Using GNU Fortran

For GCC version 4.3.6

(GCC)

The gfortran team

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

Short Contents

1 Introduction 1

Part I: Invoking GNU Fortran **5**

2 GNU Fortran Command Options 7

3 Runtime: Influencing runtime behavior with environment variables 19

Part II: Language Reference **23**

4 Fortran 2003 Status 25

5 Extensions 27

6 Intrinsic Procedures 37

7 Intrinsic Modules 163

Contributing 167

GNU General Public License 171

GNU Free Documentation License 183

Funding Free Software 191

Option Index 193

Keyword Index 195

Table of Contents

1 Introduction **1**

1.1 About GNU Fortran 1
1.2 GNU Fortran and GCC 2
1.3 Preprocessing and conditional compilation 2
1.4 GNU Fortran and G77 3
1.5 Project Status 3
1.6 Standards 4

Part I: Invoking GNU Fortran **5**

2 GNU Fortran Command Options **7**

2.1 Option summary 7
2.2 Options controlling Fortran dialect 8
2.3 Options to request or suppress errors and warnings 10
2.4 Options for debugging your program or GNU Fortran 12
2.5 Options for directory search 13
2.6 Influencing the linking step 13
2.7 Influencing runtime behavior 14
2.8 Options for code generation conventions 14
2.9 Environment variables affecting `gfortran` 18

3 Runtime: Influencing runtime behavior with environment variables **19**

3.1 `GFORTRAN_STDIN_UNIT`—Unit number for standard input 19
3.2 `GFORTRAN_STDOUT_UNIT`—Unit number for standard output 19
3.3 `GFORTRAN_STDERR_UNIT`—Unit number for standard error 19
3.4 `GFORTRAN_USE_STDERR`—Send library output to standard error 19
3.5 `GFORTRAN_TMPDIR`—Directory for scratch files 19
3.6 `GFORTRAN_UNBUFFERED_ALL`—Don't buffer I/O on all units 19
3.7 `GFORTRAN_UNBUFFERED_PRECONNECTED`—Don't buffer I/O on preconnected units 19
3.8 `GFORTRAN_SHOW_LOCUS`—Show location for runtime errors 19
3.9 `GFORTRAN_OPTIONAL_PLUS`—Print leading + where permitted 20
3.10 `GFORTRAN_DEFAULT_RECL`—Default record length for new files 20
3.11 `GFORTRAN_LIST_SEPARATOR`—Separator for list output 20
3.12 `GFORTRAN_CONVERT_UNIT`—Set endianness for unformatted I/O 20
3.13 `GFORTRAN_ERROR_DUMPCORE`—Dump core on run-time errors 21
3.14 `GFORTRAN_ERROR_BACKTRACE`—Show backtrace on run-time errors 21

Part II: Language Reference 23

4 Fortran 2003 Status 25

5 Extensions 27

5.1 Extensions implemented in GNU Fortran 27
5.1.1 Old-style kind specifications 27
5.1.2 Old-style variable initialization 27
5.1.3 Extensions to namelist 28
5.1.4 `X` format descriptor without count field 29
5.1.5 Commas in `FORMAT` specifications 29
5.1.6 Missing period in `FORMAT` specifications 29
5.1.7 I/O item lists 29
5.1.8 BOZ literal constants.................................... 29
5.1.9 Real array indices 30
5.1.10 Unary operators 30
5.1.11 Implicitly convert `LOGICAL` and `INTEGER` values 30
5.1.12 Hollerith constants support............................. 30
5.1.13 Cray pointers 31
5.1.14 `CONVERT` specifier.................................... 33
5.1.15 OpenMP.. 33
5.1.16 Argument list functions `%VAL`, `%REF` and `%LOC` 34
5.2 Extensions not implemented in GNU Fortran 34
5.2.1 `STRUCTURE` and `RECORD` 35
5.2.2 `ENCODE` and `DECODE` statements 36

