  r%z = 0
  r%a = 0
END FUNCTION
...
COMPLEX c
TYPE(quaternion) q
q = quaternion(3.14159265)  ! Structure constructor, value (~pi,0,0,0).
q = quaternion(jx=1)        ! Structure constructor, value (0,0,1,0).
q = quaternion(c)           ! "Constructor" function quat_from_complex.
```

Also, if the type is an extended type an ancestor component name can be used to provide a value for all those inherited components at once.

These extensions mean that even if a type has a private component, you can use the structure constructor if

- the component is allocatable (it will be unallocated),
- the component is default-initialised (it will have the default value), or
- the component is inherited and you use an ancestor component name to provde a value for it and the other components inherited from that ancestor.

73.10 Deferred character length [5.2]

The length of a character pointer or allocatable variable can now be declared to be deferred, by specifying the length as a colon: for example,

```
CHARACTER(LEN=:),POINTER :: ch
```

The length of a deferred-length pointer (or allocatable variable) is determined when it is allocated (see next section) or pointer-associated; for example

```
CHARACTER,TARGET :: t1*3,t2*27
CHARACTER(:),POINTER :: p
p => t1
PRINT *,LEN(p)
p => t2
PRINT *,LEN(p)
```

will first print 3 and then 27. It is not permitted to ask for the `LEN` of a disassociated pointer that has deferred length.

Note that deferred length is most useful in conjunction with the new features of typed allocation, sourced allocation, scalar allocatables and automatic reallocation.

73.11 The ERRMSG= specifier [5.1]

The `ALLOCATE` and `DEALLOCATE` statements now accept the `ERRMSG=` specifier. This specifier takes a scalar default character variable, which in the event of an allocation or deallocation error being detected will be assigned an explanatory message. If no error occurs the variable is left unchanged. Note that this is useless unless the `STAT=` specifier is also used, as otherwise the program will be terminated on error anyway.

For example,

```
ALLOCATE(w(n),STAT=ierror,ERRMSG=message)
IF (ierror/=0) THEN
  PRINT *,'Error allocating W: ',TRIM(message)
  RETURN
END IF
```

73.12 Intrinsic functions in constant expressions [5.2 partial; 5.3 complete]

It is now allowed to use any intrinsic function with constant arguments in a constant expression. (In Fortran 95 real and complex intrinsic functions were not allowed.) For example,

```
MODULE m
  REAL,PARAMETER :: e = EXP(1.0)
END
```

73.13 Specification functions can be recursive [6.2]

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

```
      PURE INTEGER FUNCTION factorial(n) RESULT(r)
        INTEGER,INTENT(IN) :: n
        IF (n>1) THEN
          r = n*factorial(n-1)
```

```
    ELSE
      r = 1
    END IF
  END FUNCTION
```

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

73.14 Access to the command line [5.1]

The intrinsic procedures COMMAND_ARGUMENT_COUNT, GET_COMMAND and GET_COMMAND_ARGUMENT have been added. These duplicate functionality previously only available via the procedures IARGC and GETARG from the F90_UNIX_ENV module.

```
INTEGER FUNCTION command_argument_count()
```

Returns the number of command-line arguments. Unlike IARGC in the F90_UNIX_ENV module, this returns 0 even if the command name cannot be retrieved.

```
SUBROUTINE get_command(command,length,status)
  CHARACTER(*),INTENT(OUT),OPTIONAL :: command
  INTEGER,INTENT(OUT),OPTIONAL :: length,status
```

Accesses the command line which invoked the program. This is formed by concatenating the command name and the arguments separated by blanks. This might differ from the command the user actually typed, and should be avoided (use GET_COMMAND_ARGUMENT instead).

If COMMAND is present, it receives the command (blank-padded or truncated as appropriate). If LENGTH is present, it receives the length of the command. If STATUS is present, it is set to -1 if COMMAND is too short to hold the whole command, a positive number if the command cannot be retrieved, and zero otherwise.

```
SUBROUTINE get_command_argument(number,value,length,status)
  INTEGER,INTENT(IN) :: number
  CHARACTER(*),INTENT(OUT),OPTIONAL :: value
  INTEGER,INTENT(OUT),OPTIONAL :: length,status
```

Accesses command-line argument number NUMBER, where argument zero is the program name. If VALUE is present, it receives the argument text (blank-padded or truncated as appropriate if the length of the argument differs from that of VALUE). If LENGTH is present, it receives the length of the argument. If STATUS is present, it is set to zero for success, -1 if VALUE is too short, and a positive number if an error occurs.

Note that it is an error for NUMBER to be less than zero or greater than the number of arguments (returned by COMMAND_ARGUMENT_COUNT).

73.15 Access to environment variables [5.1]

The intrinsic procedure GET_ENVIRONMENT_VARIABLE has been added. This duplicates the functionality previously only available via the procedure GETENV in the F90_UNIX_ENV module.

```
  SUBROUTINE get_environment_variable(name,value,length,status,trim_name)
    CHARACTER(*),INTENT(IN) :: name
    CHARACTER(*),INTENT(OUT),OPTIONAL :: value
    INTEGER,INTENT(OUT),OPTIONAL :: length,status
    LOGICAL,INTENT(IN),OPTIONAL :: trim_name
  END
```

Accesses the environment variable named by NAME; trailing blanks in NAME are ignored unless TRIM_NAME is present with the value .FALSE.. If VALUE is present, it receives the text value of the variable (blank-padded or truncated as appropriate if the length of the value differs from that of VALUE). If LENGTH is present, it receives the length of the value. If STATUS is present, it is assigned the value 1 if the environment variable does not exist, −1 if VALUE is too short, and zero for success. Other positive values might be assigned for unusual error conditions.

73.16 Character kind selection [5.1]

The intrinsic function SELECTED_CHAR_KIND has been added. At this time the only character set supported is 'ASCII'.

73.17 Argument passing relaxation [5.1]

A CHARACTER scalar actual argument may now be passed to a routine which expects to receive a CHARACTER array, provided the array is explicit-shape or assumed-size (i.e. not assumed-shape, allocatable, or pointer). This is useful for C interoperability.

73.18 The MAXLOC and MINLOC intrinsic functions [5.1]

The MAXLOC and MINLOC intrinsic functions now return zeroes for empty set locations, as required by Fortran 2003 (Fortran 95 left this result processor-dependent).

73.19 The VALUE attribute [4.x]

The VALUE attribute specifies that an argument should be passed by value.

73.19.1 Syntax

The VALUE attribute may be specified by the VALUE statement or with the VALUE keyword in a type declaration statement.

The syntax of the VALUE statement is:

VALUE [::] *name* [, *name*] ...

The VALUE attribute may only be specified for a scalar dummy argument; if the dummy argument is of type CHARACTER, its character length must be constant and equal to one.

Procedures with a VALUE dummy argument must have an explicit interface.

73.19.2 Semantics

A dummy argument with the VALUE attribute is "passed by value"; this means that a local copy is made of the argument on entry to the routine and so modifications to the dummy argument do not affect the associated actual argument and vice versa.

A VALUE dummy argument may be INTENT(IN) but cannot be INTENT(INOUT) or INTENT(OUT).

73.19.3 Example

```
PROGRAM value_example
  INTEGER :: i = 3
  CALL s(i)
  PRINT *,i  ! This will print the value 3
```

```
CONTAINS
  SUBROUTINE s(j)
    INTEGER,VALUE :: j
    j = j + 1  ! This changes the local J without affecting the actual argument
    PRINT *,j  ! This will print the value 4
  END SUBROUTINE
END
```

This example is not intended to be particularly useful, just to illustrate the functionality.

73.20 The VOLATILE attribute [5.0]

This is a horrible attribute which specifies that a variable can be modified by means outside of Fortran. Its semantics are basically the same as that of the C '`volatile`' type qualifier; essentially it disables optimisation for access to that variable.

73.21 Enhanced complex constants [5.2]

The real or imaginary part may now be a named constant, it is not limited to being a literal constant. For example:

```
  REAL,PARAMETER :: minusone = -1.0
  COMPLEX,PARAMETER :: c = (0,minusone)
```

This is not particularly useful, since the same effect can be achieved by using the `CMPLX` intrinsic function.

73.22 The `ASSOCIATE` construct [5.2]

The `ASSOCIATE` construct establishes a temporary association between the "associate names" and the specified variables or values, during execution of a block. Its syntax is

```
ASSOCIATE ( association [ , association ]... )
block
END ASSOCIATE
```

where *block* is a sequence of executable statements and constructs, and *association* is one of

```
name => expression
name => variable
name
```

The last of those is short for '*name* `=>` *name*'. The scope of each "associate name" is the *block* of the `ASSOCIATE` construct. An associate name is never allocatable or a pointer, but otherwise has the same attributes as the *variable* or *expression* (and it has the `TARGET` attribute if the *variable* or *expression* is a pointer). If it is being associated with an expression, the expression is evaluated on execution of the `ASSOCIATE` statement and its value does not change during execution of the *block* — in this case, the associate name is not permitted to appear on the left-hand-side of an assignment or any other context which might change its value. If it is being associated with a variable, the associate name can be treated as a variable.

The type of the associate name is that of the expression or variable with which it is associated. For example, in

```
ASSOCIATE(zoom=>NINT(SQRT(a+b)), alt=>state%mapval(:,i)%altitude)
  alt%x = alt%x*zoom
  alt%y = alt%y*zoom
END ASSOCIATE
```

`ALT` is associated with a variable and therefore can be modified whereas `ZOOM` cannot. The expression for `ZOOM` is of type `INTEGER` and therefore `ZOOM` is also of type `INTEGER`.

73.23 Binary, octal and hexadecimal constants [5.2]

In Fortran 95 these were restricted to `DATA` statements, but in Fortran 2003 these are now allowed to be arguments of the intrinsic functions `CMPLX`, `DBLE`, `INT` and `REAL`. The interpretation is processor-dependent, but the intent is that this specifies the internal representation of the complex or real value. The NAG Fortran compiler requires these constants to have the correct length for the specified kind of complex or real, viz 32 or 64 bits as appropriate.

For example, on a machine where default REAL is IEEE single precision,

