Using GNU Fortran 95

The gfortran team

For the 4.1.2 Version*

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

Short Contents

Table of Contents

Introduction

This manual documents the use of `gfortran`, the GNU Fortran 95 compiler. You can find in this manual how to invoke `gfortran`, as well as its features and incompatibilities.

1 Getting Started

Gfortran is the GNU Fortran 95 compiler front end, 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.

Gfortran is still in an early state of development. `gfortran` can generate code for most constructs and expressions, but much work remains to be done.

When `gfortran` 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 or Fortran 95. 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.

Gfortran 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 95 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 gfortran 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.

2 GFORTRAN and GCC

GCC used to be the GNU “C” Compiler, but is now known as the *GNU Compiler Collection*. GCC provides the GNU system with a very versatile compiler middle end (shared optimization passes), and back ends (code generators) for many different computer architectures and operating systems. The code of the middle end and back end are shared by all compiler front ends that are in the GNU Compiler Collection.

A GCC front end is essentially a source code parser and an intermediate code generator. The code generator translates the semantics of the source code into a language independent form called *GENERIC*.

The parser takes a source file written in a particular computer language, reads and parses it, and tries to make sure that the source code conforms to the language rules. Once the correctness of a program has been established, the compiler will build a data structure known as the *Abstract Syntax tree*, or just *AST* or “tree” for short. This data structure represents the whole program or a subroutine or a function. The “tree” is passed to the GCC middle end, which will perform optimization passes on it. The optimized AST is then handed off too the back end which assembles the program unit.

Different phases in this translation process can be, and in fact *are* merged in many compiler front ends. GNU Fortran 95 has a strict separation between the parser and code generator.

The goal of the gfortran project is to build a new front end for GCC. Specifically, a Fortran 95 front end. In a non-gfortran installation, `gcc` will not be able to compile Fortran 95 source code (only the “C” front end has to be compiled if you want to build GCC, all other languages are optional). If you build GCC with gfortran, `gcc` will recognize ‘`.f/.f90/.f95`’ source files and accepts Fortran 95 specific command line options.

3 GFORTRAN and G77

Why do we write a compiler front end from scratch? There's a fine Fortran 77 compiler in the GNU Compiler Collection that accepts some features of the Fortran 90 standard as extensions. Why not start from there and revamp it?

One of the reasons is that Craig Burley, the author of G77, has decided to stop working on the G77 front end. On Craig explains the reasons for his decision to stop working on G77 (`http://world.std.com/~burley/g77-why.html`) in one of the pages in his homepage. Among the reasons is a lack of interest in improvements to `g77`. Users appear to be quite satisfied with `g77` as it is. While `g77` is still being maintained (by Toon Moene), it is unlikely that sufficient people will be willing to completely rewrite the existing code.

But there are other reasons to start from scratch. Many people, including Craig Burley, no longer agreed with certain design decisions in the G77 front end. Also, the interface of `g77` to the back end is written in a style which is confusing and not up to date on recommended practice. In fact, a full rewrite had already been planned for GCC 3.0.

When Craig decided to stop, it just seemed to be a better idea to start a new project from scratch, because it was expected to be easier to maintain code we develop ourselves than to do a major overhaul of `g77` first, and then build a Fortran 95 compiler out of it.

4 GNU Fortran 95 Command Options

The `gfortran` command supports all the options supported by the `gcc` command. Only options specific to gfortran 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 `gfortran` 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 `gfortran` to the `gcc` distribution enables acceptance of `gfortran` 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.

4.1 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 4.2 [Options Controlling Fortran Dialect], page 10.

```
 -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 -frange-check
```

Warning Options

See Section 4.3 [Options to Request or Suppress Warnings], page 11.

```
 -fsyntax-only -pedantic -pedantic-errors
-w -Wall -Waliasing -Wampersand -Wconversion -Wimplicit-interface
-Wnonstd-intrinsics -Wsurprising -Wunderflow
-Wunused-labels -Wline-truncation -W
```

Debugging Options

See Section 4.4 [Options for Debugging Your Program or GCC], page 13.

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

Directory Options

See Section 4.5 [Options for Directory Search], page 13.

```
-Idir -Mdir
```

Runtime Options

See Section 4.6 [Options for influencing runtime behavior], page 14.

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

Code Generation Options

See Section 4.7 [Options for Code Generation Conventions], page 14.

```
 -fno-automatic -ff2c -fno-underscoring -fsecond-underscore
-fbounds-check -fmax-stack-var-size=n
-fpackderived -frepack-arrays -fshort-enums
```

4.2 Options Controlling Fortran Dialect

The following options control the dialect of Fortran that the compiler accepts:

`-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.

`-fd-lines-as-code`
`-fd-lines-as-comment`
: Enables special treating for lines 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.