6 Intrinsic Procedures.............................. 37

6.1 Introduction to intrinsic procedures 37
6.2 `ABORT` — Abort the program 37
6.3 `ABS` — Absolute value... 38
6.4 `ACCESS` — Checks file access modes.......................... 38
6.5 `ACHAR` — Character in ASCII collating sequence................ 39
6.6 `ACOS` — Arccosine function.................................. 40
6.7 `ACOSH` — Hyperbolic arccosine function....................... 40
6.8 `ADJUSTL` — Left adjust a string 41
6.9 `ADJUSTR` — Right adjust a string............................ 41
6.10 `AIMAG` — Imaginary part of complex number 42
6.11 `AINT` — Truncate to a whole number......................... 42
6.12 `ALARM` — Execute a routine after a given delay 43
6.13 `ALL` — All values in *MASK* along *DIM* are true 44
6.14 `ALLOCATED` — Status of an allocatable entity 45
6.15 `AND` — Bitwise logical AND 45
6.16 `ANINT` — Nearest whole number.............................. 46
6.17 `ANY` — Any value in *MASK* along *DIM* is true 46
6.18 `ASIN` — Arcsine function.................................... 47
6.19 `ASINH` — Hyperbolic arcsine function......................... 48
6.20 `ASSOCIATED` — Status of a pointer or pointer/target pair 48

6.21 ATAN — Arctangent function 49
6.22 ATAN2 — Arctangent function 50
6.23 ATANH — Hyperbolic arctangent function 50
6.24 BESJ0 — Bessel function of the first kind of order 0 51
6.25 BESJ1 — Bessel function of the first kind of order 1 52
6.26 BESJN — Bessel function of the first kind 52
6.27 BESY0 — Bessel function of the second kind of order 0 53
6.28 BESY1 — Bessel function of the second kind of order 1 53
6.29 BESYN — Bessel function of the second kind 54
6.30 BIT_SIZE — Bit size inquiry function 54
6.31 BTEST — Bit test function 55
6.32 C_ASSOCIATED — Status of a C pointer 55
6.33 C_FUNLOC — Obtain the C address of a procedure 56
6.34 C_F_PROCPOINTER — Convert C into Fortran procedure pointer 56
6.35 C_F_POINTER — Convert C into Fortran pointer 57
6.36 C_LOC — Obtain the C address of an object 58
6.37 CEILING — Integer ceiling function 58
6.38 CHAR — Character conversion function 59
6.39 CHDIR — Change working directory 59
6.40 CHMOD — Change access permissions of files 60
6.41 CMPLX — Complex conversion function 61
6.42 COMMAND_ARGUMENT_COUNT — Get number of command line arguments 62
6.43 COMPLEX — Complex conversion function 62
6.44 CONJG — Complex conjugate function 63
6.45 COS — Cosine function 63
6.46 COSH — Hyperbolic cosine function 64
6.47 COUNT — Count function 64
6.48 CPU_TIME — CPU elapsed time in seconds 65
6.49 CSHIFT — Circular shift elements of an array 66
6.50 CTIME — Convert a time into a string 66
6.51 DATE_AND_TIME — Date and time subroutine 67
6.52 DBLE — Double conversion function 68
6.53 DCMPLX — Double complex conversion function 69
6.54 DFLOAT — Double conversion function 69
6.55 DIGITS — Significant digits function 70
6.56 DIM — Positive difference 70
6.57 DOT_PRODUCT — Dot product function 71
6.58 DPROD — Double product function 71
6.59 DREAL — Double real part function 72
6.60 DTIME — Execution time subroutine (or function) 72
6.61 EOSHIFT — End-off shift elements of an array 73
6.62 EPSILON — Epsilon function 74
6.63 ERF — Error function 75
6.64 ERFC — Error function 75
6.65 ETIME — Execution time subroutine (or function) 76
6.66 EXIT — Exit the program with status. 77