```
REAL(z"41280000")
```

has the value `10.5`.

73.24 Character sets [5.1; 5.3]

The support for multiple character sets, especially for Unicode (ISO 10646) has been improved.

The default character set is now required to include lowercase letters and all the 7-bit ASCII printable characters.

The `ENCODING=` specifer for the `OPEN` and `INQUIRE` statements is described in the input/output section.

A new intrinsic function `SELECTED_CHAR_KIND(NAME)` has been added: this returns the character kind for the named character set, or -1 if there is no kind for that character set. Standard character set names are `'DEFAULT'` for the default character kind, `'ASCII'` for the 7-bit ASCII character set and `'ISO_10646'` for the UCS-4 (32-bit Unicode) character set. The name is not case-sensitive. Note that although the method of requesting UCS-4 characters is standardised, the compiler is not required to support them (in which case -1 will be returned); the NAG Fortran Compiler supports UCS-4 in release 5.3 (as well as UCS-2 and JIS X 0213).

Assignment of a character value of one kind to a character value of a different kind is permitted if each kind is one of default character, ASCII character, or UCS-4 character. Assignment to and from a UCS-4 character variable preserves the original value.

Internal file input/output to variables of UCS-4 character kind is allowed (if the kind exists), including numeric conversions (e.g. the E edit descriptor), and conversions from/to default character and ASCII character. Similarly, writing default character, ASCII character and UCS-4 character values to a UTF-8 file and reading them back is permitted and preserves the value.

Finally, the intrinsic function `IACHAR` (for converting characters to the ASCII character set) accepts characters of any kind (in Fortran 95 it only accepted default kind).

73.25 Intrinsic function changes for 64-bit machines [5.2]

Especially to support machines with greater than 32-bit address spaces, but with 32-bit default integers, several intrinsic functions now all have an optional `KIND` argument at the end of the argument list, to specify the kind of integer they return. The functions are: `COUNT`, `INDEX`, `LBOUND`, `LEN`, `LEN_TRIM`, `SCAN`, `SHAPE`, `SIZE`, `UBOUND` and `VERIFY`.

73.26 Miscellaneous intrinsic procedure changes [5.2]

The intrinsic subroutine `DATE_AND_TIME` no longer requires the three character arguments (`DATE`, `TIME` and `ZONE`) to have a minimum length: if the actual argument is too small, it merely truncates the value assigned.

The intrinsic functions `IACHAR` and `ICHAR` now accept an optional `KIND` argument to specify the kind of integer to which to convert the character value. This serves no useful purpose since there are no character sets with characters bigger than 32 bits.

The intrinsic functions `MAX`, `MAXLOC`, `MAXVAL`, `MIN`, `MINLOC` and `MINVAL` all now accept character values; the comparison used is the native (`.LT.`) one, not the ASCII (`LLT`) one.

The intrinsic subroutine `SYSTEM_CLOCK` now accepts a `COUNT_RATE` argument of type real; this is to handle systems whose clock ticks are not an integral divisor of 1 second.

Fortran 2008 Extensions

74 Fortran 2008 Overview

This part of the manual describes those parts of the Fortran 2008 language which are not in Fortran 2003, and which are currently supported by the NAG Fortran Compiler.

Fortran 2008 is a major revision to Fortran 2003: the new language features can be grouped as follows:

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

75 SPMD programming with coarrays [6.2, 7.0]

75.1 Overview

Fortran 2008 contains an SPMD (Single Program Multiple Data) programming model, where multiple copies of a program, called "images", are executed in parallel. Special variables called "coarrays" facilitate communication between images.

Release 6.2 of the NAG Fortran Compiler limited execution to a single image, with no parallel execution. Release 7.0 of the NAG Fortran Compiler can execute multiple images in parallel on SMP machines, using Co-SMP technology.

75.2 Images

Each image contains its own variables and input/output units. The number of images at execution time is not determined by the program, but by some compiler-specific method. The number of images is fixed during execution; images cannot be created or destroyed. The intrinsic function `NUM_IMAGES()` returns the number of images. Each image has an "image index"; this is a positive integer from 1 to the number of images. The intrinsic function `THIS_IMAGE()` returns the image index of the executing image.

75.3 Coarrays

Coarrays are variables that can be directly accessed by another image; they must have the `ALLOCATABLE` or `SAVE` attribute or be a dummy argument.

A coarray has a "corank", which is the number of "codimensions" it has. Each codimension has a lower "cobound" and an upper cobound, determining the "coshape". The upper cobound of the last codimension is "`*`"; rather like an assumed-size array. The "cosubscripts" determine the image index of the reference, in the same way that the subscripts of an array determine the array element number. Again, like an assumed-size array, the image index must be less than or equal to the number of images.

A coarray can be a scalar or an array. It cannot have the `POINTER` attribute, but it can have pointer components.

As well as variables, coarray components are possible. In this case, the component must be an `ALLOCATABLE` coarray, and any variable with such a component must be a dummy argument or have the `SAVE` attribute.

75.4 Declaring coarrays

A coarray has a *coarray-spec* which is declared with square brackets after the variable name, or with the `CODIMENSION` attribute or statement. For example,

```
REAL a[100,*]
REAL,CODIMENSION[-10:10,-10:*] :: b
CODIMENSION c[*]
```

declares the coarray `A` to have corank 2 with lower "cobounds" both 1 and the first upper cobound 100, the coarray `B` to have corank 2 with lower cobounds both -10 and the first upper cobound 10, and the coarray `C` to have corank 1 and lower cobound 1. Note that for non-allocatable coarrays, the *coarray-spec* must always declare the last upper cobound with an asterisk, as this will vary depending on the number of images.

An `ALLOCATABLE` coarray is declared with a *deferred-coshape-spec*, for example,

```
REAL,ALLOCATABLE :: d[:,:,:,:]
```

declares the coarray `D` to have corank 4.

75.5 Accessing coarrays on other images

To access another image's copy of a coarray, cosubscripts are used following the coarray name in square brackets; this is called "coindexing", and such an object is a "coindexed object". For example, given

```
REAL,SAVE :: e[*]
```

the coindexed object `e[1]` refers to the copy of `E` on image 1, and `e[13]` refers to the copy of `E` on image 13. For a more complicated example: given

```
REAL,SAVE :: f[10,21:30,0:*]
```

the reference `f[3,22,1]` refers to the copy of `F` on image 113. There is no correlation between image numbers and any topology of the computer, so it is probably best to avoid complicated codimensions, especially if different coarrays have different coshape.

When a coarray is an array, you cannot put the cosubscripts directly after the array name, but must use array section notation instead. For example, with

```
REAL,SAVE :: g(10,10)[*]
```

the reference `g[inum]` is invalid, to refer to the whole array `G` on image `INUM` you need to use `g(:,:)[inum]` instead.

Similarly, to access a single element of `G`, the cosubscripts follow the subscripts, e.g. `g(i,j)[inum]`.

Finally, note that when a coarray is accessed, whether by its own image or remotely, the segment ordering rules (see next section) must be obeyed. This is to avoid nonsense answers from data races.

75.6 Segments and synchronisation

Execution on each image is divided into segments, by "image control statements". The segments on a single image are ordered: each segment follows the preceding segment. Segments on different images may be ordered (one following the other) by synchronisation, otherwise they are unordered.

If a coarray is defined (assigned a value) in a segment on image I, another image J is only allowed to reference or define it in a segment that follows the segment on I.

The image control statements, and their synchronisation effects, are as follows.

`SYNC ALL` synchronises with corresponding `SYNC ALL` statement executions on other images; the segment following the n^{th} execution of a `SYNC ALL` statement on one image follows all the segments that preceded the n^{th} execution of a `SYNC ALL` statement on every other image.

`SYNC IMAGES (`***list***`)`
synchronises with corresponding `SYNC IMAGES` statement executions on the images in *list*, which is an integer expression that may be scalar or a vector. Including the invoking image number in *list* has no effect. The segment following the n^{th} execution of a `SYNC IMAGES` statement on image I with the image number J in its *list* follows the segments on image J before its n^{th} execution of `SYNC IMAGES` with I in its *list*.

`SYNC IMAGES (*)`
is equivalent to `SYNC IMAGES` with every image no. in its *list*, e.g. `SYNC IMAGES ([(i,i=1,NUM_IMAGES())])`.

`SYNC MEMORY`
This only acts as a segment divider, without synchronising with any other image. It may be useful for user-defined orderings when some other mechanism has been used to synchronise.

`ALLOCATE` **or** `DEALLOCATE`
with a coarray object being allocated or deallocated. This synchronises all images, which must execute the same `ALLOCATE` or `DEALLOCATE` statement.

`CRITICAL` **and** `END CRITICAL`
Only one image can execute a `CRITICAL` construct at a time. The code inside a `CRITICAL` construct forms a segment, which follows the previous execution (on whatever image) of the `CRITICAL` construct.

`LOCK` **and** `UNLOCK`
The segment following a `LOCK` statements that locks a particular lock variable follows the `UNLOCK` statement that previously unlocked the variable.

`END` **statement**
An `END BLOCK`, `END FUNCTION`, or `END SUBROUTINE` statement that causes automatic deallocation of a local `ALLOCATABLE` coarray, synchronises with all images (which must execute the same `END` statement).

`MOVE_ALLOC` **intrinsic**
Execution of the intrinsic subroutine `MOVE_ALLOC` with coarray arguments synchronises all images, which must execute the same `CALL` statement.

Note that image control statements have side-effects, and therefore are not permitted in pure procedures or within `DO CONCURRENT` constructs.

75.7 Allocating and deallocating coarrays

When you allocate an `ALLOCATABLE` coarray, you must give the desired cobounds in the `ALLOCATE` statement. For example,

```
REAL,ALLOCATABLE :: x(:,:,:)[:,:]
...
ALLOCATE(x(100,100,3)[1:10,*])
```

Note that the last upper cobound must be an asterisk, the same as when declaring an explicit-coshape coarray.

When allocating a coarray there is a synchronisation: all images must execute the same `ALLOCATE` statement, and all the bounds, type parameters, and cobounds of the coarray must be the same on all images.

Similarly, there is a synchronisation when a coarray is deallocated, whether by a `DEALLOCATE` statement or automatic deallocation by an `END` statement; every image must execute the same statement.

Note that the usual automatic reallocation of allocatable variables in an intrinsic assignment statement, e.g. when the expression is an array of a different shape, is not available for coarrays. An allocatable coarray variable being assigned to must already be allocated and be conformable with the expression; furthermore, if it has deferred type parameters they must have the same values, and if it is polymorphic it must have the same dynamic type.

75.8 Critical constructs

The CRITICAL construct provides a mechanism for ensuring that only one image at a time executes a code segment. For example,

```
CRITICAL
  ...do something
END CRITICAL
```

If an image I arrives at the CRITICAL statement while another image J is executing the block of the construct, it will wait until image J has executed the END CRITICAL statement before continuing. Thus the CRITICAL — END CRITICAL segment on image I follows the equivalent segment on image J.

As a construct, this may have a name, e.g.

```
critsec: CRITICAL
  ...
END CRITICAL critsec
```

The name has no effect on the operation of the construct. Each CRITICAL construct is separate from all others, and has no effect on their execution.

75.9 Lock variables

A "lock variable" is a variable of the type LOCK_TYPE, defined in the intrinsic module ISO_FORTRAN_ENV. A lock variable must be a coarray, or a component of a coarray. It is initially "unlocked"; it is locked by execution of a LOCK statement, and unlocked by execution of an UNLOCK statement. Apart from those statements, it cannot appear in any variable definition context, other than as the actual argument for an INTENT(INOUT) dummy argument.

Execution of the segment after a LOCK statement successfully locks the variable follows execution of the segment before the UNLOCK statement on the image that unlocked it. For example,

```
INTEGER FUNCTION get_sequence_number()
  USE iso_fortran_env
  INTEGER :: number = 0
  TYPE(lock_type) lock[*]
  LOCK(lock[1])
  number = number + 1
  get_sequence_number = number
  UNLOCK(lock[1])
END FUNCTION
```

If the variable lock on image 1 is locked when the LOCK statement is executed, it will wait for it to become unlocked before continuing. Thus the function get_sequence_number() provides an one-sided ordering relation: the segment following a call that returned the value N will follow every segment that preceded a call that returned a value less than N.

Conditional locking is provided with the ACQUIRED_LOCK= specifier; if this specifier is present, the executing image only acquires the lock if it was previously unlocked. For example,

```
LOGICAL gotit
LOCK(lock[1],ACQUIRED_LOCK=gotit)
IF (gotit) THEN
  ! We have the lock.
ELSE
  ! We do not have the lock - some other image does.
END IF
```

It is an error for an image to try to LOCK a variable that is already locked to that image, or to UNLOCK a variable that is already unlocked, or that is locked to another image. If the STAT= specifier is used, these errors will return the values STAT_LOCKED, STAT_UNLOCKED, or STAT_LOCKED_OTHER_IMAGE respectively (these named constants are provided by the intrinsic module ISO_FORTRAN_ENV).

75.10 Atomic coarray accessing

As an exception to the segment ordering rules, a coarray that is an integer of kind ATOMIC_INT_KIND or a logical of kind ATOMIC_LOGICAL_KIND (these named constants are provided by the intrinsic module ISO_FORTRAN_ENV), can be defined with the intrinsic subroutine ATOMIC_DEFINE, or referenced by the intrinsic subroutine ATOMIC_REF. For example,

