Using GNU Fortran

For GCC version 4.3.0 (pre-release)

The gfortran team

Published by the Free Software Foundation
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301, USA

Short Contents

Table of Contents

1 Introduction

This manual documents the use of `gfortran`, the GNU Fortran compiler. You can find in this manual how to invoke `gfortran`, as well as its features and incompatibilities.

Warning: This document, and the compiler it describes, are still under development. While efforts are made to keep it up-to-date, it might not accurately reflect the status of the most recent GNU Fortran compiler.

The GNU Fortran compiler front end was designed initially as a free replacement for, or alternative to, the unix `f95` command; `gfortran` is the command you'll use to invoke the compiler.

1.1 About GNU Fortran

The GNU Fortran compiler is still in an early state of development. It can generate code for most constructs and expressions, but much work remains to be done.

When the GNU Fortran compiler is finished, it will do everything you expect from any decent compiler:

- Read a user's program, stored in a file and containing instructions written in Fortran 77, Fortran 90, Fortran 95 or Fortran 2003. This file contains *source code*.
- Translate the user's program into instructions a computer can carry out more quickly than it takes to translate the instructions in the first place. The result after compilation of a program is *machine code*, code designed to be efficiently translated and processed by a machine such as your computer. Humans usually aren't as good writing machine code as they are at writing Fortran (or C++, Ada, or Java), because is easy to make tiny mistakes writing machine code.
- Provide the user with information about the reasons why the compiler is unable to create a binary from the source code. Usually this will be the case if the source code is flawed. When writing Fortran, it is easy to make big mistakes. The Fortran 90 requires that the compiler can point out mistakes to the user. An incorrect usage of the language causes an *error message*.

  The compiler will also attempt to diagnose cases where the user's program contains a correct usage of the language, but instructs the computer to do something questionable. This kind of diagnostics message is called a *warning message*.
- Provide optional information about the translation passes from the source code to machine code. This can help a user of the compiler to find the cause of certain bugs which may not be obvious in the source code, but may be more easily found at a lower level compiler output. It also helps developers to find bugs in the compiler itself.
- Provide information in the generated machine code that can make it easier to find bugs in the program (using a debugging tool, called a *debugger*, such as the GNU Debugger `gdb`).
- Locate and gather machine code already generated to perform actions requested by statements in the user's program. This machine code is organized into *modules* and is located and *linked* to the user program.

The GNU Fortran compiler consists of several components:

- A version of the `gcc` command (which also might be installed as the system's `cc` command) that also understands and accepts Fortran source code. The `gcc` command is the *driver* program for all the languages in the GNU Compiler Collection (GCC); With `gcc`, you can compile the source code of any language for which a front end is available in GCC.
- The `gfortran` command itself, which also might be installed as the system's `f95` command. `gfortran` is just another driver program, but specifically for the Fortran compiler only. The difference with `gcc` is that `gfortran` will automatically link the correct libraries to your program.
- A collection of run-time libraries. These libraries contain the machine code needed to support capabilities of the Fortran language that are not directly provided by the machine code generated by the `gfortran` compilation phase, such as intrinsic functions and subroutines, and routines for interaction with files and the operating system.
- The Fortran compiler itself, (`f951`). This is the GNU Fortran parser and code generator, linked to and interfaced with the GCC backend library. `f951` "translates" the source code to assembler code. You would typically not use this program directly; instead, the `gcc` or `gfortran` driver programs will call it for you.

1.2 GNU Fortran and GCC

GNU Fortran is a part of GCC, the *GNU Compiler Collection*. GCC consists of a collection of front ends for various languages, which translate the source code into a language-independent form called *GENERIC*. This is then processed by a common middle end which provides optimization, and then passed to one of a collection of back ends which generate code for different computer architectures and operating systems.

Functionally, this is implemented with a driver program (`gcc`) which provides the command-line interface for the compiler. It calls the relevant compiler front-end program (e.g., `f951` for Fortran) for each file in the source code, and then calls the assembler and linker as appropriate to produce the compiled output. In a copy of GCC which has been compiled with Fortran language support enabled, `gcc` will recognize files with '`.f`', '`.f90`', '`.f95`', and '`.f03`' extensions as Fortran source code, and compile it accordingly. A `gfortran` driver program is also provided, which is identical to `gcc` except that it automatically links the Fortran runtime libraries into the compiled program.

This manual specifically documents the Fortran front end, which handles the programming language's syntax and semantics. The aspects of GCC which relate to the optimization passes and the back-end code generation are documented in the GCC manual; see section "Introduction" in *Using the GNU Compiler Collection (GCC)*. The two manuals together provide a complete reference for the GNU Fortran compiler.

1.3 GNU Fortran and G77

The GNU Fortran compiler is the successor to G77, the Fortran 77 front end included in GCC prior to version 4. It is an entirely new program that has been designed to provide Fortran 95 support and extensibility for future Fortran language standards, as well as providing backwards compatibility for Fortran 77 and nearly all of the GNU language extensions supported by G77.

1.4 Project Status

> As soon as gfortran can parse all of the statements correctly, it will be in the "larva" state. When we generate code, the "puppa" state. When gfortran is done, we'll see if it will be a beautiful butterfly, or just a big bug....
>
> –Andy Vaught, April 2000

The start of the GNU Fortran 95 project was announced on the GCC homepage in March 18, 2000 (even though Andy had already been working on it for a while, of course).

The GNU Fortran compiler is able to compile nearly all standard-compliant Fortran 95, Fortran 90, and Fortran 77 programs, including a number of standard and non-standard extensions, and can be used on real-world programs. In particular, the supported extensions include OpenMP, Cray-style pointers, and several Fortran 2003 features such as enumeration, stream I/O, and some of the enhancements to allocatable array support from TR 15581. However, it is still under development and has a few remaining rough edges.

At present, the GNU Fortran compiler passes the NIST Fortran 77 Test Suite, and produces acceptable results on the LAPACK Test Suite. It also provides respectable performance on the Polyhedron Fortran compiler benchmarks and the Livermore Fortran Kernels test. It has been used to compile a number of large real-world programs, including the HIRLAM weather-forecasting code and the Tonto quantum chemistry package; see http://gcc.gnu.org/wiki/GfortranApps for an extended list.

Among other things, the GNU Fortran compiler is intended as a replacement for G77. At this point, nearly all programs that could be compiled with G77 can be compiled with GNU Fortran, although there are a few minor known regressions.

The primary work remaining to be done on GNU Fortran falls into three categories: bug fixing (primarily regarding the treatment of invalid code and providing useful error messages), improving the compiler optimizations and the performance of compiled code, and extending the compiler to support future standards—in particular, Fortran 2003.

1.5 Standards

The GNU Fortran compiler implements ISO/IEC 1539:1997 (Fortran 95). As such, it can also compile essentially all standard-compliant Fortran 90 and Fortran 77 programs. It also supports the ISO/IEC TR-15581 enhancements to allocatable arrays, and the OpenMP Application Program Interface v2.5 specification.

In the future, the GNU Fortran compiler may also support other standard variants of and extensions to the Fortran language. These include ISO/IEC 1539-1:2004 (Fortran 2003).

Part I: Invoking GNU Fortran

2 GNU Fortran Command Options

The `gfortran` command supports all the options supported by the `gcc` command. Only options specific to GNU Fortran are documented here.

See section "GCC Command Options" in *Using the GNU Compiler Collection (GCC)*, for information on the non-Fortran-specific aspects of the `gcc` command (and, therefore, the `gfortran` command).

All GCC and GNU Fortran options are accepted both by `gfortran` and by `gcc` (as well as any other drivers built at the same time, such as `g++`), since adding GNU Fortran to the GCC distribution enables acceptance of GNU Fortran options by all of the relevant drivers.

In some cases, options have positive and negative forms; the negative form of '`-ffoo`' would be '`-fno-foo`'. This manual documents only one of these two forms, whichever one is not the default.

2.1 Option Summary

Here is a summary of all the options specific to GNU Fortran, grouped by type. Explanations are in the following sections.

Fortran Language Options
: See Section 2.2 [Options Controlling Fortran Dialect], page 8.

```
-fall-intrinsics -ffree-form -fno-fixed-form
-fdollar-ok -fimplicit-none -fmax-identifier-length
-std=std -fd-lines-as-code -fd-lines-as-comments
-ffixed-line-length-n -ffixed-line-length-none
-ffree-line-length-n -ffree-line-length-none
-fdefault-double-8 -fdefault-integer-8 -fdefault-real-8
-fcray-pointer -fopenmp -frange-check -fno-backslash
```

Error and Warning Options
: See Section 2.3 [Options to Request or Suppress Errors and Warnings], page 9.

```
-fmax-errors=n
-fsyntax-only -pedantic -pedantic-errors
-w -Wall -Waliasing -Wampersand -Wconversion -Wimplicit-interface
-Wtabs -Wnonstd-intrinsics -Wsurprising -Wunderflow
-Wline-truncation -W
```

Debugging Options
: See Section 2.4 [Options for Debugging Your Program or GCC], page 11.

```
-fdump-parse-tree -ffpe-trap=list -fdump-core
```

Directory Options
: See Section 2.5 [Options for Directory Search], page 12.

```
-Idir -Mdir
```

Runtime Options
: See Section 2.6 [Options for influencing runtime behavior], page 12.

```
-fconvert=conversion -frecord-marker=length
```

Code Generation Options
: See Section 2.7 [Options for Code Generation Conventions], page 13.

```
-fno-automatic -ff2c -fno-underscoring -fsecond-underscore
-fbounds-check -fmax-stack-var-size=n
-fpack-derived -frepack-arrays -fshort-enums -fexternal-blas
-fblas-matmul-limit=n
```

2.2 Options Controlling Fortran Dialect

The following options control the details of the Fortran dialect accepted by the compiler:

`-ffree-form`
`-ffixed-form`
: Specify the layout used by the source file. The free form layout was introduced in Fortran 90. Fixed form was traditionally used in older Fortran programs. When neither option is specified, the source form is determined by the file extension.

`-fall-intrinsics`
: Accept all of the intrinsic procedures provided in libgfortran without regard to the setting of '`-std`'. In particular, this option can be quite useful with '`-std=f95`'. Additionally, `gfortran` will ignore '`-Wnonstd-intrinsics`'.

`-fd-lines-as-code`
`-fd-lines-as-comment`
: Enable special treatment for lines beginning with `d` or `D` in fixed form sources. If the '`-fd-lines-as-code`' option is given they are treated as if the first column contained a blank. If the '`-fd-lines-as-comments`' option is given, they are treated as comment lines.

`-fdefault-double-8`
: Set the `DOUBLE PRECISION` type to an 8 byte wide type.

`-fdefault-integer-8`
: Set the default integer and logical types to an 8 byte wide type. Do nothing if this is already the default.

`-fdefault-real-8`
: Set the default real type to an 8 byte wide type. Do nothing if this is already the default.

`-fdollar-ok`
: Allow '`$`' as a valid character in a symbol name.

`-fno-backslash`
: Change the interpretation of backslashes in string literals from "C-style" escape characters to a single backslash character.

`-ffixed-line-length-`*n*
: Set column after which characters are ignored in typical fixed-form lines in the source file, and through which spaces are assumed (as if padded to that length) after the ends of short fixed-form lines.

  Popular values for *n* include 72 (the standard and the default), 80 (card image), and 132 (corresponding to "extended-source" options in some popular compilers). *n* may also be '`none`', meaning that the entire line is meaningful and that continued character constants never have implicit spaces appended to them to fill out the line. '`-ffixed-line-length-0`' means the same thing as '`-ffixed-line-length-none`'.

`-ffree-line-length-n`
: Set column after which characters are ignored in typical free-form lines in the source file. The default value is 132. *n* may be '`none`', meaning that the entire line is meaningful. '`-ffree-line-length-0`' means the same thing as '`-ffree-line-length-none`'.

`-fmax-identifier-length=n`
: Specify the maximum allowed identifier length. Typical values are 31 (Fortran 95) and 63 (Fortran 2003).

`-fimplicit-none`
: Specify that no implicit typing is allowed, unless overridden by explicit `IMPLICIT` statements. This is the equivalent of adding `implicit none` to the start of every procedure.

`-fcray-pointer`
: Enable the Cray pointer extension, which provides C-like pointer functionality.

`-fopenmp`
: Enable the OpenMP extensions. This includes OpenMP `!$omp` directives in free form and `c$omp`, `*$omp` and `!$omp` directives in fixed form, `!$` conditional compilation sentinels in free form and `c$`, `*$` and `!$` sentinels in fixed form, and when linking arranges for the OpenMP runtime library to be linked in.

`-frange-check`
: Enable range checking on results of simplification of constant expressions during compilation. For example, by default, GNU Fortran will give an overflow error at compile time when simplifying `a = EXP(1000)`. With '`-fno-range-check`', no error will be given and the variable `a` will be assigned the value `+Infinity`. Similarly, `DATA i/Z'FFFFFFFF'/` will result in an integer overflow on most systems, but with '`-fno-range-check`' the value will "wrap around" and `i` will be initialized to -1 instead.

`-std=std`
: Conform to the specified standard. The default value for *std* is '`gnu`'; a superset of the Fortran 95 standard which includes all of the GNU extensions recommended for use in new code. The '`legacy`' value also includes obsolete extensions that may be required for old non-standard programs. Strict conformance to the Fortran 95 and Fortran 2003 standards is specified by '`f95`' and '`f2003`', respectively.

2.3 Options to Request or Suppress Errors and Warnings

Errors are diagnostic messages that report that the GNU Fortran compiler cannot compile the relevant piece of source code. The compiler will continue to process the program in an attempt to report further errors to aid in debugging, but will not produce any compiled output.

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there is likely to be a bug in the program. Unless '`-Werror`' is specified, they do not prevent compilation of the program.

You can request many specific warnings with options beginning '`-W`', for example '`-Wimplicit`' to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning '-Wno-' to turn off warnings; for example, '-Wno-implicit'. This manual lists only one of the two forms, whichever is not the default.

These options control the amount and kinds of errors and warnings produced by GNU Fortran:

-fmax-errors-*n*
: Limits the maximum number of error messages to *n*, at which point GNU Fortran bails out rather than attempting to continue processing the source code. If *n* is 0, there is no limit on the number of error messages produced.

-fsyntax-only
: Check the code for syntax errors, but don't do anything beyond that.

-pedantic
: Issue warnings for uses of extensions to Fortran 95. '-pedantic' also applies to C-language constructs where they occur in GNU Fortran source files, such as use of '\e' in a character constant within a directive like #include.

  Valid Fortran 95 programs should compile properly with or without this option. However, without this option, certain GNU extensions and traditional Fortran features are supported as well. With this option, many of them are rejected.

  Some users try to use '-pedantic' to check programs for conformance. They soon find that it does not do quite what they want—it finds some nonstandard practices, but not all. However, improvements to GNU Fortran in this area are welcome.

  This should be used in conjunction with '-std=f95' or '-std=f2003'.

-pedantic-errors
: Like '-pedantic', except that errors are produced rather than warnings.

-w
: Inhibit all warning messages.

-Wall
: Enables commonly used warning options pertaining to usage that we recommend avoiding and that we believe are easy to avoid. This currently includes '-Waliasing', '-Wampersand', '-Wsurprising', '-Wnonstd-intrinsic', '-Wno-tabs', and '-Wline-truncation'.

-Waliasing
: Warn about possible aliasing of dummy arguments. Specifically, it warns if the same actual argument is associated with a dummy argument with INTENT(IN) and a dummy argument with INTENT(OUT) in a call with an explicit interface. The following example will trigger the warning.

```
interface
  subroutine bar(a,b)
    integer, intent(in) :: a
    integer, intent(out) :: b
  end subroutine
end interface
integer :: a

call bar(a,a)
```

`-Wampersand`
: Warn about missing ampersand in continued character constants. The warning is given with '`-Wampersand`', '`-pedantic`', '`-std=f95`', and '`-std=f2003`'. Note: With no ampersand given in a continued character constant, GNU Fortran assumes continuation at the first non-comment, non-whitespace character after the ampersand that initiated the continuation.

`-Wconversion`
: Warn about implicit conversions between different types.

`-Wimplicit-interface`
: Warn if a procedure is called without an explicit interface. Note this only checks that an explicit interface is present. It does not check that the declared interfaces are consistent across program units.

`-Wnonstd-intrinsic`
: Warn if the user tries to use an intrinsic that does not belong to the standard the user has chosen via the -std option.

`-Wsurprising`
: Produce a warning when "suspicious" code constructs are encountered. While technically legal these usually indicate that an error has been made.

  This currently produces a warning under the following circumstances:

  - An INTEGER SELECT construct has a CASE that can never be matched as its lower value is greater than its upper value.
  - A LOGICAL SELECT construct has three CASE statements.

`-Wtabs`
: By default, tabs are accepted as whitespace, but tabs are not members of the Fortran Character Set. '`-Wno-tabs`' will cause a warning to be issued if a tab is encountered. Note, '`-Wno-tabs`' is active for '`-pedantic`', '`-std=f95`', '`-std=f2003`', and '`-Wall`'.

`-Wunderflow`
: Produce a warning when numerical constant expressions are encountered, which yield an UNDERFLOW during compilation.

`-Werror`
: Turns all warnings into errors.

`-W`
: Turns on "extra warnings" and, if optimization is specified via '`-O`', the '`-Wuninitialized`' option. (This might change in future versions of GNU Fortran.)

See section "Options to Request or Suppress Errors and Warnings" in *Using the GNU Compiler Collection (GCC)*, for information on more options offered by the GBE shared by `gfortran`, `gcc` and other GNU compilers.

Some of these have no effect when compiling programs written in Fortran.

2.4 Options for Debugging Your Program or GNU Fortran

GNU Fortran has various special options that are used for debugging either your program or the GNU Fortran compiler.

`-fdump-parse-tree`
: Output the internal parse tree before starting code generation. Only really useful for debugging the GNU Fortran compiler itself.

`-ffpe-trap=`*`list`*
: Specify a list of IEEE exceptions when a Floating Point Exception (FPE) should be raised. On most systems, this will result in a SIGFPE signal being sent and the program being interrupted, producing a core file useful for debugging. *list* is a (possibly empty) comma-separated list of the following IEEE exceptions: '`invalid`' (invalid floating point operation, such as `SQRT(-1.0)`), '`zero`' (division by zero), '`overflow`' (overflow in a floating point operation), '`underflow`' (underflow in a floating point operation), '`precision`' (loss of precision during operation) and '`denormal`' (operation produced a denormal value).

`-fdump-core`
: Request that a core-dump file is written to disk when a runtime error is encountered on systems that support core dumps. This option is only effective for the compilation of the Fortran main program.

See section "Options for Debugging Your Program or GCC" in *Using the GNU Compiler Collection (GCC)*, for more information on debugging options.

2.5 Options for Directory Search

These options affect how GNU Fortran searches for files specified by the `INCLUDE` directive and where it searches for previously compiled modules.

It also affects the search paths used by `cpp` when used to preprocess Fortran source.

`-I`*`dir`*
: These affect interpretation of the `INCLUDE` directive (as well as of the `#include` directive of the `cpp` preprocessor).

  Also note that the general behavior of '`-I`' and `INCLUDE` is pretty much the same as of '`-I`' with `#include` in the `cpp` preprocessor, with regard to looking for '`header.gcc`' files and other such things.

  This path is also used to search for '`.mod`' files when previously compiled modules are required by a `USE` statement.

  See section "Options for Directory Search" in *Using the GNU Compiler Collection (GCC)*, for information on the '`-I`' option.

`-M`*`dir`*

`-J`*`dir`*
: This option specifies where to put '`.mod`' files for compiled modules. It is also added to the list of directories to searched by an `USE` statement.

  The default is the current directory.

  '`-J`' is an alias for '`-M`' to avoid conflicts with existing GCC options.

2.6 Influencing runtime behavior

These options affect the runtime behavior of programs compiled with GNU Fortran.

`-fconvert=`*`conversion`*
: Specify the representation of data for unformatted files. Valid values for conversion are: '`native`', the default; '`swap`', swap between big- and little-endian; '`big-endian`', use big-endian representation for unformatted files; '`little-endian`', use little-endian representation for unformatted files.

  This option has an effect only when used in the main program. The `CONVERT` *specifier and the GFORTRAN_CONVERT_UNIT environment variable override the default specified by* '`-fconvert`'*.*

`-frecord-marker=`*`length`*
: Specify the length of record markers for unformatted files. Valid values for *length* are 4 and 8. Default is 4. *This is different from previous versions of gfortran*, which specified a default record marker length of 8 on most systems. If you want to read or write files compatible with earlier versions of gfortran, use '`-frecord-marker=8`'.

`-fmax-subrecord-length=`*`length`*
: Specify the maximum length for a subrecord. The maximum permitted value for length is 2147483639, which is also the default. Only really useful for use by the gfortran testsuite.

2.7 Options for Code Generation Conventions

These machine-independent options control the interface conventions used in code generation.

Most of them have both positive and negative forms; the negative form of '`-ffoo`' would be '`-fno-foo`'. In the table below, only one of the forms is listed—the one which is not the default. You can figure out the other form by either removing '`no-`' or adding it.

`-fno-automatic`
: Treat each program unit as if the `SAVE` statement was specified for every local variable and array referenced in it. Does not affect common blocks. (Some Fortran compilers provide this option under the name '`-static`'.)

`-ff2c`
: Generate code designed to be compatible with code generated by `g77` and `f2c`.

  The calling conventions used by `g77` (originally implemented in `f2c`) require functions that return type default `REAL` to actually return the C type `double`, and functions that return type `COMPLEX` to return the values via an extra argument in the calling sequence that points to where to store the return value. Under the default GNU calling conventions, such functions simply return their results as they would in GNU C—default `REAL` functions return the C type `float`, and `COMPLEX` functions return the GNU C type `complex`. Additionally, this option implies the '`-fsecond-underscore`' option, unless '`-fno-second-underscore`' is explicitly requested.

  This does not affect the generation of code that interfaces with the `libgfortran` library.

  Caution: It is not a good idea to mix Fortran code compiled with '`-ff2c`' with code compiled with the default '`-fno-f2c`' calling conventions as, calling

COMPLEX or default REAL functions between program parts which were compiled with different calling conventions will break at execution time.

Caution: This will break code which passes intrinsic functions of type default REAL or COMPLEX as actual arguments, as the library implementations use the '-fno-f2c' calling conventions.

-fno-underscoring

Do not transform names of entities specified in the Fortran source file by appending underscores to them.

With '-funderscoring' in effect, GNU Fortran appends one underscore to external names with no underscores. This is done to ensure compatibility with code produced by many UNIX Fortran compilers.

Caution: The default behavior of GNU Fortran is incompatible with f2c and g77, please use the '-ff2c' option if you want object files compiled with GNU Fortran to be compatible with object code created with these tools.

Use of '-fno-underscoring' is not recommended unless you are experimenting with issues such as integration of GNU Fortran into existing system environments (vis-a-vis existing libraries, tools, and so on).

For example, with '-funderscoring', and assuming other defaults like '-fcase-lower' and that j() and max_count() are external functions while my_var and lvar are local variables, a statement like

```
I = J() + MAX_COUNT (MY_VAR, LVAR)
```

is implemented as something akin to:

```
i = j_() + max_count__(&my_var__, &lvar);
```

With '-fno-underscoring', the same statement is implemented as:

```
i = j() + max_count(&my_var, &lvar);
```

Use of '-fno-underscoring' allows direct specification of user-defined names while debugging and when interfacing GNU Fortran code with other languages.

Note that just because the names match does *not* mean that the interface implemented by GNU Fortran for an external name matches the interface implemented by some other language for that same name. That is, getting code produced by GNU Fortran to link to code produced by some other compiler using this or any other method can be only a small part of the overall solution—getting the code generated by both compilers to agree on issues other than naming can require significant effort, and, unlike naming disagreements, linkers normally cannot detect disagreements in these other areas.

Also, note that with '-fno-underscoring', the lack of appended underscores introduces the very real possibility that a user-defined external name will conflict with a name in a system library, which could make finding unresolved-reference bugs quite difficult in some cases—they might occur at program run time, and show up only as buggy behavior at run time.

In future versions of GNU Fortran we hope to improve naming and linking issues so that debugging always involves using the names as they appear in the source, even if the names as seen by the linker are mangled to prevent accidental linking between procedures with incompatible interfaces.

`-fsecond-underscore`

By default, GNU Fortran appends an underscore to external names. If this option is used GNU Fortran appends two underscores to names with underscores and one underscore to external names with no underscores. GNU Fortran also appends two underscores to internal names with underscores to avoid naming collisions with external names.

This option has no effect if '`-fno-underscoring`' is in effect. It is implied by the '`-ff2c`' option.

Otherwise, with this option, an external name such as `MAX_COUNT` is implemented as a reference to the link-time external symbol `max_count__`, instead of `max_count_`. This is required for compatibility with `g77` and `f2c`, and is implied by use of the '`-ff2c`' option.

`-fbounds-check`

Enable generation of run-time checks for array subscripts and against the declared minimum and maximum values. It also checks array indices for assumed and deferred shape arrays against the actual allocated bounds.

In the future this may also include other forms of checking, e.g., checking substring references.

`-fmax-stack-var-size=`*n*

This option specifies the size in bytes of the largest array that will be put on the stack.

This option currently only affects local arrays declared with constant bounds, and may not apply to all character variables. Future versions of GNU Fortran may improve this behavior.

The default value for *n* is 32768.