6.67 EXP — Exponential function 77
6.68 EXPONENT — Exponent function 78
6.69 FDATE — Get the current time as a string 78
6.70 FLOAT — Convert integer to default real 79
6.71 FGET — Read a single character in stream mode from stdin ... 79
6.72 FGETC — Read a single character in stream mode 80
6.73 FLOOR — Integer floor function 81
6.74 FLUSH — Flush I/O unit(s) 81
6.75 FNUM — File number function 82
6.76 FPUT — Write a single character in stream mode to stdout 82
6.77 FPUTC — Write a single character in stream mode 83
6.78 FRACTION — Fractional part of the model representation 84
6.79 FREE — Frees memory 84
6.80 FSEEK — Low level file positioning subroutine 85
6.81 FSTAT — Get file status 86
6.82 FTELL — Current stream position 86
6.83 GAMMA — Gamma function 87
6.84 GERROR — Get last system error message 87
6.85 GETARG — Get command line arguments 88
6.86 GET_COMMAND — Get the entire command line 89
6.87 GET_COMMAND_ARGUMENT — Get command line arguments 89
6.88 GETCWD — Get current working directory 90
6.89 GETENV — Get an environmental variable 90
6.90 GET_ENVIRONMENT_VARIABLE — Get an environmental variable 91
6.91 GETGID — Group ID function 91
6.92 GETLOG — Get login name 92
6.93 GETPID — Process ID function 92
6.94 GETUID — User ID function 93
6.95 GMTIME — Convert time to GMT info 93
6.96 HOSTNM — Get system host name 94
6.97 HUGE — Largest number of a kind 94
6.98 IACHAR — Code in ASCII collating sequence 94
6.99 IAND — Bitwise logical and 95
6.100 IARGC — Get the number of command line arguments 96
6.101 IBCLR — Clear bit 96
6.102 IBITS — Bit extraction 97
6.103 IBSET — Set bit 97
6.104 ICHAR — Character-to-integer conversion function 97
6.105 IDATE — Get current local time subroutine (day/month/year) 98
6.106 IEOR — Bitwise logical exclusive or 99
6.107 IERRNO — Get the last system error number 99
6.108 INDEX — Position of a substring within a string 100
6.109 INT — Convert to integer type 100
6.110 INT2 — Convert to 16-bit integer type 101
6.111 INT8 — Convert to 64-bit integer type 101
6.112 IOR — Bitwise logical or 102

6.113 IRAND — Integer pseudo-random number 102
6.114 IS_IOSTAT_END — Test for end-of-file value 103
6.115 IS_IOSTAT_EOR — Test for end-of-record value 103
6.116 ISATTY — Whether a unit is a terminal device. 104
6.117 ISHFT — Shift bits 104
6.118 ISHFTC — Shift bits circularly 105
6.119 ISNAN — Test for a NaN 105
6.120 ITIME — Get current local time subroutine (hour/minutes/seconds) 106
6.121 KILL — Send a signal to a process 106
6.122 KIND — Kind of an entity 106
6.123 LBOUND — Lower dimension bounds of an array 107
6.124 LEN — Length of a character entity 107
6.125 LEN_TRIM — Length of a character entity without trailing blank characters 108
6.126 LGAMMA — Logarithm of the Gamma function 108
6.127 LGE — Lexical greater than or equal 109
6.128 LGT — Lexical greater than 110
6.129 LINK — Create a hard link 110
6.130 LLE — Lexical less than or equal 111
6.131 LLT — Lexical less than 111
6.132 LNBLNK — Index of the last non-blank character in a string . . 112
6.133 LOC — Returns the address of a variable 112
6.134 LOG — Logarithm function 113
6.135 LOG10 — Base 10 logarithm function 113
6.136 LOGICAL — Convert to logical type 114
6.137 LONG — Convert to integer type 114
6.138 LSHIFT — Left shift bits 115
6.139 LSTAT — Get file status 115
6.140 LTIME — Convert time to local time info 116
6.141 MALLOC — Allocate dynamic memory 116
6.142 MATMUL — matrix multiplication 117
6.143 MAX — Maximum value of an argument list 118
6.144 MAXEXPONENT — Maximum exponent of a real kind 118
6.145 MAXLOC — Location of the maximum value within an array . . 119
6.146 MAXVAL — Maximum value of an array 119
6.147 MCLOCK — Time function 120
6.148 MCLOCK8 — Time function (64-bit) 120
6.149 MERGE — Merge variables 121
6.150 MIN — Minimum value of an argument list 121
6.151 MINEXPONENT — Minimum exponent of a real kind 122
6.152 MINLOC — Location of the minimum value within an array . . 122
6.153 MINVAL — Minimum value of an array 123
6.154 MOD — Remainder function 124
6.155 MODULO — Modulo function 124
6.156 MOVE_ALLOC — Move allocation from one object to another 125
6.157 MVBITS — Move bits from one integer to another 126