```
MODULE stopping
  USE iso_fortran_env
  LOGICAL(atomic_logical_kind),PRIVATE :: stop_flag[*] = .FALSE.
CONTAINS
  SUBROUTINE make_it_stop
    CALL atomic_define(stop_flag[1],.TRUE._atomic_logical_kind)
  END SUBROUTINE
  LOGICAL FUNCTION please_stop()
    CALL atomic_ref(please_stop,stop_flag[1])
  END FUNCTION
END MODULE
```

In this example, it is perfectly valid for any image to call make_it_stop, and for any other image to invoke the function please_stop(), without any regard for segments. (On a distributed memory machine it might take some time for changes to the atomic variable to be visible on other images, but they should eventually get the message.)

Note that ordinary assignment and referencing should not be mixed with calls to the atomic subroutines, as ordinary assignment and referencing are always subject to the segment ordering rules.

75.11 Normal termination of execution

If an image executes a STOP statement, or the END PROGRAM statement, normal termination is initiated. The other images continue execution, and all data on the "stopped" image remains; other images can continue to reference and define coarrays on the stopped image.

When normal termination has been initiated on all images, the program terminates.

75.12 Error termination

If any image terminates due to an error, for example an input/output error in an input/output statement that does not have any IOSTAT= or ERR= specifier, the entire program is error terminated. On a distributed memory machine it may take some time for the error termination messages to reach every image, so the termination might not be immediate.

The ERROR STOP statement initiates error termination.

75.13 Fault tolerance

The Fortran 2018 standard adds many features for detecting, simulating, and recovering from image failure. For example, the FAIL IMAGE statement causes the executing image to fail (stop responding to accesses from other images). These extensions are listed in the detailed syntax below, even though they are not part of the Fortran 2008 standard.

The FAIL IMAGE statement itself is not very useful when the number of images is equal to one, as it inevitably causes complete program failure.

75.14 Detailed syntax of coarray features

Coindexed object (data object designator):

> In a data object designator, a part (component or base object) that is a coarray can include an image selector: *part-name [* `(` *section-subscript-list* `)` *] [image-selector]*

where *part-name* identifies a coarray, and *image-selector* is

> *left-bracket cosubscript-list [* `,` *image-selector-spec] right-bracket*

The number of *cosubscript*s must be equal to the corank of *part-name*. If *image-selector* appears and *part-name* is an array, *section-subscript-list* must also appear. The optional *image-selector-spec* is Fortran 2018 (part of the fault tolerance feature), and is a comma-separated list of one or more of the following specifiers:

> `STAT =` *scalar-int-variable*
> `TEAM =` *team-value*
> `TEAM_NUMBER =` *scalar-int-expression*

A *team-value* must be a scalar expression of type `TEAM_TYPE` from the intrinsic module `ISO_FORTRAN_ENV`. The `STAT=` variable is assigned zero if the reference or definition was successful, and the value `STAT_FAILED` if the image referenced has failed.

`CRITICAL` construct:

> *[construct-name* `:` *]* `CRITICAL` *[* `(` *[sync-stat-list]* `)` *]*
> *block*
> `END CRITICAL` *[construct-name]*

where the optional *sync-stat-list* is a `STAT=` specifier, an `ERRMSG=` specifier, or both (separated by a comma). Note: The optional parentheses and *sync-stat-list* are Fortran 2018.

The *block* is not permitted to contain:

- a `RETURN` or `STOP` statement;
- an image control statement;
- a branch whose target is outside the construct.

`FAIL IMAGE` statement:

> `FAIL IMAGE`

Note: This statement is Fortran 2018.

`LOCK` statement:

> `LOCK (` *lock-variable [* `,` *lock-stat-list]* `)`

where *lock-stat-list* is a comma-separated list of one or more of the following:

> `ACQUIRED_LOCK =` *scalar-logical-variable*
> `ERRMSG =` *scalar-default-character-variable*
> `STAT =` *scalar-int-variable*

and *lock-variable* is a scalar variable of type `LOCK_TYPE` from the intrinsic module `ISO_FORTRAN_ENV`.

`SYNC ALL` statement:

`SYNC ALL` *[* `(` *[sync-stat-list]* `)` *]*

`SYNC IMAGES` statement:

`SYNC IMAGES (` *image-set [*, *sync-stat-list]* `)`

where *image-set* is an asterisk, or an integer expression that is scalar or of rank one.

`SYNC MEMORY` statement:

`SYNC MEMORY` *[* `(` *[sync-stat-list]* `)` *]*

`UNLOCK` statement:

`UNLOCK (` *lock-variable [*, *sync-stat-list]* `)`

Note:

- The variables in *sync-stat-list* or *lock-stat-list* are not permitted to be coindexed objects, nor may they depend on anything else in the statement.

75.15 Intrinsic procedures and coarrays

```
SUBROUTINE ATOMIC_DEFINE(ATOM, VALUE, STAT)
```

`ATOM` is `INTENT(OUT)` scalar `INTEGER(ATOMIC_INT_KIND)` or `LOGICAL(ATOMIC_LOGICAL_KIND)`, and must be a coarray or a coindexed object.

`VALUE` is scalar with the same type as `ATOM`.

`STAT` *(Optional)* is scalar Integer and must have a decimal exponent range of at least four. It must not be coindexed.

The variable `ATOM` is atomically assigned the value of `VALUE`, without regard to the segment rules. If `STAT` is present, it is assigned a positive value if an error occurs, and zero otherwise. Note: `STAT` is part of Fortran 2018.

```
SUBROUTINE ATOMIC_REF(VALUE, ATOM, STAT)
```

`VALUE` is `INTENT(OUT)` scalar with the same type as `ATOM`.

`ATOM` is scalar `INTEGER(ATOMIC_INT_KIND)` or `LOGICAL(ATOMIC_LOGICAL_KIND)`, and must be a coarray or a coindexed object.

`STAT` *(Optional)* is scalar Integer and must have a decimal exponent range of at least four. It must not be coindexed.

The value of `ATOM` is atomically read, without regard to the segment rules, and then assigned to the variable `VALUE`. If `STAT` is present, it is assigned a positive value if an error occurs, and zero otherwise. Note: `STAT` is part of Fortran 2018.

```
INTEGER FUNCTION IMAGE_INDEX(COARRAY, SUB)
```

`COARRAY` a coarray of any type.

`SUB` an integer vector whose size is equal to the corank of `COARRAY`.

If the value of SUB is a valid set of cosubscripts for COARRAY, the value of the result is the image index of the image they will reference, otherwise the result has the value zero. For example, if X is declared with cobounds [11:20,13:*], the result of IMAGE_INDEX(X,[11,13]) will be equal to one, and the result of IMAGE_INDEX(x,[1,1]) will be equal to zero.

```
FUNCTION LCOBOUND(COARRAY, DIM , KIND)
```

COARRAY coarray of any type and corank N;

DIM *(Optional)* scalar Integer in the range 1 to N;

KIND *(Optional)* scalar Integer constant expression;

Result Integer or Integer(Kind=KIND).

If DIM appears, the result is scalar, being the value of the lower cobound of that codimension of COARRAY. If DIM does not appear, the result is a vector of length N containing all the lower cobounds of COARRAY. The actual argument for DIM must not itself be an optional dummy argument.

```
SUBROUTINE MOVE_ALLOC(FROM, TO, STAT, ERRMSG)  ! Revised
```

FROM an allocatable variable of any type.

TO an allocatable with the same declared type, type parameters, rank and corank, as FROM.

STAT INTENT(OUT) scalar Integer with a decimal exponent range of at least four.

ERRMSG INTENT(INOUT) scalar default character variable.

If FROM and TO are coarrays, the CALL statement is an image control statement that synchronises all images. If STAT is present, it is assigned a positive value if any error occurs, otherwise it is assigned the value zero. If ERRMSG is present and an error occurs, it is assigned an explanatory message. Note: The STAT and ERRMSG arguments are Fortran 2018.

```
INTEGER FUNCTION NUM_IMAGES()
```

This intrinsic function returns the number of images. In this release of the NAG Fortran Compiler, the value will always be equal to one.

```
INTEGER FUNCTION THIS_IMAGE()
```

Returns the image index of the executing image.

```
FUNCTION THIS_IMAGE(COARRAY)
```

Returns an array of type Integer with default kind, with the size equal to the corank of COARRAY, which may be a coarray of any type. The values returned are the cosubscripts for COARRAY that correspond to the executing image.

```
INTEGER FUNCTION THIS_IMAGE(COARRAY, DIM)
```

COARRAY is a coarray of any type.

DIM is scalar Integer.

Returns the cosubscript for the codimension DIM that corresponds to the executing image. Note: In Fortran 2008 DIM was not permitted to be an optional dummy argument; Fortran 2018 permits that.

```
FUNCTION UCOBOUND(COARRAY, DIM, KIND)
```

`COARRAY` coarray of any type and corank N;

`DIM` *(Optional)* scalar Integer in the range 1 to N;

`KIND` *(Optional)* scalar Integer constant expression;

Result Integer or Integer(Kind=KIND).

If `DIM` appears, the result is scalar, being the value of the upper cobound of that codimension of `COARRAY`. If `DIM` does not appear, the result is a vector of length N containing all the upper cobounds of `COARRAY`. The actual argument for `DIM` must not itself be an optional dummy argument.

Note that if `COARRAY` has corank $N>1$, and the number of images in the current execution is not an integer multiple of the coextents up to codimension $N-1$, the images do not make a full rectangular pattern. In this case, the value of the last upper cobound is the maximum value that a cosubscript can take for that codimension; e.g. with a coarray-spec of `[1:3,1:*]` and four images in the execution, the last upper cobound will be equal to 2 because the cosubscripts `[1,2]` are valid even though `[2,2]` and `[2,3]` are not.

76 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

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

the single statement

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

will suffice.

- [5.3 for C_ASSOCIATED, 7.0 for C_LOC and C_FUNLOC] A specification expression may now use the C_ASSOCIATED, C_LOC and C_FUNLOC functions from the ISO_C_BINDING module. For example, given a TYPE(C_PTR) variable X and another interoperable variable Y with the TARGET attribute,

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

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

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

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

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

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

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

- [7.1] An allocatable component can forward-reference a type, for example:

```
Type t2
  Type(t),Pointer :: p
  Type(t),Allocatable :: a
End Type
Type t
  Integer c
End Type
```

An allocatable component can also be of recursive type, or two types can be mutually recursive. For example,

```
Type t
  Integer v
  Type(t),Allocatable :: a
End Type
```

This allows lists and trees to be built using allocatable components. Building or traversing such data structures will usually require recursive procedure calls, as there is no allocatable analogue of pointer assignment.

No matter how deeply nested such recursive data structures become, they can never be circular (again, because there is no pointer assignment). As usual, deallocating the top object of such a structure will recursively deallocate all its allocatable components.

- [7.1] A dummy argument can be used in a specification expression in an elemental subprogram, as long as it is not used to specify a type parameter (such as character length) of a function result. For type parameters of function results, they remain limited to appearing in specification enquiries (such as LEN) when the enquiry is not about a deferred characteristic.

For example, in