`-fpack-derived`

This option tells GNU Fortran to pack derived type members as closely as possible. Code compiled with this option is likely to be incompatible with code compiled without this option, and may execute slower.

`-frepack-arrays`

In some circumstances GNU Fortran may pass assumed shape array sections via a descriptor describing a noncontiguous area of memory. This option adds code to the function prologue to repack the data into a contiguous block at runtime.

This should result in faster accesses to the array. However it can introduce significant overhead to the function call, especially when the passed data is noncontiguous.

`-fshort-enums`

This option is provided for interoperability with C code that was compiled with the '`-fshort-enums`' option. It will make GNU Fortran choose the smallest `INTEGER` kind a given enumerator set will fit in, and give all its enumerators this kind.

`-fexternal-blas`
: This option will make gfortran generate calls to BLAS functions for some matrix operations like `MATMUL`, instead of using our own algorithms, if the size of the matrices involved is larger than a given limit (see '`-fblas-matmul-limit`'). This may be profitable if an optimized vendor BLAS library is available. The BLAS library will have to be specified at link time.

`-fblas-matmul-limit=`*n*
: Only significant when '`-fexternal-blas`' is in effect. Matrix multiplication of matrices with size larger than (or equal to) *n* will be performed by calls to BLAS functions, while others will be handled by `gfortran` internal algorithms. If the matrices involved are not square, the size comparison is performed using the geometric mean of the dimensions of the argument and result matrices.

  The default value for *n* is 30.

See section "Options for Code Generation Conventions" in *Using the GNU Compiler Collection (GCC)*, for information on more options offered by the GBE shared by `gfortran`, `gcc`, and other GNU compilers.

2.8 Environment Variables Affecting gfortran

The `gfortran` compiler currently does not make use of any environment variables to control its operation above and beyond those that affect the operation of `gcc`.

See section "Environment Variables Affecting GCC" in *Using the GNU Compiler Collection (GCC)*, for information on environment variables.

See Chapter 3 [Runtime], page 17, for environment variables that affect the run-time behavior of programs compiled with GNU Fortran.

3 Runtime: Influencing runtime behavior with environment variables

The behavior of the `gfortran` can be influenced by environment variables.

Malformed environment variables are silently ignored.

3.1 `GFORTRAN_STDIN_UNIT`—Unit number for standard input

This environment variable can be used to select the unit number preconnected to standard input. This must be a positive integer. The default value is 5.

3.2 `GFORTRAN_STDOUT_UNIT`—Unit number for standard output

This environment variable can be used to select the unit number preconnected to standard output. This must be a positive integer. The default value is 6.

3.3 `GFORTRAN_STDERR_UNIT`—Unit number for standard error

This environment variable can be used to select the unit number preconnected to standard error. This must be a positive integer. The default value is 0.

3.4 `GFORTRAN_USE_STDERR`—Send library output to standard error

This environment variable controls where library output is sent. If the first letter is '`y`', '`Y`' or '`1`', standard error is used. If the first letter is '`n`', '`N`' or '`0`', standard output is used.

3.5 `GFORTRAN_TMPDIR`—Directory for scratch files

This environment variable controls where scratch files are created. If this environment variable is missing, GNU Fortran searches for the environment variable `TMP`. If this is also missing, the default is '`/tmp`'.

3.6 `GFORTRAN_UNBUFFERED_ALL`—Don't buffer output

This environment variable controls whether all output is unbuffered. If the first letter is '`y`', '`Y`' or '`1`', all output is unbuffered. This will slow down large writes. If the first letter is '`n`', '`N`' or '`0`', output is buffered. This is the default.

3.7 `GFORTRAN_SHOW_LOCUS`—Show location for runtime errors

If the first letter is '`y`', '`Y`' or '`1`', filename and line numbers for runtime errors are printed. If the first letter is '`n`', '`N`' or '`0`', don't print filename and line numbers for runtime errors. The default is to print the location.

3.8 `GFORTRAN_OPTIONAL_PLUS`—Print leading + where permitted

If the first letter is '`y`', '`Y`' or '`1`', a plus sign is printed where permitted by the Fortran standard. If the first letter is '`n`', '`N`' or '`0`', a plus sign is not printed in most cases. Default is not to print plus signs.

3.9 GFORTRAN_DEFAULT_RECL—Default record length for new files

This environment variable specifies the default record length, in bytes, for files which are opened without a RECL tag in the OPEN statement. This must be a positive integer. The default value is 1073741824 bytes (1 GB).

3.10 GFORTRAN_LIST_SEPARATOR—Separator for list output

This environment variable specifies the separator when writing list-directed output. It may contain any number of spaces and at most one comma. If you specify this on the command line, be sure to quote spaces, as in

```
$ GFORTRAN_LIST_SEPARATOR='  ,  ' ./a.out
```

when a.out is the compiled Fortran program that you want to run. Default is a single space.

3.11 GFORTRAN_CONVERT_UNIT—Set endianness for unformatted I/O

By setting the GFORTRAN_CONVERT_UNIT variable, it is possible to change the representation of data for unformatted files. The syntax for the GFORTRAN_CONVERT_UNIT variable is:

```
GFORTRAN_CONVERT_UNIT: mode | mode ';' exception ;
mode: 'native' | 'swap' | 'big_endian' | 'little_endian' ;
exception: mode ':' unit_list | unit_list ;
unit_list: unit_spec | unit_list unit_spec ;
unit_spec: INTEGER | INTEGER '-' INTEGER ;
```

The variable consists of an optional default mode, followed by a list of optional exceptions, which are separated by semicolons from the preceding default and each other. Each exception consists of a format and a comma-separated list of units. Valid values for the modes are the same as for the CONVERT specifier:

NATIVE Use the native format. This is the default.

SWAP Swap between little- and big-endian.

LITTLE_ENDIAN Use the little-endian format for unformatted files.

BIG_ENDIAN Use the big-endian format for unformatted files.

A missing mode for an exception is taken to mean BIG_ENDIAN. Examples of values for GFORTRAN_CONVERT_UNIT are:

'big_endian' Do all unformatted I/O in big_endian mode.

'little_endian;native:10-20,25' Do all unformatted I/O in little_endian mode, except for units 10 to 20 and 25, which are in native format.

'10-20' Units 10 to 20 are big-endian, the rest is native.

Setting the environment variables should be done on the command line or via the export command for sh-compatible shells and via setenv for csh-compatible shells.

Example for sh:

```
$ gfortran foo.f90
$ GFORTRAN_CONVERT_UNIT='big_endian;native:10-20' ./a.out
```

Example code for csh:

```
% gfortran foo.f90
% setenv GFORTRAN_CONVERT_UNIT 'big_endian;native:10-20'
```

```
% ./a.out
```

Using anything but the native representation for unformatted data carries a significant speed overhead. If speed in this area matters to you, it is best if you use this only for data that needs to be portable.

See Section 5.14 [CONVERT specifier], page 30, for an alternative way to specify the data representation for unformatted files. See Section 2.6 [Runtime Options], page 12, for setting a default data representation for the whole program. The `CONVERT` specifier overrides the '`-fconvert`' compile options.

Part II: Language Reference

4 Fortran 2003 Status

Although GNU Fortran focuses on implementing the Fortran 95 standard for the time being, a few Fortran 2003 features are currently available.

- Intrinsics `command_argument_count`, `get_command`, `get_command_argument`, `get_environment_variable`, and `move_alloc`.
- Array constructors using square brackets. That is, `[...]` rather than `(/.../)`.
- `FLUSH` statement.
- `IOMSG=` specifier for I/O statements.
- Support for the declaration of enumeration constants via the `ENUM` and `ENUMERATOR` statements. Interoperability with `gcc` is guaranteed also for the case where the `-fshort-enums` command line option is given.
- TR 15581:
  - `ALLOCATABLE` dummy arguments.
  - `ALLOCATABLE` function results
  - `ALLOCATABLE` components of derived types
- The `OPEN` statement supports the `ACCESS='STREAM'` specifier, allowing I/O without any record structure.
- Namelist input/output for internal files.
- The `PROTECTED` statement and attribute.
- The `VALUE` statement and attribute.
- The `VOLATILE` statement and attribute.
- The `IMPORT` statement, allowing to import host-associated derived types.
- `USE` statement with `INTRINSIC` and `NON_INTRINSIC` attribute; supported intrinsic modules: `ISO_FORTRAN_ENV`, `OMP_LIB` and `OMP_LIB_KINDS`.
- Renaming of operators in the `USE` statement.

5 Extensions

GNU Fortran implements a number of extensions over standard Fortran. This chapter contains information on their syntax and meaning. There are currently two categories of GNU Fortran extensions, those that provide functionality beyond that provided by any standard, and those that are supported by GNU Fortran purely for backward compatibility with legacy compilers. By default, '-std=gnu' allows the compiler to accept both types of extensions, but to warn about the use of the latter. Specifying either '-std=f95' or '-std=f2003' disables both types of extensions, and '-std=legacy' allows both without warning.

5.1 Old-style kind specifications

GNU Fortran allows old-style kind specifications in declarations. These look like:

```
TYPESPEC*size x,y,z
```

where `TYPESPEC` is a basic type (`INTEGER`, `REAL`, etc.), and where `size` is a byte count corresponding to the storage size of a valid kind for that type. (For `COMPLEX` variables, `size` is the total size of the real and imaginary parts.) The statement then declares `x`, `y` and `z` to be of type `TYPESPEC` with the appropriate kind. This is equivalent to the standard-conforming declaration

```
TYPESPEC(k) x,y,z
```

where `k` is equal to `size` for most types, but is equal to `size/2` for the `COMPLEX` type.

5.2 Old-style variable initialization

GNU Fortran allows old-style initialization of variables of the form:

```
INTEGER i/1/,j/2/
REAL x(2,2) /3*0.,1./
```

The syntax for the initializers is as for the `DATA` statement, but unlike in a `DATA` statement, an initializer only applies to the variable immediately preceding the initialization. In other words, something like `INTEGER I,J/2,3/` is not valid. This style of initialization is only allowed in declarations without double colons (`::`); the double colons were introduced in Fortran 90, which also introduced a standard syntax for initializing variables in type declarations.

Examples of standard-conforming code equivalent to the above example are:

```
! Fortran 90
      INTEGER :: i = 1, j = 2
      REAL :: x(2,2) = RESHAPE((/0.,0.,0.,1./),SHAPE(x))
! Fortran 77
      INTEGER i, j
      REAL x(2,2)
      DATA i/1/, j/2/, x/3*0.,1./
```

Note that variables which are explicitly initialized in declarations or in `DATA` statements automatically acquire the `SAVE` attribute.

5.3 Extensions to namelist

GNU Fortran fully supports the Fortran 95 standard for namelist I/O including array qualifiers, substrings and fully qualified derived types. The output from a namelist write is compatible with namelist read. The output has all names in upper case and indentation to column 1 after the namelist name. Two extensions are permitted:

Old-style use of '$' instead of '&'

```
$MYNML
 X(:)%Y(2) = 1.0 2.0 3.0
 CH(1:4) = "abcd"
$END
```

It should be noted that the default terminator is '/' rather than '&END'.

Querying of the namelist when inputting from stdin. After at least one space, entering '?' sends to stdout the namelist name and the names of the variables in the namelist:

```
 ?

&mynml
 x
 x%y
 ch
&end
```

Entering '=?' outputs the namelist to stdout, as if WRITE(*,NML = mynml) had been called:

```
=?

&MYNML
 X(1)%Y=  0.000000    ,  1.000000    ,  0.000000    ,
 X(2)%Y=  0.000000    ,  2.000000    ,  0.000000    ,
 X(3)%Y=  0.000000    ,  3.000000    ,  0.000000    ,
 CH=abcd,  /
```

To aid this dialog, when input is from stdin, errors send their messages to stderr and execution continues, even if IOSTAT is set.

PRINT namelist is permitted. This causes an error if '-std=f95' is used.

```
PROGRAM test_print
  REAL, dimension (4)  ::  x = (/1.0, 2.0, 3.0, 4.0/)
  NAMELIST /mynml/ x
  PRINT mynml
END PROGRAM test_print
```

Expanded namelist reads are permitted. This causes an error if '-std=f95' is used. In the following example, the first element of the array will be given the value 0.00 and the two succeeding elements will be given the values 1.00 and 2.00.

```
&MYNML
  X(1,1) = 0.00 , 1.00 , 2.00
/
```

5.4 X format descriptor without count field

To support legacy codes, GNU Fortran permits the count field of the X edit descriptor in FORMAT statements to be omitted. When omitted, the count is implicitly assumed to be one.

```
       PRINT 10, 2, 3
10     FORMAT (I1, X, I1)
```

5.5 Commas in FORMAT specifications

To support legacy codes, GNU Fortran allows the comma separator to be omitted immediately before and after character string edit descriptors in FORMAT statements.

```
       PRINT 10, 2, 3
10     FORMAT ('FOO='I1' BAR='I2)
```

5.6 Missing period in FORMAT specifications

To support legacy codes, GNU Fortran allows missing periods in format specifications if and only if '-std=legacy' is given on the command line. This is considered non-conforming code and is discouraged.

```
       REAL :: value
       READ(*,10) value
10     FORMAT ('F4')
```

5.7 I/O item lists

To support legacy codes, GNU Fortran allows the input item list of the READ statement, and the output item lists of the WRITE and PRINT statements, to start with a comma.

5.8 BOZ literal constants

As an extension, GNU Fortran allows hexadecimal BOZ literal constants to be specified using the X prefix, in addition to the standard Z prefix. BOZ literal constants can also be specified by adding a suffix to the string. For example, Z'ABC' and 'ABC'Z are equivalent.

The Fortran standard restricts the appearance of a BOZ literal constant to the DATA statement, and it is expected to be assigned to an INTEGER variable. GNU Fortran permits a BOZ literal to appear in any initialization expression as well as assignment statements.

Attempts to use a BOZ literal constant to do a bitwise initialization of a variable can lead to confusion. A BOZ literal constant is converted to an INTEGER value with the kind type with the largest decimal representation, and this value is then converted numerically to the type and kind of the variable in question. Thus, one should not expect a bitwise copy of the BOZ literal constant to be assigned to a REAL variable.

Similarly, initializing an INTEGER variable with a statement such as DATA i/Z'FFFFFFFF'/ will produce an integer overflow rather than the desired result of −1 when i is a 32-bit integer on a system that supports 64-bit integers. The '-fno-range-check' option can be used as a workaround for legacy code that initializes integers in this manner.

5.9 Real array indices

As an extension, GNU Fortran allows the use of REAL expressions or variables as array indices.

5.10 Unary operators

As an extension, GNU Fortran allows unary plus and unary minus operators to appear as the second operand of binary arithmetic operators without the need for parenthesis.

```
       X = Y * -Z
```

5.11 Implicitly convert `LOGICAL` and `INTEGER` values

As an extension for backwards compatibility with other compilers, GNU Fortran allows the implicit conversion of `LOGICAL` values to `INTEGER` values and vice versa. When converting from a `LOGICAL` to an `INTEGER`, `.FALSE.` is interpreted as zero, and `.TRUE.` is interpreted as one. When converting from `INTEGER` to `LOGICAL`, the value zero is interpreted as `.FALSE.` and any nonzero value is interpreted as `.TRUE.`.

```
INTEGER :: i = 1
IF (i) PRINT *, 'True'
```

5.12 Hollerith constants support

GNU Fortran supports Hollerith constants in assignments, function arguments, and `DATA` and `ASSIGN` statements. A Hollerith constant is written as a string of characters preceded by an integer constant indicating the character count, and the letter `H` or `h`, and stored in bytewise fashion in a numeric (`INTEGER`, `REAL`, or `complex`) or `LOGICAL` variable. The constant will be padded or truncated to fit the size of the variable in which it is stored.

Examples of valid uses of Hollerith constants:

```
complex*16 x(2)
data x /16Habcdefghijklmnop, 16Hqrstuvwxyz012345/
x(1) = 16HABCDEFGHIJKLMNOP
call foo (4h abc)
```

Invalid Hollerith constants examples:

```
integer*4 a
a = 8H12345678 ! Valid, but the Hollerith constant will be truncated.
a = 0H         ! At least one character is needed.
```

In general, Hollerith constants were used to provide a rudimentary facility for handling character strings in early Fortran compilers, prior to the introduction of `CHARACTER` variables in Fortran 77; in those cases, the standard-compliant equivalent is to convert the program to use proper character strings. On occasion, there may be a case where the intent is specifically to initialize a numeric variable with a given byte sequence. In these cases, the same result can be obtained by using the `TRANSFER` statement, as in this example.

```
INTEGER(KIND=4) :: a
a = TRANSFER ("abcd", a)     ! equivalent to: a = 4Habcd
```

5.13 Cray pointers

Cray pointers are part of a non-standard extension that provides a C-like pointer in Fortran. This is accomplished through a pair of variables: an integer "pointer" that holds a memory address, and a "pointee" that is used to dereference the pointer.

Pointer/pointee pairs are declared in statements of the form:

```
pointer ( <pointer> , <pointee> )
```

or,

```
pointer ( <pointer1> , <pointee1> ), ( <pointer2> , <pointee2> ), ...
```

The pointer is an integer that is intended to hold a memory address. The pointee may be an array or scalar. A pointee can be an assumed size array—that is, the last dimension may be left unspecified by using a `*` in place of a value—but a pointee cannot be an assumed shape array. No space is allocated for the pointee.

The pointee may have its type declared before or after the pointer statement, and its array specification (if any) may be declared before, during, or after the pointer statement. The pointer may be declared as an integer prior to the pointer statement. However, some machines have default integer sizes that are different than the size of a pointer, and so the following code is not portable:

```
integer ipt
pointer (ipt, iarr)
```

If a pointer is declared with a kind that is too small, the compiler will issue a warning; the resulting binary will probably not work correctly, because the memory addresses stored in the pointers may be truncated. It is safer to omit the first line of the above example; if explicit declaration of ipt's type is omitted, then the compiler will ensure that ipt is an integer variable large enough to hold a pointer.

Pointer arithmetic is valid with Cray pointers, but it is not the same as C pointer arithmetic. Cray pointers are just ordinary integers, so the user is responsible for determining how many bytes to add to a pointer in order to increment it. Consider the following example:

```
real target(10)
real pointee(10)
pointer (ipt, pointee)
ipt = loc (target)
ipt = ipt + 1
```

The last statement does not set `ipt` to the address of `target(1)`, as it would in C pointer arithmetic. Adding `1` to `ipt` just adds one byte to the address stored in `ipt`.

Any expression involving the pointee will be translated to use the value stored in the pointer as the base address.

To get the address of elements, this extension provides an intrinsic function `LOC()`. The `LOC()` function is equivalent to the `&` operator in C, except the address is cast to an integer type:

```
real ar(10)
pointer(ipt, arpte(10))
real arpte
ipt = loc(ar)  ! Makes arpte is an alias for ar
arpte(1) = 1.0 ! Sets ar(1) to 1.0
```

The pointer can also be set by a call to the `MALLOC` intrinsic (see Section 6.127 [MALLOC], page 102).

Cray pointees often are used to alias an existing variable. For example:

```
integer target(10)
integer iarr(10)
pointer (ipt, iarr)
ipt = loc(target)
```

As long as `ipt` remains unchanged, `iarr` is now an alias for `target`. The optimizer, however, will not detect this aliasing, so it is unsafe to use `iarr` and `target` simultaneously. Using a pointee in any way that violates the Fortran aliasing rules or assumptions is illegal. It is the user's responsibility to avoid doing this; the compiler works under the assumption that no such aliasing occurs.

Cray pointers will work correctly when there is no aliasing (i.e., when they are used to access a dynamically allocated block of memory), and also in any routine where a pointee is used, but any variable with which it shares storage is not used. Code that violates these rules may not run as the user intends. This is not a bug in the optimizer; any code that violates the aliasing rules is illegal. (Note that this is not unique to GNU Fortran; any Fortran compiler that supports Cray pointers will "incorrectly" optimize code with illegal aliasing.)

There are a number of restrictions on the attributes that can be applied to Cray pointers and pointees. Pointees may not have the `ALLOCATABLE`, `INTENT`, `OPTIONAL`, `DUMMY`, `TARGET`, `INTRINSIC`, or `POINTER` attributes. Pointers may not have the `DIMENSION`, `POINTER`, `TARGET`, `ALLOCATABLE`, `EXTERNAL`, or `INTRINSIC` attributes. Pointees may not occur in more than one pointer statement. A pointee cannot be a pointer. Pointees cannot occur in equivalence, common, or data statements.

A Cray pointer may also point to a function or a subroutine. For example, the following excerpt is valid:

```
implicit none
external sub
pointer (subptr,subpte)
external subpte
subptr = loc(sub)
call subpte()
[...]
subroutine sub
[...]
end subroutine sub
```

A pointer may be modified during the course of a program, and this will change the location to which the pointee refers. However, when pointees are passed as arguments, they are treated as ordinary variables in the invoked function. Subsequent changes to the pointer will not change the base address of the array that was passed.

5.14 CONVERT specifier

GNU Fortran allows the conversion of unformatted data between little- and big-endian representation to facilitate moving of data between different systems. The conversion can be indicated with the `CONVERT` specifier on the `OPEN` statement. See Section 3.11 [GFORTRAN_CONVERT_UNIT], page 18, for an alternative way of specifying the data format via an environment variable.

Valid values for `CONVERT` are:

`CONVERT='NATIVE'` Use the native format. This is the default.

`CONVERT='SWAP'` Swap between little- and big-endian.

`CONVERT='LITTLE_ENDIAN'` Use the little-endian representation for unformatted files.

`CONVERT='BIG_ENDIAN'` Use the big-endian representation for unformatted files.

Using the option could look like this:

```
open(file='big.dat',form='unformatted',access='sequential', &
     convert='big_endian')
```

The value of the conversion can be queried by using `INQUIRE(CONVERT=ch)`. The values returned are `'BIG_ENDIAN'` and `'LITTLE_ENDIAN'`.

`CONVERT` works between big- and little-endian for `INTEGER` values of all supported kinds and for `REAL` on IEEE systems of kinds 4 and 8. Conversion between different "extended double" types on different architectures such as m68k and x86_64, which GNU Fortran supports as `REAL(KIND=10)` and `REAL(KIND=16)`, will probably not work.

Note that the values specified via the GFORTRAN_CONVERT_UNIT environment variable will override the CONVERT specifier in the open statement. This is to give control over data formats to users who do not have the source code of their program available.

Using anything but the native representation for unformatted data carries a significant speed overhead. If speed in this area matters to you, it is best if you use this only for data that needs to be portable.

5.15 OpenMP

GNU Fortran attempts to be OpenMP Application Program Interface v2.5 compatible when invoked with the '`-fopenmp`' option. GNU Fortran then generates parallelized code according to the OpenMP directives used in the source. The OpenMP Fortran runtime library routines are provided both in a form of a Fortran 90 module named `omp_lib` and in a form of a Fortran `include` file named '`omp_lib.h`'.

For details refer to the actual OpenMP Application Program Interface v2.5 specification.

6 Intrinsic Procedures

6.1 Introduction to intrinsic procedures

The intrinsic procedures provided by GNU Fortran include all of the intrinsic procedures required by the Fortran 95 standard, a set of intrinsic procedures for backwards compatibility with G77, and a small selection of intrinsic procedures from the Fortran 2003 standard. Any conflict between a description here and a description in either the Fortran 95 standard or the Fortran 2003 standard is unintentional, and the standard(s) should be considered authoritative.

The enumeration of the `KIND` type parameter is processor defined in the Fortran 95 standard. GNU Fortran defines the default integer type and default real type by `INTEGER(KIND=4)` and `REAL(KIND=4)`, respectively. The standard mandates that both data types shall have another kind, which have more precision. On typical target architectures supported by `gfortran`, this kind type parameter is `KIND=8`. Hence, `REAL(KIND=8)` and `DOUBLE PRECISION` are equivalent. In the description of generic intrinsic procedures, the kind type parameter will be specified by `KIND=*`, and in the description of specific names for an intrinsic procedure the kind type parameter will be explicitly given (e.g., `REAL(KIND=4)` or `REAL(KIND=8)`). Finally, for brevity the optional `KIND=` syntax will be omitted.

Many of the intrinsic procedures take one or more optional arguments. This document follows the convention used in the Fortran 95 standard, and denotes such arguments by square brackets.

GNU Fortran offers the '`-std=f95`' and '`-std=gnu`' options, which can be used to restrict the set of intrinsic procedures to a given standard. By default, `gfortran` sets the '`-std=gnu`' option, and so all intrinsic procedures described here are accepted. There is one caveat. For a select group of intrinsic procedures, `g77` implemented both a function and a subroutine. Both classes have been implemented in `gfortran` for backwards compatibility with `g77`. It is noted here that these functions and subroutines cannot be intermixed in a given subprogram. In the descriptions that follow, the applicable standard for each intrinsic procedure is noted.

6.2 `ABORT` — Abort the program

Description:
: `ABORT` causes immediate termination of the program. On operating systems that support a core dump, `ABORT` will produce a core dump, which is suitable for debugging purposes.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL ABORT`

Return value:
: Does not return.

Example:

```
program test_abort
  integer :: i = 1, j = 2
  if (i /= j) call abort
end program test_abort
```

See also: Section 6.60 [EXIT], page 68, Section 6.109 [KILL], page 93

6.3 ABS — Absolute value

Description:
`ABS(X)` computes the absolute value of `X`.