`-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`
: Compile switch to change the interpretation of a backslash 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 (corresponds to "extended-source" options in some popular compilers). *n* may 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. For free-form, 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 200x).

`-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`
: Enables the Cray pointer extension, which provides a C-like pointer.

`-frange-check`
: Enable range checking on results of simplification of constant expressions during compilation. For example, by default, `gfortran` 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`.

`-std=`*`std`*
: Conform to the specified standard. Allowed values for *std* are '`gnu`', '`f95`', '`f2003`' and '`legacy`'.

4.3 Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there might have been an error.

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 warnings produced by GNU Fortran:

`-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 `gfortran` in this area are welcome.

: This should be used in conjunction with -std=*std*.

`-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 '`-Wunused-labels`', '`-Waliasing`', '`-Wampersand`', '`-Wsurprising`', '`-Wnonstd-intrinsic`', 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 literals. The warning is given with '`-Wampersand`', '`-pedantic`', and '`-std=f95`'. Note: With no ampersand given in a continued character literal, gfortran assumes continuation at the first non-comment, non-whitespace character.

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

`-Wimplicit-interface`
Warn about when procedure are 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.

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

`-Wunused-labels`
Warn whenever a label is defined but never referenced.

`-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 `gfortran`

See section "Options to Request or Suppress 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.

4.4 Options for Debugging Your Program or GNU Fortran

GNU Fortran has various special options that are used for debugging either your program or `gfortran`

`-fdump-parse-tree`
: Output the internal parse tree before starting code generation. Only really useful for debugging gfortran 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 denormal value).

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

4.5 Options for Directory Search

These options affect how `gfortran` 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.

`-Idir` 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.

`-Mdir`

`-Jdir`
: 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.

4.6 Influencing runtime behavior

These options affect the runtime behavior of `gfortran`.

`-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 whatever `off_t` is specified to be on that particular system. Note that specifying *length* as 4 limits the record length of unformatted files to 2 GB. This option does not extend the maximum possible record length on systems where `off_t` is a four_byte quantity.

4.7 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, `gfortran` 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 `gfortran` is incompatible with `f2c` and `g77`, please use the '`-ff2c`' option if you want object files compiled with '`gfortran`' 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 `gfortran` code with other languages.

Note that just because the names match does *not* mean that the interface implemented by `gfortran` for an external name matches the interface implemented by some other language for that same name. That is, getting code produced by `gfortran` 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 `gfortran` 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, `gfortran` appends an underscore to external names. If this option is used `gfortran` appends two underscores to names with underscores and one underscore to external names with no underscores. (`gfortran` 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, eg. 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 `gfortran` may improve this behavior.

  The default value for *n* is 32768.

`-fpackderived`
: This option tells gfortran 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 `gfortran` may pass assumed shape array sections via a descriptor describing a discontiguous 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 discontiguous.

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

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.

4.8 Environment Variables Affecting GNU Fortran

GNU Fortran 95 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 6 [Runtime], page 23, for environment variables that affect the run-time behavior of `gfortran` programs.

5 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).

Gfortran is currently reaching the stage where is is able to compile real world programs. However it is still under development and has many rough edges.

5.1 Compiler Status

Front end This is the part of gfortran which parses a source file, verifies that it is valid Fortran 95, performs compile time replacement of constants (PARAMETER variables) and reads and generate module files. This is almost complete. Every Fortran 95 source should be accepted, and most none-Fortran 95 source should be rejected. If you find a source file where this is not true, please tell us. You can use the -fsyntax-only switch to make gfortran quit after running the front end, effectively reducing it to a syntax checker.

Middle end interface
These are the parts of gfortran that take the parse tree generated by the front end and translate it to the GENERIC form required by the GCC back end. Work is ongoing in these parts of gfortran, but a large part has already been completed.

5.2 Library Status

Some intrinsic functions map directly to library functions, and in most cases the name of the library function used depends on the type of the arguments. For some intrinsics we generate inline code, and for others, such as sin, cos and sqrt, we rely on the backend to use special instructions in the floating point unit of the CPU if available, or to fall back to a call to libm if these are not available.

Implementation of some non-elemental intrinsic functions (eg. DOT_PRODUCT, AVERAGE) is not yet optimal. This is hard because we have to make decisions whether to use inline code (good for small arrays as no function call overhead occurs) or generate function calls (good for large arrays as it allows use of hand-optimized assembly routines, SIMD instructions, etc.)

The IO library is in a mostly usable state. Unformatted I/O for `REAL(KIND=10)` variables is currently not recommended.

Array intrinsics mostly work.

5.3 Proposed Extensions

Here's a list of proposed extensions for `gfortran`, 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.