```
Elemental Subroutine s(x,n,y)
  Real,Intent(In) :: x
  Integer,Intent(In) :: n
  Real,Intent(Out) :: y
  Real temp(n)
  ...
```

the dummy argument `N` can be used to declare the local array `TEMP`.

- [7.1] Pointers and pointer components can be initialised to point to a target. The target must be valid for that pointer (e.g. same type, rank, etc.). The main cases are:

  Named pointer initialisation

  For data pointers, the target must have the `SAVE` attribute (variables in modules and the main program have this attribute implicitly). For procedure pointers, the target must be a module procedure or external procedure, not a dummy procedure, internal procedure, or statement function.

  For example,

  ```
  Module m
    Real,Target :: x
    Real,Pointer :: p => x
  End Module
  Program test
    Use m
    p = 3
    Print *,x ! Will print the value 3.0
  End Program
  ```

  Component default initialisation

  Pointer components can be default-initialised to point to a target. The requirements on the target are the same as for named pointer initialisation.

  For example,

  ```
  Module m
    Real,Target :: x
    Type t
      Real,Pointer :: p => x
    End Type
  End Module
  Program test
    Use m
    Type(t) y
    y%p = 3
    Print *,x ! Will print the value 3.0
  End Program
  ```

  Component initialisation with structure constructors

  A structure constructor in a constant expression can specify a target for any pointer component. The requirements on the target are the same as for named pointer initialisation.

  For example,

  ```
  Module m
    Real,Target :: x
    Type t
      Real,Pointer :: p
    End Type
  End Module
  Program test
    Use m
    Type(t) :: y = t(x)
    y%p = 3
    Print *,x ! Will print the value 3.0
  End Program
  ```

- [7.1] A reference to a function that returns a pointer can be used as a variable in many contexts. In particular, it can be used as the variable in an assignment statement, as the actual argument corresponding to an `INTENT(OUT)` or `INTENT(INOUT)` dummy argument, and as the selector in an `ASSOCIATE` or `SELECT TYPE` construct that modifies the associate-name.

  For example, with this module,

```
Module m
  Real,Target,Save :: table(100) = 0
Contains
  Function f(n)
    Integer,Intent(In) :: n
    Real,Pointer :: f
    f => table(Min(Max(1,n),Size(table)))
  End Function
End Module
```

  the program below will print "`-1.23E+02`".

```
Program example
  Use m
  f(13) = -123
  Print 1,f(13)
1 Format(ES10.3)
End Program
```

  It should be noted that the syntax of a statement function definition is identical to part of the syntax of a pointer function reference as a variable; the existence of a pointer-valued function that is accessible in the scope determines which of these it is. This may lead to confusing error messages in some situations.

  With the above module, this program demonstrates the use of the feature with an `ASSOCIATE` construct.

```
Program assoc_eg
  Use m
  Associate(x=>f(3), y=>f(4))
    x = 0.5
    y = 3/x
  End Associate
  Print 1,table(3:4) ! Will print "  5.00E-01  6.00E+00"
1 Format(2ES10.2)
End Program
```

  Finally, here is an example using argument passing.

```
Program argument_eg
  Use m
  Call set(f(7))
  Print 1,table(7) ! Will print "1.41421"
1 Format(F7.5)
Contains
  Subroutine set(x)
    Real,Intent(Out) :: x
    x = Sqrt(2.0)
  End Subroutine
End Program
```

  Other contexts where a reference to a pointer-valued function may be used instead of a variable designator include:

  – as an internal file specifier in a `WRITE` statement (the function must return a pointer to a character string or array for this);

  - as an input-item in a `READ` statement;
  - as a `STAT=` or `ERRMSG=` variable in an `ALLOCATE` or `DEALLOCATE` statement, or in an image control statement such as `EVENT WAIT`;
  - as the team variable in a `FORM TEAM` statement.

- [7.1] The result of a function can be a procedure pointer. For example,

```
Module ppfun
  Private
  Abstract Interface
    Subroutine charsub(string)
      Character(*),Intent(In) :: string
    End Subroutine
  End Interface
  Public charsub,hello_goodbye
Contains
  Subroutine hello(string)
    Character(*),Intent(In) :: string
    Print *,'Hello: ',string
  End Subroutine
  Subroutine bye(string)
    Character(*),Intent(In) :: string
    Print *,'Goodbye: ',string
    Stop
  End Subroutine
  Function hello_goodbye(flag)
    Logical,Intent(In) :: flag
    Procedure(hello),Pointer :: hello_goodbye
    If (flag) Then
      hello_goodbye => hello
    Else
      hello_goodbye => bye
    End If
  End Function
End Module
Program example
  Use ppfun
  Procedure(charsub),Pointer :: pp
  pp => hello_goodbye(.True.)
  Call pp('One')
  pp => hello_goodbye(.False.)
  Call pp('Two')
End Program
```

  The function `hello_goodbye` in module `ppfun` returns a pointer to a procedure, which needs to be pointer-assigned to a procedure pointer to be invoked. When executed, this example will print

```
Hello: One
Goodbye: Two
```

  Use of this feature is not recommended, as it blurs the lines between data objects and procedures; this may lead to confusion or misunderstandings during code maintenance. The feature provides no functionality that was not already provided by procedure pointer components.

77 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,

```
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
```

- [6.2] 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.

- [6.1] The real and imaginary parts of a `COMPLEX` object can be accessed using the complex part designators '`%RE`' and '`%IM`'. For example, given

```
COMPLEX,PARAMETER :: c = (1,2), ca(2) = [ (3,4),(5,6) ]
```

  the designators `c%re` and `c%im` have the values 1 and 2 respectively, and `ca%re` and `ca%im` are arrays with the values `[ 3,5 ]` and `[ 4,6 ]` respectively. In the case of variables, for example

```
COMPLEX :: v, va(10)
```

  the real and imaginary parts can also be assigned to directly; the statement

```
va%im = 0
```

  will set the imaginary part of each element of `va` to zero without affecting the real part.

- 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,

```
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 operaton of this feature is similar to the way that `ALLOCATE(`*variable*`,SOURCE=`*expr*`)` works. For example, given

```
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).

- [6.1] Rank-remapping pointer assignment is now permitted when the target has rank greater than one, provided it is "simply contiguous" (a term which means that it must be easily seen at compile-time to be contiguous). For example, the pointer assignment in

```
REAL,TARGET :: x(100,100)
REAL,POINTER :: x1(:)
x1(1:Size(x)) => x
```

establishes `X1` as a single-dimensional alias for the whole of `X`.

78 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'
```

- [6.1] The FORALL construct now has an optional type specifier in the initial statement of the construct, which can be used to specify the type (which must be INTEGER) and kind of the index variables. When this is specified, the existence or otherwise of any entity in the outer scope that has the same name as an index variable does not affect the index variable in any way. For example,

```
Complex i(100)
Real x(200)
...
Forall (Integer :: i=1:Size(x)) x(i) = i
```

  Note that the FORALL construct is still not recommended for high performance, as the semantics imply evaluating the right-hand sides into array temps the size of the iteration space, and then assigning to the variables; this usually performs worse than ordinary DO loops.

- [6.1] The DO CONCURRENT construct is a DO loop with restrictions and semantics intended to allow efficient execution. The iterations of a DO CONCURRENT construct may be executed in any order, and possibly even in parallel. The loop index variables are local to the construct.

  The DO CONCURRENT header has similar syntax to the FORALL header, including the ability to explicitly specify the type and kind of the loop index variables, and including the scalar mask.

  The restrictions on the DO CONCURRENT construct are:

  - no branch is allowed from within the construct to outside of it (this includes the RETURN and STOP statements, but ERROR STOP is allowed);
  - the EXIT statement cannot be used to terminate the loop;
  - the CYCLE statement cannot refer to an outer loop;
  - there must be no dependencies between loop iterations, and if a variable is assigned to by any iteration, it is not allowed to be referenced by another iteration unless that iteration assigns it a value first;
  - all procedures referenced within the construct must be pure;
  - no image control statements can appear within the loop;
  - no reference to IEEE_GET_FLAG or IEEE_SET_HALTING_MODE is allowed.

  For example,

```
Integer vsub(n)
...
Do Concurrent (i=1:n)
  ! Safe because vsub has no duplicate values.
  x(vsub(i)) = i
End Do
```

  The full syntax of the DO CONCURRENT statement is:

  [do-construct-name **:** *]* **DO** *[label] [* **,** *]* **CONCURRENT** *forall-header*

  where *forall-header* is

  (*[integer-type-spec* **::** *] triplet-spec [* **,** *triplet-spec]... [* **,** *mask-expr]* **)**

  where *mask-expr* is a scalar logical expression, and *triplet-spec* is

  name = *expr* : *expr [* : *expr]*

79 Intrinsic procedures and modules

79.1 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 $0 \leq im \leq \pi$ for the `ACOSH` function, and $-\pi/2 \leq im \leq \pi/2$ 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`.

- [6.1] The new elemental intrinsic functions `BESSEL_J0`, `BESSEL_Y0`, `BESSEL_J1` and `BESSEL_Y1` compute the Bessel functions J_0, Y_0, J_1 and Y_1 respectively. These functions are solutions to Bessel's differential equation. The J functions are of the 1^{st} kind and the Y functions are of the 2^{nd} kind; the following subscript indicates the order (0 or 1). There is a single argument `X`, which must be of type Real; the result of the function has the same type and kind. For functions of the 2^{nd} kind (`BESSEL_Y0` and `BESSEL_Y1`), the argument `X` must be positive.

  For example, `BESSEL_J0(1.5)` is approximately `0.5118276`, `BESSEL_Y0(1.5)` is approximately `0.3824489`, `BESSEL_J1(1.5)` is approximately `0.5579365` and `BESSEL_Y1(1.5)` is approximately `-0.4123086`.

- [6.1] The new intrinsic functions `BESSEL_JN` and `BESSEL_YN` compute the Bessel functions J_n and Y_n respectively. These functions come in two forms: an elemental form and a transformational form.

  The elemental form has two arguments: `N`, the order of the function to compute, and `X`, the argument of the Bessel function. `BESSEL_JN(0,X)` is identical to `BESSEL_J0(X)`, etc..

  The transformational form has three scalar arguments: `N1`, `N2` and `X`. The result is a vector of size `MAX(N2-N1+1,0)`, containing approximations to the Bessel functions of orders `N1` to `N2` applied to `X`.

  For example, `BESSEL_JN(5,7.5)` is approximately `0.283474`, `BESSEL_YN(5,7.5)` is approximately `0.175418`, `BESSEL_JN(3,5,7.5)` is approximately `[ -0.258061, 0.023825, 0.283474 ]` and `BESSEL_YN(3,5,7.5)` is approximately `[ 0.159708, 0.314180, 0.175418 ]`.

- [6.0] The elemental intrinsic functions `ERF`, `ERFC` and `ERFC_SCALED` 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`(π); 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**2)`; 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.

- 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(CMPLX(X,Y,KIND(X)))`.

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

- The array reduction intrinsic function `NORM2(X,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)`.

79.2 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(INT(Z'FF',INT8),128)` is true, while `INT(Z'FF',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(INT(B'11000101',1),B'11001001',2) has the value INT(B'00010111',1) (decimal value 23), whereas DSHIFTR(INT(B'11000101',1),B'11001001',2) has the value INT(B'01110010',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 + or * operation, they reduce it by the IAND, IOR and IEOR intrinsic functions respectively. That it, 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.

79.3 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.

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

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

where

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

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

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

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

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

79.4 Changes to existing intrinsic procedures [mostly 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).

- [6.2] 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].

- 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 IEEE_SELECTED_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.

79.5 ISO_C_BINDING additions [6.2]

The standard intrinsic module ISO_C_BINDING contains an additional procedure as follows.

```
INTERFACE c_sizeof
  PURE INTEGER(c_size_t) FUNCTION c_sizeof...(x) ! Specific name not visible
    TYPE(*) :: x(..)
  END FUNCTION
END INTERFACE
```

The actual argument x must be interoperable. The result is the same as the C sizeof operator applied to the conceptually corresponding C entity; that is, the size of x in bytes. If x is an array, it is the size of the whole array, not just one element. Note that x cannot be an assumed-size array.