Standard: F77 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: `RESULT = ABS(X)`

Arguments:
X The type of the argument shall be an `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)`.

Return value:
The return value is of the same type and kind as the argument except the return value is `REAL(*)` for a `COMPLEX(*)` argument.

Example:

```
program test_abs
  integer :: i = -1
  real :: x = -1.e0
  complex :: z = (-1.e0,0.e0)
  i = abs(i)
  x = abs(x)
  x = abs(z)
end program test_abs
```

Specific names:

Name	Argument	Return type	Standard
CABS(Z)	COMPLEX(4) Z	REAL(4)	F77 and later
DABS(X)	REAL(8) X	REAL(8)	F77 and later
IABS(I)	INTEGER(4) I	INTEGER(4)	F77 and later
ZABS(Z)	COMPLEX(8) Z	COMPLEX(8)	GNU extension
CDABS(Z)	COMPLEX(8) Z	COMPLEX(8)	GNU extension

6.4 ACCESS — Checks file access modes

Description:
`ACCESS(NAME, MODE)` checks whether the file *NAME* exists, is readable, writable or executable. Except for the executable check, `ACCESS` can be replaced by Fortran 95's `INQUIRE`.

Standard: GNU extension

Class: Inquiry function

Syntax: `RESULT = ACCESS(NAME, MODE)`

Arguments:

NAME	Scalar CHARACTER with the file name. Tailing blank are ignored unless the character achar(0) is present, then all characters up to and excluding achar(0) are used as file name.
MODE	Scalar CHARACTER with the file access mode, may be any concatenation of "r" (readable), "w" (writable) and "x" (executable), or " " to check for existence.

Return value:
Returns a scalar INTEGER, which is 0 if the file is accessible in the given mode; otherwise or if an invalid argument has been given for MODE the value 1 is returned.

Example:

```
program access_test
  implicit none
  character(len=*), parameter :: file  = 'test.dat'
  character(len=*), parameter :: file2 = 'test.dat  '//achar(0)
  if(access(file,' ') == 0) print *, trim(file),' is exists'
  if(access(file,'r') == 0) print *, trim(file),' is readable'
  if(access(file,'w') == 0) print *, trim(file),' is writable'
  if(access(file,'x') == 0) print *, trim(file),' is executable'
  if(access(file2,'rwx') == 0) &
    print *, trim(file2),' is readable, writable and executable'
end program access_test
```

Specific names:
See also:

6.5 ACHAR — Character in ASCII collating sequence

Description:
ACHAR(I) returns the character located at position I in the ASCII collating sequence.

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = ACHAR(I)

Arguments:

I	The type shall be INTEGER(*).

Return value:
The return value is of type CHARACTER with a length of one. The kind type parameter is the same as KIND('A').

Example:

```
program test_achar
  character c
  c = achar(32)
end program test_achar
```

Note: See Section 6.97 [ICHAR], page 87 for a discussion of converting between numerical values and formatted string representations.

See also: Section 6.33 [CHAR], page 51, Section 6.91 [IACHAR], page 84, Section 6.97 [ICHAR], page 87

6.6 ACOS — Arccosine function

Description:
`ACOS(X)` computes the arccosine of *X* (inverse of `COS(X)`).

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = ACOS(X)`

Arguments:

X	The type shall be `REAL(*)` with a magnitude that is less than one.

Return value:
The return value is of type `REAL(*)` and it lies in the range $0 \leq \acos(x) \leq \pi$. The kind type parameter is the same as *X*.

Example:

```
program test_acos
  real(8) :: x = 0.866_8
  x = acos(x)
end program test_acos
```

Specific names:

Name	Argument	Return type	Standard
`DACOS(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

See also: Inverse function: Section 6.39 [COS], page 55

6.7 ACOSH — Hyperbolic arccosine function

Description:
`ACOSH(X)` computes the area hyperbolic cosine of *X* (inverse of `COSH(X)`).

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = ACOSH(X)`

Arguments:

X	The type shall be `REAL(*)` with a magnitude that is greater or equal to one.

Return value:
The return value is of type `REAL(*)` and it lies in the range $0 \leq \mathrm{acosh}(x) \leq \infty$.

Example:

```
PROGRAM test_acosh
  REAL(8), DIMENSION(3) :: x = (/ 1.0, 2.0, 3.0 /)
  WRITE (*,*) ACOSH(x)
END PROGRAM
```

See also: Inverse function: Section 6.40 [COSH], page 55

6.8 ADJUSTL — Left adjust a string

Description:
`ADJUSTL(STR)` will left adjust a string by removing leading spaces. Spaces are inserted at the end of the string as needed.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = ADJUSTL(STR)`

Arguments:

STR	The type shall be `CHARACTER`.

Return value:
The return value is of type `CHARACTER` where leading spaces are removed and the same number of spaces are inserted on the end of *STR*.

Example:

```
program test_adjustl
  character(len=20) :: str = '   gfortran'
  str = adjustl(str)
  print *, str
end program test_adjustl
```

6.9 ADJUSTR — Right adjust a string

Description:
`ADJUSTR(STR)` will right adjust a string by removing trailing spaces. Spaces are inserted at the start of the string as needed.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = ADJUSTR(STR)`

Arguments:

STR	The type shall be `CHARACTER`.

Return value:
The return value is of type `CHARACTER` where trailing spaces are removed and the same number of spaces are inserted at the start of *STR*.

Example:

```
program test_adjustr
  character(len=20) :: str = 'gfortran'
  str = adjustr(str)
  print *, str
end program test_adjustr
```

6.10 AIMAG — Imaginary part of complex number

Description:
`AIMAG(Z)` yields the imaginary part of complex argument `Z`. The `IMAG(Z)` and `IMAGPART(Z)` intrinsic functions are provided for compatibility with `g77`, and their use in new code is strongly discouraged.

Standard: F77 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: `RESULT = AIMAG(Z)`

Arguments:

Z	The type of the argument shall be `COMPLEX(*)`.

Return value:
The return value is of type real with the kind type parameter of the argument.

Example:

```
program test_aimag
  complex(4) z4
  complex(8) z8
  z4 = cmplx(1.e0_4, 0.e0_4)
  z8 = cmplx(0.e0_8, 1.e0_8)
  print *, aimag(z4), dimag(z8)
end program test_aimag
```

Specific names:

Name	Argument	Return type	Standard
`DIMAG(Z)`	`COMPLEX(8) Z`	`REAL(8)`	GNU extension
`IMAG(Z)`	`COMPLEX(*) Z`	`REAL(*)`	GNU extension
`IMAGPART(Z)`	`COMPLEX(*) Z`	`REAL(*)`	GNU extension

6.11 AINT — Truncate to a whole number

Description:
`AINT(X [, KIND])` truncates its argument to a whole number.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = AINT(X [, KIND])`

Arguments:

X	The type of the argument shall be `REAL(*)`.
KIND	(Optional) An `INTEGER(*)` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type real with the kind type parameter of the argument if the optional *KIND* is absent; otherwise, the kind type parameter will be given by *KIND*. If the magnitude of *X* is less than one, then `AINT(X)` returns zero. If the magnitude is equal to or greater than one, then it returns the largest whole number that does not exceed its magnitude. The sign is the same as the sign of *X*.

Example:

```
program test_aint
  real(4) x4
  real(8) x8
  x4 = 1.234E0_4
  x8 = 4.321_8
  print *, aint(x4), dint(x8)
  x8 = aint(x4,8)
end program test_aint
```

Specific names:

Name	Argument	Return type	Standard
DINT(X)	REAL(8) X	REAL(8)	F77 and later

6.12 ALARM — Execute a routine after a given delay

Description:
: ALARM(SECONDS, HANDLER [, STATUS]) causes external subroutine *HANDLER* to be executed after a delay of *SECONDS* by using alarm(2) to set up a signal and signal(2) to catch it. If *STATUS* is supplied, it will be returned with the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

Standard: GNU extension

Class: Subroutine

Syntax: CALL ALARM(SECONDS, HANDLER [, STATUS])

Arguments:

SECONDS	The type of the argument shall be a scalar INTEGER. It is INTENT(IN).
HANDLER	Signal handler (INTEGER FUNCTION or SUBROUTINE) or dummy/global INTEGER scalar. The scalar values may be either SIG_IGN=1 to ignore the alarm generated or SIG_DFL=0 to set the default action. It is INTENT(IN).
STATUS	(Optional) *STATUS* shall be a scalar variable of the default INTEGER kind. It is INTENT(OUT).

Example:

```
program test_alarm
  external handler_print
  integer i
  call alarm (3, handler_print, i)
  print *, i
  call sleep(10)
end program test_alarm
```

This will cause the external routine *handler_print* to be called after 3 seconds.

6.13 ALL — All values in *MASK* along *DIM* are true

Description:
: ALL(MASK [, DIM]) determines if all the values are true in *MASK* in the array along dimension *DIM*.

Standard: F95 and later

Class: transformational function

Syntax: `RESULT = ALL(MASK [, DIM])`

Arguments:

MASK	The type of the argument shall be `LOGICAL(*)` and it shall not be scalar.
DIM	(Optional) *DIM* shall be a scalar integer with a value that lies between one and the rank of *MASK*.

Return value:

`ALL(MASK)` returns a scalar value of type `LOGICAL(*)` where the kind type parameter is the same as the kind type parameter of *MASK*. If *DIM* is present, then `ALL(MASK, DIM)` returns an array with the rank of *MASK* minus 1. The shape is determined from the shape of *MASK* where the *DIM* dimension is elided.

(A) `ALL(MASK)` is true if all elements of *MASK* are true. It also is true if *MASK* has zero size; otherwise, it is false.

(B) If the rank of *MASK* is one, then `ALL(MASK,DIM)` is equivalent to `ALL(MASK)`. If the rank is greater than one, then `ALL(MASK,DIM)` is determined by applying `ALL` to the array sections.

Example:

```
program test_all
  logical l
  l = all((/.true., .true., .true./))
  print *, l
  call section
  contains
    subroutine section
      integer a(2,3), b(2,3)
      a = 1
      b = 1
      b(2,2) = 2
      print *, all(a .eq. b, 1)
      print *, all(a .eq. b, 2)
    end subroutine section
end program test_all
```

6.14 `ALLOCATED` — Status of an allocatable entity

Description:

`ALLOCATED(X)` checks the status of whether *X* is allocated.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = ALLOCATED(X)`

Arguments:

X	The argument shall be an `ALLOCATABLE` array.

Return value:
The return value is a scalar LOGICAL with the default logical kind type parameter. If *X* is allocated, ALLOCATED(X) is .TRUE.; otherwise, it returns the .TRUE.

Example:
```
program test_allocated
  integer :: i = 4
  real(4), allocatable :: x(:)
  if (allocated(x) .eqv. .false.) allocate(x(i))
end program test_allocated
```

6.15 AND — Bitwise logical AND

Description:
Bitwise logical AND.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. For integer arguments, programmers should consider the use of the Section 6.92 [IAND], page 85 intrinsic defined by the Fortran standard.

Standard: GNU extension

Class: Non-elemental function

Syntax: RESULT = AND(I, J)

Arguments:

I	The type shall be either INTEGER(*) or LOGICAL.
J	The type shall be either INTEGER(*) or LOGICAL.

Return value:
The return type is either INTEGER(*) or LOGICAL after cross-promotion of the arguments.

Example:
```
PROGRAM test_and
  LOGICAL :: T = .TRUE., F = ..FALSE.
  INTEGER :: a, b
  DATA a / Z'F' /, b / Z'3' /

  WRITE (*,*) AND(T, T), AND(T, F), AND(F, T), AND(F, F)
  WRITE (*,*) AND(a, b)
END PROGRAM
```

See also: F95 elemental function: Section 6.92 [IAND], page 85

6.16 ANINT — Nearest whole number

Description:
ANINT(X [, KIND]) rounds its argument to the nearest whole number.

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = ANINT(X [, KIND])

Arguments:

X	The type of the argument shall be `REAL(*)`.
KIND	(Optional) An `INTEGER(*)` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type real with the kind type parameter of the argument if the optional *KIND* is absent; otherwise, the kind type parameter will be given by *KIND*. If *X* is greater than zero, then `ANINT(X)` returns `AINT(X+0.5)`. If *X* is less than or equal to zero, then it returns `AINT(X-0.5)`.

Example:

```
program test_anint
  real(4) x4
  real(8) x8
  x4 = 1.234E0_4
  x8 = 4.321_8
  print *, anint(x4), dnint(x8)
  x8 = anint(x4,8)
end program test_anint
```

Specific names:

Name	Argument	Return type	Standard
`DNINT(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

6.17 `ANY` — Any value in *MASK* along *DIM* is true

Description:
`ANY(MASK [, DIM])` determines if any of the values in the logical array *MASK* along dimension *DIM* are `.TRUE.`.

Standard: F95 and later

Class: transformational function

Syntax: `RESULT = ANY(MASK [, DIM])`

Arguments:

MASK	The type of the argument shall be `LOGICAL(*)` and it shall not be scalar.
DIM	(Optional) *DIM* shall be a scalar integer with a value that lies between one and the rank of *MASK*.

Return value:
`ANY(MASK)` returns a scalar value of type `LOGICAL(*)` where the kind type parameter is the same as the kind type parameter of *MASK*. If *DIM* is present, then `ANY(MASK, DIM)` returns an array with the rank of *MASK* minus 1. The shape is determined from the shape of *MASK* where the *DIM* dimension is elided.

(A) `ANY(MASK)` is true if any element of *MASK* is true; otherwise, it is false. It also is false if *MASK* has zero size.

(B) If the rank of *MASK* is one, then `ANY(MASK,DIM)` is equivalent to `ANY(MASK)`. If the rank is greater than one, then `ANY(MASK,DIM)` is determined by applying `ANY` to the array sections.

Example:

```
program test_any
  logical l
  l = any((/.true., .true., .true./))
  print *, l
  call section
  contains
    subroutine section
      integer a(2,3), b(2,3)
      a = 1
      b = 1
      b(2,2) = 2
      print *, any(a .eq. b, 1)
      print *, any(a .eq. b, 2)
    end subroutine section
end program test_any
```

6.18 ASIN — Arcsine function

Description:
: `ASIN(X)` computes the arcsine of its X (inverse of `SIN(X)`).

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = ASIN(X)`

Arguments:

X	The type shall be `REAL(*)`, and a magnitude that is less than one.

Return value:
: The return value is of type `REAL(*)` and it lies in the range $-\pi/2 \leq \operatorname{asin}(x) \leq \pi/2$. The kind type parameter is the same as X.

Example:

```
program test_asin
  real(8) :: x = 0.866_8
  x = asin(x)
end program test_asin
```

Specific names:

Name	Argument	Return type	Standard
`DASIN(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

See also: Inverse function: Section 6.174 [SIN], page 126

6.19 ASINH — Hyperbolic arcsine function

Description:
: `ASINH(X)` computes the area hyperbolic sine of X (inverse of `SINH(X)`).

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = ASINH(X)`

Arguments:

X	The type shall be `REAL(*)`, with *X* a real number.

Return value:
The return value is of type `REAL(*)` and it lies in the range $-\infty \leq \operatorname{asinh}(x) \leq \infty$.

Example:

```
PROGRAM test_asinh
  REAL(8), DIMENSION(3) :: x = (/ -1.0, 0.0, 1.0 /)
  WRITE (*,*) ASINH(x)
END PROGRAM
```

See also: Inverse function: Section 6.175 [SINH], page 126

6.20 ASSOCIATED — Status of a pointer or pointer/target pair

Description:
`ASSOCIATED(PTR [, TGT])` determines the status of the pointer *PTR* or if *PTR* is associated with the target *TGT*.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = ASSOCIATED(PTR [, TGT])`

Arguments:

PTR	*PTR* shall have the `POINTER` attribute and it can be of any type.
TGT	(Optional) *TGT* shall be a `POINTER` or a `TARGET`. It must have the same type, kind type parameter, and array rank as *PTR*.

The status of neither *PTR* nor *TGT* can be undefined.

Return value:
`ASSOCIATED(PTR)` returns a scalar value of type `LOGICAL(4)`. There are several cases:

(A) If the optional *TGT* is not present, then `ASSOCIATED(PTR)`
is true if *PTR* is associated with a target; otherwise, it returns false.

(B) If *TGT* is present and a scalar target, the result is true if
TGT is not a 0 sized storage sequence and the target associated with *PTR* occupies the same storage units. If *PTR* is disassociated, then the result is false.

(C) If *TGT* is present and an array target, the result is true if
TGT and *PTR* have the same shape, are not 0 sized arrays, are arrays whose elements are not 0 sized storage sequences, and *TGT* and *PTR* occupy the same storage units in array element order. As in case(B), the result is false, if *PTR* is disassociated.

(D) If *TGT* is present and an scalar pointer, the result is true if target associated with *PTR* and the target associated with *TGT* are not 0 sized storage sequences and occupy the same storage units. The result is false, if either *TGT* or *PTR* is disassociated.

(E) If *TGT* is present and an array pointer, the result is true if target associated with *PTR* and the target associated with *TGT* have the same shape, are not 0 sized arrays, are arrays whose elements are not 0 sized storage sequences, and *TGT* and *PTR* occupy the same storage units in array element order. The result is false, if either *TGT* or *PTR* is disassociated.

Example:

```
program test_associated
   implicit none
   real, target  :: tgt(2) = (/1., 2./)
   real, pointer :: ptr(:)
   ptr => tgt
   if (associated(ptr)     .eqv. .false.) call abort
   if (associated(ptr,tgt) .eqv. .false.) call abort
end program test_associated
```

See also: Section 6.146 [NULL], page 113

6.21 `ATAN` — Arctangent function

Description:
`ATAN(X)` computes the arctangent of *X*.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = ATAN(X)`

Arguments:

X	The type shall be `REAL(*)`.

Return value:
The return value is of type `REAL(*)` and it lies in the range $-\pi/2 \leq \text{atan}(x) \leq \pi/2$.

Example:

```
program test_atan
  real(8) :: x = 2.866_8
  x = atan(x)
end program test_atan
```

Specific names:

Name	Argument	Return type	Standard
`DATAN(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

See also: Inverse function: Section 6.188 [TAN], page 132

6.22 ATAN2 — Arctangent function

Description:
: `ATAN2(Y,X)` computes the arctangent of the complex number $X + iY$.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = ATAN2(Y,X)`

Arguments:

Y	The type shall be `REAL(*)`.
X	The type and kind type parameter shall be the same as *Y*. If *Y* is zero, then *X* must be nonzero.

Return value:
: The return value has the same type and kind type parameter as *Y*. It is the principal value of the complex number $X + iY$. If *X* is nonzero, then it lies in the range $-\pi \le \mathrm{atan}(x) \le \pi$. The sign is positive if *Y* is positive. If *Y* is zero, then the return value is zero if *X* is positive and π if *X* is negative. Finally, if *X* is zero, then the magnitude of the result is $\pi/2$.

Example:

```
program test_atan2
  real(4) :: x = 1.e0_4, y = 0.5e0_4
  x = atan2(y,x)
end program test_atan2
```

Specific names:

Name	Argument	Return type	Standard
`DATAN2(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

6.23 ATANH — Hyperbolic arctangent function

Description:
: `ATANH(X)` computes the area hyperbolic sine of *X* (inverse of `TANH(X)`).

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = ATANH(X)`

Arguments:

X	The type shall be `REAL(*)` with a magnitude that is less than or equal to one.

Return value:
: The return value is of type `REAL(*)` and it lies in the range $-\infty \le \mathrm{atanh}(x) \le \infty$.

Example:

```
PROGRAM test_atanh
  REAL, DIMENSION(3) :: x = (/ -1.0, 0.0, 1.0 /)
  WRITE (*,*) ATANH(x)
END PROGRAM
```

See also: Inverse function: Section 6.189 [TANH], page 133

6.24 BESJ0 — Bessel function of the first kind of order 0

Description:
`BESJ0(X)` computes the Bessel function of the first kind of order 0 of *X*.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = BESJ0(X)`

Arguments:

X	The type shall be `REAL(*)`, and it shall be scalar.

Return value:
The return value is of type `REAL(*)` and it lies in the range $-0.4027... \leq Bessel(0,x) \leq 1$.

Example:

```
program test_besj0
  real(8) :: x = 0.0_8
  x = besj0(x)
end program test_besj0
```

Specific names:

Name	Argument	Return type	Standard
`DBESJ0(X)`	`REAL(8) X`	`REAL(8)`	GNU extension

6.25 BESJ1 — Bessel function of the first kind of order 1

Description:
`BESJ1(X)` computes the Bessel function of the first kind of order 1 of *X*.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = BESJ1(X)`

Arguments:

X	The type shall be `REAL(*)`, and it shall be scalar.

Return value:
The return value is of type `REAL(*)` and it lies in the range $-0.5818... \leq Bessel(0,x) \leq 0.5818$.

Example:

```
program test_besj1
  real(8) :: x = 1.0_8
  x = besj1(x)
end program test_besj1
```

Specific names:

Name	Argument	Return type	Standard
`DBESJ1(X)`	`REAL(8) X`	`REAL(8)`	GNU extension

6.26 BESJN — Bessel function of the first kind

Description:
BESJN(N, X) computes the Bessel function of the first kind of order *N* of *X*.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = BESJN(N, X)

Arguments:

N	The type shall be INTEGER(*), and it shall be scalar.
X	The type shall be REAL(*), and it shall be scalar.

Return value:
The return value is a scalar of type REAL(*).

Example:

```
program test_besjn
  real(8) :: x = 1.0_8
  x = besjn(5,x)
end program test_besjn
```

Specific names:

Name	Argument	Return type	Standard
DBESJN(X)	INTEGER(*) N REAL(8) X	REAL(8)	GNU extension

6.27 BESY0 — Bessel function of the second kind of order 0

Description:
BESY0(X) computes the Bessel function of the second kind of order 0 of *X*.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = BESY0(X)

Arguments:

X	The type shall be REAL(*), and it shall be scalar.

Return value:
The return value is a scalar of type REAL(*).

Example:

```
program test_besy0
  real(8) :: x = 0.0_8
  x = besy0(x)
end program test_besy0
```

Specific names:

Name	Argument	Return type	Standard
DBESY0(X)	REAL(8) X	REAL(8)	GNU extension

6.28 BESY1 — Bessel function of the second kind of order 1

Description:
BESY1(X) computes the Bessel function of the second kind of order 1 of *X*.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = BESY1(X)

Arguments:

X	The type shall be REAL(*), and it shall be scalar.

Return value:
The return value is a scalar of type REAL(*).

Example:

```
program test_besy1
  real(8) :: x = 1.0_8
  x = besy1(x)
end program test_besy1
```

Specific names:

Name	Argument	Return type	Standard
DBESY1(X)	REAL(8) X	REAL(8)	GNU extension

6.29 BESYN — Bessel function of the second kind

Description:
BESYN(N, X) computes the Bessel function of the second kind of order *N* of *X*.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = BESYN(N, X)

Arguments:

N	The type shall be INTEGER(*), and it shall be scalar.
X	The type shall be REAL(*), and it shall be scalar.

Return value:
The return value is a scalar of type REAL(*).

Example:

```
program test_besyn
  real(8) :: x = 1.0_8
  x = besyn(5,x)
end program test_besyn
```

Specific names:

Name	Argument	Return type	Standard
DBESYN(N,X)	INTEGER(*) N REAL(8) X	REAL(8)	GNU extension

6.30 BIT_SIZE — Bit size inquiry function

Description:
BIT_SIZE(I) returns the number of bits (integer precision plus sign bit) represented by the type of *I*.

Standard: F95 and later

Class: Inquiry function

Syntax: RESULT = BIT_SIZE(I)

Arguments:

I	The type shall be INTEGER(*).

Return value:
The return value is of type INTEGER(*)

Example:

```
program test_bit_size
    integer :: i = 123
    integer :: size
    size = bit_size(i)
    print *, size
end program test_bit_size
```

6.31 BTEST — Bit test function

Description:
BTEST(I,POS) returns logical .TRUE. if the bit at *POS* in *I* is set.

Standard: F95 and later

Class: Elemental function

Syntax: RESULT = BTEST(I, POS)

Arguments:

I	The type shall be INTEGER(*).
POS	The type shall be INTEGER(*).

Return value:
The return value is of type LOGICAL

Example:

```
program test_btest
    integer :: i = 32768 + 1024 + 64
    integer :: pos
    logical :: bool
    do pos=0,16
        bool = btest(i, pos)
        print *, pos, bool
    end do
end program test_btest
```

6.32 CEILING — Integer ceiling function

Description:
`CEILING(X)` returns the least integer greater than or equal to *X*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = CEILING(X [, KIND])`

Arguments:

X	The type shall be `REAL(*)`.
KIND	(Optional) An `INTEGER(*)` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER(KIND)`

Example:

```
program test_ceiling
    real :: x = 63.29
    real :: y = -63.59
    print *, ceiling(x) ! returns 64
    print *, ceiling(y) ! returns -63
end program test_ceiling
```

See also: Section 6.67 [FLOOR], page 72, Section 6.144 [NINT], page 112

6.33 CHAR — Character conversion function

Description:
`CHAR(I [, KIND])` returns the character represented by the integer *I*.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = CHAR(I [, KIND])`

Arguments:

I	The type shall be `INTEGER(*)`.
KIND	(Optional) An `INTEGER(*)` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `CHARACTER(1)`

Example:

```
program test_char
    integer :: i = 74
    character(1) :: c
    c = char(i)
    print *, i, c ! returns 'J'
end program test_char
```

Note: See Section 6.97 [ICHAR], page 87 for a discussion of converting between numerical values and formatted string representations.