5.3.1 Compiler extensions:

- Flag for defining the kind number for default logicals.
- 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.
- Flag to cause the compiler to distinguish between upper and lower case names. The Fortran 95 standard does not distinguish them.
- 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 to compile lines beginning with “D”.
- Flag to ignore lines beginning with “D”.
- Flag for maximum errors before ending compile.
- Generate code to check for null pointer dereferences – prints locus of dereference instead of segfaulting. There was some discussion about this option in the g95 development mailing list.
- Allow setting the default unit number.
- Option to initialize otherwise uninitialized integer and floating point variables.
- Support for OpenMP directives. This also requires support from the runtime library and the rest of the compiler.
- Support for Fortran 200x. This includes several new features including floating point exceptions, extended use of allocatable arrays, C interoperability, Parameterizer data types and function pointers.

5.3.2 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).

6 Runtime: Influencing runtime behavior with environment variables

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

Malformed environment variables are silently ignored.

6.1 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.

6.2 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.

6.3 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.

6.4 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.

6.5 GFORTRAN_TMPDIR – Directory for scratch files

This environment variable controls where scratch files are created. Default is '/tmp'.

6.6 GFORTRAN_UNBUFFERED_ALL – Don't buffer output

This environment variable controls wether 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 bufferred. This is the default.

6.7 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.

6.8 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 lettter is 'n', 'N' or '0', a plus sign is not printed in most cases. Default is not to print plus signs.

6.9 GFORTRAN_DEFAULT_RECL – Default record lenght for new files

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

6.10 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 gfortran program that you want to run. Default is a single space.

6.11 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 7.13 [CONVERT specifier], page 31, for an alternative way to specify the data representation for unformatted files. See Section 4.6 [Runtime Options], page 14, for setting a default data representation for the whole program. The `CONVERT` specifier overrides the `-fconvert` compile options.

7 Extensions

gfortran implements a number of extensions over standard Fortran. This chapter contains information on their syntax and meaning. There are currently two categories of gfortran extensions, those that provide functionality beyond that provided by any standard, and those that are supported by gfortran 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.

7.1 Old-style kind specifications

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

```
TYPESPEC*k x,y,z
```

where TYPESPEC is a basic type, and where k is a valid kind number for that type. The statement then declares x, y and z to be of type TYPESPEC with kind k. In other words, it is equivalent to the standard conforming declaration

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

7.2 Old-style variable initialization

gfortran allows old-style initialization of variables of the form:

```
INTEGER*4 i/1/,j/2/
REAL*8 x(2,2) /3*0.,1./
```

These are only allowed in declarations without double colons (::), as these were introduced in Fortran 90 which also introduced a new syntax for variable initializations. The syntax for the individual initializers is as for the DATA statement, but unlike in a DATA statement, an initializer only applies to the variable immediately preceding. In other words, something like INTEGER I,J/2,3/ is not valid.

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

```
! Fortran 90
      INTEGER(4) :: i = 1, j = 2
      REAL(8) :: x(2,2) = RESHAPE((/0.,0.,0.,1./),SHAPE(x))
! Fortran 77
      INTEGER  i, j
      DOUBLE PRECISION x(2,2)
      DATA i,j,x /1,2,3*0.,1./
```

7.3 Extensions to namelist

gfortran 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 noticed 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
```

7.4 X format descriptor

To support legacy codes, `gfortran` 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)
```

7.5 Commas in FORMAT specifications

To support legacy codes, `gfortran` 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)
```

7.6 I/O item lists

To support legacy codes, `gfortran` 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.

7.7 Hexadecimal constants

As a GNU extension, `gfortran` allows hexadecimal constants to be specified using the X prefix, in addition to the standard Z prefix.

7.8 Real array indices

As a GNU extension, `gfortran` allows arrays to be indexed using real types, whose values are implicitly converted to integers.

7.9 Unary operators

As a GNU extension, `gfortran` 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
```

7.10 Implicitly interconvert LOGICAL and INTEGER

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

```
INTEGER*4 i
i = .FALSE.
```

7.11 Hollerith constants support

A Hollerith constant is a string of characters preceded by the letter '`H`' or '`h`', and there must be an literal, unsigned, nonzero default integer constant indicating the number of characters in the string. Hollerith constants are stored as byte strings, one character per byte.

`gfortran` supports Hollerith constants. They can be used as the right hands in the `DATA` statement and `ASSIGN` statement, also as the arguments. The left hands can be of Integer, Real, Complex and Logical type. The constant will be padded or truncated to fit the size of left hand.