79.6 ISO_FORTRAN_ENV additions

[5.3] The standard 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).

[6.1] The standard intrinsic module ISO_FORTRAN_ENV contains two new functions as follows.

- COMPILER_VERSION. This function is pure, has no arguments, and returns a scalar default character string that identifies the version of the compiler that was used to compile the source file. This function may be used in a constant expression, e.g. to initialise a variable or named constant with this information. For example,

  ```
  Module version_info
    Use Iso_Fortran_Env
    Character(Len(Compiler_Version())) :: compiler = Compiler_Version()
  End Module
  Program show_version_info
    Use version_info
    Print *,compiler
  End Program
  ```

  With release 6.1 of the NAG Fortran Compiler, this program will print something like

  ```
  NAG Fortran Compiler Release 6.1(Tozai) Build 6105
  ```

- COMPILER_OPTIONS. This function is pure, has no arguments, and returns a scalar default character string that identifies the options supplied to the compiler when the source file was compiled. This function may be used in a constant expression, e.g. to initialise a variable or named constant with this information. For example,

  ```
  Module options_info
    Use Iso_Fortran_Env
    Character(Len(Compiler_Options())) :: compiler = Compiler_Options()
  End Module
  Program show_options_info
    Use options_info
    Print *,compiler
  End Program
  ```

  If compiled with the options *−C=array −C=pointer −O*, this program will print something like

  ```
  -C=array -C=pointer -O
  ```

80 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,

```
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

```
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

```
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,

```
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 indefinite repeat count is only allowed at the top level of the format specification, and must be the last format item.

- [6.0] The G0 and G0.*d* edit descriptors perform generalised editing with all leading and trailing blanks (except those within a character value itself) omitted. For example,

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

  produces the output

```
1.250000,T,Hi !,123456789,
```

81 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,

```
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,

```
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,

```
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.
- [6.2] 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
```

- [6.2] 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=''`).
- [6.2] 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
```

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

  A "separate module procedure" is a procedure whose interface is declared in the module specification part, but whose definition may provided either in the module itself, or in a submodule of that module. The interface of

a separate module procedure is declared by using the MODULE keyword in the prefix of the interface body. For example,

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

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

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

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

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

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

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

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

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

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

The initial *submodule-stmt* has the form

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

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

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

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

For example,

```
MODULE mymod
  INTERFACE
    MODULE INTEGER FUNCTION next_number() RESULT(r)
    END FUNCTION
    MODULE SUBROUTINE reset()
    END SUBROUTINE
  END INTERFACE
END MODULE
SUBMODULE (mymod) variables
  INTEGER :: next = 1
END SUBMODULE
SUBMODULE (mymod:variables) functions
CONTAINS
  MODULE PROCEDURE next_number
    r = next
    next = next + 1
  END PROCEDURE
END SUBMODULE
SUBMODULE (mymod:variables) subroutines
CONTAINS
  MODULE SUBROUTINE reset()
    PRINT *,'Resetting'
    next = 1
  END SUBROUTINE
END SUBMODULE
PROGRAM demo
  USE mymod
  PRINT *,'Hello',next_number()
  PRINT *,'Hello again',next_number()
  CALL reset
  PRINT *,'Hello last',next_number()
END PROGRAM
```

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

Fortran 2018 Extensions

82 Fortran 2018 Overview

The new features of Fortran 2018 that are supported by the NAG Fortran Compiler can be grouped as follows:

- Data declaration
- Data usage and computation
- Input/output
- Execution control
- Intrinsic procedures and modules
- Program units and procedures
- Advanced C interoperability
- Updated IEEE arithmetic capabilities
- Advanced coarray programming

83 Data declaration

- [5.3] If an object is initialised (in a type declaration statement or component definition statement), its array bounds and character length can be used in its initialisation expression.
- [7.0] The `EQUIVALENCE` and `COMMON` statements, and the `BLOCK DATA` program unit, are considered to be obsolescent (and reported as such when the *–f2018* option is used).
- [7.1] Assumed-rank dummy arguments accept actual arguments of any rank; they **assume** the rank from the actual argument. This rank may be zero; that is, the actual argument may be scalar. Furthermore, assumed-rank dummy arguments may have the `ALLOCATABLE` or `POINTER` attribute, and thus accept allocatable/pointer variables of any rank.

  The syntax is as follows:

```
Real,Dimension(..) :: a, b
Integer :: c(..)
```

  That declares three variables (which must be dummy arguments) to be assumed-rank.

  The use of assumed-rank dummy arguments within Fortran is extremely limited; basically, the intrinsic inquiry functions can be used, and there is a `SELECT RANK` construct, but other than that they may only appear as actual arguments to other procedures where they correspond to another assumed-rank argument.

  The main use of assumed rank is for advanced C interoperability (see later section).

  Here is an extremely simple example of use within Fortran:

```
Program assumed_rank_example
  Real x(1,2),y(3,4,5,6,7)
  Call showrank(1.5)
  Call showrank(x)
  Call showrank(y)
Contains
  Subroutine showrank(a)
    Real,Intent(In) :: a(..)
    Print *,'Rank is',Rank(a)
  End Subroutine
End Program
```

That will produce the output

```
Rank is 0
Rank is 2
Rank is 5
```

- [7.1] The `TYPE(*)` type specifier can be used to declare scalar, assumed-size, and assumed-rank dummy arguments. Such an argument is called **assumed-type**; the corresponding actual argument may be of any type. It must not have the `ALLOCATABLE`, `CODIMENSION`, `INTENT (OUT)`, `POINTER`, or `VALUE` attribute.

  An assumed-type variable is extremely limited in the ways it can be used directly in Fortran:

  – it may be passed as an actual argument to another assumed-type dummy argument;
  – it may appear as the first argument to the intrinsic functions `IS_CONTIGUOUS`, `LBOUND`, `PRESENT`, `SHAPE`, `SIZE`, or `UBOUND`;
  – it may be used as the argument of the function `C_LOC` (in the `ISO_C_BINDING` intrinsic module}.

  Other than these contexts, it cannot be used in any other way at all. Note that if it is an array, you cannot subscript it or create an array section from it.

  This is mostly useful for interoperating with C programs (see later section). Note that in a non-generic procedure reference, a scalar argument can be passed to an assumed-type argument that is an assumed-size array.

84 Data usage and computation

- [7.1] The `SELECT RANK` construct facilitates use of assumed rank objects within Fortran. It has the syntax

```
[ construct-name ] SELECT RANK ( [ assoc_name => ] assumed-rank-variable-name )
      [ rank-stmt
          block ]...
END SELECT [ construct-name ]
```

  where *rank-stmt* is one of:

```
RANK ( scalar-int-constant-expression ) [ construct-name ]
RANK ( * ) [ construct-name ]
RANK DEFAULT [ construct-name ]
```

  In any particular `SELECT RANK` construct, there must not be more than one `RANK DEFAULT` statement, or more than one `RANK (*)` statement, or more than `RANK (integer)` with the same value integer expression. If the assumed-rank variable has the `ALLOCATABLE` or `POINTER` attribute, the `RANK (*)` statement is not permitted.

  The *block* following a `RANK` statement with an integer constant expression is executed if the assumed-rank variable is associated with a non-assumed-rank actual argument that has that rank, and is not an assumed-size array. Within the *block* it acts as if it were an assumed-shape array with that rank.

  The *block* following a `RANK (*)` is executed if the ultimate argument is an assumed-size array. Within the *block* it acts as if it were declared with bounds '`(1:*)`'; if different bounds or rank are desired, this can be passed to another procedure using sequence association.

  The *block* following a `RANK DEFAULT` statement is executed if no other block is selected. Within its *block*, it is still an assumed-rank variable, i.e. there is no change.

  Here is a simple example of the `SELECT RANK` construct.

```
Program select_rank_example
  Integer :: a = 123, b(1,2) = Reshape( [ 10,20 ], [ 1,2 ] ), c(1,3,1) = 777, d(1,1,1,1,1)
  Call show(a)
  Call show(b)
  Call show(c)
  Call show(d)
Contains
```

```
  Subroutine show(x)
    Integer x(..)
    Select Rank(x)
    Rank (0)
      Print 1,'scalar',x
    Rank (1)
      Print 1,'vector',x
    Rank (2)
      Print 1,'matrix',x
    Rank (3)
      Print 1,'3D array',x
    Rank Default
      Print *,'Rank',Rank(x),'not supported'
    End Select
  1 Format(1x,a,*(1x,i0,:))
  End Subroutine
End Program
```

This will produce the output

```
scalar 123
matrix 10 20
3D array 777 777 777
Rank 5 not supported
```

85 Input/output

- [7.0] The `RECL=` specifier in an `INQUIRE` statement for an unconnected unit or file now assigns the value −1 to the variable. For a unit or file connected with `ACCESS='STREAM'`, it assigns the value −2 to the variable. Under previous Fortran standards, the variable became undefined.
- [7.1] The `SIZE=` specifier can be used in a `READ` statement without `ADVANCE='NO'`, that is, in a `READ` statement with no `ADVANCE=` specifier, or one with an explicit `ADVANCE='YES'`. For example,

```
Character(65536) buf
Integer nc
Read(*,'(A)',Size=nc) buf
Print *,'The number of characters on that line was',nc
```

Note that `SIZE=` is not permitted with list-directed or namelist formatting; that would be pointless, as there are no edit descriptors with such formatting and thus no characters to be counted by `SIZE=`.

86 Execution control

- [6.2] The expression in an `ERROR STOP` or `STOP` statement can be non-constant. It is still required to be default Integer or default Character.
- [6.2] 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.
```

87 Intrinsic procedures and modules

- [6.2] 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.

- [7.1] The DIM argument to the intrinsic functions ALL, ANY, FINDLOC, IALL, IANY, IPARITY, MAXLOC, MAXVAL, MINLOC, MINVAL, NORM2, PARITY, PRODUCT and SUM can be an optional dummy argument, as long as it is present at execution time. For example,

```
Subroutine sub(x,n)
  Real,Intent(In) :: x(:,:,:)
  Integer,Intent(In),Optional :: n
  If (Present(n)) Then
    Print *,Norm2(x,n) ! Rank two array result.
  Else
    Print *,Norm2(x)   ! Scalar result.
  End If
End Subroutine
```

- [7.1] Specific intrinsic functions are considered to be obsolescent (and reported as such with the *–f2018* option). In the case of a function that is both specific and generic, e.g. SQRT, the obsolescent usage is passing as an actual argument, use as a procedure interface, or being the target of a procedure pointer assignment.

- [7.1] The intrinsic inquiry function RANK returns the dimensionality of its argument. It has the following syntax:

```
RANK ( A )
```

  A : data object of any type:
  Result : scalar Integer of default kind.

  The result is the rank of A, that is, zero for scalar A, one if A is a one-dimensional array, and so on.

  This function can be used in a constant expression except when A is an assumed-rank variable.