See also: Section 6.5 [ACHAR], page 35, Section 6.91 [IACHAR], page 84, Section 6.97 [ICHAR], page 87

6.34 CHDIR — Change working directory

Description:
Change current working directory to a specified *PATH*.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL CHDIR(PATH [, STATUS])`

Arguments:

PATH	The type shall be `CHARACTER(*)` and shall specify a valid path within the file system.
STATUS	(Optional) status flag. Returns 0 on success, a system specific and non-zero error code otherwise.

Example:

```
PROGRAM test_chdir
  CHARACTER(len=255) :: path
  CALL getcwd(path)
  WRITE(*,*) TRIM(path)
  CALL chdir("/tmp")
  CALL getcwd(path)
  WRITE(*,*) TRIM(path)
END PROGRAM
```

See also: Section 6.81 [GETCWD], page 79

6.35 CHMOD — Change access permissions of files

Description:
`CHMOD` changes the permissions of a file. This function invokes `/bin/chmod` and might therefore not work on all platforms.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, non-elemental function

Syntax:

```
CALL CHMOD(NAME, MODE[, STATUS])
STATUS = CHMOD(NAME, MODE)
```

Arguments:

NAME	Scalar `CHARACTER` with the file name. Trailing blanks are ignored unless the character `achar(0)` is present, then all characters up to and excluding `achar(0)` are used as the file name.

MODE Scalar `CHARACTER` giving the file permission. *MODE* uses the same syntax as the *MODE* argument of `/bin/chmod`.

STATUS (optional) scalar `INTEGER`, which is `0` on success and non-zero otherwise.

Return value:
In either syntax, *STATUS* is set to `0` on success and non-zero otherwise.

Example: `CHMOD` as subroutine

```
program chmod_test
  implicit none
  integer :: status
  call chmod('test.dat','u+x',status)
  print *, 'Status: ', status
end program chmod_test
```

`CHMOD` as non-elemental function:

```
program chmod_test
  implicit none
  integer :: status
  status = chmod('test.dat','u+x')
  print *, 'Status: ', status
end program chmod_test
```

6.36 `CMPLX` — Complex conversion function

Description:
`CMPLX(X [, Y [, KIND]])` returns a complex number where *X* is converted to the real component. If *Y* is present it is converted to the imaginary component. If *Y* is not present then the imaginary component is set to 0.0. If *X* is complex then *Y* must not be present.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = CMPLX(X [, Y [, KIND]])`

Arguments:

X The type may be `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)`.

Y (Optional; only allowed if *X* is not `COMPLEX(*)`.) May be `INTEGER(*)` or `REAL(*)`.

KIND (Optional) An `INTEGER(*)` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `COMPLEX(*)`

Example:

```
program test_cmplx
    integer :: i = 42
    real :: x = 3.14
    complex :: z
    z = cmplx(i, x)
    print *, z, cmplx(x)
end program test_cmplx
```

6.37 COMMAND_ARGUMENT_COUNT — Get number of command line arguments

Description:
: COMMAND_ARGUMENT_COUNT() returns the number of arguments passed on the command line when the containing program was invoked.

Standard: F2003

Class: Inquiry function

Syntax: RESULT = COMMAND_ARGUMENT_COUNT()

Arguments:
: None

Return value:
: The return value is of type INTEGER(4)

Example:

```
program test_command_argument_count
    integer :: count
    count = command_argument_count()
    print *, count
end program test_command_argument_count
```

See also: Section 6.79 [GET_COMMAND], page 78, Section 6.80 [GET_COMMAND_ARGUMENT], page 79

6.38 CONJG — Complex conjugate function

Description:
: CONJG(Z) returns the conjugate of *Z*. If *Z* is (x, y) then the result is (x, -y)

Standard: F77 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: Z = CONJG(Z)

Arguments:

Z	The type shall be COMPLEX(*).

Return value:
: The return value is of type COMPLEX(*).

Example:

```
program test_conjg
    complex :: z = (2.0, 3.0)
    complex(8) :: dz = (2.71_8, -3.14_8)
    z= conjg(z)
    print *, z
    dz = dconjg(dz)
    print *, dz
end program test_conjg
```

Specific names:

Name	Argument	Return type	Standard
DCONJG(Z)	COMPLEX(8) Z	COMPLEX(8)	GNU extension

6.39 COS — Cosine function

Description:
`COS(X)` computes the cosine of *X*.

Standard: F77 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: `RESULT = COS(X)`

Arguments:

X	The type shall be `REAL(*)` or `COMPLEX(*)`.

Return value:
The return value is of type `REAL(*)` and it lies in the range $-1 \leq \cos(x) \leq 1$. The kind type parameter is the same as *X*.

Example:

```
program test_cos
  real :: x = 0.0
  x = cos(x)
end program test_cos
```

Specific names:

Name	Argument	Return type	Standard
`DCOS(X)`	`REAL(8) X`	`REAL(8)`	F77 and later
`CCOS(X)`	`COMPLEX(4) X`	`COMPLEX(4)`	F77 and later
`ZCOS(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	GNU extension
`CDCOS(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	GNU extension

See also: Inverse function: Section 6.6 [ACOS], page 36

6.40 COSH — Hyperbolic cosine function

Description:
`COSH(X)` computes the hyperbolic cosine of *X*.

Standard: F77 and later

Class: Elemental function

Syntax: `X = COSH(X)`

Arguments:

X	The type shall be `REAL(*)`.

Return value:
The return value is of type `REAL(*)` and it is positive $(\cosh(x) \geq 0$.

Example:

```
program test_cosh
  real(8) :: x = 1.0_8
  x = cosh(x)
end program test_cosh
```

Specific names:

Name	Argument	Return type	Standard
`DCOSH(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

See also: Inverse function: Section 6.7 [ACOSH], page 36

6.41 `COUNT` — Count function

Description:
`COUNT(MASK [, DIM])` counts the number of `.TRUE.` elements of *MASK* along the dimension of *DIM*. If *DIM* is omitted it is taken to be `1`. *DIM* is a scaler of type `INTEGER` in the range of $1/leqDIM/leqn)$ where n is the rank of *MASK*.

Standard: F95 and later

Class: transformational function

Syntax: `RESULT = COUNT(MASK [, DIM])`

Arguments:

MASK	The type shall be `LOGICAL`.
DIM	The type shall be `INTEGER`.

Return value:
The return value is of type `INTEGER` with rank equal to that of *MASK*.

Example:

```
program test_count
    integer, dimension(2,3) :: a, b
    logical, dimension(2,3) :: mask
    a = reshape( (/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /))
    b = reshape( (/ 0, 7, 3, 4, 5, 8 /), (/ 2, 3 /))
    print '(3i3)', a(1,:)
    print '(3i3)', a(2,:)
    print *
    print '(3i3)', b(1,:)
    print '(3i3)', b(2,:)
    print *
    mask = a.ne.b
    print '(3l3)', mask(1,:)
    print '(3l3)', mask(2,:)
    print *
    print '(3i3)', count(mask)
    print *
    print '(3i3)', count(mask, 1)
    print *
    print '(3i3)', count(mask, 2)
end program test_count
```

6.42 `CPU_TIME` — CPU elapsed time in seconds

Description:
Returns a `REAL` value representing the elapsed CPU time in seconds. This is useful for testing segments of code to determine execution time.

Standard: F95 and later

Class: Subroutine

Syntax: `CALL CPU_TIME(TIME)`

Arguments:

TIME	The type shall be REAL with INTENT(OUT).

Return value:
None

Example:

```
program test_cpu_time
    real :: start, finish
    call cpu_time(start)
        ! put code to test here
    call cpu_time(finish)
    print '("Time = ",f6.3," seconds.")',finish-start
end program test_cpu_time
```

6.43 CSHIFT — Circular shift function

Description:
CSHIFT(ARRAY, SHIFT [, DIM]) performs a circular shift on elements of *ARRAY* along the dimension of *DIM*. If *DIM* is omitted it is taken to be 1. *DIM* is a scaler of type INTEGER in the range of $1/leqDIM/leqn)$ where n is the rank of *ARRAY*. If the rank of *ARRAY* is one, then all elements of *ARRAY* are shifted by *SHIFT* places. If rank is greater than one, then all complete rank one sections of *ARRAY* along the given dimension are shifted. Elements shifted out one end of each rank one section are shifted back in the other end.

Standard: F95 and later

Class: transformational function

Syntax: RESULT = CSHIFT(A, SHIFT [, DIM])

Arguments:

ARRAY	May be any type, not scaler.
SHIFT	The type shall be INTEGER.
DIM	The type shall be INTEGER.

Return value:
Returns an array of same type and rank as the *ARRAY* argument.

Example:

```
program test_cshift
    integer, dimension(3,3) :: a
    a = reshape( (/ 1, 2, 3, 4, 5, 6, 7, 8, 9 /), (/ 3, 3 /))
    print '(3i3)', a(1,:)
    print '(3i3)', a(2,:)
    print '(3i3)', a(3,:)
    a = cshift(a, SHIFT=(/1, 2, -1/), DIM=2)
    print *
    print '(3i3)', a(1,:)
    print '(3i3)', a(2,:)
    print '(3i3)', a(3,:)
end program test_cshift
```

6.44 CTIME — Convert a time into a string

Description:

CTIME(T,S) converts *T*, a system time value, such as returned by TIME8(), to a string of the form 'Sat Aug 19 18:13:14 1995', and returns that string into *S*.

If CTIME is invoked as a function, it can not be invoked as a subroutine, and vice versa.

T is an INTENT(IN) INTEGER(KIND=8) variable. *S* is an INTENT(OUT) CHARACTER variable.

Standard: GNU extension

Class: Subroutine

Syntax:

CALL CTIME(T,S).
S = CTIME(T), (not recommended).

Arguments:

S	The type shall be of type CHARACTER.
T	The type shall be of type INTEGER(KIND=8).

Return value:

The converted date and time as a string.

Example:

```
program test_ctime
    integer(8) :: i
    character(len=30) :: date
    i = time8()

    ! Do something, main part of the program

    call ctime(i,date)
    print *, 'Program was started on ', date
end program test_ctime
```

See Also: Section 6.88 [GMTIME], page 82, Section 6.126 [LTIME], page 102, Section 6.190 [TIME], page 133, Section 6.191 [TIME8], page 134

6.45 DATE_AND_TIME — Date and time subroutine

Description:

DATE_AND_TIME(DATE, TIME, ZONE, VALUES) gets the corresponding date and time information from the real-time system clock. *DATE* is INTENT(OUT) and has form ccyymmdd. *TIME* is INTENT(OUT) and has form hhmmss.sss. *ZONE* is INTENT(OUT) and has form (+-)hhmm, representing the difference with respect to Coordinated Universal Time (UTC). Unavailable time and date parameters return blanks.

VALUES is INTENT(OUT) and provides the following:

VALUE(1):	The year

`VALUE(2)`:	The month
`VALUE(3)`:	The day of the month
`VALUE(4)`:	Time difference with UTC in minutes
`VALUE(5)`:	The hour of the day
`VALUE(6)`:	The minutes of the hour
`VALUE(7)`:	The seconds of the minute
`VALUE(8)`:	The milliseconds of the second

Standard: F95 and later

Class: Subroutine

Syntax: `CALL DATE_AND_TIME([DATE, TIME, ZONE, VALUES])`

Arguments:

DATE	(Optional) The type shall be `CHARACTER(8)` or larger.
TIME	(Optional) The type shall be `CHARACTER(10)` or larger.
ZONE	(Optional) The type shall be `CHARACTER(5)` or larger.
VALUES	(Optional) The type shall be `INTEGER(8)`.

Return value:
None

Example:

```
program test_time_and_date
    character(8)  :: date
    character(10) :: time
    character(5)  :: zone
    integer,dimension(8) :: values
    ! using keyword arguments
    call date_and_time(date,time,zone,values)
    call date_and_time(DATE=date,ZONE=zone)
    call date_and_time(TIME=time)
    call date_and_time(VALUES=values)
    print '(a,2x,a,2x,a)', date, time, zone
    print '(8i5))', values
end program test_time_and_date
```

6.46 `DBLE` — Double conversion function

Description:
`DBLE(X)` Converts *X* to double precision real type.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = DBLE(X)`

Arguments:

X	The type shall be `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)`.

Return value:
The return value is of type double precision real.

Example:

```
program test_dble
    real    :: x = 2.18
    integer :: i = 5
    complex :: z = (2.3,1.14)
    print *, dble(x), dble(i), dble(z)
end program test_dble
```

See also: Section 6.48 [DFLOAT], page 60, Section 6.64 [FLOAT], page 70, Section 6.159 [REAL], page 118

6.47 DCMPLX — Double complex conversion function

Description:
`DCMPLX(X [,Y])` returns a double complex number where *X* is converted to the real component. If *Y* is present it is converted to the imaginary component. If *Y* is not present then the imaginary component is set to 0.0. If *X* is complex then *Y* must not be present.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = DCMPLX(X [, Y])`

Arguments:

X	The type may be `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)`.
Y	(Optional if *X* is not `COMPLEX(*)`.) May be `INTEGER(*)` or `REAL(*)`.

Return value:
The return value is of type `COMPLEX(8)`

Example:

```
program test_dcmplx
    integer :: i = 42
    real :: x = 3.14
    complex :: z
    z = cmplx(i, x)
    print *, dcmplx(i)
    print *, dcmplx(x)
    print *, dcmplx(z)
    print *, dcmplx(x,i)
end program test_dcmplx
```

6.48 DFLOAT — Double conversion function

Description:
`DFLOAT(X)` Converts *X* to double precision real type.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = DFLOAT(X)`

Arguments:

X	The type shall be `INTEGER(*)`.

Return value:
The return value is of type double precision real.

Example:

```
program test_dfloat
    integer :: i = 5
    print *, dfloat(i)
end program test_dfloat
```

See also: Section 6.46 [DBLE], page 59, Section 6.64 [FLOAT], page 70, Section 6.159 [REAL], page 118

6.49 DIGITS — Significant digits function

Description:
DIGITS(X) returns the number of significant digits of the internal model representation of *X*. For example, on a system using a 32-bit floating point representation, a default real number would likely return 24.

Standard: F95 and later

Class: Inquiry function

Syntax: RESULT = DIGITS(X)

Arguments:

X	The type may be INTEGER(*) or REAL(*).

Return value:
The return value is of type INTEGER.

Example:

```
program test_digits
    integer :: i = 12345
    real :: x = 3.143
    real(8) :: y = 2.33
    print *, digits(i)
    print *, digits(x)
    print *, digits(y)
end program test_digits
```

6.50 DIM — Dim function

Description:
DIM(X,Y) returns the difference X-Y if the result is positive; otherwise returns zero.

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = DIM(X, Y)

Arguments:

X	The type shall be INTEGER(*) or REAL(*)
Y	The type shall be the same type and kind as *X*.

Return value:
The return value is of type INTEGER(*) or REAL(*).

Example:

```
program test_dim
    integer :: i
    real(8) :: x
    i = dim(4, 15)
    x = dim(4.345_8, 2.111_8)
    print *, i
    print *, x
end program test_dim
```

Specific names:

Name	Argument	Return type	Standard
IDIM(X,Y)	INTEGER(4) X,Y	INTEGER(4)	F77 and later
DDIM(X,Y)	REAL(8) X,Y	REAL(8)	F77 and later

6.51 DOT_PRODUCT — Dot product function

Description:
DOT_PRODUCT(X,Y) computes the dot product multiplication of two vectors *X* and *Y*. The two vectors may be either numeric or logical and must be arrays of rank one and of equal size. If the vectors are INTEGER(*) or REAL(*), the result is SUM(X*Y). If the vectors are COMPLEX(*), the result is SUM(CONJG(X)*Y). If the vectors are LOGICAL, the result is ANY(X.AND.Y).

Standard: F95 and later

Class: transformational function

Syntax: RESULT = DOT_PRODUCT(X, Y)

Arguments:

X	The type shall be numeric or LOGICAL, rank 1.
Y	The type shall be numeric or LOGICAL, rank 1.

Return value:
If the arguments are numeric, the return value is a scaler of numeric type, INTEGER(*), REAL(*), or COMPLEX(*). If the arguments are LOGICAL, the return value is .TRUE. or .FALSE..

Example:

```
program test_dot_prod
    integer, dimension(3) :: a, b
    a = (/ 1, 2, 3 /)
    b = (/ 4, 5, 6 /)
    print '(3i3)', a
    print *
    print '(3i3)', b
    print *
    print *, dot_product(a,b)
end program test_dot_prod
```

6.52 DPROD — Double product function

Description:
DPROD(X,Y) returns the product X*Y.

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = DPROD(X, Y)

Arguments:

X	The type shall be REAL.
Y	The type shall be REAL.

Return value:
The return value is of type REAL(8).

Example:

```
program test_dprod
    integer :: i
    real :: x = 5.2
    real :: y = 2.3
    real(8) :: d
    d = dprod(x,y)
    print *, d
end program test_dprod
```

6.53 DREAL — Double real part function

Description:
DREAL(Z) returns the real part of complex variable *Z*.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = DREAL(Z)

Arguments:

Z	The type shall be COMPLEX(8).

Return value:
The return value is of type REAL(8).

Example:

```
program test_dreal
    complex(8) :: z = (1.3_8,7.2_8)
    print *, dreal(z)
end program test_dreal
```

See also: Section 6.10 [AIMAG], page 38

6.54 DTIME — Execution time subroutine (or function)

Description:

DTIME(TARRAY, RESULT) initially returns the number of seconds of runtime since the start of the process's execution in *RESULT*. *TARRAY* returns the user and system components of this time in TARRAY(1) and TARRAY(2) respectively. *RESULT* is equal to TARRAY(1) + TARRAY(2).

Subsequent invocations of DTIME return values accumulated since the previous invocation.

On some systems, the underlying timings are represented using types with sufficiently small limits that overflows (wrap around) are possible, such as 32-bit types. Therefore, the values returned by this intrinsic might be, or become, negative, or numerically less than previous values, during a single run of the compiled program.

If DTIME is invoked as a function, it can not be invoked as a subroutine, and vice versa.

TARRAY and *RESULT* are INTENT(OUT) and provide the following:

TARRAY(1):	User time in seconds.
TARRAY(2):	System time in seconds.
RESULT:	Run time since start in seconds.

Standard: GNU extension

Class: Subroutine

Syntax:

CALL DTIME(TARRAY, RESULT).
RESULT = DTIME(TARRAY), (not recommended).

Arguments:

TARRAY	The type shall be REAL, DIMENSION(2).
RESULT	The type shall be REAL.

Return value:

Elapsed time in seconds since the start of program execution.

Example:

```
program test_dtime
    integer(8) :: i, j
    real, dimension(2) :: tarray
    real :: result
    call dtime(tarray, result)
    print *, result
    print *, tarray(1)
    print *, tarray(2)
    do i=1,100000000    ! Just a delay
        j = i * i - i
    end do
    call dtime(tarray, result)
    print *, result
    print *, tarray(1)
    print *, tarray(2)
end program test_dtime
```

6.55 EOSHIFT — End-off shift function

Description:

EOSHIFT(ARRAY, SHIFT[,BOUNDARY, DIM]) performs an end-off shift on elements of *ARRAY* along the dimension of *DIM*. If *DIM* is omitted it is taken to be 1. *DIM* is a scaler of type INTEGER in the range of $1/leqDIM/leqn$) where n is the rank of *ARRAY*. If the rank of *ARRAY* is one, then all elements of *ARRAY* are shifted by *SHIFT* places. If rank is greater than one, then all complete rank one sections of *ARRAY* along the given dimension are shifted. Elements shifted out one end of each rank one section are dropped. If *BOUNDARY* is present then the corresponding value of from *BOUNDARY* is copied back in the other end. If *BOUNDARY* is not present then the following are copied in depending on the type of *ARRAY*.

Array Type	*Boundary Value*
Numeric	0 of the type and kind of *ARRAY*.
Logical	.FALSE..
Character(*len*)	*len* blanks.

Standard: F95 and later

Class: transformational function

Syntax: RESULT = EOSHIFT(A, SHIFT [, BOUNDARY, DIM])

Arguments:

ARRAY	May be any type, not scaler.
SHIFT	The type shall be INTEGER.
BOUNDARY	Same type as *ARRAY*.
DIM	The type shall be INTEGER.

Return value:

Returns an array of same type and rank as the *ARRAY* argument.

Example:

```
program test_eoshift
    integer, dimension(3,3) :: a
    a = reshape( (/ 1, 2, 3, 4, 5, 6, 7, 8, 9 /), (/ 3, 3 /))
    print '(3i3)', a(1,:)
    print '(3i3)', a(2,:)
    print '(3i3)', a(3,:)
    a = EOSHIFT(a, SHIFT=(/1, 2, 1/), BOUNDARY=-5, DIM=2)
    print *
    print '(3i3)', a(1,:)
    print '(3i3)', a(2,:)
    print '(3i3)', a(3,:)
end program test_eoshift
```

6.56 EPSILON — Epsilon function

Description:

EPSILON(X) returns a nearly negligible number relative to 1.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = EPSILON(X)`

Arguments:

X	The type shall be `REAL(*)`.

Return value:
The return value is of same type as the argument.

Example:

```
program test_epsilon
    real :: x = 3.143
    real(8) :: y = 2.33
    print *, EPSILON(x)
    print *, EPSILON(y)
end program test_epsilon
```

6.57 `ERF` — Error function

Description:
`ERF(X)` computes the error function of *X*.

Standard: GNU Extension

Class: Elemental function

Syntax: `RESULT = ERF(X)`

Arguments:

X	The type shall be `REAL(*)`, and it shall be scalar.

Return value:
The return value is a scalar of type `REAL(*)` and it is positive ($-1 \leq erf(x) \leq 1$.

Example:

```
program test_erf
  real(8) :: x = 0.17_8
  x = erf(x)
end program test_erf
```

Specific names:

Name	Argument	Return type	Standard
`DERF(X)`	`REAL(8) X`	`REAL(8)`	GNU extension

6.58 `ERFC` — Error function

Description:
`ERFC(X)` computes the complementary error function of *X*.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = ERFC(X)`

Arguments:

X	The type shall be `REAL(*)`, and it shall be scalar.

Return value:
The return value is a scalar of type `REAL(*)` and it is positive ($0 \leq erfc(x) \leq 2$.

Example:

```
program test_erfc
  real(8) :: x = 0.17_8
  x = erfc(x)
end program test_erfc
```

Specific names:

Name	Argument	Return type	Standard
`DERFC(X)`	`REAL(8) X`	`REAL(8)`	GNU extension

6.59 `ETIME` — Execution time subroutine (or function)

Description:
`ETIME(TARRAY, RESULT)` returns the number of seconds of runtime since the start of the process's execution in *RESULT*. *TARRAY* returns the user and system components of this time in `TARRAY(1)` and `TARRAY(2)` respectively. *RESULT* is equal to `TARRAY(1) + TARRAY(2)`.

On some systems, the underlying timings are represented using types with sufficiently small limits that overflows (wrap around) are possible, such as 32-bit types. Therefore, the values returned by this intrinsic might be, or become, negative, or numerically less than previous values, during a single run of the compiled program.

If `ETIME` is invoked as a function, it can not be invoked as a subroutine, and vice versa.

TARRAY and *RESULT* are `INTENT(OUT)` and provide the following:

`TARRAY(1)`:	User time in seconds.
`TARRAY(2)`:	System time in seconds.
`RESULT`:	Run time since start in seconds.

Standard: GNU extension

Class: Subroutine

Syntax:
`CALL ETIME(TARRAY, RESULT)`.
`RESULT = ETIME(TARRAY)`, (not recommended).

Arguments:

TARRAY	The type shall be `REAL, DIMENSION(2)`.
RESULT	The type shall be `REAL`.

Return value:
Elapsed time in seconds since the start of program execution.

Example:

```
program test_etime
    integer(8) :: i, j
    real, dimension(2) :: tarray
    real :: result
```

```
        call ETIME(tarray, result)
        print *, result
        print *, tarray(1)
        print *, tarray(2)
        do i=1,100000000    ! Just a delay
            j = i * i - i
        end do
        call ETIME(tarray, result)
        print *, result
        print *, tarray(1)
        print *, tarray(2)
    end program test_etime
```

See also: Section 6.42 [CPU_TIME], page 56

6.60 EXIT — Exit the program with status.

Description:
EXIT causes immediate termination of the program with status. If status is omitted it returns the canonical *success* for the system. All Fortran I/O units are closed.

Standard: GNU extension

Class: Subroutine

Syntax: CALL EXIT([STATUS])

Arguments:

STATUS	The type of the argument shall be INTEGER(*).

Return value:
STATUS is passed to the parent process on exit.

Example:

```
    program test_exit
      integer :: STATUS = 0
      print *, 'This program is going to exit.'
      call EXIT(STATUS)
    end program test_exit
```

See also: Section 6.2 [ABORT], page 33, Section 6.109 [KILL], page 93

6.61 EXP — Exponential function

Description:
EXP(X) computes the base e exponential of X.

Standard: F77 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: RESULT = EXP(X)

Arguments:

X	The type shall be REAL(*) or COMPLEX(*).

Return value:
The return value has same type and kind as *X*.