Valid Hollerith constants examples:

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

Invalid Hollerith constants examples:

```
integer*4 a
a = 8H12345678 ! The Hollerith constant is too long. It will be truncated.
a = 0H         ! At least one character needed.
```

7.12 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 one familiar with C pointer arithmetic might expect. 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(), loc() is essentially the C '&' operator, 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 a malloc-type function. There is no malloc intrinsic implemented as part of the Cray pointer extension, but it might be a useful future addition to `gfortran`. Even without an intrinsic malloc function, dynamic memory allocation can be combined with Cray pointers by calling a short C function:

```
mymalloc.c:

        void mymalloc_(void **ptr, int *nbytes)
        {
            *ptr = malloc(*nbytes);
            return;
        }

caller.f:

        program caller
        integer ipinfo;
```

```
real*4 data
pointer (ipdata, data(1024))
call mymalloc(ipdata,4*1024)
end
```

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're 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 gfortran; 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 attributes ALLOCATABLE, INTENT, OPTIONAL, DUMMY, TARGET, EXTERNAL, INTRINSIC, or POINTER. Pointers may not have the attributes DIMENSION, POINTER, TARGET, ALLOCATABLE, EXTERNAL, or INTRINSIC. 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 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.

7.13 CONVERT specifier

gfortran 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 6.11 [GFORTRAN_CONVERT_UNIT], page 24, 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 sytems of kinds 4 and 8. Conversion between different "extended double" types on different architectures such as m68k and x86_64, which gfortran supports as `REAL(KIND=10)` 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 a user who does not have the source code of his 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.

8 Intrinsic Procedures

This portion of the document is incomplete and undergoing massive expansion and editing. All contributions and corrections are strongly encouraged.

8.1 Introduction to intrinsic procedures

Gfortran provides a rich set of intrinsic procedures that includes all the intrinsic procedures required by the Fortran 95 standard, a set of intrinsic procedures for backwards compatibility with Gnu Fortran 77 (i.e., `g77`), and a small selection of intrinsic procedures from the Fortran 2003 standard. Any description here, which conflicts with 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. Gfortran 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 intrinsics 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.

`Gfortran` 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 option(s) is noted.

8.2 `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.

Option: gnu

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
```

8.3 ABS — Absolute value

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

Option: f95, gnu

Class: elemental function

Syntax: X = 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	Option
CABS(Z)	COMPLEX(4) Z	REAL(4)	f95, gnu
DABS(X)	REAL(8) X	REAL(8)	f95, gnu
IABS(I)	INTEGER(4) I	INTEGER(4)	f95, gnu
ZABS(Z)	COMPLEX(8) Z	COMPLEX(8)	gnu
CDABS(Z)	COMPLEX(8) Z	COMPLEX(8)	gnu

8.4 ACHAR — Character in ASCII collating sequence

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

Option: f95, gnu

Class: elemental function

Syntax: C = 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
```

8.5 ACOS — Arc cosine function

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

Option: f95, gnu

Class: elemental function

Syntax: `X = 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 $0 \leq \arccos(x) \leq \pi$. The kind type parameter is the same as *X*.

Example:

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

Specific names:

Name	Argument	Return type	Option
`DACOS(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu

8.6 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.

Option: f95, gnu

Class: elemental function

Syntax: `STR = 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
```

8.7 `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.

Option: f95, gnu

Class: elemental function

Syntax: `STR = 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
```

8.8 `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.

Option: f95, gnu

Class: elemental function

Syntax: `X = 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	Option
`DIMAG(Z)`	`COMPLEX(8) Z`	`REAL(8)`	f95, gnu
`IMAG(Z)`	`COMPLEX(*) Z`	`REAL(*)`	gnu
`IMAGPART(Z)`	`COMPLEX(*) Z`	`REAL(*)`	gnu

8.9 `AINT` — Imaginary part of complex number

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

Option: f95, gnu

Class: elemental function

Syntax: `X = AINT(X) X = AINT(X, KIND)`

Arguments:

X	The type of the argument shall be `REAL(*)`.
KIND	(Optional) *KIND* shall be a scalar integer initialization expression.

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	Option
`DINT(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu

8.10 `ALARM` — Execute a routine after a given delay

Description:
`ALARM(SECONDS [, STATUS])` causes external subroutine *HANDLER* to be executed after a delay of *SECONDS* by using `alarm(1)` 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.

Option: gnu

Class: subroutine

Syntax: `CALL ALARM(SECONDS, HANDLER) 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. `INTEGER`. It is `INTENT(IN)`.
STATUS	(Optional) *STATUS* shall be a scalar `INTEGER` variable. 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.

8.11 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*.