6.158 NEAREST — Nearest representable number 126
6.159 NEW_LINE — New line character 127
6.160 NINT — Nearest whole number 127
6.161 NOT — Logical negation 128
6.162 NULL — Function that returns an disassociated pointer 128
6.163 OR — Bitwise logical OR 129
6.164 PACK — Pack an array into an array of rank one 129
6.165 PERROR — Print system error message 130
6.166 PRECISION — Decimal precision of a real kind 130
6.167 PRESENT — Determine whether an optional dummy argument is specified 131
6.168 PRODUCT — Product of array elements 131
6.169 RADIX — Base of a model number 132
6.170 RAN — Real pseudo-random number 133
6.171 RAND — Real pseudo-random number 133
6.172 RANDOM_NUMBER — Pseudo-random number 133
6.173 RANDOM_SEED — Initialize a pseudo-random number sequence 134
6.174 RANGE — Decimal exponent range of a real kind 135
6.175 REAL — Convert to real type 135
6.176 RENAME — Rename a file 136
6.177 REPEAT — Repeated string concatenation 137
6.178 RESHAPE — Function to reshape an array 137
6.179 RRSPACING — Reciprocal of the relative spacing 138
6.180 RSHIFT — Right shift bits 138
6.181 SCALE — Scale a real value 138
6.182 SCAN — Scan a string for the presence of a set of characters 139
6.183 SECNDS — Time function 140
6.184 SECOND — CPU time function 140
6.185 SELECTED_INT_KIND — Choose integer kind 141
6.186 SELECTED_REAL_KIND — Choose real kind 141
6.187 SET_EXPONENT — Set the exponent of the model 142
6.188 SHAPE — Determine the shape of an array 142
6.189 SIGN — Sign copying function 143
6.190 SIGNAL — Signal handling subroutine (or function) 144
6.191 SIN — Sine function 144
6.192 SINH — Hyperbolic sine function 145
6.193 SIZE — Determine the size of an array 145
6.194 SIZEOF — Size in bytes of an expression 146
6.195 SLEEP — Sleep for the specified number of seconds 147
6.196 SNGL — Convert double precision real to default real 147
6.197 SPACING — Smallest distance between two numbers of a given type 147
6.198 SPREAD — Add a dimension to an array 148
6.199 SQRT — Square-root function 148
6.200 SRAND — Reinitialize the random number generator 149
6.201 STAT — Get file status 149

6.202 SUM — Sum of array elements 151
6.203 SYMLNK — Create a symbolic link 151
6.204 SYSTEM — Execute a shell command 152
6.205 SYSTEM_CLOCK — Time function 152
6.206 TAN — Tangent function 153
6.207 TANH — Hyperbolic tangent function 153
6.208 TIME — Time function 154
6.209 TIME8 — Time function (64-bit) 155
6.210 TINY — Smallest positive number of a real kind 155
6.211 TRANSFER — Transfer bit patterns 155
6.212 TRANSPOSE — Transpose an array of rank two 156
6.213 TRIM — Remove trailing blank characters of a string 157
6.214 TTYNAM — Get the name of a terminal device 157
6.215 UBOUND — Upper dimension bounds of an array 158
6.216 UMASK — Set the file creation mask 158
6.217 UNLINK — Remove a file from the file system 158
6.218 UNPACK — Unpack an array of rank one into an array 159
6.219 VERIFY — Scan a string for the absence of a set of characters 159
6.220 XOR — Bitwise logical exclusive OR 160

7 Intrinsic Modules 163

7.1 ISO_FORTRAN_ENV 163
7.2 ISO_C_BINDING 163
7.3 OpenMP Modules OMP_LIB and OMP_LIB_KINDS 165

Contributing 167

Contributors to GNU Fortran 167
Projects 168
Proposed Extensions 168
Compiler extensions: 168
Environment Options 169

GNU General Public License 171

GNU Free Documentation License 183

ADDENDUM: How to use this License for your documents 189

Funding Free Software 191

Option Index 193

Keyword Index 195

1 Introduction

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

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

1.1 About GNU Fortran

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

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

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

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

The GNU Fortran compiler consists of several components:

- A version of the `gcc` command (which also might be installed as the system's `cc` command) that also understands and accepts Fortran source code. The `gcc` command is the *driver* program for all the languages in the GNU Compiler Collection (GCC); With gcc, you can compile the source code of any language for which a front end is available in GCC.