Example:

```
program test_exp
  real :: x = 1.0
  x = exp(x)
end program test_exp
```

Specific names:

Name	Argument	Return type	Standard
DEXP(X)	REAL(8) X	REAL(8)	F77 and later
CEXP(X)	COMPLEX(4) X	COMPLEX(4)	F77 and later
ZEXP(X)	COMPLEX(8) X	COMPLEX(8)	GNU extension
CDEXP(X)	COMPLEX(8) X	COMPLEX(8)	GNU extension

6.62 EXPONENT — Exponent function

Description:
EXPONENT(X) returns the value of the exponent part of *X*. If *X* is zero the value returned is zero.

Standard: F95 and later

Class: Elemental function

Syntax: RESULT = EXPONENT(X)

Arguments:

X	The type shall be REAL(*).

Return value:
The return value is of type default INTEGER.

Example:

```
program test_exponent
  real :: x = 1.0
  integer :: i
  i = exponent(x)
  print *, i
  print *, exponent(0.0)
end program test_exponent
```

6.63 FDATE — Get the current time as a string

Description:
FDATE(DATE) returns the current date (using the same format as CTIME) in *DATE*. It is equivalent to CALL CTIME(DATE, TIME()).

If FDATE is invoked as a function, it can not be invoked as a subroutine, and vice versa.

DATE is an INTENT(OUT) CHARACTER variable.

Standard: GNU extension

Class: Subroutine

Syntax:

CALL FDATE(DATE).
DATE = FDATE(), (not recommended).

Arguments:

DATE	The type shall be of type CHARACTER.

Return value:

The current date as a string.

Example:

```
program test_fdate
    integer(8) :: i, j
    character(len=30) :: date
    call fdate(date)
    print *, 'Program started on ', date
    do i = 1, 100000000 ! Just a delay
        j = i * i - i
    end do
    call fdate(date)
    print *, 'Program ended on ', date
end program test_fdate
```

6.64 FLOAT — Convert integer to default real

Description:

FLOAT(I) converts the integer *I* to a default real value.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = FLOAT(I)

Arguments:

I	The type shall be INTEGER(*).

Return value:

The return value is of type default REAL.

Example:

```
program test_float
    integer :: i = 1
    if (float(i) /= 1.) call abort
end program test_float
```

See also: Section 6.46 [DBLE], page 59, Section 6.48 [DFLOAT], page 60, Section 6.159 [REAL], page 118

6.65 FGET — Read a single character in stream mode from stdin

Description:

Read a single character in stream mode from stdin by bypassing normal formatted output. Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

This intrinsic routine is provided for backwards compatibility with g77. GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability. See also Chapter 4 [Fortran 2003 status], page 23.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL FGET(C [, STATUS])`

Arguments:

C	The type shall be `CHARACTER`.
STATUS	(Optional) status flag of type `INTEGER`. Returns 0 on success, -1 on end-of-file, and a system specific positive error code otherwise.

Example:

```
PROGRAM test_fget
  INTEGER, PARAMETER :: strlen = 100
  INTEGER :: status, i = 1
  CHARACTER(len=strlen) :: str = ""

  WRITE (*,*) 'Enter text:'
  DO
    CALL fget(str(i:i), status)
    if (status /= 0 .OR. i > strlen) exit
    i = i + 1
  END DO
  WRITE (*,*) TRIM(str)
END PROGRAM
```

See also: Section 6.66 [FGETC], page 71, Section 6.70 [FPUT], page 73, Section 6.71 [FPUTC], page 74

6.66 FGETC — Read a single character in stream mode

Description:

Read a single character in stream mode by bypassing normal formatted output. Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

This intrinsic routine is provided for backwards compatibility with g77. GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability. See also Chapter 4 [Fortran 2003 status], page 23.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL FGETC(UNIT, C [, STATUS])`

Arguments:

UNIT	The type shall be `INTEGER`.
C	The type shall be `CHARACTER`.

STATUS (Optional) status flag of type `INTEGER`. Returns 0 on success, -1 on end-of-file and a system specific positive error code otherwise.

Example:

```
PROGRAM test_fgetc
  INTEGER :: fd = 42, status
  CHARACTER :: c

  OPEN(UNIT=fd, FILE="/etc/passwd", ACTION="READ", STATUS = "OLD")
  DO
    CALL fgetc(fd, c, status)
    IF (status /= 0) EXIT
    call fput(c)
  END DO
  CLOSE(UNIT=fd)
END PROGRAM
```

See also: Section 6.65 [FGET], page 70, Section 6.70 [FPUT], page 73, Section 6.71 [FPUTC], page 74

6.67 `FLOOR` — Integer floor function

Description:
`FLOOR(X)` returns the greatest integer less than or equal to *X*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = FLOOR(X [, KIND])`

Arguments:

X	The type shall be `REAL(*)`.
KIND	(Optional) An `INTEGER(*)` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER(KIND)`

Example:

```
program test_floor
    real :: x = 63.29
    real :: y = -63.59
    print *, floor(x) ! returns 63
    print *, floor(y) ! returns -64
end program test_floor
```

See also: Section 6.32 [CEILING], page 51, Section 6.144 [NINT], page 112

6.68 `FLUSH` — Flush I/O unit(s)

Description:
Flushes Fortran unit(s) currently open for output. Without the optional argument, all units are flushed, otherwise just the unit specified.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL FLUSH(UNIT)`

Arguments:

UNIT	(Optional) The type shall be `INTEGER`.

Note: Beginning with the Fortran 2003 standard, there is a `FLUSH` statement that should be preferred over the `FLUSH` intrinsic.

6.69 `FNUM` — File number function

Description:
`FNUM(UNIT)` returns the POSIX file descriptor number corresponding to the open Fortran I/O unit `UNIT`.

Standard: GNU extension

Class: Non-elemental function

Syntax: `RESULT = FNUM(UNIT)`

Arguments:

UNIT	The type shall be `INTEGER`.

Return value:
The return value is of type `INTEGER`

Example:

```
program test_fnum
  integer :: i
  open (unit=10, status = "scratch")
  i = fnum(10)
  print *, i
  close (10)
end program test_fnum
```

6.70 `FPUT` — Write a single character in stream mode to stdout

Description:
Write a single character in stream mode to stdout by bypassing normal formatted output. Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

This intrinsic routine is provided for backwards compatibility with `g77`. GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability. See also Chapter 4 [Fortran 2003 status], page 23.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL FPUT(C [, STATUS])`

Arguments:

C	The type shall be `CHARACTER`.
STATUS	(Optional) status flag of type `INTEGER`. Returns 0 on success, -1 on end-of-file and a system specific positive error code otherwise.

Example:

```
PROGRAM test_fput
  CHARACTER(len=*) :: str = "gfortran"
  INTEGER :: i
  DO i = 1, len_trim(str)
    CALL fput(str(i:i))
  END DO
END PROGRAM
```

See also: Section 6.71 [FPUTC], page 74, Section 6.65 [FGET], page 70, Section 6.66 [FGETC], page 71

6.71 `FPUTC` — Write a single character in stream mode

Description:

Write a single character in stream mode by bypassing normal formatted output. Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

This intrinsic routine is provided for backwards compatibility with `g77`. GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability. See also Chapter 4 [Fortran 2003 status], page 23.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL FPUTC(UNIT, C [, STATUS])`

Arguments:

UNIT	The type shall be `INTEGER`.
C	The type shall be `CHARACTER`.
STATUS	(Optional) status flag of type `INTEGER`. Returns 0 on success, -1 on end-of-file and a system specific positive error code otherwise.

Example:

```
PROGRAM test_fputc
  CHARACTER(len=*) :: str = "gfortran"
  INTEGER :: fd = 42, i

  OPEN(UNIT = fd, FILE = "out", ACTION = "WRITE", STATUS="NEW")
  DO i = 1, len_trim(str)
    CALL fputc(fd, str(i:i))
  END DO
  CLOSE(fd)
END PROGRAM
```

See also: Section 6.70 [FPUT], page 73, Section 6.65 [FGET], page 70, Section 6.66 [FGETC], page 71

6.72 FRACTION — Fractional part of the model representation

Description:
`FRACTION(X)` returns the fractional part of the model representation of `X`.

Standard: F95 and later

Class: Elemental function

Syntax: `Y = FRACTION(X)`

Arguments:

X	The type of the argument shall be a `REAL`.

Return value:
The return value is of the same type and kind as the argument. The fractional part of the model representation of `X` is returned; it is `X * RADIX(X)**(-EXPONENT(X))`.

Example:

```
program test_fraction
  real :: x
  x = 178.1387e-4
  print *, fraction(x), x * radix(x)**(-exponent(x))
end program test_fraction
```

6.73 FREE — Frees memory

Description:
Frees memory previously allocated by `MALLOC()`. The `FREE` intrinsic is an extension intended to be used with Cray pointers, and is provided in GNU Fortran to allow user to compile legacy code. For new code using Fortran 95 pointers, the memory de-allocation intrinsic is `DEALLOCATE`.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL FREE(PTR)`

Arguments:

PTR	The type shall be `INTEGER`. It represents the location of the memory that should be de-allocated.

Return value:
None

Example: See `MALLOC` for an example.

See also: Section 6.127 [MALLOC], page 102

6.74 FSEEK — Low level file positioning subroutine

Not yet implemented in GNU Fortran.

Description:

Standard: GNU extension

Class: Subroutine

Syntax:

Arguments:

Return value:

Example:

Specific names:

See also: g77 features lacking in gfortran

6.75 FSTAT — Get file status

Description:

`FSTAT` is identical to Section 6.183 [STAT], page 129, except that information about an already opened file is obtained.

The elements in `BUFF` are the same as described by Section 6.183 [STAT], page 129.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL FSTAT(UNIT, BUFF [, STATUS])`

Arguments:

UNIT	An open I/O unit number of type `INTEGER`.
BUFF	The type shall be `INTEGER(4), DIMENSION(13)`.
STATUS	(Optional) status flag of type `INTEGER(4)`. Returns 0 on success and a system specific error code otherwise.

Example: See Section 6.183 [STAT], page 129 for an example.

See also: To stat a link: Section 6.125 [LSTAT], page 101, to stat a file: Section 6.183 [STAT], page 129

6.76 FTELL — Current stream position

Description:

Retrieves the current position within an open file.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, function

Syntax:

`CALL FTELL(UNIT, OFFSET)`
`OFFSET = FTELL(UNIT)`

Arguments:

OFFSET	Shall of type `INTEGER`.
UNIT	Shall of type `INTEGER`.

Return value:
In either syntax, *OFFSET* is set to the current offset of unit number *UNIT*, or to -1 if the unit is not currently open.

Example:

```
PROGRAM test_ftell
  INTEGER :: i
  OPEN(10, FILE="temp.dat")
  CALL ftell(10,i)
  WRITE(*,*) i
END PROGRAM
```

See also: Section 6.74 [FSEEK], page 76

6.77 `GERROR` — Get last system error message

Description:
Returns the system error message corresponding to the last system error. This resembles the functionality of `strerror(3)` in C.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL GERROR(RESULT)`

Arguments:

RESULT	Shall of type `CHARACTER(*)`.

Example:

```
PROGRAM test_gerror
  CHARACTER(len=100) :: msg
  CALL gerror(msg)
  WRITE(*,*) msg
END PROGRAM
```

See also: Section 6.100 [IERRNO], page 89, Section 6.149 [PERROR], page 115

6.78 `GETARG` — Get command line arguments

Description:
Retrieve the Nth argument that was passed on the command line when the containing program was invoked.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 6.80 [GET_COMMAND_ARGUMENT], page 79 intrinsic defined by the Fortran 2003 standard.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL GETARG(N, ARG)`

Arguments:

N	Shall be of type `INTEGER(4)`, $N \geq 0$
ARG	Shall be of type `CHARACTER(*)`.

Return value:
After `GETARG` returns, the *ARG* argument holds the *N*th command line argument. If *ARG* can not hold the argument, it is truncated to fit the length of *ARG*. If there are less than *N* arguments specified at the command line, *ARG* will be filled with blanks. If $N = 0$, *ARG* is set to the name of the program (on systems that support this feature).

Example:

```
PROGRAM test_getarg
  INTEGER :: i
  CHARACTER(len=32) :: arg

  DO i = 1, iargc()
    CALL getarg(i, arg)
    WRITE (*,*) arg
  END DO
END PROGRAM
```

See also: GNU Fortran 77 compatibility function: Section 6.93 [IARGC], page 85

F2003 functions and subroutines: Section 6.79 [GET_COMMAND], page 78, Section 6.80 [GET_COMMAND_ARGUMENT], page 79, Section 6.37 [COMMAND_ARGUMENT_COUNT], page 54

6.79 `GET_COMMAND` — Get the entire command line

Description:
Retrieve the entire command line that was used to invoke the program.

Standard: F2003

Class: Subroutine

Syntax: `CALL GET_COMMAND(CMD)`

Arguments:

CMD	Shall be of type `CHARACTER(*)`.

Return value:
Stores the entire command line that was used to invoke the program in *ARG*. If *ARG* is not large enough, the command will be truncated.

Example:

```
PROGRAM test_get_command
  CHARACTER(len=255) :: cmd
  CALL get_command(cmd)
  WRITE (*,*) TRIM(cmd)
END PROGRAM
```

See also: Section 6.80 [GET_COMMAND_ARGUMENT], page 79, Section 6.37 [COMMAND_ARGUMENT_COUNT], page 54

6.80 GET_COMMAND_ARGUMENT — Get command line arguments

Description:
Retrieve the Nth argument that was passed on the command line when the containing program was invoked.

Standard: F2003

Class: Subroutine

Syntax: `CALL GET_COMMAND_ARGUMENT(N, ARG)`

Arguments:

N	Shall be of type `INTEGER(4)`, $N \geq 0$
ARG	Shall be of type `CHARACTER(*)`.

Return value:
After `GET_COMMAND_ARGUMENT` returns, the *ARG* argument holds the Nth command line argument. If *ARG* can not hold the argument, it is truncated to fit the length of *ARG*. If there are less than N arguments specified at the command line, *ARG* will be filled with blanks. If $N = 0$, *ARG* is set to the name of the program (on systems that support this feature).

Example:

```
PROGRAM test_get_command_argument
  INTEGER :: i
  CHARACTER(len=32) :: arg

  i = 0
  DO
    CALL get_command_argument(i, arg)
    IF (LEN_TRIM(arg) == 0) EXIT

    WRITE (*,*) TRIM(arg)
    i = i+1
  END DO
END PROGRAM
```

See also: Section 6.79 [GET_COMMAND], page 78, Section 6.37 [COMMAND_ARGUMENT_COUNT], page 54

6.81 GETCWD — Get current working directory

Description:
Get current working directory.

Standard: GNU extension

Class: Non-elemental subroutine.

Syntax: `CALL GETCWD(CWD [, STATUS])`

Arguments:

CWD	The type shall be `CHARACTER(*)`.
STATUS	(Optional) status flag. Returns 0 on success, a system specific and non-zero error code otherwise.

Example:

```
PROGRAM test_getcwd
  CHARACTER(len=255) :: cwd
  CALL getcwd(cwd)
  WRITE(*,*) TRIM(cwd)
END PROGRAM
```

See also: Section 6.34 [CHDIR], page 52

6.82 `GETENV` — Get an environmental variable

Description:

Get the *VALUE* of the environmental variable *ENVVAR*.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 6.83 [GET_ENVIRONMENT_VARIABLE], page 80 intrinsic defined by the Fortran 2003 standard.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL GETENV(ENVVAR, VALUE)`

Arguments:

ENVVAR	Shall be of type `CHARACTER(*)`.
VALUE	Shall be of type `CHARACTER(*)`.

Return value:

Stores the value of *ENVVAR* in *VALUE*. If *VALUE* is not large enough to hold the data, it is truncated. If *ENVVAR* is not set, *VALUE* will be filled with blanks.

Example:

```
PROGRAM test_getenv
  CHARACTER(len=255) :: homedir
  CALL getenv("HOME", homedir)
  WRITE (*,*) TRIM(homedir)
END PROGRAM
```

See also: Section 6.83 [GET_ENVIRONMENT_VARIABLE], page 80

6.83 `GET_ENVIRONMENT_VARIABLE` — Get an environmental variable

Description:

Get the *VALUE* of the environmental variable *ENVVAR*.

Standard: F2003

Class: Subroutine

Syntax: `CALL GET_ENVIRONMENT_VARIABLE(ENVVAR, VALUE)`

Arguments:

ENVVAR	Shall be of type `CHARACTER(*)`.
VALUE	Shall be of type `CHARACTER(*)`.

Return value:
Stores the value of *ENVVAR* in *VALUE*. If *VALUE* is not large enough to hold the data, it is truncated. If *ENVVAR* is not set, *VALUE* will be filled with blanks.

Example:

```
PROGRAM test_getenv
  CHARACTER(len=255) :: homedir
  CALL get_environment_variable("HOME", homedir)
  WRITE (*,*) TRIM(homedir)
END PROGRAM
```

6.84 `GETGID` — Group ID function

Description:
Returns the numerical group ID of the current process.

Standard: GNU extension

Class: function

Syntax: `RESULT = GETGID()`

Return value:
The return value of `GETGID` is an `INTEGER` of the default kind.

Example: See `GETPID` for an example.

See also: Section 6.86 [GETPID], page 82, Section 6.87 [GETUID], page 82

6.85 `GETLOG` — Get login name

Description:
Gets the username under which the program is running.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL GETLOG(LOGIN)`

Arguments:

LOGIN	Shall be of type `CHARACTER(*)`.

Return value:
Stores the current user name in *LOGIN*. (On systems where the `getlogin(3)` function is not implemented, this will return a blank string.)

Example:

```
PROGRAM TEST_GETLOG
  CHARACTER(32) :: login
  CALL GETLOG(login)
  WRITE(*,*) login
END PROGRAM
```

See also: Section 6.87 [GETUID], page 82

6.86 GETPID — Process ID function

Description:
Returns the numerical process identifier of the current process.

Standard: GNU extension

Class: function

Syntax: `RESULT = GETPID()`

Return value:
The return value of `GETPID` is an `INTEGER` of the default kind.

Example:

```
program info
  print *, "The current process ID is ", getpid()
  print *, "Your numerical user ID is ", getuid()
  print *, "Your numerical group ID is ", getgid()
end program info
```

See also: Section 6.84 [GETGID], page 81, Section 6.87 [GETUID], page 82

6.87 GETUID — User ID function

Description:
Returns the numerical user ID of the current process.

Standard: GNU extension

Class: function

Syntax: `RESULT = GETUID()`

Return value:
The return value of `GETUID` is an `INTEGER` of the default kind.

Example: See `GETPID` for an example.

See also: Section 6.86 [GETPID], page 82, Section 6.85 [GETLOG], page 81

6.88 GMTIME — Convert time to GMT info

Description:
Given a system time value *STIME* (as provided by the `TIME8()` intrinsic), fills *TARRAY* with values extracted from it appropriate to the UTC time zone (Universal Coordinated Time, also known in some countries as GMT, Greenwich Mean Time), using `gmtime(3)`.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL GMTIME(STIME, TARRAY)`

Arguments:

STIME	An `INTEGER(*)` scalar expression corresponding to a system time, with `INTENT(IN)`.
TARRAY	A default `INTEGER` array with 9 elements, with `INTENT(OUT)`.

Return value:

The elements of *TARRAY* are assigned as follows:

1. Seconds after the minute, range 0–59 or 0–61 to allow for leap seconds
2. Minutes after the hour, range 0–59
3. Hours past midnight, range 0–23
4. Day of month, range 0–31
5. Number of months since January, range 0–12
6. Years since 1900
7. Number of days since Sunday, range 0–6
8. Days since January 1
9. Daylight savings indicator: positive if daylight savings is in effect, zero if not, and negative if the information is not available.

See also: Section 6.44 [CTIME], page 58, Section 6.126 [LTIME], page 102, Section 6.190 [TIME], page 133, Section 6.191 [TIME8], page 134

6.89 `HOSTNM` — Get system host name

Description:

Retrieves the host name of the system on which the program is running.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, function

Syntax:

```
CALL HOSTNM(NAME[, STATUS])
STATUS = HOSTNM(NAME)
```

Arguments:

NAME	Shall of type `CHARACTER(*)`.
STATUS	(Optional) status flag of type `INTEGER`. Returns 0 on success, or a system specific error code otherwise.

Return value:

In either syntax, *NAME* is set to the current hostname if it can be obtained, or to a blank string otherwise.

6.90 HUGE — Largest number of a kind

Description:
: `HUGE(X)` returns the largest number that is not an infinity in the model of the type of `X`.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = HUGE(X)`

Arguments:

X	Shall be of type `REAL` or `INTEGER`.

Return value:
: The return value is of the same type and kind as *X*

Example:

```
program test_huge_tiny
  print *, huge(0), huge(0.0), huge(0.0d0)
  print *, tiny(0.0), tiny(0.0d0)
end program test_huge_tiny
```

6.91 IACHAR — Code in ASCII collating sequence

Description:
: `IACHAR(C)` returns the code for the ASCII character in the first character position of `C`.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = IACHAR(C)`

Arguments:

C	Shall be a scalar `CHARACTER`, with `INTENT(IN)`

Return value:
: The return value is of type `INTEGER` and of the default integer kind.

Example:

```
program test_iachar
  integer i
  i = iachar(' ')
end program test_iachar
```

Note: See Section 6.97 [ICHAR], page 87 for a discussion of converting between numerical values and formatted string representations.

See also: Section 6.5 [ACHAR], page 35, Section 6.33 [CHAR], page 51, Section 6.97 [ICHAR], page 87

6.92 IAND — Bitwise logical and

Description:
Bitwise logical AND.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = IAND(I, J)`

Arguments:

I	The type shall be `INTEGER(*)`.
J	The type shall be `INTEGER(*)`, of the same kind as *I*. (As a GNU extension, different kinds are also permitted.)

Return value:
The return type is `INTEGER(*)`, of the same kind as the arguments. (If the argument kinds differ, it is of the same kind as the larger argument.)

Example:

```
PROGRAM test_iand
  INTEGER :: a, b
  DATA a / Z'F' /, b / Z'3' /
  WRITE (*,*) IAND(a, b)
END PROGRAM
```

See also: Section 6.103 [IOR], page 90, Section 6.99 [IEOR], page 88, Section 6.95 [IBITS], page 86, Section 6.96 [IBSET], page 87, Section 6.94 [IBCLR], page 86, Section 6.145 [NOT], page 113

6.93 IARGC — Get the number of command line arguments

Description:
`IARGC()` returns the number of arguments passed on the command line when the containing program was invoked.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 6.37 [COMMAND_ARGUMENT_COUNT], page 54 intrinsic defined by the Fortran 2003 standard.

Standard: GNU extension

Class: Non-elemental Function

Syntax: `RESULT = IARGC()`

Arguments:
None.

Return value:
The number of command line arguments, type `INTEGER(4)`.

Example: See Section 6.78 [GETARG], page 77

See also: GNU Fortran 77 compatibility subroutine: Section 6.78 [GETARG], page 77

F2003 functions and subroutines: Section 6.79 [GET_COMMAND], page 78, Section 6.80 [GET_COMMAND_ARGUMENT], page 79, Section 6.37 [COMMAND_ARGUMENT_COUNT], page 54

6.94 IBCLR — Clear bit

Description:
`IBCLR` returns the value of *I* with the bit at position *POS* set to zero.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = IBCLR(I, POS)`

Arguments:

I	The type shall be `INTEGER(*)`.
POS	The type shall be `INTEGER(*)`.

Return value:
The return value is of type `INTEGER(*)` and of the same kind as *I*.

See also: Section 6.95 [IBITS], page 86, Section 6.96 [IBSET], page 87, Section 6.92 [IAND], page 85, Section 6.103 [IOR], page 90, Section 6.99 [IEOR], page 88, Section 6.141 [MVBITS], page 111

6.95 IBITS — Bit extraction

Description:
`IBITS` extracts a field of length *LEN* from *I*, starting from bit position *POS* and extending left for *LEN* bits. The result is right-justified and the remaining bits are zeroed. The value of `POS+LEN` must be less than or equal to the value `BIT_SIZE(I)`.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = IBITS(I, POS, LEN)`

Arguments:

I	The type shall be `INTEGER(*)`.
POS	The type shall be `INTEGER(*)`.
LEN	The type shall be `INTEGER(*)`.

Return value:
The return value is of type `INTEGER(*)` and of the same kind as *I*.

See also: Section 6.30 [BIT_SIZE], page 50, Section 6.94 [IBCLR], page 86, Section 6.96 [IBSET], page 87, Section 6.92 [IAND], page 85, Section 6.103 [IOR], page 90, Section 6.99 [IEOR], page 88

6.96 IBSET — Set bit

Description:
IBSET returns the value of *I* with the bit at position *POS* set to one.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = IBSET(I, POS)`

Arguments:

I	The type shall be `INTEGER(*)`.
POS	The type shall be `INTEGER(*)`.

Return value:
The return value is of type `INTEGER(*)` and of the same kind as *I*.

See also: Section 6.94 [IBCLR], page 86, Section 6.95 [IBITS], page 86, Section 6.92 [IAND], page 85, Section 6.103 [IOR], page 90, Section 6.99 [IEOR], page 88, Section 6.141 [MVBITS], page 111

6.97 ICHAR — Character-to-integer conversion function

Description:
`ICHAR(C)` returns the code for the character in the first character position of `C` in the system's native character set. The correspondence between characters and their codes is not necessarily the same across different GNU Fortran implementations.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = ICHAR(C)`

Arguments:

C	Shall be a scalar `CHARACTER`, with `INTENT(IN)`

Return value:
The return value is of type `INTEGER` and of the default integer kind.