Option: f95, gnu

Class: transformational function

Syntax: `L = ALL(MASK) L = 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
```

8.12 `ALLOCATED` — Status of an allocatable entity

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

Option: f95, gnu

Class: inquiry function

Syntax: `L = 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
```

8.13 `ANINT` — Nearest whole number

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

Option: f95, gnu

Class: elemental function

Syntax: `X = ANINT(X) X = ANINT(X, KIND)`

Arguments:

X	The type of the argument shall be `REAL(*)`.
KIND	(Optional) *KIND* shall be a scalar integer initialization expression.

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 return `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	Option
`DNINT(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu

8.14 `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.`.

Option: f95, gnu

Class: transformational function

Syntax: `L = ANY(MASK) L = 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
```

8.15 `ASIN` — Arcsine function

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

Option: f95, gnu

Class: elemental function

Syntax: `X = 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 $-\pi/2 \leq \arccos(x) \leq \pi/2$. 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	Option
`DASIN(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu

8.16 `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*.

Option: f95, gnu

Class: inquiry function

Syntax: `L = ASSOCIATED(PTR) L = 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
```

8.17 `ATAN` — Arctangent function

Description:

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

Option: f95, gnu

Class: elemental function

Syntax: `X = ATAN(X)`

Arguments:

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

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

Example:

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

Specific names:

Name	Argument	Return type	Option
`DATAN(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu

8.18 `ATAN2` — Arctangent function

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

Option: f95, gnu

Class: elemental function

Syntax: `X = 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 principle value of the complex number $X + iY$. If *X* is nonzero, then it lies in the range $-\pi \leq \arccos(x) \leq \pi$. 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	Option
`DATAN2(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu

8.19 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*.

Option: gnu

Class: elemental function

Syntax: `X = 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	Option
`DBESJ0(X)`	`REAL(8) X`	`REAL(8)`	gnu

8.20 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*.

Option: gnu

Class: elemental function

Syntax: `X = 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	Option
`DBESJ1(X)`	`REAL(8) X`	`REAL(8)`	gnu

8.21 BESJN — Bessel function of the first kind

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

Option: gnu

Class: elemental function

Syntax: `Y = 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	Option
`DBESJN(X)`	`INTEGER(*) N` `REAL(8) X`	`REAL(8)`	gnu

8.22 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*.

Option: gnu

Class: elemental function

Syntax: `X = 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	Option
`DBESY0(X)`	`REAL(8) X`	`REAL(8)`	gnu

8.23 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*.

Option: gnu

Class: elemental function

Syntax: X = 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	Option
DBESY1(X)	REAL(8) X	REAL(8)	gnu

8.24 BESYN — Bessel function of the second kind

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

Option: gnu

Class: elemental function

Syntax: Y = 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	Option
DBESYN(N,X)	INTEGER(*) N REAL(8) X	REAL(8)	gnu

8.25 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*.

Option: f95, gnu

Class: elemental function

Syntax: I = 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
```

8.26 BTEST — Bit test function

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

Option: f95, gnu

Class: elemental function

Syntax: I = 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
```

8.27 `CEILING` — Integer ceiling function

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

Option: f95, gnu

Class: elemental function

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

Arguments:

X	The type shall be `REAL(*)`.
KIND	Optional scaler integer initialization expression.

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
```

8.28 `CHAR` — Character conversion function

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

Option: f95, gnu

Class: elemental function

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

Arguments:

I	The type shall be `INTEGER(*)`.
KIND	Optional scaler integer initialization expression.

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
```

8.29 `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.

Option: f95, gnu

Class: elemental function

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

Arguments:

X	The type may be `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)`.
Y	Optional, allowed if X is not `COMPLEX(*)`. May be `INTEGER(*)` or `REAL(*)`.
KIND	Optional scaler integer initialization expression.

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
```

8.30 `COMMAND_ARGUMENT_COUNT` — Argument count function

Description:

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

Option: f2003, gnu

Class: non-elemental function

Syntax: `I = 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
```

8.31 `CONJG` — Complex conjugate function

Description:

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

Option: f95, gnu

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	Option
`DCONJG(Z)`	`COMPLEX(8) Z`	`COMPLEX(8)`	gnu

8.32 `COS` — Cosine function

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

Option: f95, gnu

Class: elemental function

Syntax: `X = COS(X)`

Arguments:

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

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

Example:

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

Specific names:

Name	Argument	Return type	Option
`DCOS(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu
`CCOS(X)`	`COMPLEX(4) X`	`COMPLEX(4)`	f95, gnu
`ZCOS(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	f95, gnu
`CDCOS(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	f95, gnu

8.33 COSH — Hyperbolic cosine function

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

Option: f95, gnu

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	Option
`DCOSH(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu

8.34 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*.

Option: f95, gnu

Class: transformational function

Syntax: `I = 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
```

8.35 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.

Option: f95, gnu

Class: subroutine

Syntax: CPU_TIME(X)

Arguments:

X	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
```

8.36 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.