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

1.2 GNU Fortran and GCC

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

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

Source files with '.f', '.for', '.fpp', '.ftn', '.F', '.FOR', '.FPP', and '.FTN' extensions are treated as fixed form. Source files with '.f90', '.f95', '.f03', '.F90', '.F95', and '.F03' extensions are treated as free form. The capitalized versions of either form are run through preprocessing. Source files with the lower case '.fpp' extension are also run through preprocessing.

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

1.3 Preprocessing and conditional compilation

Many Fortran compilers including GNU Fortran allow passing the source code through a C preprocessor (CPP; sometimes also called the Fortran preprocessor, FPP) to allow for conditional compilation. In the case of GNU Fortran, this is the GNU C Preprocessor in the traditional mode. On systems with case-preserving file names, the preprocessor is automatically invoked if the file extension is `.F`, `.FOR`, `.FTN`, `.F90`, `.F95` or `.F03`; otherwise use for fixed-format code the option `-x f77-cpp-input` and for free-format code `-x f95-cpp-input`. Invocation of the preprocessor can be suppressed using `-x f77` or `-x f95`.

If the GNU Fortran invoked the preprocessor, `__GFORTRAN__` is defined and `__GNUC__`, `__GNUC_MINOR__` and `__GNUC_PATCHLEVEL__` can be used to determine the version of the compiler. See Section "Overview" in *The C Preprocessor* for details.

While CPP is the de-facto standard for preprocessing Fortran code, Part 3 of the Fortran 95 standard (ISO/IEC 1539-3:1998) defines Conditional Compilation, which is not widely used and not directly supported by the GNU Fortran compiler. You can use the program coco to preprocess such files (`http://users.erols.com/dnagle/coco.html`).

1.4 GNU Fortran and G77

The GNU Fortran compiler is the successor to `g77`, the Fortran 77 front end included in GCC prior to version 4. It is an entirely new program that has been designed to provide Fortran 95 support and extensibility for future Fortran language standards, as well as providing backwards compatibility for Fortran 77 and nearly all of the GNU language extensions supported by `g77`.

1.5 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).

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

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

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

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

1.6 Standards

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

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

Part I: Invoking GNU Fortran

2 GNU Fortran Command Options

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

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

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

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

2.1 Option summary

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

Fortran Language Options

See Section 2.2 [Options controlling Fortran dialect], page 8.

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

Error and Warning Options

See Section 2.3 [Options to request or suppress errors and warnings], page 10.

```
-fmax-errors=n
-fsyntax-only -pedantic -pedantic-errors
-Wall -Waliasing -Wampersand -Wcharacter-truncation -Wconversion
-Wimplicit-interface -Wline-truncation -Wnonstd-intrinsics -Wsurprising
-Wno-tabs -Wunderflow -Wunused-parameter
```

Debugging Options

See Section 2.4 [Options for debugging your program or GNU Fortran], page 12.

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

Directory Options

See Section 2.5 [Options for directory search], page 13.

```
-Idir -Jdir -Mdir -fintrinsic-modules-path dir
```

Link Options

See Section 2.6 [Options for influencing the linking step], page 13.

```
-static-libgfortran
```

Runtime Options

See Section 2.7 [Options for influencing runtime behavior], page 14.

```
-fconvert=conversion -frecord-marker=length
-fmax-subrecord-length=length -fsign-zero
```