Example:

```
program test_ichar
  integer i
  i = ichar(' ')
end program test_ichar
```

Note: No intrinsic exists to convert between a numeric value and a formatted character string representation – for instance, given the `CHARACTER` value `'154'`, obtaining an `INTEGER` or `REAL` value with the value 154, or vice versa. Instead, this functionality is provided by internal-file I/O, as in the following example:

```
program read_val
  integer value
  character(len=10) string, string2
  string = '154'
```

```
  ! Convert a string to a numeric value
  read (string,'(I10)') value
  print *, value

  ! Convert a value to a formatted string
  write (string2,'(I10)') value
  print *, string2
end program read_val
```

See also: Section 6.5 [ACHAR], page 35, Section 6.33 [CHAR], page 51, Section 6.91 [IACHAR], page 84

6.98 IDATE — Get current local time subroutine (day/month/year)

Description:
`IDATE(TARRAY)` Fills *TARRAY* with the numerical values at the current local time. The day (in the range 1-31), month (in the range 1-12), and year appear in elements 1, 2, and 3 of *TARRAY*, respectively. The year has four significant digits.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL IDATE(TARRAY)`

Arguments:

TARRAY	The type shall be `INTEGER, DIMENSION(3)` and the kind shall be the default integer kind.

Return value:
Does not return.

Example:

```
program test_idate
  integer, dimension(3) :: tarray
  call idate(tarray)
  print *, tarray(1)
  print *, tarray(2)
  print *, tarray(3)
end program test_idate
```

6.99 IEOR — Bitwise logical exclusive or

Description:
`IEOR` returns the bitwise boolean exclusive-OR of *I* and *J*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = IEOR(I, J)`

Arguments:

I	The type shall be `INTEGER(*)`.

J	The type shall be `INTEGER(*)`, of the same kind as *I*. (As a GNU extension, different kinds are also permitted.)

Return value:
The return type is `INTEGER(*)`, of the same kind as the arguments. (If the argument kinds differ, it is of the same kind as the larger argument.)

See also: Section 6.103 [IOR], page 90, Section 6.92 [IAND], page 85, Section 6.95 [IBITS], page 86, Section 6.96 [IBSET], page 87, Section 6.94 [IBCLR], page 86, Section 6.145 [NOT], page 113

6.100 `IERRNO` — Get the last system error number

Description:
Returns the last system error number, as given by the C `errno()` function.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = IERRNO()`

Arguments:
None.

Return value:
The return value is of type `INTEGER` and of the default integer kind.

See also: Section 6.149 [PERROR], page 115

6.101 `INDEX` — Position of a substring within a string

Description:
Returns the position of the start of the first occurrence of string *SUBSTRING* as a substring in *STRING*, counting from one. If *SUBSTRING* is not present in *STRING*, zero is returned. If the *BACK* argument is present and true, the return value is the start of the last occurrence rather than the first.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = INDEX(STRING, SUBSTRING [, BACK])`

Arguments:

STRING	Shall be a scalar `CHARACTER(*)`, with `INTENT(IN)`
SUBSTRING	Shall be a scalar `CHARACTER(*)`, with `INTENT(IN)`
BACK	(Optional) Shall be a scalar `LOGICAL(*)`, with `INTENT(IN)`

Return value:
The return value is of type `INTEGER` and of the default integer kind.

See also:

6.102 INT — Convert to integer type

Description:
Convert to integer type

Standard: F77 and later

Class: Elemental function

Syntax:

`RESULT = INT(X [, KIND))`

Arguments:

X	Shall be of type `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)`.
KIND	(Optional) An `INTEGER(*)` initialization expression indicating the kind parameter of the result.

Return value:
These functions return a `INTEGER(*)` variable or array under the following rules:

(A) If X is of type `INTEGER(*)`, `INT(X) = X`

(B) If X is of type `REAL(*)` and $|X| < 1$, `INT(X)` equals *0*. If $|X| \geq 1$, then `INT(X)` equals the largest integer that does not exceed the range of X and whose sign is the same as the sign of X.

(C) If X is of type `COMPLEX(*)`, rule B is applied to the real part of X.

Example:

```
program test_int
  integer :: i = 42
  complex :: z = (-3.7, 1.0)
  print *, int(i)
  print *, int(z), int(z,8)
end program
```

Specific names:

Name	Argument	Return type	Standard
`IFIX(X)`	`REAL(4) X`	`INTEGER`	F77 and later
`IDINT(X)`	`REAL(8) X`	`INTEGER`	F77 and later

6.103 IOR — Bitwise logical or

Description:
`IEOR` returns the bitwise boolean OR of *I* and *J*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = IEOR(I, J)`

Arguments:

I	The type shall be `INTEGER(*)`.
J	The type shall be `INTEGER(*)`, of the same kind as *I*. (As a GNU extension, different kinds are also permitted.)

Return value:
The return type is `INTEGER(*)`, of the same kind as the arguments. (If the argument kinds differ, it is of the same kind as the larger argument.)

See also: Section 6.99 [IEOR], page 88, Section 6.92 [IAND], page 85, Section 6.95 [IBITS], page 86, Section 6.96 [IBSET], page 87, Section 6.94 [IBCLR], page 86, Section 6.145 [NOT], page 113

6.104 IRAND — Integer pseudo-random number

Description:
`IRAND(FLAG)` returns a pseudo-random number from a uniform distribution between 0 and a system-dependent limit (which is in most cases 2147483647). If *FLAG* is 0, the next number in the current sequence is returned; if *FLAG* is 1, the generator is restarted by `CALL SRAND(0)`; if *FLAG* has any other value, it is used as a new seed with `SRAND`.

Standard: GNU extension

Class: Non-elemental function

Syntax: `RESULT = IRAND(FLAG)`

Arguments:

FLAG	Shall be a scalar `INTEGER` of kind 4.

Return value:
The return value is of `INTEGER(kind=4)` type.

Example:

```
program test_irand
  integer,parameter :: seed = 86456

  call srand(seed)
  print *, irand(), irand(), irand(), irand()
  print *, irand(seed), irand(), irand(), irand()
end program test_irand
```

6.105 ISATTY — Whether a unit is a terminal device.

Description:
Determine whether a unit is connected to a terminal device.

Standard: GNU extension.

Class: Non-elemental function.

Syntax: `RESULT = ISATTY(UNIT)`

Arguments:

UNIT	Shall be a scalar `INTEGER(*)`.

Return value:
Returns `.TRUE.` if the *UNIT* is connected to a terminal device, `.FALSE.` otherwise.

Example:

```
PROGRAM test_isatty
  INTEGER(kind=1) :: unit
  DO unit = 1, 10
    write(*,*) isatty(unit=unit)
  END DO
END PROGRAM
```

See also: Section 6.196 [TTYNAM], page 135

6.106 ISHFT — Shift bits

Description:

ISHFT returns a value corresponding to *I* with all of the bits shifted *SHIFT* places. A value of *SHIFT* greater than zero corresponds to a left shift, a value of zero corresponds to no shift, and a value less than zero corresponds to a right shift. If the absolute value of *SHIFT* is greater than BIT_SIZE(I), the value is undefined. Bits shifted out from the left end or right end are lost; zeros are shifted in from the opposite end.

Standard: F95 and later

Class: Elemental function

Syntax: RESULT = ISHFT(I, SHIFT)

Arguments:

I	The type shall be INTEGER(*).
SHIFT	The type shall be INTEGER(*).

Return value:

The return value is of type INTEGER(*) and of the same kind as *I*.

See also: Section 6.107 [ISHFTC], page 92

6.107 ISHFTC — Shift bits circularly

Description:

ISHFTC returns a value corresponding to *I* with the rightmost *SIZE* bits shifted circularly *SHIFT* places; that is, bits shifted out one end are shifted into the opposite end. A value of *SHIFT* greater than zero corresponds to a left shift, a value of zero corresponds to no shift, and a value less than zero corresponds to a right shift. The absolute value of *SHIFT* must be less than *SIZE*. If the *SIZE* argument is omitted, it is taken to be equivalent to BIT_SIZE(I).

Standard: F95 and later

Class: Elemental function

Syntax: RESULT = ISHFTC(I, SHIFT [, SIZE])

Arguments:

I	The type shall be INTEGER(*).
SHIFT	The type shall be INTEGER(*).
SIZE	(Optional) The type shall be INTEGER(*); the value must be greater than zero and less than or equal to BIT_SIZE(I).

Return value:
The return value is of type `INTEGER(*)` and of the same kind as *I*.

See also: Section 6.106 [ISHFT], page 92

6.108 `ITIME` — Get current local time subroutine (hour/minutes/seconds)

Description:
`IDATE(TARRAY)` Fills *TARRAY* with the numerical values at the current local time. The hour (in the range 1-24), minute (in the range 1-60), and seconds (in the range 1-60) appear in elements 1, 2, and 3 of *TARRAY*, respectively.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL ITIME(TARRAY)`

Arguments:

TARRAY	The type shall be `INTEGER, DIMENSION(3)` and the kind shall be the default integer kind.

Return value:
Does not return.

Example:

```
program test_itime
  integer, dimension(3) :: tarray
  call itime(tarray)
  print *, tarray(1)
  print *, tarray(2)
  print *, tarray(3)
end program test_itime
```

6.109 `KILL` — Send a signal to a process

Description:
Standard: Sends the signal specified by *SIGNAL* to the process *PID*. See `kill(2)`.

Class: Subroutine

Syntax: `CALL KILL(PID, SIGNAL [, STATUS])`

Arguments:

PID	Shall be a scalar `INTEGER`, with `INTENT(IN)`
SIGNAL	Shall be a scalar `INTEGER`, with `INTENT(IN)`
STATUS	(Optional) status flag of type `INTEGER(4)` or `INTEGER(8)`. Returns 0 on success, or a system-specific error code otherwise.

See also: Section 6.2 [ABORT], page 33, Section 6.60 [EXIT], page 68

6.110 KIND — Kind of an entity

Description:
KIND(X) returns the kind value of the entity *X*.

Standard: F95 and later

Class: Inquiry function

Syntax: K = KIND(X)

Arguments:

X	Shall be of type LOGICAL, INTEGER, REAL, COMPLEX or CHARACTER.

Return value:
The return value is a scalar of type INTEGER and of the default integer kind.

Example:

```
program test_kind
  integer,parameter :: kc = kind(' ')
  integer,parameter :: kl = kind(.true.)

  print *, "The default character kind is ", kc
  print *, "The default logical kind is ", kl
end program test_kind
```

6.111 LBOUND — Lower dimension bounds of an array

Description:
Returns the lower bounds of an array, or a single lower bound along the *DIM* dimension.

Standard: F95 and later

Class: Inquiry function

Syntax: RESULT = LBOUND(ARRAY [, DIM])

Arguments:

ARRAY	Shall be an array, of any type.
DIM	(Optional) Shall be a scalar INTEGER(*).

Return value:
If *DIM* is absent, the result is an array of the lower bounds of *ARRAY*. If *DIM* is present, the result is a scalar corresponding to the lower bound of the array along that dimension. If *ARRAY* is an expression rather than a whole array or array structure component, or if it has a zero extent along the relevant dimension, the lower bound is taken to be 1.

See also: Section 6.197 [UBOUND], page 136

6.112 LEN — Length of a character entity

Description:
Returns the length of a character string. If *STRING* is an array, the length of an element of *STRING* is returned. Note that *STRING* need not be defined when this intrinsic is invoked, since only the length, not the content, of *STRING* is needed.

Standard: F77 and later

Class: Inquiry function

Syntax: `L = LEN(STRING)`

Arguments:

STRING	Shall be a scalar or array of type `CHARACTER(*)`, with `INTENT(IN)`

Return value:
The return value is an `INTEGER` of the default kind.

See also: Section 6.113 [LEN_TRIM], page 95, Section 6.8 [ADJUSTL], page 37, Section 6.9 [ADJUSTR], page 37

6.113 LEN_TRIM — Length of a character entity without trailing blank characters

Description:
Returns the length of a character string, ignoring any trailing blanks.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = LEN_TRIM(STRING)`

Arguments:

STRING	Shall be a scalar of type `CHARACTER(*)`, with `INTENT(IN)`

Return value:
The return value is an `INTEGER` of the default kind.

See also: Section 6.112 [LEN], page 95, Section 6.8 [ADJUSTL], page 37, Section 6.9 [ADJUSTR], page 37

6.114 LGE — Lexical greater than or equal

Description:
Determines whether one string is lexically greater than or equal to another string, where the two strings are interpreted as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if spaces were appended to it to form a value that has the same length as the longer.

In general, the lexical comparison intrinsics `LGE`, `LGT`, `LLE`, and `LLT` differ from the corresponding intrinsic operators `.GE.`, `.GT.`, `.LE.`, and `.LT.`, in that the latter use the processor's character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = LGE(STRING_A, STRING_B)`

Arguments:

STRING_A	Shall be of default `CHARACTER` type.
STRING_B	Shall be of default `CHARACTER` type.

Return value:
Returns `.TRUE.` if `STRING_A >= STRING_B`, and `.FALSE.` otherwise, based on the ASCII ordering.

See also: Section 6.115 [LGT], page 96, Section 6.117 [LLE], page 97, Section 6.118 [LLT], page 98

6.115 `LGT` — Lexical greater than

Description:
Determines whether one string is lexically greater than another string, where the two strings are interpreted as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if spaces were appended to it to form a value that has the same length as the longer.

In general, the lexical comparison intrinsics `LGE`, `LGT`, `LLE`, and `LLT` differ from the corresponding intrinsic operators `.GE.`, `.GT.`, `.LE.`, and `.LT.`, in that the latter use the processor's character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = LGT(STRING_A, STRING_B)`

Arguments:

STRING_A	Shall be of default `CHARACTER` type.
STRING_B	Shall be of default `CHARACTER` type.

Return value:
Returns `.TRUE.` if `STRING_A > STRING_B`, and `.FALSE.` otherwise, based on the ASCII ordering.

See also: Section 6.114 [LGE], page 95, Section 6.117 [LLE], page 97, Section 6.118 [LLT], page 98

6.116 LINK — Create a hard link

Description:

Makes a (hard) link from file *PATH1* to *PATH2*. A null character (`CHAR(0)`) can be used to mark the end of the names in *PATH1* and *PATH2*; otherwise, trailing blanks in the file names are ignored. If the *STATUS* argument is supplied, it contains 0 on success or a nonzero error code upon return; see `link(2)`.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, non-elemental function

Syntax:

```
CALL LINK(PATH1, PATH2 [, STATUS])
STATUS = LINK(PATH1, PATH2)
```

Arguments:

PATH1	Shall be of default `CHARACTER` type.
PATH2	Shall be of default `CHARACTER` type.
STATUS	(Optional) Shall be of default `INTEGER` type.

See also: Section 6.185 [SYMLNK], page 131, Section 6.199 [UNLINK], page 137

6.117 LLE — Lexical less than or equal

Description:

Determines whether one string is lexically less than or equal to another string, where the two strings are interpreted as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if spaces were appended to it to form a value that has the same length as the longer.

In general, the lexical comparison intrinsics `LGE`, `LGT`, `LLE`, and `LLT` differ from the corresponding intrinsic operators `.GE.`, `.GT.`, `.LE.`, and `.LT.`, in that the latter use the processor's character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = LLE(STRING_A, STRING_B)`

Arguments:

STRING_A	Shall be of default `CHARACTER` type.
STRING_B	Shall be of default `CHARACTER` type.

Return value:

Returns `.TRUE.` if `STRING_A <= STRING_B`, and `.FALSE.` otherwise, based on the ASCII ordering.

See also: Section 6.114 [LGE], page 95, Section 6.115 [LGT], page 96, Section 6.118 [LLT], page 98

6.118 LLT — Lexical less than

Description:
Determines whether one string is lexically less than another string, where the two strings are interpreted as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if spaces were appended to it to form a value that has the same length as the longer.

In general, the lexical comparison intrinsics `LGE`, `LGT`, `LLE`, and `LLT` differ from the corresponding intrinsic operators `.GE.`, `.GT.`, `.LE.`, and `.LT.`, in that the latter use the processor's character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = LLT(STRING_A, STRING_B)`

Arguments:

STRING_A	Shall be of default `CHARACTER` type.
STRING_B	Shall be of default `CHARACTER` type.

Return value:
Returns `.TRUE.` if `STRING_A < STRING_B`, and `.FALSE.` otherwise, based on the ASCII ordering.

See also: Section 6.114 [LGE], page 95, Section 6.115 [LGT], page 96, Section 6.117 [LLE], page 97

6.119 LNBLNK — Index of the last non-blank character in a string

Description:
Returns the length of a character string, ignoring any trailing blanks. This is identical to the standard `LEN_TRIM` intrinsic, and is only included for backwards compatibility.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = LNBLNK(STRING)`

Arguments:

STRING	Shall be a scalar of type `CHARACTER(*)`, with `INTENT(IN)`

Return value:
The return value is of `INTEGER(kind=4)` type.

See also: Section 6.101 [INDEX], page 89, Section 6.113 [LEN_TRIM], page 95

6.120 LOC — Returns the address of a variable

Description:
: `LOC(X)` returns the address of *X* as an integer.

Standard: GNU extension

Class: Inquiry function

Syntax: `RESULT = LOC(X)`

Arguments:

X	Variable of any type.

Return value:
: The return value is of type `INTEGER`, with a `KIND` corresponding to the size (in bytes) of a memory address on the target machine.

Example:

```
program test_loc
  integer :: i
  real :: r
  i = loc(r)
  print *, i
end program test_loc
```

6.121 LOG — Logarithm function

Description:
: `LOG(X)` computes the logarithm of *X*.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = LOG(X)`

Arguments:

X	The type shall be `REAL(*)` or `COMPLEX(*)`.

Return value:
: The return value is of type `REAL(*)` or `COMPLEX(*)`. The kind type parameter is the same as *X*.

Example:

```
program test_log
  real(8) :: x = 1.0_8
  complex :: z = (1.0, 2.0)
  x = log(x)
  z = log(z)
end program test_log
```

Specific names:

Name	Argument	Return type	Standard
`ALOG(X)`	`REAL(4) X`	`REAL(4)`	f95, gnu
`DLOG(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu
`CLOG(X)`	`COMPLEX(4) X`	`COMPLEX(4)`	f95, gnu
`ZLOG(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	f95, gnu
`CDLOG(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	f95, gnu

6.122 LOG10 — Base 10 logarithm function

Description:
LOG10(X) computes the base 10 logarithm of *X*.

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = LOG10(X)

Arguments:

X	The type shall be REAL(*) or COMPLEX(*).

Return value:
The return value is of type REAL(*) or COMPLEX(*). The kind type parameter is the same as *X*.

Example:

```
program test_log10
  real(8) :: x = 10.0_8
  x = log10(x)
end program test_log10
```

Specific names:

Name	Argument	Return type	Standard
ALOG10(X)	REAL(4) X	REAL(4)	F95 and later
DLOG10(X)	REAL(8) X	REAL(8)	F95 and later

6.123 LOGICAL — Convert to logical type

Description:
Converts one kind of LOGICAL variable to another.

Standard: F95 and later

Class: Elemental function

Syntax: RESULT = LOGICAL(L [, KIND])

Arguments:

L	The type shall be LOGICAL(*).
KIND	(Optional) An INTEGER(*) initialization expression indicating the kind parameter of the result.

Return value:
The return value is a LOGICAL value equal to *L*, with a kind corresponding to *KIND*, or of the default logical kind if *KIND* is not given.

See also: Section 6.102 [INT], page 90, Section 6.159 [REAL], page 118, Section 6.36 [CMPLX], page 53

6.124 LSHIFT — Left shift bits

Description:
LSHIFT returns a value corresponding to *I* with all of the bits shifted left by *SHIFT* places. If the absolute value of *SHIFT* is greater than BIT_SIZE(I), the value is undefined. Bits shifted out from the left end are lost; zeros are shifted in from the opposite end.

This function has been superseded by the ISHFT intrinsic, which is standard in Fortran 95 and later.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = LSHIFT(I, SHIFT)`

Arguments:

I	The type shall be INTEGER(*).
SHIFT	The type shall be INTEGER(*).

Return value:
The return value is of type INTEGER(*) and of the same kind as *I*.

See also: Section 6.106 [ISHFT], page 92, Section 6.107 [ISHFTC], page 92, Section 6.164 [RSHIFT], page 120

6.125 LSTAT — Get file status

Description:
LSTAT is identical to Section 6.183 [STAT], page 129, except that if path is a symbolic link, then the link itself is statted, not the file that it refers to.

The elements in BUFF are the same as described by Section 6.183 [STAT], page 129.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL LSTAT(FILE, BUFF [, STATUS])`

Arguments:

FILE	The type shall be CHARACTER(*), a valid path within the file system.
BUFF	The type shall be INTEGER(4), DIMENSION(13).
STATUS	(Optional) status flag of type INTEGER(4). Returns 0 on success and a system specific error code otherwise.

Example: See Section 6.183 [STAT], page 129 for an example.

See also: To stat an open file: Section 6.75 [FSTAT], page 76, to stat a file: Section 6.183 [STAT], page 129

6.126 `LTIME` — Convert time to local time info

Description:
Given a system time value *STIME* (as provided by the `TIME8()` intrinsic), fills *TARRAY* with values extracted from it appropriate to the local time zone using `localtime(3)`.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL LTIME(STIME, TARRAY)`

Arguments:

STIME	An `INTEGER(*)` scalar expression corresponding to a system time, with `INTENT(IN)`.
TARRAY	A default `INTEGER` array with 9 elements, with `INTENT(OUT)`.

Return value:
The elements of *TARRAY* are assigned as follows:
1. Seconds after the minute, range 0–59 or 0–61 to allow for leap seconds
2. Minutes after the hour, range 0–59
3. Hours past midnight, range 0–23
4. Day of month, range 0–31
5. Number of months since January, range 0–12
6. Years since 1900
7. Number of days since Sunday, range 0–6
8. Days since January 1
9. Daylight savings indicator: positive if daylight savings is in effect, zero if not, and negative if the information is not available.

See also: Section 6.44 [CTIME], page 58, Section 6.88 [GMTIME], page 82, Section 6.190 [TIME], page 133, Section 6.191 [TIME8], page 134

6.127 `MALLOC` — Allocate dynamic memory

Description:
`MALLOC(SIZE)` allocates *SIZE* bytes of dynamic memory and returns the address of the allocated memory. The `MALLOC` intrinsic is an extension intended to be used with Cray pointers, and is provided in GNU Fortran to allow the user to compile legacy code. For new code using Fortran 95 pointers, the memory allocation intrinsic is `ALLOCATE`.

Standard: GNU extension

Class: Non-elemental function

Syntax: `PTR = MALLOC(SIZE)`

Arguments:

SIZE	The type shall be `INTEGER(*)`.

Return value:
The return value is of type `INTEGER(K)`, with K such that variables of type `INTEGER(K)` have the same size as C pointers (`sizeof(void *)`).

Example: The following example demonstrates the use of `MALLOC` and `FREE` with Cray pointers. This example is intended to run on 32-bit systems, where the default integer kind is suitable to store pointers; on 64-bit systems, ptr_x would need to be declared as `integer(kind=8)`.

```
program test_malloc
  integer i
  integer ptr_x
  real*8 x(*), z
  pointer(ptr_x,x)

  ptr_x = malloc(20*8)
  do i = 1, 20
    x(i) = sqrt(1.0d0 / i)
  end do
  z = 0
  do i = 1, 20
    z = z + x(i)
    print *, z
  end do
  call free(ptr_x)
end program test_malloc
```

See also: Section 6.73 [FREE], page 75

6.128 `MATMUL` — matrix multiplication

Description:
Performs a matrix multiplication on numeric or logical arguments.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = MATMUL(MATRIX_A, MATRIX_B)`

Arguments:

MATRIX_A	An array of `INTEGER(*)`, `REAL(*)`, `COMPLEX(*)`, or `LOGICAL(*)` type, with a rank of one or two.
MATRIX_B	An array of `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)` type if *MATRIX_A* is of a numeric type; otherwise, an array of `LOGICAL(*)` type. The rank shall be one or two, and the first (or only) dimension of *MATRIX_B* shall be equal to the last (or only) dimension of *MATRIX_A*.

Return value:
The matrix product of *MATRIX_A* and *MATRIX_B*. The type and kind of the result follow the usual type and kind promotion rules, as for the `*` or `.AND.` operators.

See also:

6.129 MAX — Maximum value of an argument list

Description:
Returns the argument with the largest (most positive) value.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = MAX(A1, A2 [, A3 [, ...]])`

Arguments:

A1	The type shall be `INTEGER(*)` or `REAL(*)`.
A2, *A3*, ...	An expression of the same type and kind as *A1*. (As a GNU extension, arguments of different kinds are permitted.)

Return value:
The return value corresponds to the maximum value among the arguments, and has the same type and kind as the first argument.

Specific names:

Name	Argument	Return type	Standard
`MAX0(I)`	`INTEGER(4) I`	`INTEGER(4)`	F77 and later
`AMAX0(I)`	`INTEGER(4) I`	`REAL(MAX(X))`	F77 and later
`MAX1(X)`	`REAL(*) X`	`INT(MAX(X))`	F77 and later
`AMAX1(X)`	`REAL(4) X`	`REAL(4)`	F77 and later
`DMAX1(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

See also: Section 6.131 [MAXLOC], page 105 Section 6.132 [MAXVAL], page 105, Section 6.134 [MIN], page 106

6.130 MAXEXPONENT — Maximum exponent of a real kind

Description:
`MAXEXPONENT(X)` returns the maximum exponent in the model of the type of `X`.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = MAXEXPONENT(X)`

Arguments:

X	Shall be of type `REAL`.