Option: f95, gnu

Class: transformational function

Syntax: A = 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
```

8.37 `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.

Option: gnu

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
```

8.38 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

Option: f95, gnu

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
```

8.39 DBLE — Double conversion function

Description:
`DBLE(X)` Converts X to double precision real type. `DFLOAT` is an alias for `DBLE`

Option: f95, gnu

Class: elemental function

Syntax: `X = DBLE(X) X = DFLOAT(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), dfloat(z)
end program test_dble
```

8.40 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.

Option: f95, gnu

Class: elemental function

Syntax: `C = DCMPLX(X) C = 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
```

8.41 DFLOAT — Double conversion function

Description:
DFLOAT(X) Converts *X* to double precision real type. DFLOAT is an alias for DBLE. See DBLE.

8.42 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.

Option: f95, gnu

Class: inquiry function

Syntax: C = 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
```

8.43 DIM — Dim function

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

Option: f95, gnu

Class: elemental function

Syntax: X = 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	Option
IDIM(X,Y)	INTEGER(4) X,Y	INTEGER(4)	gnu
DDIM(X,Y)	REAL(8) X,Y	REAL(8)	gnu

8.44 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).

Option: f95

Class: transformational function

Syntax: S = 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
```

8.45 DPROD — Double product function

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

Option: f95, gnu

Class: elemental function

Syntax: `D = 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
```

8.46 `DREAL` — Double real part function

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

Option: gnu

Class: elemental function

Syntax: `D = 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
```

8.47 `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 (wraparounds) 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.

Option: gnu

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
```

8.48 `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.

Option: f95, gnu

Class: transformational function

Syntax: `A = 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
```

8.49 `EPSILON` — Epsilon function

Description:

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

Option: f95, gnu

Class: inquiry function

Syntax: `C = 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
```

8.50 ERF — Error function

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

Option: gnu

Class: elemental function

Syntax: `X = 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	Option
`DERF(X)`	`REAL(8) X`	`REAL(8)`	gnu

8.51 ERFC — Error function

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

Option: gnu

Class: elemental function

Syntax: `X = 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	Option
`DERFC(X)`	`REAL(8) X`	`REAL(8)`	gnu

8.52 `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 (wraparounds) 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.

Option: gnu

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
```

8.53 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.

Option: gnu

Class: non-elemental 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
```

8.54 EXP — Exponential function

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

Option: f95, gnu

Class: elemental function

Syntax: X = 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	Option
DEXP(X)	REAL(8) X	REAL(8)	f95, gnu
CEXP(X)	COMPLEX(4) X	COMPLEX(4)	f95, gnu

ZEXP(X)	COMPLEX(8) X	COMPLEX(8)	f95, gnu
CDEXP(X)	COMPLEX(8) X	COMPLEX(8)	f95, gnu

8.55 EXPONENT — Exponent function

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

Option: f95, gnu

Class: elemental function

Syntax: I = 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
```

8.56 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, TIME8()).

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.

Option: gnu

Class: subroutine

Syntax:

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

Arguments:

DATE	The type shall be of type CHARACTER.

Return value:
The current date and time 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
```

8.57 FLOAT — Convert integer to default real

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

Option: gnu

Class: function

Syntax: X = 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
```

8.58 FLOOR — Integer floor function

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

Option: f95, gnu

Class: elemental function

Syntax: I = FLOOR(X[,KIND])

Arguments:

X	The type shall be REAL(*).
KIND	Optional scaler integer initialization expression.

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
```

8.59 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.

Option: gnu

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 prefered over the `FLUSH` intrinsic.

8.60 FNUM — File number function

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

Option: gnu

Class: non-elemental function

Syntax: `I = 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
```

8.61 FRACTION — Fractional part of the model representation

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

Option: f95, gnu

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
```

8.62 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 `gfortran` to allow user to compile legacy code. For new code using Fortran 95 pointers, the memory de-allocation intrinsic is `DEALLOCATE`.

Option: gnu

Class: subroutine

Syntax: `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.

8.63 GETGID — Group ID function

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

Option: gnu

Class: function

Syntax: `I = GETGID()`

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

Example: See `GETPID` for an example.

8.64 GETPID — Process ID function

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

Option: gnu

Class: function

Syntax: `I = 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
```

8.65 GETUID — User ID function

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

Option: gnu

Class: function

Syntax: `GETUID()`

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

Example: See `GETPID` for an example.

8.66 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`.

Option: f95, gnu

Class: elemental function

Syntax: `Y = 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
```

8.67 IACHAR — Code in ASCII collating sequence

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

Option: f95, gnu

Class: elemental function

Syntax: `I = 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
```

8.68 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 character and their codes is not necessarily the same between GNU Fortran implementations.

Option: f95, gnu

Class: elemental function

Syntax: `I = 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 a printable character string to a numerical value. For example, there is no intrinsic that, given the `CHARACTER` value 154, returns an `INTEGER` or `REAL` value with the value 154.
Instead, you can use internal-file I/O to do this kind of conversion. For example:

```
program read_val
  integer value
  character(len=10) string
```

```
  string = '154'
  read (string,'(I10)') value
  print *, value
end program read_val
```

8.69 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.

Option: gnu

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
```

8.70 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.