Code Generation Options

See Section 2.8 [Options for code generation conventions], page 14.

```
-fno-automatic -ff2c -fno-underscoring -fsecond-underscore
-fbounds-check -fmax-stack-var-size=n
-fpack-derived -frepack-arrays -fshort-enums -fexternal-blas
-fblas-matmul-limit=n -frecursive -finit-local-zero
-finit-integer=n -finit-real=<zero|inf|-inf|nan>
-finit-logical=<true|false> -finit-character=n
```

2.2 Options controlling Fortran dialect

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

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

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

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

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

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

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

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

`-fbackslash`
: Change the interpretation of backslashes in string literals from a single backslash character to "C-style" escape characters. The following combinations are expanded \a, \b, \f, \n, \r, \t, \v, \\, and \0 to the ASCII characters alert, backspace, form feed, newline, carriage return, horizontal tab, vertical

tab, backslash, and NUL, respectively. All other combinations of a character preceded by \ are unexpanded.

`-fmodule-private`
: Set the default accessibility of module entities to `PRIVATE`. Use-associated entities will not be accessible unless they are explicitly declared as `PUBLIC`.

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

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

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

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

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

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

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

`-fno-range-check`
: Disable range checking on results of simplification of constant expressions during compilation. For example, GNU Fortran will give an error at compile time when simplifying `a = 1. / 0`. With this option, no error will be given and `a` will be assigned the value `+Infinity`. If an expression evaluates to a value outside of the relevant range of [`-HUGE()`:`HUGE()`], then the expression will be replaced by `-Inf` or `+Inf` as appropriate. Similarly, `DATA i/Z'FFFFFFFF'/` will result in an integer overflow on most systems, but with '`-fno-range-check`' the value will "wrap around" and `i` will be initialized to -1 instead.

`-std=`*std* Specify the standard to which the program is expected to conform, which may be one of '`f95`', '`f2003`', '`gnu`', or '`legacy`'. The default value for *std* is '`gnu`', which specifies a superset of the Fortran 95 standard that includes all of the extensions supported by GNU Fortran, although warnings will be given for obsolete extensions not recommended for use in new code. The '`legacy`' value is equivalent but without the warnings for obsolete extensions, and may be useful for old non-standard programs. The '`f95`' and '`f2003`' values specify strict conformance to the Fortran 95 and Fortran 2003 standards, respectively; errors are given for all extensions beyond the relevant language standard, and warnings are given for the Fortran 77 features that are permitted but obsolescent in later standards.

2.3 Options to request or suppress errors and warnings

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

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

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

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

`-fmax-errors=`*n*

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

`-fsyntax-only`

Check the code for syntax errors, but don't actually compile it. This will generate module files for each module present in the code, but no other output file.

`-pedantic`

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

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

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

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

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

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

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

The following example will trigger the warning.