Return value:
The return value is of type `INTEGER` and of the default integer kind.

Example:

```
program exponents
  real(kind=4) :: x
  real(kind=8) :: y

  print *, minexponent(x), maxexponent(x)
  print *, minexponent(y), maxexponent(y)
end program exponents
```

6.131 `MAXLOC` — Location of the maximum value within an array

Description:
Determines the location of the element in the array with the maximum value, or, if the *DIM* argument is supplied, determines the locations of the maximum element along each row of the array in the *DIM* direction. If *MASK* is present, only the elements for which *MASK* is `.TRUE.` are considered. If more than one element in the array has the maximum value, the location returned is that of the first such element in array element order. If the array has zero size, or all of the elements of *MASK* are `.FALSE.`, then the result is an array of zeroes. Similarly, if *DIM* is supplied and all of the elements of *MASK* along a given row are zero, the result value for that row is zero.

Standard: F95 and later

Class: Transformational function

Syntax:

```
RESULT = MAXLOC(ARRAY, DIM [, MASK])
RESULT = MAXLOC(ARRAY [, MASK])
```

Arguments:

ARRAY	Shall be an array of type `INTEGER(*)`, `REAL(*)`, or `CHARACTER(*)`.
DIM	(Optional) Shall be a scalar of type `INTEGER(*)`, with a value between one and the rank of *ARRAY*, inclusive. It may not be an optional dummy argument.
MASK	Shall be an array of type `LOGICAL(*)`, and conformable with *ARRAY*.

Return value:
If *DIM* is absent, the result is a rank-one array with a length equal to the rank of *ARRAY*. If *DIM* is present, the result is an array with a rank one less than the rank of *ARRAY*, and a size corresponding to the size of *ARRAY* with the *DIM* dimension removed. If *DIM* is present and *ARRAY* has a rank of one, the result is a scalar. In all cases, the result is of default `INTEGER` type.

See also: Section 6.129 [MAX], page 104, Section 6.132 [MAXVAL], page 105

6.132 `MAXVAL` — Maximum value of an array

Description:
Determines the maximum value of the elements in an array value, or, if the *DIM* argument is supplied, determines the maximum value along each row of the array in the *DIM* direction. If *MASK* is present, only the elements for which *MASK* is `.TRUE.` are considered. If the array has zero size, or all of the elements of *MASK* are `.FALSE.`, then the result is the most negative number of the type and kind of *ARRAY* if *ARRAY* is numeric, or a string of nulls if *ARRAY* is of character type.

Standard: F95 and later

Class: Transformational function

Syntax:

```
RESULT = MAXVAL(ARRAY, DIM [, MASK])
RESULT = MAXVAL(ARRAY [, MASK])
```

Arguments:

ARRAY	Shall be an array of type `INTEGER(*)`, `REAL(*)`, or `CHARACTER(*)`.
DIM	(Optional) Shall be a scalar of type `INTEGER(*)`, with a value between one and the rank of *ARRAY*, inclusive. It may not be an optional dummy argument.
MASK	Shall be an array of type `LOGICAL(*)`, and conformable with *ARRAY*.

Return value:

If *DIM* is absent, or if *ARRAY* has a rank of one, the result is a scalar. If *DIM* is present, the result is an array with a rank one less than the rank of *ARRAY*, and a size corresponding to the size of *ARRAY* with the *DIM* dimension removed. In all cases, the result is of the same type and kind as *ARRAY*.

See also: Section 6.129 [MAX], page 104, Section 6.131 [MAXLOC], page 105

6.133 `MERGE` — Merge variables

Description:

Select values from two arrays according to a logical mask. The result is equal to *TSOURCE* if *MASK* is `.TRUE.`, or equal to *FSOURCE* if it is `.FALSE.`.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = MERGE(TSOURCE, FSOURCE, MASK)`

Arguments:

TSOURCE	May be of any type.
FSOURCE	Shall be of the same type and type parameters as *TSOURCE*.
MASK	Shall be of type `LOGICAL(*)`.

Return value:

The result is of the same type and type parameters as *TSOURCE*.

6.134 `MIN` — Minimum value of an argument list

Description:

Returns the argument with the smallest (most negative) value.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = MIN(A1, A2 [, A3, ...])`

Arguments:

A1	The type shall be `INTEGER(*)` or `REAL(*)`.
A2, *A3*, ...	An expression of the same type and kind as *A1*. (As a GNU extension, arguments of different kinds are permitted.)

Return value:
The return value corresponds to the maximum value among the arguments, and has the same type and kind as the first argument.

Specific names:

Name	Argument	Return type	Standard
`MIN0(I)`	`INTEGER(4) I`	`INTEGER(4)`	F77 and later
`AMIN0(I)`	`INTEGER(4) I`	`REAL(MIN(X))`	F77 and later
`MIN1(X)`	`REAL(*) X`	`INT(MIN(X))`	F77 and later
`AMIN1(X)`	`REAL(4) X`	`REAL(4)`	F77 and later
`DMIN1(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

See also: Section 6.129 [MAX], page 104, Section 6.136 [MINLOC], page 107, Section 6.137 [MINVAL], page 108

6.135 `MINEXPONENT` — Minimum exponent of a real kind

Description:
`MINEXPONENT(X)` returns the minimum exponent in the model of the type of `X`.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = MINEXPONENT(X)`

Arguments:

X	Shall be of type `REAL`.

Return value:
The return value is of type `INTEGER` and of the default integer kind.

Example: See `MAXEXPONENT` for an example.

6.136 `MINLOC` — Location of the minimum value within an array

Description:
Determines the location of the element in the array with the minimum value, or, if the *DIM* argument is supplied, determines the locations of the minimum element along each row of the array in the *DIM* direction. If *MASK* is present, only the elements for which *MASK* is `.TRUE.` are considered. If more than one element in the array has the minimum value, the location returned is that of the first such element in array element order. If the array has zero size, or all of the elements of *MASK* are `.FALSE.`, then the result is an array of zeroes. Similarly, if *DIM* is supplied and all of the elements of *MASK* along a given row are zero, the result value for that row is zero.

Standard: F95 and later

Class: Transformational function

Syntax:

```
RESULT = MINLOC(ARRAY, DIM [, MASK])
RESULT = MINLOC(ARRAY [, MASK])
```

Arguments:

ARRAY	Shall be an array of type `INTEGER(*)`, `REAL(*)`, or `CHARACTER(*)`.
DIM	(Optional) Shall be a scalar of type `INTEGER(*)`, with a value between one and the rank of *ARRAY*, inclusive. It may not be an optional dummy argument.
MASK	Shall be an array of type `LOGICAL(*)`, and conformable with *ARRAY*.

Return value:

If *DIM* is absent, the result is a rank-one array with a length equal to the rank of *ARRAY*. If *DIM* is present, the result is an array with a rank one less than the rank of *ARRAY*, and a size corresponding to the size of *ARRAY* with the *DIM* dimension removed. If *DIM* is present and *ARRAY* has a rank of one, the result is a scalar. In all cases, the result is of default `INTEGER` type.

See also: Section 6.134 [MIN], page 106, Section 6.137 [MINVAL], page 108

6.137 `MINVAL` — Minimum value of an array

Description:

Determines the minimum value of the elements in an array value, or, if the *DIM* argument is supplied, determines the minimum value along each row of the array in the *DIM* direction. If *MASK* is present, only the elements for which *MASK* is `.TRUE.` are considered. If the array has zero size, or all of the elements of *MASK* are `.FALSE.`, then the result is `HUGE(ARRAY)` if *ARRAY* is numeric, or a string of `CHAR(255)` characters if *ARRAY* is of character type.

Standard: F95 and later

Class: Transformational function

Syntax:

```
RESULT = MINVAL(ARRAY, DIM [, MASK])
RESULT = MINVAL(ARRAY [, MASK])
```

Arguments:

ARRAY	Shall be an array of type `INTEGER(*)`, `REAL(*)`, or `CHARACTER(*)`.
DIM	(Optional) Shall be a scalar of type `INTEGER(*)`, with a value between one and the rank of *ARRAY*, inclusive. It may not be an optional dummy argument.
MASK	Shall be an array of type `LOGICAL(*)`, and conformable with *ARRAY*.

Return value:
If *DIM* is absent, or if *ARRAY* has a rank of one, the result is a scalar. If *DIM* is present, the result is an array with a rank one less than the rank of *ARRAY*, and a size corresponding to the size of *ARRAY* with the *DIM* dimension removed. In all cases, the result is of the same type and kind as *ARRAY*.

See also: Section 6.134 [MIN], page 106, Section 6.136 [MINLOC], page 107

6.138 MOD — Remainder function

Description:
`MOD(A,P)` computes the remainder of the division of A by P. It is calculated as `A - (INT(A/P) * P)`.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = MOD(A, P)`

Arguments:

A	Shall be a scalar of type `INTEGER` or `REAL`
P	Shall be a scalar of the same type as *A* and not equal to zero

Return value:
The kind of the return value is the result of cross-promoting the kinds of the arguments.

Example:

```
program test_mod
  print *, mod(17,3)
  print *, mod(17.5,5.5)
  print *, mod(17.5d0,5.5)
  print *, mod(17.5,5.5d0)

  print *, mod(-17,3)
  print *, mod(-17.5,5.5)
  print *, mod(-17.5d0,5.5)
  print *, mod(-17.5,5.5d0)

  print *, mod(17,-3)
  print *, mod(17.5,-5.5)
  print *, mod(17.5d0,-5.5)
  print *, mod(17.5,-5.5d0)
end program test_mod
```

Specific names:

Name	Arguments	Return type	Standard
`AMOD(A,P)`	`REAL(4)`	`REAL(4)`	F95 and later
`DMOD(A,P)`	`REAL(8)`	`REAL(8)`	F95 and later

6.139 MODULO — Modulo function

Description:
`MODULO(A,P)` computes the *A* modulo *P*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = MODULO(A, P)`

Arguments:

A	Shall be a scalar of type `INTEGER` or `REAL`
P	Shall be a scalar of the same type and kind as *A*

Return value:

The type and kind of the result are those of the arguments.

If *A* and *P* are of type `INTEGER`:
`MODULO(A,P)` has the value *R* such that `A=Q*P+R`, where *Q* is an integer and *R* is between 0 (inclusive) and *P* (exclusive).

If *A* and *P* are of type `REAL`:
`MODULO(A,P)` has the value of `A - FLOOR (A / P) * P`.

In all cases, if *P* is zero the result is processor-dependent.

Example:

```
program test_modulo
  print *, modulo(17,3)
  print *, modulo(17.5,5.5)

  print *, modulo(-17,3)
  print *, modulo(-17.5,5.5)

  print *, modulo(17,-3)
  print *, modulo(17.5,-5.5)
end program test_mod
```

6.140 `MOVE_ALLOC` — Move allocation from one object to another

Description:

`MOVE_ALLOC(SRC, DEST)` moves the allocation from *SRC* to *DEST*. *SRC* will become deallocated in the process.

Standard: F2003 and later

Class: Subroutine

Syntax: `CALL MOVE_ALLOC(SRC, DEST)`

Arguments:

SRC	`ALLOCATABLE`, `INTENT(INOUT)`, may be of any type and kind.
DEST	`ALLOCATABLE`, `INTENT(OUT)`, shall be of the same type, kind and rank as *SRC*

Return value:

None

Example:

```
program test_move_alloc
    integer, allocatable :: a(:), b(:)

    allocate(a(3))
    a = [ 1, 2, 3 ]
    call move_alloc(a, b)
    print *, allocated(a), allocated(b)
    print *, b
end program test_move_alloc
```

6.141 MVBITS — Move bits from one integer to another

Description:
Moves *LEN* bits from positions *FROMPOS* through FROMPOS+LEN-1 of *FROM* to positions *TOPOS* through TOPOS+LEN-1 of *TO*. The portion of argument *TO* not affected by the movement of bits is unchanged. The values of FROMPOS+LEN-1 and TOPOS+LEN-1 must be less than BIT_SIZE(FROM).

Standard: F95 and later

Class: Elemental function

Syntax: RESULT = MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)

Arguments:

FROM	The type shall be INTEGER(*).
FROMPOS	The type shall be INTEGER(*).
LEN	The type shall be INTEGER(*).
TO	The type shall be INTEGER(*), of the same kind as *FROM*.
TOPOS	The type shall be INTEGER(*).

Return value:
The return value is of type INTEGER(*) and of the same kind as *FROM*.

See also: Section 6.94 [IBCLR], page 86, Section 6.96 [IBSET], page 87, Section 6.95 [IBITS], page 86, Section 6.92 [IAND], page 85, Section 6.103 [IOR], page 90, Section 6.99 [IEOR], page 88

6.142 NEAREST — Nearest representable number

Description:
NEAREST(X, S) returns the processor-representable number nearest to X in the direction indicated by the sign of S.

Standard: F95 and later

Class: Elemental function

Syntax: RESULT = NEAREST(X, S)

Arguments:

X	Shall be of type REAL.
S	(Optional) shall be of type REAL and not equal to zero.

Return value:
The return value is of the same type as `X`. If `S` is positive, `NEAREST` returns the processor-representable number greater than `X` and nearest to it. If `S` is negative, `NEAREST` returns the processor-representable number smaller than `X` and nearest to it.

Example:

```
program test_nearest
  real :: x, y
  x = nearest(42.0, 1.0)
  y = nearest(42.0, -1.0)
  write (*,"(3(G20.15))") x, y, x - y
end program test_nearest
```

6.143 `NEW_LINE` — New line character

Description:
`NEW_LINE(C)` returns the new-line character.

Standard: F2003 and later

Class: Elemental function

Syntax: `RESULT = NEW_LINE(C)`

Arguments:

C	The argument shall be a scalar or array of the type `CHARACTER`.

Return value:
Returns a *CHARACTER* scalar of length one with the new-line character of the same kind as parameter *C*.

Example:

```
program newline
  implicit none
  write(*,'(A)') 'This is record 1.'//NEW_LINE('A')//'This is record 2.'
end program newline
```

6.144 `NINT` — Nearest whole number

Description:
`NINT(X)` rounds its argument to the nearest whole number.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = NINT(X)`

Arguments:

X	The type of the argument shall be `REAL`.

Return value:
Returns *A* with the fractional portion of its magnitude eliminated by rounding to the nearest whole number and with its sign preserved, converted to an `INTEGER` of the default kind.

Example:

```
program test_nint
  real(4) x4
  real(8) x8
  x4 = 1.234E0_4
  x8 = 4.321_8
  print *, nint(x4), idnint(x8)
end program test_nint
```

Specific names:

Name	Argument	Standard
`IDNINT(X)`	`REAL(8)`	F95 and later

See also: Section 6.32 [CEILING], page 51, Section 6.67 [FLOOR], page 72

6.145 `NOT` — Logical negation

Description:
`NOT` returns the bitwise boolean inverse of *I*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = NOT(I)`

Arguments:

I	The type shall be `INTEGER(*)`.

Return value:
The return type is `INTEGER(*)`, of the same kind as the argument.

See also: Section 6.92 [IAND], page 85, Section 6.99 [IEOR], page 88, Section 6.103 [IOR], page 90, Section 6.95 [IBITS], page 86, Section 6.96 [IBSET], page 87, Section 6.94 [IBCLR], page 86

6.146 `NULL` — Function that returns an disassociated pointer

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:
Return value:
Example:

See also: Section 6.20 [ASSOCIATED], page 44

6.147 OR — Bitwise logical OR

Description:

Bitwise logical OR.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. For integer arguments, programmers should consider the use of the Section 6.103 [IOR], page 90 intrinsic defined by the Fortran standard.

Standard: GNU extension

Class: Non-elemental function

Syntax: `RESULT = OR(X, Y)`

Arguments:

X	The type shall be either INTEGER(*) or LOGICAL.
Y	The type shall be either INTEGER(*) or LOGICAL.

Return value:

The return type is either INTEGER(*) or LOGICAL after cross-promotion of the arguments.

Example:

```
PROGRAM test_or
  LOGICAL :: T = .TRUE., F = ..FALSE.
  INTEGER :: a, b
  DATA a / Z'F' /, b / Z'3' /

  WRITE (*,*) OR(T, T), OR(T, F), OR(F, T), OR(F, F)
  WRITE (*,*) OR(a, b)
END PROGRAM
```

See also: F95 elemental function: Section 6.103 [IOR], page 90

6.148 PACK — Pack an array into an array of rank one

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:
Return value:
Example:

Specific names:
See also: Section 6.200 [UNPACK], page 137

6.149 PERROR — Print system error message

Description:
Prints (on the C `stderr` stream) a newline-terminated error message corresponding to the last system error. This is prefixed by *STRING*, a colon and a space. See `perror(3)`.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL PERROR(STRING)`

Arguments:

STRING	A scalar of default `CHARACTER` type.

See also: Section 6.100 [IERRNO], page 89

6.150 PRECISION — Decimal precision of a real kind

Description:
`PRECISION(X)` returns the decimal precision in the model of the type of `X`.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = PRECISION(X)`

Arguments:

X	Shall be of type `REAL` or `COMPLEX`.

Return value:
The return value is of type `INTEGER` and of the default integer kind.

Example:

```
program prec_and_range
  real(kind=4) :: x(2)
  complex(kind=8) :: y

  print *, precision(x), range(x)
  print *, precision(y), range(y)
end program prec_and_range
```

6.151 PRESENT — Determine whether an optional argument is specified

Intrinsic implemented, documentation pending.

Description:

Standard: F95 and later

Class: Inquiry function

Syntax:

Arguments:

Return value:

Example:

See also:

6.152 PRODUCT — Product of array elements

Intrinsic implemented, documentation pending.

Description:

Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:

Return value:

Example:

Specific names:

See also: Section 6.184 [SUM], page 130

6.153 RADIX — Base of a model number

Description:
`RADIX(X)` returns the base of the model representing the entity *X*.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = RADIX(X)`

Arguments:

X	Shall be of type `INTEGER` or `REAL`

Return value:
The return value is a scalar of type `INTEGER` and of the default integer kind.

Example:

```
program test_radix
  print *, "The radix for the default integer kind is", radix(0)
  print *, "The radix for the default real kind is", radix(0.0)
end program test_radix
```

6.154 RAN — Real pseudo-random number

Description:
For compatibility with HP FORTRAN 77/iX, the `RAN` intrinsic is provided as an alias for `RAND`. See Section 6.155 [RAND], page 117 for complete documentation.

Standard: GNU extension

Class: Non-elemental function

See also: Section 6.155 [RAND], page 117, Section 6.156 [RANDOM_NUMBER], page 117

6.155 RAND — Real pseudo-random number

Description:
RAND(FLAG) returns a pseudo-random number from a uniform distribution between 0 and 1. If *FLAG* is 0, the next number in the current sequence is returned; if *FLAG* is 1, the generator is restarted by CALL SRAND(0); if *FLAG* has any other value, it is used as a new seed with SRAND.

Standard: GNU extension

Class: Non-elemental function

Syntax: RESULT = RAND(FLAG)

Arguments:

FLAG	Shall be a scalar INTEGER of kind 4.

Return value:
The return value is of REAL type and the default kind.

Example:

```
program test_rand
  integer,parameter :: seed = 86456

  call srand(seed)
  print *, rand(), rand(), rand(), rand()
  print *, rand(seed), rand(), rand(), rand()
end program test_rand
```

See also: Section 6.182 [SRAND], page 129, Section 6.156 [RANDOM_NUMBER], page 117

6.156 RANDOM_NUMBER — Pseudo-random number

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Elemental subroutine

Syntax:

Arguments:
Return value:
Example:

See also: Section 6.157 [RANDOM_SEED], page 117

6.157 RANDOM_SEED — Initialize a pseudo-random number sequence

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Subroutine

Syntax:

Arguments:

Return value:

Example:

See also: Section 6.156 [RANDOM_NUMBER], page 117

6.158 `RANGE` — Decimal exponent range of a real kind

Description:
`RANGE(X)` returns the decimal exponent range in the model of the type of `X`.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = RANGE(X)`

Arguments:

X	Shall be of type `REAL` or `COMPLEX`.

Return value:
The return value is of type `INTEGER` and of the default integer kind.

Example: See `PRECISION` for an example.

6.159 `REAL` — Convert to real type

Description:
`REAL(X [, KIND])` converts its argument *X* to a real type. The `REALPART(X)` function is provided for compatibility with `g77`, and its use is strongly discouraged.

Standard: F77 and later

Class: Elemental function

Syntax:

```
RESULT = REAL(X [, KIND])
RESULT = REALPART(Z)
```

Arguments:

X	Shall be `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)`.
KIND	(Optional) An `INTEGER(*)` initialization expression indicating the kind parameter of the result.

Return value:
These functions return a `REAL(*)` variable or array under the following rules:

(A) `REAL(X)` is converted to a default real type if *X* is an integer or real variable.

(B) `REAL(X)` is converted to a real type with the kind type parameter of *X* if *X* is a complex variable.

(C) `REAL(X, KIND)` is converted to a real type with kind type parameter *KIND* if *X* is a complex, integer, or real variable.

Example:

```
program test_real
  complex :: x = (1.0, 2.0)
  print *, real(x), real(x,8), realpart(x)
end program test_real
```

See also: Section 6.46 [DBLE], page 59, Section 6.48 [DFLOAT], page 60, Section 6.64 [FLOAT], page 70

6.160 RENAME — Rename a file

Description:

Renames a file from file *PATH1* to *PATH2*. A null character (`CHAR(0)`) can be used to mark the end of the names in *PATH1* and *PATH2*; otherwise, trailing blanks in the file names are ignored. If the *STATUS* argument is supplied, it contains 0 on success or a nonzero error code upon return; see `rename(2)`.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, non-elemental function

Syntax:

```
CALL RENAME(PATH1, PATH2 [, STATUS])
STATUS = RENAME(PATH1, PATH2)
```

Arguments:

PATH1	Shall be of default `CHARACTER` type.
PATH2	Shall be of default `CHARACTER` type.
STATUS	(Optional) Shall be of default `INTEGER` type.

See also: Section 6.116 [LINK], page 97

6.161 REPEAT — Repeated string concatenation

Intrinsic implemented, documentation pending.

Description:

Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:

Return value:

Example:

See also:

6.162 RESHAPE — Function to reshape an array

Intrinsic implemented, documentation pending.

Description:

Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:

Return value:

Example:

See also: Section 6.171 [SHAPE], page 124

6.163 RRSPACING — Reciprocal of the relative spacing

Description:
`RRSPACING(X)` returns the reciprocal of the relative spacing of model numbers near *X*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = RRSPACING(X)`

Arguments:

X	Shall be of type `REAL`.

Return value:
The return value is of the same type and kind as *X*. The value returned is equal to `ABS(FRACTION(X)) * FLOAT(RADIX(X))**DIGITS(X)`.

6.164 RSHIFT — Right shift bits

Description:
`RSHIFT` returns a value corresponding to *I* with all of the bits shifted right by *SHIFT* places. If the absolute value of *SHIFT* is greater than `BIT_SIZE(I)`, the value is undefined. Bits shifted out from the left end are lost; zeros are shifted in from the opposite end.

This function has been superseded by the `ISHFT` intrinsic, which is standard in Fortran 95 and later.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = RSHIFT(I, SHIFT)`

Arguments:

I	The type shall be `INTEGER(*)`.
SHIFT	The type shall be `INTEGER(*)`.

Return value:
The return value is of type `INTEGER(*)` and of the same kind as *I*.

See also: Section 6.106 [ISHFT], page 92, Section 6.107 [ISHFTC], page 92, Section 6.124 [LSHIFT], page 101

6.165 SCALE — Scale a real value

Description:
`SCALE(X,I)` returns `X * RADIX(X)**I`.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = SCALE(X, I)`

Arguments:

X	The type of the argument shall be a `REAL`.
I	The type of the argument shall be a `INTEGER`.

Return value:
The return value is of the same type and kind as *X*. Its value is `X * RADIX(X)**I`.

Example:

```
program test_scale
  real :: x = 178.1387e-4
  integer :: i = 5
  print *, scale(x,i), x*radix(x)**i
end program test_scale
```

6.166 SCAN — Scan a string for the presence of a set of characters

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Elemental function

Syntax:

Arguments:
Return value:
Example:

See also:

6.167 SECNDS — Time function

Description:
`SECNDS(X)` gets the time in seconds from the real-time system clock. *X* is a reference time, also in seconds. If this is zero, the time in seconds from midnight is returned. This function is non-standard and its use is discouraged.

Standard: GNU extension

Class: function

Syntax: `RESULT = SECNDS (X)`

Arguments:

Name	Type
T	REAL(4)
X	REAL(4)

Return value:
None

Example:

```
program test_secnds
    real(4) :: t1, t2
    print *, secnds (0.0)   ! seconds since midnight
    t1 = secnds (0.0)       ! reference time
    do i = 1, 10000000      ! do something
    end do
    t2 = secnds (t1)        ! elapsed time
    print *, "Something took ", t2, " seconds."
end program test_secnds
```

6.168 SELECTED_INT_KIND — Choose integer kind

Description:
`SELECTED_INT_KIND(I)` return the kind value of the smallest integer type that can represent all values ranging from -10^I (exclusive) to 10^I (exclusive). If there is no integer kind that accommodates this range, `SELECTED_INT_KIND` returns -1.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = SELECTED_INT_KIND(I)`

Arguments:

I	Shall be a scalar and of type `INTEGER`.