- [7.1] The intrinsic function REDUCE performs user-defined array reductions. It has the following syntax:

```
REDUCE ( ARRAY, OPERATION [, MASK, IDENTITY, ORDERED ] ) or
REDUCE ( ARRAY, OPERATION DIM [, MASK, IDENTITY, ORDERED ] )
```

  ARRAY : array of any type;
  OPERATION : pure function with two arguments, each argument being scalar, non-allocatable, non-pointer, non-polymorphic non-optional variables with the same declared type and type parameters as ARRAY; if one argument has the ASYNCHRONOUS, TARGET or VALUE attribute, the other must also have that attribute; the result must be a non-polymorphic scalar variable with the same type and type parameters as ARRAY;
  DIM : scalar Integer in the range 1 to N, where N is the rank of ARRAY;

MASK : type Logical, and either scalar or an array with the same shape as ARRAY;

IDENTITY : scalar with the same declared type and type parameters as ARRAY;

ORDERED : scalar of type Logical;

Result : Same type and type parameters as ARRAY.

The result is ARRAY reduced by the user-supplied OPERATION. If DIM is absent, the whole (masked) ARRAY is reduced to a scalar result. If DIM is present, the result has rank N-1 and the shape of ARRAY with dimension DIM removed; each element of the result is the reduction of the masked elements in that dimension.

If exactly one element contributes to a result value, that value is equal to the element; that is, OPERATION is only invoked when more that one element appears.

If no elements contribute to a result value, the IDENTITY argument must be present, and that value is equal to IDENTITY.

For example,

```
Module triplet_m
  Type triplet
    Integer i,j,k
  End Type
Contains
  Pure Type(triplet) Function tadd(a,b)
    Type(triplet),Intent(In) :: a,b
    tadd%i = a%i + b%i
    tadd%j = a%j + b%j
    tadd%k = a%k + b%k
  End Function
End Module
Program reduce_example
  Use triplet_m
  Type(triplet) a(2,3)
  a = Reshape( [ triplet(1,2,3),triplet(1,2,4), &
                 triplet(2,2,5),triplet(2,2,6), &
                 triplet(3,2,7),triplet(3,2,8) ], [ 2,3 ] )
  Print 1, Reduce(a,tadd)
  Print 1, Reduce(a,tadd,1)
  Print 1, Reduce(a,tadd,a%i/=2)
  Print 1, Reduce(Array=a,Dim=2,Operation=tadd)
  Print 1, Reduce(a, Mask=a%i/=2, Dim=1, Operation=tadd, Identity=triplet(0,0,0))
1 Format(1x,6('triplet(',I0,',',I0,',',I0,')',:,'; '))
End Program
```

This will produce the output:

```
 triplet(12,12,33)
 triplet(2,4,7); triplet(4,4,11); triplet(6,4,15)
 triplet(8,8,22)
 triplet(6,6,15); triplet(6,6,18)
 triplet(2,4,7); triplet(0,0,0); triplet(6,4,15)
```

- [7.0] The intrinsic **atomic subroutines** ATOMIC_ADD, ATOMIC_AND, ATOMIC_CAS, ATOMIC_FETCH_ADD, ATOMIC_FETCH_AND, ATOMIC_FETCH_OR, ATOMIC_FETCH_XOR, ATOMIC_OR and ATOMIC_XOR are described under **Advanced coarray programming**.

- [7.1] The intrinsic **collective subroutines** CO_BROADCAST, CO_MAX, CO_MIN, CO_REDUCE and CO_SUM are described under **Advanced coarray programming**.

- [7.0] The intrinsic functions COSHAPE, EVENT_QUERY, FAILED_IMAGES, GET_TEAM, IMAGE_STATUS, STOPPED_IMAGES, and TEAM_NUMBER, and the changes to the intrinsic functions NUM_IMAGES and THIS_IMAGE, are described under **Advanced coarray programming**.

88 Program units and procedures

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

  For example,

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

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

  For example,

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

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

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

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

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

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

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

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

may be invoked with

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

to print the first five factorials.

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

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

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

- Generic resolution can use the number of procedure arguments; that is, if one procedure has more non-optional procedure arguments than the other has optional plus non-optional procedure arguments, the procedures are considered to be unambiguous.

For example,

```
MODULE npa_example
  INTERFACE g
    MODULE PROCEDURE s1,s2
  END INTERFACE
CONTAINS
  SUBROUTINE s1(a)
    EXTERNAL a
    CALL a
  END SUBROUTINE
  SUBROUTINE s2(b,a)
    EXTERNAL b,a
    CALL b
    CALL a
  END SUBROUTINE
END MODULE
```

This example does not conform to the Fortran 2008 rules for unambiguous generic procedures, because the argument A distinguishes by position but not by keyword, the argument B distinguish by keyword but not by position, and the positional disambiguator (A) does not appear earlier in the list than the keyword disambiguator (B).

89 Advanced C interoperability

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

```
USE ISO_C_BINDING
ABSTRACT INTERFACE
  SUBROUTINE my_callback_interface(arg)
    CLASS(*) arg
  END SUBROUTINE
END INTERFACE
TYPE,BIND(C) :: mycallback
```

```
  TYPE(C_FUNPTR) :: callback
END TYPE
...
TYPE(mycallback) cb
PROCEDURE(my_callback_interface),EXTERNAL :: sub
cb%callback = C_FUNLOC(sub)
...
PROCEDURE(my_callback_interface),POINTER :: pp
CALL C_F_PROCPOINTER(cb%callback,pp)
CALL pp(...)
```

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

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

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

- [7.1] Assumed-rank variables are permitted to be dummy arguments of a `BIND(C)` routine, even those with the `ALLOCATABLE` or `POINTER` attribute. An assumed-rank argument is passed by reference as a "C descriptor"; it is then up to the C routine to decode what that means. The C descriptor, along with several utility functions for manipulating it, is defined by the source file `ISO_Fortran_binding.h`; this can be found in the compiler's library directory (on Linux this is usually `/usr/local/lib/NAG_Fortran`, but that can be changed at installation time).

  This topic is highly complex, and beyond the scope of this document. The reader should direct their attention to the Fortran 2018 standard, or to a good textbook.

- [7.1] A `TYPE(*)` ("assumed type") dummy argument is permitted in a `BIND(C)` procedure. It interoperates with a C argument declared as "`void *`". There is no difference between scalar and assumed-size on the C side, but on the Fortran side, if the dummy argument is scalar the actual argument must also be scalar, and if the dummy argument is an array, the actual argument must also be an array.

  Because an actual argument can be passed directly to a `TYPE(*)` dummy, the `C_LOC` function is not required, and so there is no need for the `TARGET` attribute on the actual argument.

  For example,

```
Program type_star_example
  Interface
    Function checksum(scalar,size) Bind(C)
      Use Iso_C_Binding
      Type(*) scalar
      Integer(C_int),Value :: size
      Integer(C_int) checksum
    End Function
  End Interface
  Type myvec3
    Double Precision v(3)
  End Type
  Type(myvec3) x
  Call Random_Number(x%v)
  Print *,checksum(x,Storage_Size(x)/8)
End Program
int checksum(void *a,int n)
{
  int i;
  int res = 0;
  unsigned char *p = a;
  for (i=0; i<n; i++) res = 0x3fffffff&((res<<1) + p[i]);
```

```
  return res;
}
```

- A BIND(C) procedure can have optional arguments. Such arguments cannot also have the VALUE attribute.

  An absent optional argument of a BIND(C) procedure is indicated by passing a null pointer argument.

  For example,

```
Program optional_example
  Use Iso_C_Binding
  Interface
    Function f(a,b) Bind(C)
      Import
      Integer(C_int),Intent(In) :: a
      Integer(C_int),Intent(In),Optional :: b
      Integer(C_int) f
    End Function
  End Interface
  Integer(C_int) x,y
  x = f(3,14)
  y = f(23)
  Print *,x,y
End Program

int f(int *arg1,int *arg2)
{
  int res = *arg1;
  if (arg2) res += *arg2;
  return res;
}
```

  The second reference to f is missing the optional argument b, so a null pointer will be passed for it. This will result in the output:

```
 17 23
```

90 Updated IEEE arithmetic capabilities

- [7.0] The module IEEE_ARITHMETIC has new functions IEEE_NEXT_DOWN and IEEE_NEXT_UP. These are elemental with a single argument, which must be a REAL of an IEEE kind (that is, IEEE_SUPPORT_DATATYPE must return .TRUE. for that kind of REAL). They return the next IEEE value, that does not compare equal to the argument, in the downwards and upwards directions respectively, except that the next down from $-\infty$ is $-\infty$ itself, and the next up from $+\infty$ is $+\infty$ itself. These functions are superior to the old IEEE_NEXT_AFTER function in that they do not signal any exception unless the argument is a signalling NaN (in which case IEEE_INVALID is signalled).

  For example, IEEE_NEXT_UP(-0.0) and IEEE_NEXT_UP(+0.0) both return the smallest positive subnormal value (provided subnormal values are supported), without signalling IEEE_UNDERFLOW (which IEEE_NEXT_AFTER does).

  Similarly, IEEE_NEXT_UP(HUGE(0.0)) returns $+\infty$ without signalling overflow.

- [7.0] The module IEEE_ARITHMETIC has new named constants IEEE_NEGATIVE_SUBNORMAL, IEEE_POSITIVE_SUBNORMAL, and the new function IEEE_SUPPORT_SUBNORMAL. These are from Fortran 2018, and reflect the change of terminology in the IEEE arithmetic standard in 2008. They are equivalent to the old functions IEEE_NEGATIVE_DENORMAL, IEEE_POSITIVE_DENORMAL and IEEE_SUPPORT_DENORMAL.

- [7.0] The requirement that the FLAG_VALUE argument to IEEE_GET_FLAG and IEEE_SET_FLAG, the HALTING argument to IEEE_GET_HALTING_MODE and IEEE_SET_HALTING_MODE, and the GRADUAL argument to IEEE_GET_UNDERFLOW_MODE and IEEE_SET_UNDERFLOW_MODE, be default LOGICAL has been dropped; any kind of LOGICAL is now permitted.

  For example,

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

will retrieve the current IEEE flags into an array of one-byte LOGICALs.

91 Advanced coarray programming

- [7.0] Additional intrinsic atomic subroutines provide a means for multiple images to update atomic variables without synchronisation. These are:

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

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

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

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

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

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

- [7.0] The intrinsic function COSHAPE returns a vector of the co-extents of a coarray; its syntax is as follows.

  COSHAPE(COARRAY *[*, KIND *]*)

  > COARRAY : coarray of any type; if it is ALLOCATABLE, it must be allocated; if it is a structure component, the rightmost component must be a coarray component;
  > KIND (optional) : scalar Integer constant expression that is a valid Integer kind number;
  > Result : vector of type Integer, or Integer(KIND)} if KIND is present; the size of the result is equal to the co-rank of COARRAY.

  For example, if a coarray is declared

  ```
  REAL x[5,*]
  ```

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

- [7.0] The intrinsic elemental function IMAGE_STATUS enquires whether another image has stopped or failed; its syntax is as follows.

  IMAGE_STATUS(IMAGE *[*, TEAM *]*)

  > IMAGE : positive integer that is a valid image number;
  > TEAM (optional) : scalar TEAM_TYPE value that identifies the current or an ancestor team;
  > Result : default Integer.

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

- [7.0] The intrinsic function STOPPED_IMAGES returns an array listing the images that have initiated normal termination (i.e. "stopped"); its syntax is as follows.

  STOPPED_IMAGES(*[* TEAM, KIND *]*)

  > TEAM (optional) : scalar TEAM_TYPE value that identifies the current or an ancestor team;
  > KIND (optional) : scalar INTEGER constant expression that is a valid Integer kind number;
  > Result : vector of type Integer, or Integer(KIND) if KIND is present.

  The elements of the result are the stopped image numbers in ascending order.