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

call bar(a,a)
```

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

-Wcharacter-truncation
: Warn when a character assignment will truncate the assigned string.

-Wconversion
: Warn about implicit conversions between different types.

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

-Wnonstd-intrinsics
: 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.
- A TRANSFER specifies a source that is shorter than the destination.

`-Wtabs` By default, tabs are accepted as whitespace, but tabs are not members of the Fortran Character Set. For continuation lines, a tab followed by a digit between 1 and 9 is supported. '`-Wno-tabs`' will cause a warning to be issued if a tab is encountered. Note, '`-Wno-tabs`' is active for '`-pedantic`', '`-std=f95`', '`-std=f2003`', and '`-Wall`'.

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

`-Wunused-parameter`
Contrary to `gcc`'s meaning of '`-Wunused-parameter`', `gfortran`'s implementation of this option does not warn about unused dummy arguments, but about unused `PARAMETER` values. '`-Wunused-parameter`' is not included in '`-Wall`' but is implied by '`-Wall -Wextra`'.

`-Werror` Turns all warnings into errors.

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

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

2.4 Options for debugging your program or GNU Fortran

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

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

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

Some of the routines in the Fortran runtime library, like '`CPU_TIME`', are likely to to trigger floating point exceptions when `ffpe-trap=precision` is used. For this reason, the use of `ffpe-trap=precision` is not recommended.

`-fbacktrace`
: Specify that, when a runtime error is encountered or a deadly signal is emitted (segmentation fault, illegal instruction, bus error or floating-point exception), the Fortran runtime library should output a backtrace of the error. This option only has influence for compilation of the Fortran main program.

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

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

2.5 Options for directory search

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

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

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

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

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

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

`-M`*dir*

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

  The default is the current directory.

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

`-fintrinsic-modules-path` *dir*
: This option specifies the location of pre-compiled intrinsic modules, if they are not in the default location expected by the compiler.

2.6 Influencing the linking step

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler is not doing a link step.

`-static-libgfortran`
: On systems that provide '`libgfortran`' as a shared and a static library, this option forces the use of the static version. If no shared version of '`libgfortran`' was built when the compiler was configured, this option has no effect.

2.7 Influencing runtime behavior

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

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

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

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

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

`-fsign-zero`
: When writing zero values, show the negative sign if the sign bit is set. `fno-sign-zero` does not print the negative sign of zero values for compatibility with F77. Default behavior is to show the negative sign.

2.8 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 (except those marked as RECURSIVE) as if the `SAVE` statement were 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`' or '`-save`'.) The default, which is '`-fautomatic`', uses the stack for local variables smaller than the value given by '`-fmax-stack-var-size`'. Use the option '`-frecursive`' to use no static memory.

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

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

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

Caution: It is not a good idea to mix Fortran code compiled with '`-ff2c`' with code compiled with the default '`-fno-f2c`' calling conventions as, calling `COMPLEX` or default `REAL` functions between program parts which were compiled with different calling conventions will break at execution time.

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

`-fno-underscoring`

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

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

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

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

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

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

is implemented as something akin to:

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

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

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

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

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

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

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

`-fsecond-underscore`

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

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

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

`-fbounds-check`

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

Some checks require that '`-fbounds-check`' is set for the compilation of the main program.

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

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

This option specifies the size in bytes of the largest array that will be put on the stack; if the size is exceeded static memory is used (except in procedures marked as RECURSIVE). Use the option '`-frecursive`' to allow for recursive procedures which do not have a RECURSIVE attribute or for parallel programs. Use '`-fno-automatic`' to never use the stack.

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

The default value for *n* is 32768.

`-fpack-derived`

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

`-frepack-arrays`

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

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

`-fshort-enums`

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

`-fexternal-blas`

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

`-fblas-matmul-limit=`*n*

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

The default value for *n* is 30.

`-frecursive`

Allow indirect recursion by forcing all local arrays to be allocated on the stack. This flag cannot be used together with '`-fmax-stack-var-size=`' or '`-fno-automatic`'.

`-finit-local-zero`
`-finit-integer=`*n*
`-finit-real=`<*zero*|*inf*|*-inf*|*nan*>
`-finit-logical=`<*true*|*false*>
`-finit-character=`*n*

The '`-finit-local-zero`' option instructs the compiler to initialize local `INTEGER`, `REAL`, and `COMPLEX` variables to zero, `LOGICAL` variables to false, and `CHARACTER` variables to a string of null bytes. Finer-grained initialization options are provided by the '`-finit-integer=`*n*', '`-finit-real=`<*zero*|*inf*|*-inf*|*nan*>' (which also initializes the real and imaginary parts of local `COMPLEX` variables), ‘-finit-logical=<*true*|*false*>’, and ‘-finit-character=*n*’ (where *n* is an ASCII character value) options. These options do not initialize components of derived type variables, nor do they initialize variables that appear in an EQUIVALENCE statement. (This limitation may be removed in future releases).