Option: gnu

Class: non-elemental function

Syntax: I = 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
```

8.71 `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.

Option: gnu

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
```

8.72 `KIND` — Kind of an entity

Description:
: `KIND(X)` returns the kind value of the entity *X*.

Option: f95, gnu

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
```

8.73 LOC — Returns the address of a variable

Description:
`LOC(X)` returns the address of *X* as an integer.

Option: gnu

Class: inquiry function

Syntax: `I = LOC(X)`

Arguments:

X	Variable of any type.

Return value:
The return value is of type `INTEGER(n)`, where `n` is 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
```

8.74 LOG — Logarithm function

Description:
`LOG(X)` computes the logarithm of *X*.

Option: f95, gnu

Class: elemental function

Syntax: `X = 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	Option
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

8.75 LOG10 — Base 10 logarithm function

Description:
: `LOG10(X)` computes the base 10 logarithm of *X*.

Option: f95, gnu

Class: elemental function

Syntax: `X = 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	Option
ALOG10(X)	REAL(4) X	REAL(4)	f95, gnu
DLOG10(X)	REAL(8) X	REAL(8)	f95, gnu

8.76 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 `gfortran` to allow user to compile legacy code. For new code using Fortran 95 pointers, the memory allocation intrinsic is `ALLOCATE`.

Option: gnu

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
```

8.77 `MAXEXPONENT` — Maximum exponent of a real kind

Description:
`MAXEXPONENT(X)` returns the maximum exponent in the model of the type of `X`.

Option: f95, gnu

Class: elemental function

Syntax: `I = 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
```

8.78 `MINEXPONENT` — Minimum exponent of a real kind

Description:
`MINEXPONENT(X)` returns the minimum exponent in the model of the type of `X`.

Option: f95, gnu

Class: elemental function

Syntax: `I = 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.

8.79 `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)`.

Option: f95, gnu

Class: elemental function

Syntax: `X = 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	Option
`AMOD(A,P)`	`REAL(4)`	`REAL(4)`	f95, gnu
`DMOD(A,P)`	`REAL(8)`	`REAL(8)`	f95, gnu

8.80 MODULO — Modulo function

Description:
`MODULO(A,P)` computes the *A* modulo *P*.

Option: f95, gnu

Class: elemental function

Syntax: `X = 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_mod
  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
```

Specific names:

Name	Arguments	Return type	Option
`AMOD(A,P)`	`REAL(4)`	`REAL(4)`	f95, gnu
`DMOD(A,P)`	`REAL(8)`	`REAL(8)`	f95, gnu

8.81 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`.

Option: f95, gnu

Class: elemental function

Syntax: `Y = 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
```

8.82 `NINT` — Nearest whole number

Description:

`NINT(X)` rounds its argument to the nearest whole number.

Option: f95, gnu

Class: elemental function

Syntax: `X = 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	Option
`IDNINT(X)`	`REAL(8)`	f95, gnu

8.83 `PRECISION` — Decimal precision of a real kind

Description:

`PRECISION(X)` returns the decimal precision in the model of the type of `X`.

Option: f95, gnu

Class: elemental function

Syntax: `I = 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
```

8.84 `RADIX` — Base of a model number

Description:

`RADIX(X)` returns the base of the model representing the entity *X*.

Option: f95, gnu

Class: inquiry function

Syntax: `R = 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
```

8.85 `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`.

Option: gnu

Class: non-elemental function

Syntax: `X = 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
```

Note: For compatibility with HP FORTRAN 77/iX, the `RAN` intrinsic is provided as an alias for `RAND`.

8.86 `RANGE` — Decimal exponent range of a real kind

Description:
`RANGE(X)` returns the decimal exponent range in the model of the type of `X`.

Option: f95, gnu

Class: elemental function

Syntax: `I = 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.

8.87 `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.

Option: f95, gnu

Class: transformational function

Syntax:

```
X = REAL(X)
X = REAL(X, KIND)
X = REALPART(Z)
```

Arguments:

X	shall be `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)`.
KIND	(Optional) *KIND* shall be a scalar integer.

Return value:
These functions return the 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
```

8.88 RRSPACING — Reciprocal of the relative spacing

Description:
RRSPACING(X) returns the reciprocal of the relative spacing of model numbers near *X*.