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

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

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

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

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

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

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

  EVENT WAIT (*event-variable [, event-wait-spec-list]*)

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

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

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

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

  EVENT_QUERY (EVENT, COUNT *[*, STAT *]*)

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

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

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

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

  Creating teams

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

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

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

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

  For example,

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

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

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

  Changing to a subteam

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

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

  where *team-value* is a value of type TEAM_TYPE, and the optional *sync-stat-list* is a comma-separated list containing at most one STAT= and ERRMSG= specifier; the STAT= and ERRMSG= specifiers have their usual form and semantics. Execution of the *statements* within the construct are with the current team set to the team

identified by *team-value*; this must be a subteam of the current team outside the construct. The setting of the current team remains in effect during procedure calls, so any procedure referenced by the construct will also be executed with the new team current.

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

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

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

Synchronising parent or ancestor teams

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

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

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

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

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

Team-related intrinsic functions

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

  ```
  GET_TEAM( [ LEVEL ] )
  ```

  where the optional `LEVEL` argument is a scalar integer value that is equal to one of the named constants `CURRENT_TEAM`, `INITIAL_TEAM` or `PARENT_TEAM`, in the intrinsic module `ISO_FORTRAN_ENV`. This argument specifies which team the returned `TEAM_TYPE` value should identify; if it is absent, the value for the current team is returned. If the current team is the initial team, the `LEVEL` argument must not be equal to `PARENT_TEAM`, as the initial team has no parent.
- The intrinsic function `TEAM_NUMBER` returns a team number value (that was used in `FORM TEAM` by the executing image). It has the form

  ```
  TEAM_NUMBER( [ TEAM ] )
  ```

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

Information about sibling and ancestor teams

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

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

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

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

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

Establishing coarrays

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

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

Allocating and deallocating coarrays in teams

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

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

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

Accessing coarrays in sibling teams

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

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

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

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

Accessing coarrays in ancestor teams

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

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

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

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

Coarray association in `CHANGE TEAM`

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

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

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

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

and *explicit-coshape-spec* is

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

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

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

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

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

  The form of the `FAIL IMAGE` statement is simply

  ```
  FAIL IMAGE
  ```

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

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

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

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

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

  The intrinsic function `FAILED_IMAGES` returns an array of images that are known to have failed (it is possible that an image might fail and no other image realise until it tries to synchronise with it). Its syntax is as follows.

  `FAILED_IMAGES( [ TEAM, KIND ] )`

  `TEAM` (optional) : scalar `TEAM_TYPE` value that identifies the current or an ancestor team;
  `KIND` (optional) : scalar Integer constant expression that is a valid Integer kind number;
  Result : vector of type Integer, or Integer(`KIND`) if `KIND` is present.

  The elements of the result are the failed image numbers in ascending order.

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

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

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

  The fault tolerance features are in principle intended to permit recovery from hardware failure, with the `FAIL IMAGE` statement allowing some testing of recovery scenarios. The NAG Fortran Compiler does not support recovery from hardware failure (at Release 7.1).

- [7.1] The intrinsic subroutines `CO_BROADCAST`, `CO_MAX`, `CO_MIN`, `CO_REDUCE` and `CO_SUM` perform **collective** operations. These are for coarray parallelism: they compute values across all images in the current team, without explicit synchronisation.

  All of these subroutines have optional `STAT` and `ERRMSG` arguments. On successful execution, the `STAT` argument is assigned the value zero and the `ERRMSG` argument is left unchanged. If an error occurs, a positive value is

assigned to STAT and an explanatory message is assigned to ERRMSG. Only the errors STAT_FAILED_IMAGE and STAT_STOPPED_IMAGE are likely to be able to be caught in this way. Because there is not full synchronisation (see below), different images may receive different errors, or none at all. If an error occurs and STAT is not present, execution is terminated. Note that if the actual arguments for STAT or ERRMSG are optional dummy arguments, they must be present on all images or absent on all images.

A reference (CALL) to one of these subroutines is **not** an image control statement, does not end the current segment, and does not imply synchronisation (though some partial synchronisation will occur during the computation). However, such calls are only permitted where an image control statement is permitted.

Each image in a team must execute the same sequence of CALL statements to collective subroutines as the other images in the team. There must be no synchronisation between the images at the time of the call; the invocations must come from unordered segments.

All collective subroutines have the first argument "A", which is INTENT(INOUT), and must not be a coindexed object. This argument contains the data for the calculation, and must have the same type, type parameters, and shape on all images in the current team. If it is a coarray that is a dummy argument, it must have the same ultimate argument on all images.

SUBROUTINE CO_BROADCAST (A, SOURCE_IMAGE *[*, STAT, ERRMSG *]*)

> A : variable of any type; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if A is a coarray that is a dummy argument, it must have the same ultimate argument on each image;
>
> SOURCE_IMAGE : integer scalar, in the range one to NUM_IMAGES(), this argument must have the same value on all images in the current team;
>
> STAT (optional) : integer scalar variable, not coindexed;
>
> ERRMSG (optional) : character scalar variable of default kind, not coindexed.
>
> The value of argument A on image SOURCE_IMAGE is assigned to the argument A on all the other images.

SUBROUTINE CO_MAX (A *[*, RESULT_IMAGE, STAT, ERRMSG *]*)

> A: variable of type Integer, Real, or Character; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if A is a coarray that is a dummy argument, it must have the same ultimate argument on each image;
>
> RESULT_IMAGE (optional) : integer scalar, in the range one to NUM_IMAGES(), this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;
>
> STAT (optional) : integer scalar variable, not coindexed;
>
> ERRMSG (optional) : character scalar variable of default kind, not coindexed.
>
> This subroutine computes the maximum value of A across all images; if A is an array, the value is computed elementally. If RESULT_IMAGE is present, the result is assigned to argument A on that image, otherwise it is assigned to argument A on all images.

SUBROUTINE CO_MIN (A *[*, RESULT_IMAGE, STAT, ERRMSG *]*)

> A: variable of type Integer, Real, or Character; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if A is a coarray that is a dummy argument, it must have the same ultimate argument on each image;
>
> RESULT_IMAGE (optional) : integer scalar, in the range one to NUM_IMAGES(), this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;
>
> STAT (optional) : integer scalar variable, not coindexed;
>
> ERRMSG (optional) : character scalar variable of default kind, not coindexed.
>
> This subroutine computes the minimum value of A across all images; if A is an array, the value is computed elementally. If RESULT_IMAGE is present, the result is assigned to argument A on that image, otherwise it is assigned to argument A on all images.

SUBROUTINE CO_REDUCE (A, OPERATION *[*, RESULT_IMAGE, STAT, ERRMSG *]*)

> A: non-polymorphic variable of any type; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if A is a coarray that is a dummy argument, it must have the same ultimate argument on each image;

`OPERATION` : pure function with exactly two arguments; the dummy arguments of `OPERATION` must be non-allocatable, non-optional, non-pointer, non-polymorphic dummy variables, and each argument and the result of the function must be scalar with the same type and type parameters as `A`;

`RESULT_IMAGE` (optional) : integer scalar, in the range one to `NUM_IMAGES()`, this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;

`STAT` (optional) : integer scalar variable, not coindexed;

`ERRMSG` (optional) : character scalar variable of default kind, not coindexed.

This subroutine computes an arbitrary reduction of `A` across all images; if `A` is an array, the value is computed elementally. The reduction is computed starting with the set of corresponding values of `A` on all images; this is an iterative process, taking two values from the set and converting them to a single value by applying the `OPERATION` function; the process continues until the set contains only a single value — that value is the result. If `RESULT_IMAGE` is present, the result is assigned to argument `A` on that image, otherwise it is assigned to argument `A` on all images.

`SUBROUTINE CO_SUM ( A [, RESULT_IMAGE, STAT, ERRMSG ] )`

`A`: variable of type Integer, Real, or Complex; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if `A` is a coarray that is a dummy argument, it must have the same ultimate argument on each image;

`RESULT_IMAGE` (optional) : integer scalar, in the range one to `NUM_IMAGES()`, this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;

`STAT` (optional) : integer scalar variable, not coindexed;

`ERRMSG` (optional) : character scalar variable of default kind, not coindexed.

This subroutine computes the sum of `A` across all images; if `A` is an array, the value is computed elementally. If `RESULT_IMAGE` is present, the result is assigned to argument `A` on that image, otherwise it is assigned to argument `A` on all images.

Appendices

92 Mixing Fortran and C

When mixing Fortran source code with C source code, without using the C interoperability features of Fortran 2003, the following points should be noted.

92.1 Naming Conventions

External procedure names and common block names are in lower case with an underscore appended (this is the same as the standard UNIX f77). The main program-unit is called '`main`'.

Module variables and module procedure names are formed by taking the module name (in lower case), appending '`_MP_`' and then appending the entity's name (in lower case).

These conventions differ when the *−compatible* option is used. On Windows this causes external procedures and common blocks to be in upper case with no trailing underscore, while module entities have the module name in upper case instead of lower case (64-bit Windows is always *−compatible*). On most other platforms this option adds an extra underscore (after the one that is always added) for names that already had an underscore.

92.2 Initialisation and Termination

If the main program is not written in Fortran then either the initialisation routine

```
__NAGf90_rts_init(int argc,char*argv[])
```

or the i/o initialisation routine

```
__NAGf90_io_init(void)
```

should be called from C. The `__NAGf90_rts_init` routine initialises the Fortran floating-point environment as well as allowing command-line arguments to be accessed via the Fortran 2003 intrinsic functions and also via the `F90_UNIX` module.

Additionally, the program should be terminated with `__NAGf90_finish(int status)`, or alternatively `__NAGf90_io_finish(void)` may be called before the usual C termination routine to close all Fortran files and flush the Fortran i/o buffers (failing to do this might corrupt an output file that is still open).

92.3 Calling Conventions

The following sections describe in detail the calling conventions used by Fortran programs compiled with the NAG Fortran Compiler in C terms. This information is thus mostly useful to those wishing to mix Fortran and C code.

The conventions used for the Fortran 77 subset of Modern Fortran are compatible with the de facto UNIX f77 conventions (except for `COMPLEX` functions compiled without the *−compatible* option).

92.4 Data Types

Definitions of data types useful in communicating from C to Fortran are in the files `dope.h` and `nagfortran.h`, which are located in the compiler library directory (usually `/usr/local/lib/NAG_Fortran` on UNIX-like systems).

Fortran Data Type	Fortran Precision	C `typedef` name
INTEGER	8 bits	Integer1
	16 bits	Integer2
	32 bits	Integer *or* Integer3
	64 bits	Integer4
LOGICAL	8 bits	Logical1
	16 bits	Logical2
	32 bits	Logical
	64 bits	Logical4
REAL	half	__NAGf90_HReal
	single	Real
	double	Double
	double-double	DDReal
	quadruple	QReal
COMPLEX	half	__NAGf90_HComplex
	single	Complex
	double	DComplex
	double-double	DDComplex
	quadruple	QComplex

Note that DDReal and QReal are the same on most systems; on Sun Solaris these are different (the latter being an IEEE-conformant 128-bit floating-point type).

92.4.1 Pointers

Scalar non-polymorphic non-CHARACTER `POINTER` types are simply C pointers to the object.

An array `POINTER` is a dope vector describing the array pointed to. Unlike a simple address, these dope vectors are capable of directly describing non-contiguous array sections of contiguous arrays. See below (`Dope`N and `ChDope`N) for further details. Polymorphic dope vectors are `NPDope`N except for `CLASS(*)` pointers which are `CSDope`N.

92.4.2 Derived types

Fortran derived types are translated into C `struct`s. If `BIND(C)` or `SEQUENCE` is used, the order of the items within the struct is the same as the order within the derived type definition. Otherwise, the order of items is permuted to put larger types at the front of the struct so as to improve alignment and reduce storage; the C output code can be inspected to determine the ordering used.