Example:

```
program large_integers
  integer,parameter :: k5 = selected_int_kind(5)
  integer,parameter :: k15 = selected_int_kind(15)
  integer(kind=k5) :: i5
  integer(kind=k15) :: i15

  print *, huge(i5), huge(i15)

  ! The following inequalities are always true
  print *, huge(i5) >= 10_k5**5-1
  print *, huge(i15) >= 10_k15**15-1
end program large_integers
```

6.169 SELECTED_REAL_KIND — Choose real kind

Description:
`SELECTED_REAL_KIND(P,R)` return the kind value of a real data type with decimal precision greater of at least `P` digits and exponent range greater at least `R`.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = SELECTED_REAL_KIND(P, R)`

Arguments:

P	(Optional) shall be a scalar and of type `INTEGER`.
R	(Optional) shall be a scalar and of type `INTEGER`.

At least one argument shall be present.

Return value:
`SELECTED_REAL_KIND` returns the value of the kind type parameter of a real data type with decimal precision of at least `P` digits and a decimal exponent range of at least `R`. If more than one real data type meet the criteria, the kind of the data type with the smallest decimal precision is returned. If no real data type matches the criteria, the result is

-1 if the processor does not support a real data type with a precision greater than or equal to `P`

-2 if the processor does not support a real type with an exponent range greater than or equal to `R`

-3 if neither is supported.

Example:

```
program real_kinds
  integer,parameter :: p6 = selected_real_kind(6)
  integer,parameter :: p10r100 = selected_real_kind(10,100)
  integer,parameter :: r400 = selected_real_kind(r=400)
  real(kind=p6) :: x
  real(kind=p10r100) :: y
  real(kind=r400) :: z

  print *, precision(x), range(x)
  print *, precision(y), range(y)
  print *, precision(z), range(z)
end program real_kinds
```

6.170 SET_EXPONENT — Set the exponent of the model

Description:
`SET_EXPONENT(X, I)` returns the real number whose fractional part is that that of *X* and whose exponent part is *I*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = SET_EXPONENT(X, I)`

Arguments:

X	Shall be of type `REAL`.
I	Shall be of type `INTEGER`.

Return value:
The return value is of the same type and kind as *X*. The real number whose fractional part is that that of *X* and whose exponent part if *I* is returned; it is `FRACTION(X) * RADIX(X)**I`.

Example:

```
program test_setexp
  real :: x = 178.1387e-4
  integer :: i = 17
  print *, set_exponent(x), fraction(x) * radix(x)**i
end program test_setexp
```

6.171 SHAPE — Determine the shape of an array

Intrinsic implemented, documentation pending.

Description:

Standard: F95 and later

Class: Inquiry function

Syntax:

Arguments:

Return value:

Example:

See also: Section 6.162 [RESHAPE], page 120

6.172 SIGN — Sign copying function

Description:
`SIGN(A,B)` returns the value of *A* with the sign of *B*.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = SIGN(A, B)`

Arguments:

A	Shall be a scalar of type `INTEGER` or `REAL`
B	Shall be a scalar of the same type and kind as *A*

Return value:
The kind of the return value is that of *A* and *B*. If $B \geq 0$ then the result is `ABS(A)`, else it is `-ABS(A)`.

Example:

```
program test_sign
  print *, sign(-12,1)
  print *, sign(-12,0)
  print *, sign(-12,-1)

  print *, sign(-12.,1.)
  print *, sign(-12.,0.)
  print *, sign(-12.,-1.)
end program test_sign
```

Specific names:

Name	Arguments	Return type	Standard
`ISIGN(A,P)`	`INTEGER(4)`	`INTEGER(4)`	f95, gnu
`DSIGN(A,P)`	`REAL(8)`	`REAL(8)`	f95, gnu

6.173 `SIGNAL` — Signal handling subroutine (or function)

Description:

`SIGNAL(NUMBER, HANDLER [, STATUS])` causes external subroutine *HANDLER* to be executed with a single integer argument when signal *NUMBER* occurs. If *HANDLER* is an integer, it can be used to turn off handling of signal *NUMBER* or revert to its default action. See `signal(2)`.

If `SIGNAL` is called as a subroutine and the *STATUS* argument is supplied, it is set to the value returned by `signal(2)`.

Standard: GNU extension

Class: Subroutine, non-elemental function

Syntax:

`CALL SIGNAL(NUMBER, HANDLER [, STATUS])`
`STATUS = SIGNAL(NUMBER, HANDLER)`

Arguments:

NUMBER	Shall be a scalar integer, with `INTENT(IN)`
HANDLER	Signal handler (`INTEGER FUNCTION` or `SUBROUTINE`) or dummy/global `INTEGER` scalar. `INTEGER`. It is `INTENT(IN)`.
STATUS	(Optional) *STATUS* shall be a scalar integer. It has `INTENT(OUT)`.

Return value:

The `SIGNAL` function returns the value returned by `signal(2)`.

Example:

```
program test_signal
  intrinsic signal
  external handler_print

  call signal (12, handler_print)
  call signal (10, 1)

  call sleep (30)
end program test_signal
```

6.174 SIN — Sine function

Description:
`SIN(X)` computes the sine of *X*.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = SIN(X)`

Arguments:

X	The type shall be `REAL(*)` or `COMPLEX(*)`.

Return value:
The return value has same type and kind as *X*.

Example:

```
program test_sin
  real :: x = 0.0
  x = sin(x)
end program test_sin
```

Specific names:

Name	Argument	Return type	Standard
`DSIN(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu
`CSIN(X)`	`COMPLEX(4) X`	`COMPLEX(4)`	f95, gnu
`ZSIN(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	f95, gnu
`CDSIN(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	f95, gnu

See also: Section 6.18 [ASIN], page 43

6.175 SINH — Hyperbolic sine function

Description:
`SINH(X)` computes the hyperbolic sine of *X*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = SINH(X)`

Arguments:

X	The type shall be `REAL(*)`.

Return value:
The return value is of type `REAL(*)`.

Example:

```
program test_sinh
  real(8) :: x = - 1.0_8
  x = sinh(x)
end program test_sinh
```

Specific names:

Name	Argument	Return type	Standard
`DSINH(X)`	`REAL(8) X`	`REAL(8)`	F95 and later

See also: Section 6.19 [ASINH], page 43

6.176 SIZE — Determine the size of an array

Intrinsic implemented, documentation pending.

Description:

Standard: F95 and later

Class: Inquiry function

Syntax:

Arguments:

Return value:

Example:

See also:

6.177 SLEEP — Sleep for the specified number of seconds

Description:
Calling this subroutine causes the process to pause for *SECONDS* seconds.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL SLEEP(SECONDS)`

Arguments:

SECONDS	The type shall be of default `INTEGER`.

Example:

```
program test_sleep
  call sleep(5)
end
```

6.178 SNGL — Convert double precision real to default real

Description:
`SNGL(A)` converts the double precision real *A* to a default real value. This is an archaic form of `REAL` that is specific to one type for *A*.

Standard: GNU extension

Class: function

Syntax: `RESULT = SNGL(A)`

Arguments:

A	The type shall be a double precision `REAL`.

Return value:
The return value is of type default `REAL`.

See also: Section 6.46 [DBLE], page 59

6.179 SPACING — Smallest distance between two numbers of a given type

Intrinsic implemented, documentation pending.

Description:

Standard: F95 and later

Class: Elemental function

Syntax:

Arguments:

Return value:

Example:

See also:

6.180 SPREAD — Add a dimension to an array

Intrinsic implemented, documentation pending.

Description:

Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:

Return value:

Example:

See also:

6.181 SQRT — Square-root function

Description:
SQRT(X) computes the square root of *X*.

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = SQRT(X)

Arguments:

X	The type shall be REAL(*) or COMPLEX(*).

Return value:
The return value is of type REAL(*) or COMPLEX(*). The kind type parameter is the same as *X*.

Example:

```
program test_sqrt
  real(8) :: x = 2.0_8
  complex :: z = (1.0, 2.0)
  x = sqrt(x)
  z = sqrt(z)
end program test_sqrt
```

Specific names:

Name	Argument	Return type	Standard
`DSQRT(X)`	`REAL(8) X`	`REAL(8)`	F95 and later
`CSQRT(X)`	`COMPLEX(4) X`	`COMPLEX(4)`	F95 and later
`ZSQRT(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	GNU extension
`CDSQRT(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	GNU extension

6.182 `SRAND` — Reinitialize the random number generator

Description:
`SRAND` reinitializes the pseudo-random number generator called by `RAND` and `IRAND`. The new seed used by the generator is specified by the required argument *SEED*.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL SRAND(SEED)`

Arguments:

SEED	Shall be a scalar `INTEGER(kind=4)`.

Return value:
Does not return.

Example: See `RAND` and `IRAND` for examples.

Notes: The Fortran 2003 standard specifies the intrinsic `RANDOM_SEED` to initialize the pseudo-random numbers generator and `RANDOM_NUMBER` to generate pseudo-random numbers. Please note that in GNU Fortran, these two sets of intrinsics (`RAND`, `IRAND` and `SRAND` on the one hand, `RANDOM_NUMBER` and `RANDOM_SEED` on the other hand) access two independent pseudo-random number generators.

See also: Section 6.155 [RAND], page 117, Section 6.157 [RANDOM_SEED], page 117, Section 6.156 [RANDOM_NUMBER], page 117

6.183 `STAT` — Get file status

Description:
This function returns information about a file. No permissions are required on the file itself, but execute (search) permission is required on all of the directories in path that lead to the file.

The elements that are obtained and stored in the array `BUFF`:

`buff(1)`	Device ID
`buff(2)`	Inode number
`buff(3)`	File mode
`buff(4)`	Number of links
`buff(5)`	Owner's uid
`buff(6)`	Owner's gid
`buff(7)`	ID of device containing directory entry for file (0 if not available)

`buff(8)`	File size (bytes)
`buff(9)`	Last access time
`buff(10)`	Last modification time
`buff(11)`	Last file status change time
`buff(12)`	Preferred I/O block size (-1 if not available)
`buff(13)`	Number of blocks allocated (-1 if not available)

Not all these elements are relevant on all systems. If an element is not relevant, it is returned as 0.

Standard: GNU extension

Class: Non-elemental subroutine

Syntax: `CALL STAT(FILE,BUFF[,STATUS])`

Arguments:

FILE	The type shall be `CHARACTER(*)`, a valid path within the file system.
BUFF	The type shall be `INTEGER(4), DIMENSION(13)`.
STATUS	(Optional) status flag of type `INTEGER(4)`. Returns 0 on success and a system specific error code otherwise.

Example:

```
PROGRAM test_stat
  INTEGER, DIMENSION(13) :: buff
  INTEGER :: status

  CALL STAT("/etc/passwd", buff, status)

  IF (status == 0) THEN
    WRITE (*, FMT="('Device ID:',               T30, I19)") buff(1)
    WRITE (*, FMT="('Inode number:',            T30, I19)") buff(2)
    WRITE (*, FMT="('File mode (octal):',       T30, O19)") buff(3)
    WRITE (*, FMT="('Number of links:',         T30, I19)") buff(4)
    WRITE (*, FMT="('Owner''s uid:',            T30, I19)") buff(5)
    WRITE (*, FMT="('Owner''s gid:',            T30, I19)") buff(6)
    WRITE (*, FMT="('Device where located:',    T30, I19)") buff(7)
    WRITE (*, FMT="('File size:',               T30, I19)") buff(8)
    WRITE (*, FMT="('Last access time:',        T30, A19)") CTIME(buff(9))
    WRITE (*, FMT="('Last modification time',   T30, A19)") CTIME(buff(10))
    WRITE (*, FMT="('Last status change time:', T30, A19)") CTIME(buff(11))
    WRITE (*, FMT="('Preferred block size:',    T30, I19)") buff(12)
    WRITE (*, FMT="('No. of blocks allocated:', T30, I19)") buff(13)
  END IF
END PROGRAM
```

See also: To stat an open file: Section 6.75 [FSTAT], page 76, to stat a link: Section 6.125 [LSTAT], page 101

6.184 `SUM` — Sum of array elements

Intrinsic implemented, documentation pending.

Description:

Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:
Return value:
Example:

See also: Section 6.152 [PRODUCT], page 116

6.185 SYMLNK — Create a symbolic link

Description:

Makes a symbolic link from file *PATH1* to *PATH2*. A null character (`CHAR(0)`) can be used to mark the end of the names in *PATH1* and *PATH2*; otherwise, trailing blanks in the file names are ignored. If the *STATUS* argument is supplied, it contains 0 on success or a nonzero error code upon return; see `symlink(2)`. If the system does not supply `symlink(2)`, `ENOSYS` is returned.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, non-elemental function

Syntax:

```
CALL SYMLNK(PATH1, PATH2 [, STATUS])
STATUS = SYMLNK(PATH1, PATH2)
```

Arguments:

PATH1	Shall be of default `CHARACTER` type.
PATH2	Shall be of default `CHARACTER` type.
STATUS	(Optional) Shall be of default `INTEGER` type.

See also: Section 6.116 [LINK], page 97, Section 6.199 [UNLINK], page 137

6.186 SYSTEM — Execute a shell command

Description:

Passes the command *COMMAND* to a shell (see `system(3)`). If argument *STATUS* is present, it contains the value returned by `system(3)`, which is presumably 0 if the shell command succeeded. Note that which shell is used to invoke the command is system-dependent and environment-dependent.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, non-elemental function

Syntax:

```
CALL SYSTEM(COMMAND [, STATUS])
STATUS = SYSTEM(COMMAND)
```

Arguments:

COMMAND	Shall be of default `CHARACTER` type.
STATUS	(Optional) Shall be of default `INTEGER` type.

See also:

6.187 `SYSTEM_CLOCK` — Time function

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Subroutine

Syntax:

Arguments:
Return value:
Example:

See also:

6.188 `TAN` — Tangent function

Description:
`TAN(X)` computes the tangent of *X*.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = TAN(X)`

Arguments:

X	The type shall be `REAL(*)`.

Return value:
The return value is of type `REAL(*)`. The kind type parameter is the same as *X*.

Example:

```
program test_tan
  real(8) :: x = 0.165_8
  x = tan(x)
end program test_tan
```

Specific names:

Name	Argument	Return type	Standard
`DTAN(X)`	`REAL(8) X`	`REAL(8)`	F95 and later

See also: Section 6.21 [ATAN], page 45

6.189 TANH — Hyperbolic tangent function

Description:
TANH(X) computes the hyperbolic tangent of *X*.

Standard: F77 and later

Class: Elemental function

Syntax: X = TANH(X)

Arguments:
X The type shall be REAL(*).

Return value:
The return value is of type REAL(*) and lies in the range $-1 \leq tanh(x) \leq 1$.

Example:

```
program test_tanh
  real(8) :: x = 2.1_8
  x = tanh(x)
end program test_tanh
```

Specific names:

Name	Argument	Return type	Standard
DTANH(X)	REAL(8) X	REAL(8)	F95 and later

See also: Section 6.23 [ATANH], page 46

6.190 TIME — Time function

Description:
Returns the current time encoded as an integer (in the manner of the UNIX function time(3)). This value is suitable for passing to CTIME(), GMTIME(), and LTIME().

This intrinsic is not fully portable, such as to systems with 32-bit INTEGER types but supporting times wider than 32 bits. Therefore, the values returned by this intrinsic might be, or become, negative, or numerically less than previous values, during a single run of the compiled program.

See Section 6.191 [TIME8], page 134, for information on a similar intrinsic that might be portable to more GNU Fortran implementations, though to fewer Fortran compilers.

Standard: GNU extension

Class: Non-elemental function

Syntax: RESULT = TIME()

Return value:
The return value is a scalar of type INTEGER(4).

See also: Section 6.44 [CTIME], page 58, Section 6.88 [GMTIME], page 82, Section 6.126 [LTIME], page 102, Section 6.191 [TIME8], page 134

6.191 TIME8 — Time function (64-bit)

Description:
Returns the current time encoded as an integer (in the manner of the UNIX function `time(3)`). This value is suitable for passing to `CTIME()`, `GMTIME()`, and `LTIME()`.

Warning: this intrinsic does not increase the range of the timing values over that returned by `time(3)`. On a system with a 32-bit `time(3)`, `TIME8()` will return a 32-bit value, even though it is converted to a 64-bit `INTEGER(8)` value. That means overflows of the 32-bit value can still occur. Therefore, the values returned by this intrinsic might be or become negative or numerically less than previous values during a single run of the compiled program.

Standard: GNU extension

Class: Non-elemental function

Syntax: `RESULT = TIME8()`

Return value:
The return value is a scalar of type `INTEGER(8)`.

See also: Section 6.44 [CTIME], page 58, Section 6.88 [GMTIME], page 82, Section 6.126 [LTIME], page 102, Section 6.190 [TIME], page 133

6.192 TINY — Smallest positive number of a real kind

Description:
`TINY(X)` returns the smallest positive (non zero) number in the model of the type of `X`.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = TINY(X)`

Arguments:

X	Shall be of type `REAL`.

Return value:
The return value is of the same type and kind as *X*

Example: See `HUGE` for an example.

6.193 TRANSFER — Transfer bit patterns

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:
Return value:
Example:

See also:

6.194 TRANSPOSE — Transpose an array of rank two

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:
Return value:
Example:

See also:

6.195 TRIM — Function to remove trailing blank characters of a string

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:
Return value:
Example:

See also:

6.196 TTYNAM — Get the name of a terminal device.

Description:
Get the name of a terminal device. For more information, see `ttyname(3)`.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, non-elemental function

Syntax:

```
CALL TTYNAM(UNIT, NAME)
NAME = TTYNAM(UNIT)
```

Arguments:

UNIT	Shall be a scalar `INTEGER(*)`.
NAME	Shall be of type `CHARACTER(*)`.

Example:

```
PROGRAM test_ttynam
  INTEGER :: unit
  DO unit = 1, 10
    IF (isatty(unit=unit)) write(*,*) ttynam(unit)
  END DO
END PROGRAM
```

See also: Section 6.105 [ISATTY], page 91

6.197 UBOUND — Upper dimension bounds of an array

Description:
Returns the upper bounds of an array, or a single upper bound along the *DIM* dimension.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = UBOUND(ARRAY [, DIM])`

Arguments:

ARRAY	Shall be an array, of any type.
DIM	(Optional) Shall be a scalar `INTEGER(*)`.

Return value:
If *DIM* is absent, the result is an array of the upper bounds of *ARRAY*. If *DIM* is present, the result is a scalar corresponding to the upper bound of the array along that dimension. If *ARRAY* is an expression rather than a whole array or array structure component, or if it has a zero extent along the relevant dimension, the upper bound is taken to be the number of elements along the relevant dimension.

See also: Section 6.111 [LBOUND], page 94

6.198 UMASK — Set the file creation mask

Description:
Sets the file creation mask to *MASK* and returns the old value in argument *OLD* if it is supplied. See `umask(2)`.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL UMASK(MASK [, OLD])`

Arguments:

MASK	Shall be a scalar of type `INTEGER(*)`.
MASK	(Optional) Shall be a scalar of type `INTEGER(*)`.

6.199 `UNLINK` — Remove a file from the file system

Description:
Unlinks the file *PATH*. A null character (`CHAR(0)`) can be used to mark the end of the name in *PATH*; otherwise, trailing blanks in the file name are ignored. If the *STATUS* argument is supplied, it contains 0 on success or a nonzero error code upon return; see `unlink(2)`.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, non-elemental function

Syntax:

```
CALL UNLINK(PATH [, STATUS])
STATUS = UNLINK(PATH)
```

Arguments:

PATH	Shall be of default `CHARACTER` type.
STATUS	(Optional) Shall be of default `INTEGER` type.

See also: Section 6.116 [LINK], page 97, Section 6.185 [SYMLNK], page 131

6.200 `UNPACK` — Unpack an array of rank one into an array

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Transformational function

Syntax:

Arguments:
Return value:
Example:

See also: Section 6.148 [PACK], page 114

6.201 `VERIFY` — Scan a string for the absence of a set of characters

Intrinsic implemented, documentation pending.

Description:
Standard: F95 and later

Class: Elemental function

Syntax:

Arguments:
Return value:
Example:

Specific names:
See also:

6.202 XOR — Bitwise logical exclusive OR

Description:
Bitwise logical exclusive or.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. For integer arguments, programmers should consider the use of the Section 6.99 [IEOR], page 88 intrinsic defined by the Fortran standard.

Standard: GNU extension

Class: Non-elemental function

Syntax: `RESULT = XOR(X, Y)`

Arguments:

X	The type shall be either `INTEGER(*)` or `LOGICAL`.
Y	The type shall be either `INTEGER(*)` or `LOGICAL`.

Return value:
The return type is either `INTEGER(*)` or `LOGICAL` after cross-promotion of the arguments.

Example:

```
PROGRAM test_xor
  LOGICAL :: T = .TRUE., F = .FALSE.
  INTEGER :: a, b
  DATA a / Z,'F' /, b / Z'3' /

  WRITE (*,*) XOR(T, T), XOR(T, F), XOR(F, T), XOR(F, F)
  WRITE (*,*) XOR(a, b)
END PROGRAM
```

See also: F95 elemental function: Section 6.99 [IEOR], page 88

Contributing

Free software is only possible if people contribute to efforts to create it. We're always in need of more people helping out with ideas and comments, writing documentation and contributing code.

If you want to contribute to GNU Fortran, have a look at the long lists of projects you can take on. Some of these projects are small, some of them are large; some are completely orthogonal to the rest of what is happening on GNU Fortran, but others are "mainstream" projects in need of enthusiastic hackers. All of these projects are important! We'll eventually get around to the things here, but they are also things doable by someone who is willing and able.

Contributors to GNU Fortran

Most of the parser was hand-crafted by *Andy Vaught*, who is also the initiator of the whole project. Thanks Andy! Most of the interface with GCC was written by *Paul Brook*.

The following individuals have contributed code and/or ideas and significant help to the GNU Fortran project (in no particular order):

- Andy Vaught
- Katherine Holcomb
- Tobias Schlüter
- Steven Bosscher
- Toon Moene
- Tim Prince
- Niels Kristian Bech Jensen
- Steven Johnson
- Paul Brook
- Feng Wang
- Bud Davis
- Paul Thomas
- François-Xavier Coudert
- Steven G. Kargl
- Jerry Delisle
- Janne Blomqvist
- Erik Edelmann
- Thomas Koenig
- Asher Langton
- Jakub Jelinek
- Roger Sayle
- H.J. Lu
- Richard Henderson
- Richard Sandiford

- Richard Guenther
- Bernhard Fischer

The following people have contributed bug reports, smaller or larger patches, and much needed feedback and encouragement for the GNU Fortran project:

- Erik Schnetter
- Bill Clodius
- Kate Hedstrom

Many other individuals have helped debug, test and improve the GNU Fortran compiler over the past few years, and we welcome you to do the same! If you already have done so, and you would like to see your name listed in the list above, please contact us.

Projects

Help build the test suite
: Solicit more code for donation to the test suite. We can keep code private on request.

Bug hunting/squishing
: Find bugs and write more test cases! Test cases are especially very welcome, because it allows us to concentrate on fixing bugs instead of isolating them.

Smaller projects ("bug" fixes):
- Allow init exprs to be numbers raised to integer powers.
- Implement correct rounding.
- Implement F restrictions on Fortran 95 syntax.
- See about making Emacs-parsable error messages.

If you wish to work on the runtime libraries, please contact a project maintainer.

Proposed Extensions

Here's a list of proposed extensions for the GNU Fortran compiler, in no particular order. Most of these are necessary to be fully compatible with existing Fortran compilers, but they are not part of the official J3 Fortran 95 standard.

Compiler extensions:

- User-specified alignment rules for structures.
- Flag to generate `Makefile` info.
- Automatically extend single precision constants to double.
- Compile code that conserves memory by dynamically allocating common and module storage either on stack or heap.
- Compile flag to generate code for array conformance checking (suggest -CC).
- User control of symbol names (underscores, etc).
- Compile setting for maximum size of stack frame size before spilling parts to static or heap.

- Flag to force local variables into static space.
- Flag to force local variables onto stack.
- Flag for maximum errors before ending compile.
- Option to initialize otherwise uninitialized integer and floating point variables.

Environment Options

- Pluggable library modules for random numbers, linear algebra. LA should use BLAS calling conventions.
- Environment variables controlling actions on arithmetic exceptions like overflow, underflow, precision loss—Generate NaN, abort, default. action.
- Set precision for fp units that support it (i387).
- Variable for setting fp rounding mode.
- Variable to fill uninitialized variables with a user-defined bit pattern.
- Environment variable controlling filename that is opened for that unit number.
- Environment variable to clear/trash memory being freed.
- Environment variable to control tracing of allocations and frees.
- Environment variable to display allocated memory at normal program end.
- Environment variable for filename for * IO-unit.
- Environment variable for temporary file directory.
- Environment variable forcing standard output to be line buffered (unix).

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software

which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

   The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

   Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Funding Free Software

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers—the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, "We will donate ten dollars to the Frobnitz project for each disk sold." Don't be satisfied with a vague promise, such as "A portion of the profits are donated," since it doesn't give a basis for comparison.

Even a precise fraction "of the profits from this disk" is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU Compiler Collection contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is "the proper thing to do" when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z