Option: f95, gnu

Class: elemental function

Syntax: Y = 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).

8.89 SCALE — Scale a real value

Description:
SCALE(X,I) returns X * RADIX(X)**I.

Option: f95, gnu

Class: elemental function

Syntax: Y = 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
```

8.90 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 accomodates this range, `SELECTED_INT_KIND` returns -1.

Option: f95

Class: transformational function

Syntax:
`J = 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
```

8.91 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`.

Option: f95

Class: transformational function

Syntax:
`I = 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
```

8.92 SECNDS — Time subroutine

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.

Option: gnu

Class: function

Syntax: T = 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
```

8.93 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 if *I*.

Option: f95, gnu

Class: elemental function

Syntax: Y = 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
```

8.94 SIGN — Sign copying function

Description:
SIGN(A,B) returns the value of *A* with the sign of *B*.

Option: f95, gnu

Class: elemental function

Syntax: X = 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	Option
`ISIGN(A,P)`	`INTEGER(4)`	`INTEGER(4)`	f95, gnu
`DSIGN(A,P)`	`REAL(8)`	`REAL(8)`	f95, gnu

8.95 `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)`.

Option: gnu

Class: subroutine, non-elemental function

Syntax:

`CALL ALARM(NUMBER, HANDLER)`

`CALL ALARM(NUMBER, HANDLER, STATUS)`

`STATUS = ALARM(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` functions 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
```

8.96 `SIN` — Sine function

Description:
`SIN(X)` computes the sine of *X*.

Option: f95, gnu

Class: elemental function

Syntax: `X = SIN(X)`

Arguments:

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

Return value:

The return value has same type and king than *X*.

Example:

```
program test_sin
  real :: x = 0.0
  x = sin(x)
end program test_sin
```

Specific names:

Name	Argument	Return type	Option
`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

8.97 `SINH` — Hyperbolic sine function

Description:

`SINH(X)` computes the hyperbolic sine of *X*.

Option: f95, gnu

Class: elemental function

Syntax: `X = 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	Option
`DSINH(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu

8.98 `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*.

Option: gnu

Class: function

Syntax: `X = SNGL(A)`

Arguments:

A	The type shall be a double precision `REAL`.

Return value:
The return value is of type default `REAL`.

8.99 `SQRT` — Square-root function

Description:
`SQRT(X)` computes the square root of *X*.

Option: f95, gnu

Class: elemental function

Syntax: `X = 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	Option
`DSQRT(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu
`CSQRT(X)`	`COMPLEX(4) X`	`COMPLEX(4)`	f95, gnu
`ZSQRT(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	f95, gnu
`CDSQRT(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	f95, gnu

8.100 `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*.

Option: gnu

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 `gfortran`, 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 numbers generators.

8.101 `TAN` — Tangent function

Description:
`TAN(X)` computes the tangent of *X*.

Option: f95, gnu

Class: elemental function

Syntax: `X = 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	Option
`DTAN(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu

8.102 `TANH` — Hyperbolic tangent function

Description:
`TANH(X)` computes the hyperbolic tangent of *X*.

Option: f95, gnu

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	Option
`DTANH(X)`	`REAL(8) X`	`REAL(8)`	f95, gnu

8.103 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`.

Option: f95, gnu

Class: elemental function

Syntax: `Y = 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.

9 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 95, 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 `gfortran`, 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.

9.1 Contributors to GNU Fortran 95

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 gfortran project (in no particular order):

- Andy Vaught
- Katherine Holcomb
- Tobias Schlter
- Steven Bosscher
- Toon Moene
- Tim Prince
- Niels Kristian Bech Jensen
- Steven Johnson
- Paul Brook
- Feng Wang
- Bud Davis
- Paul Thomas
- Franois-Xavier Coudert
- Steve Kargl
- Jerry Delisle
- Janne Blomqvist
- Erik Edelmann
- Thomas Koenig
- Asher Langton

The following people have contributed bug reports, smaller or larger patches, and much needed feedback and encouragement for the `gfortran` project:

- Erik Schnetter
- Bill Clodius
- Kate Hedstrom

Many other individuals have helped debug, test and improve `gfortran` 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.

9.2 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.

10 Standards

The GNU Fortran 95 Compiler aims to be a conforming implementation of ISO/IEC 1539:1997 (Fortran 95).

In the future it may also support other variants of and extensions to the Fortran language. These include ANSI Fortran 77, ISO Fortran 90, ISO Fortran 2003 and OpenMP.

10.1 Fortran 2003 status

Although `gfortran` 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`, and `get_environment_variable`.
- 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.

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

-

[

A

B

C

D

E

N

O

P

R

S