92.4.3 Supporting types

`Char` `unsigned char`
Data type for default (single-byte) character storage.

`Char2` `unsigned short`
Data type for 16-bit (JIS and UCS-2) character storage.

`Char4` `unsigned int`
Data type for 32-bit (UCS-4) character storage.

`Substring`
`struct { Char *addr; Chrlen len;}`
Describes a single-byte (default) `CHARACTER` string; used for deferred-length default `CHARACTER` variables and as the return type of variable-length scalar non-`POINTER` default `CHARACTER` functions and all `POINTER` default `CHARACTER` functions.

`Substring2`
: `struct { Char2 *addr; Chrlen len;}`
Describes a 16-bit (JIS or UCS-2) `CHARACTER` string; used for deferred-length 16-bit `CHARACTER` variables and as the return type of variable-length scalar non-`POINTER` 16-bit `CHARACTER` functions and all `POINTER` 16-bit `CHARACTER` functions.

`Substring4`
: `struct { Char4 *addr; Chrlen len;}`
Describes a 32-bit ISO_10646 (UCS-4) `CHARACTER` string; used for deferred-length 32-bit `CHARACTER` variables and as the return type of variable-length scalar non-`POINTER` UCS-4 `CHARACTER` functions and all `POINTER` UCS-4 `CHARACTER` functions.

`Offset` `int`, `long` or `long long`
: An integer type for addressing and subscript calculations; this is `int` (32-bit) on 32-bit systems and small model 64-bit systems, and a 64-bit integer type on large model 64-bit systems.

`Chrlen` usually `int`, or `long long` on 64-bit Windows.
: An integer type for representing character length.

`Pointer` usually `char *`.
: A byte pointer used to refer to any type and for pointer arithmetic.

`Triplet` `struct { Offset extent,mult,lower;}`
: Contains the parameters of an array dimension. `extent` is the size of that dimension, `mult` is the stride (i.e., the distance between successive elements in bytes) and `lower` is the lower bound. It is a component of the `Dope`N and `ChDope`N structs.

`Dope`***N*** `struct { Pointer addr; Offset offset; Triplet dim[`N`];}`
: Dope vectors for all non-polymorphic non-`CHARACTER` arrays (including arrays of derived type). N is the rank and is from 1 to 7. `addr` is the address of the first element, `dim` describes each dimension and `offset` is the offset to add to the subscript calculation, i.e., `SUM(mult*lower)`. This is used as the return type for `POINTER` array functions; a pointer to it is used as the argument type for assumed-shape and `POINTER` array arguments.

  An array pointer which has been nullified has an `addr` field which is a null pointer; note that zero-sized arrays have an `addr` field which is not a null pointer.

`ChDope`***N*** `struct { Pointer addr; Chrlen len; Offset offset; Triplet dim[`N`];}`
: These are exactly the same as the `Dope`N structs with the addition of the `len` component which specifies the `CHARACTER` length.

`ArrayTemp_`*`type`*
: `struct { `*`type`*` *addr; Offset extent[7];}`
Describes a contiguous array of *type*, which is one of: `Integer1`, `Integer2`, `Integer`, `Integer4`, `Logical1`, `Logical2`, `Logical`, `Logical4`, `Real`, `Double`, `QReal`, `Complex`, `DComplex` or `QComplex`. It is used as the return type for non-`POINTER` array functions. Note that `extent` values after the rank of the array being described are undefined.

`ArrayTemp_Character`
: `struct { Char *addr; Chrlen len; Offset extent[7];}`
Describes a contiguous `CHARACTER` array; it is the same as the other array types with the addition of the `len` component for the `CHARACTER` length.

`ArrayTemp_Derived`
: synonym for `ArrayTemp_Character`.
Describes a contiguous array of any derived type. The `len` field in this case is the size of the derived type array element.

92.5 SUBROUTINE return types

92.5.1 SUBROUTINEs with label arguments

The return type is `int`; its value is the index of the label to which control is to be transferred (1 for the first label, etc.). Zero or an out-of-range value indicates no control transfer is to take place.

92.5.2 SUBROUTINEs with no label arguments

Return type is `void`.

92.6 FUNCTION return types

92.6.1 Scalar

1. `INTEGER`, `LOGICAL` and `REAL`.
   The intrinsic type as listed above.

2. `COMPLEX`, *−compatible* option not used, and not Windows 64-bit mode.
   `Complex` or `DComplex` according to the precision.

3. `COMPLEX`, *-compatible* option used, or Windows 64-bit mode.
   Return type is `void`. The address of a temporary of type `Complex` or `DComplex` is passed as the initial argument of the function (the result is written to this location).

4. `CHARACTER` with fixed or assumed length.
   Return type is `void`. Two additional initial arguments are passed to the function, the first (`Char*`, `Char2*` or `Char4*`) being the address to which the result is to be written and the second (`Chrlen`) being the length of the result (in case the called function is of assumed length).

5. `CHARACTER` with variable length.
   Return type is `Substring`, `Substring2` or `Substring4` (described above). The called function allocates the storage for the string being returned; it is the caller's responsibility to deallocate it when it is no longer required.

6. Derived type.
   The derived-type struct.

92.6.2 Scalar `POINTER` functions

Note that with all `POINTER`-valued functions the storage to which the result points may have been allocated within the called function or the result may point to pre-existing storage.

1. `INTEGER`, `LOGICAL`, `REAL` and `COMPLEX`.
   A pointer to the appropriate intrinsic type (e.g., `Complex*`).

2. `CHARACTER`
   Return type is `Substring`, `Substring2` or `Substring4`.

3. Derived type.
   A pointer to the derived-type struct.

92.6.3 Array non-`POINTER` functions

1. Intrinsic types.
   The appropriate `ArrayTemp_` struct for the intrinsic type, as described above.

2. Derived types.
   `ArrayTemp_Derived` is returned with the `len` component set to the size of the derived-type struct.

92.6.4 Array `POINTER` functions

1. `CLASS(*)`
   `CSDope1`, `CSDope2`, ... or `CSDope7`, depending on the rank of the array.

2. `CLASS(`*derived-type-name*`)` `NPDope1`, `NPDope2`, ... or `NPDope7`, depending on the rank of the array.

3. Non-polymorphic non-CHARACTER type.
   Dope1, Dope2, ... or Dope7, depending on the rank of the array.
4. CHARACTER.
   ChDope1, ChDope2, ... or ChDope7, depending on the rank of the array.

Note that non-polymorphic derived-type arrays are returned as Dope$N$ structs.

92.7 Argument types

92.7.1 CHARACTER type

All normal arguments of CHARACTER type, whether default CHARACTER or multi-byte CHARACTER, have an additional Chrlen argument being the length of the CHARACTER entity; this additional argument is passed at the end of the argument list after all the normal arguments. When there are several CHARACTER arguments their lengths are passed in order from left to right.

This is except on 32-bit Windows when the *−compatible* option is specified, the additional argument immediately follows the CHARACTER argument.

The other exception to this rule is for assumed-shape CHARACTER arrays; in this case the length of the dummy argument is taken directly from the field in the dope vector and is not passed separately.

92.7.2 non-POINTER non-ALLOCATABLE Scalar

1. non-CHARACTER type.
   The address of the argument is passed (e.g., Integer* for an INTEGER argument).
2. CHARACTER.
   The address of the argument is passed, and additionally the length of the argument is passed as a separate Chrlen argument at the end of the argument list.

92.7.3 POINTER and ALLOCATABLE Scalar

These are passed exactly the same as the normal case except that the address of the pointer (or allocatable descriptor) is passed (e.g., Integer** for an INTEGER POINTER).

92.7.4 non-POINTER Array

1. Assumed shape.
   The address of an appropriate dope vector describing the array is passed (i.e., a CSDope$N$*, NPDope$N$*, Dope$N$* or a ChDope$N$*, depending on the polymorphism and type of the dummy argument). There is no need for the array to be contiguous as long as it can be described by an array section.
2. Other.
   The address of the first element of the array is passed. For CHARACTER arrays the length of each array element is passed as an Chrlen at the end of the argument list, the same as for scalars. The array must be contiguous.

92.7.5 POINTER Array

The address of an appropriate dope vector is passed, the same as for assumed-shape arrays. For CHARACTER arrays the length is passed at the end as a separate argument.

92.7.6 Procedures

The address of the procedure is passed. CHARACTER functions have the length passed as a separate argument at the end of the list; if the function is of variable length this length will be negative.

92.7.7 `OPTIONAL` arguments

If an `OPTIONAL` argument is `.NOT.PRESENT()` a null pointer of the appropriate type is passed (e.g., for an `INTEGER` scalar, an `(Integer*)0` is passed).

93 ASCII Collating Sequence

93.1 Printing Characters

Decimal value and character. The value 32 is a space.

	30	40	50	60	70	80	90	100	110	120
0		(	2	<	F	P	Z	d	n	x
1		)	3	=	G	Q	[	e	o	y
2		*	4	>	H	R	\	f	p	z
3	!	+	5	?	I	S	]	g	q	{
4	"	,	6	@	J	T	^	h	r	|
5	#	-	7	A	K	U	_	i	s	}
6	$	.	8	B	L	V	‘	j	t	~
7	%	/	9	C	M	W	a	k	u	
8	&	0	:	D	N	X	b	l	v	
9	’	1	;	E	O	Y	c	m	w	

Octal value and character. The value 40 is a space.

	40	50	60	70	100	110	120	130	140	150	160	170
0		(	0	8	@	H	P	X	’	h	p	x
1	!	)	1	9	A	I	Q	Y	a	i	q	y
2	"	*	2	:	B	J	R	Z	b	j	r	z
3	#	+	3	;	C	K	S	[	c	k	s	{
4	$	,	4	<	D	L	T	\	d	l	t	|
5	%	-	5	=	E	M	U	]	e	m	u	}
6	&	.	6	>	F	N	V	^	f	n	v	~
7	’	/	7	?	G	O	W	_	g	o	w	

93.2 Non-printing Characters

Decimal	Octal	Mnemonic	Control	Description
0	000	NUL	ctrl/@	Null character
1	001	SOH	ctrl/A	Start of heading
2	002	STX	ctrl/B	Start of text
3	003	ETX	ctrl/C	End of text
4	004	EOT	ctrl/D	End of transmission
5	005	ENQ	ctrl/E	Enquire
6	006	ACK	ctrl/F	Acknowledge
7	007	BEL	ctrl/G	Bell
8	010	BS	ctrl/H	Backspace
9	011	HT	ctrl/I	Horizontal tab
10	012	LF	ctrl/J	Line feed
11	013	VT	ctrl/K	Vertical tab
12	014	FF	ctrl/L	Form feed
13	015	CR	ctrl/M	Carriage return
14	016	SO	ctrl/N	Shift out
15	017	SI	ctrl/O	Shift in
16	020	DLE	ctrl/P	Data link escape
17	021	DC1	ctrl/Q	Device control 1 (XON)
18	022	DC2	ctrl/R	Device control 2
19	023	DC3	ctrl/S	Device control 3 (XOFF)
20	024	DC4	ctrl/T	Device control 4
21	025	NAK	ctrl/U	Negative acknowledge
22	026	SYN	ctrl/V	Synchronise
23	027	ETB	ctrl/W	End transmission of block
24	030	CAN	ctrl/X	Cancel
25	031	EM	ctrl/Y	End of medium
26	032	EOF	ctrl/Z	End-of-file
27	033	ESC	ctrl/[	Escape
28	034	FS	ctrl/\	File separator
29	035	GS	ctrl/]	Group separator
30	036	RS	ctrl/^	Record separator
31	037	US	ctrl/_	Unit separator
127	177	DEL		Delete

Detailed Contents

Contents