Note that the ‘-finit-real=nan’ option initializes REAL and COMPLEX variables with a quiet NaN.

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

2.9 Environment variables affecting gfortran

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

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

See Chapter 3 [Runtime], page 19, for environment variables that affect the run-time behavior of programs compiled with GNU Fortran.

3 Runtime: Influencing runtime behavior with environment variables

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

Malformed environment variables are silently ignored.

3.1 `GFORTRAN_STDIN_UNIT`—Unit number for standard input

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

3.2 `GFORTRAN_STDOUT_UNIT`—Unit number for standard output

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

3.3 `GFORTRAN_STDERR_UNIT`—Unit number for standard error

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

3.4 `GFORTRAN_USE_STDERR`—Send library output to standard error

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

3.5 `GFORTRAN_TMPDIR`—Directory for scratch files

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

3.6 `GFORTRAN_UNBUFFERED_ALL`—Don't buffer I/O on all units

This environment variable controls whether all I/O is unbuffered. If the first letter is '`y`', '`Y`' or '`1`', all I/O is unbuffered. This will slow down small sequential reads and writes. If the first letter is '`n`', '`N`' or '`0`', I/O is buffered. This is the default.

3.7 `GFORTRAN_UNBUFFERED_PRECONNECTED`—Don't buffer I/O on preconnected units

The environment variable named `GFORTRAN_UNBUFFERED_PRECONNECTED` controls whether I/O on a preconnected unit (i.e STDOUT or STDERR) is unbuffered. If the first letter is '`y`', '`Y`' or '`1`', I/O is unbuffered. This will slow down small sequential reads and writes. If the first letter is '`n`', '`N`' or '`0`', I/O is buffered. This is the default.

3.8 `GFORTRAN_SHOW_LOCUS`—Show location for runtime errors

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

3.9 GFORTRAN_OPTIONAL_PLUS—Print leading + where permitted

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

3.10 GFORTRAN_DEFAULT_RECL—Default record length for new files

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

3.11 GFORTRAN_LIST_SEPARATOR—Separator for list output

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

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

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

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

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

NATIVE Use the native format. This is the default.

SWAP Swap between little- and big-endian.

LITTLE_ENDIAN Use the little-endian format for unformatted files.

BIG_ENDIAN Use the big-endian format for unformatted files.

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

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

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

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

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

Example for `sh`:

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

Example code for `csh`:

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

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

See Section 5.1.14 [CONVERT specifier], page 33, for an alternative way to specify the data representation for unformatted files. See Section 2.7 [Runtime Options], page 14, for setting a default data representation for the whole program. The `CONVERT` specifier overrides the '`-fconvert`' compile options.

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

3.13 `GFORTRAN_ERROR_DUMPCORE`—Dump core on run-time errors

If the `GFORTRAN_ERROR_DUMPCORE` variable is set to '`y`', '`Y`' or '`1`' (only the first letter is relevant) then library run-time errors cause core dumps. To disable the core dumps, set the variable to '`n`', '`N`', '`0`'. Default is not to core dump unless the '`-fdump-core`' compile option was used.

3.14 `GFORTRAN_ERROR_BACKTRACE`—Show backtrace on run-time errors

If the `GFORTRAN_ERROR_BACKTRACE` variable is set to '`y`', '`Y`' or '`1`' (only the first letter is relevant) then a backtrace is printed when a run-time error occurs. To disable the backtracing, set the variable to '`n`', '`N`', '`0`'. Default is not to print a backtrace unless the '`-fbacktrace`' compile option was used.

Part II: Language Reference

4 Fortran 2003 Status

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

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

5 Extensions

The two sections below detail the extensions to standard Fortran that are implemented in GNU Fortran, as well as some of the popular or historically important extensions that are not (or not yet) implemented. For the latter case, we explain the alternatives available to GNU Fortran users, including replacement by standard-conforming code or GNU extensions.

5.1 Extensions implemented in GNU Fortran

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

5.1.1 Old-style kind specifications

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

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

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

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

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

5.1.2 Old-style variable initialization

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

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

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

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

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

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

5.1.3 Extensions to namelist

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

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

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

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

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

```
 ?

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

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

```
=?

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

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

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

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

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

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

5.1.4 `X` format descriptor without count field

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

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

5.1.5 Commas in `FORMAT` specifications

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

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

5.1.6 Missing period in `FORMAT` specifications

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

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

5.1.7 I/O item lists

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

5.1.8 BOZ literal constants

Besides decimal constants, Fortran also supports binary (`b`), octal (`o`) and hexadecimal (`z`) integer constants. The syntax is: '`prefix quote digits quote`', were the prefix is either `b`, `o` or `z`, quote is either `'` or `"` and the digits are for binary `0` or `1`, for octal between `0` and `7`, and for hexadecimal between `0` and `F`. (Example: `b'01011101'`.)

Up to Fortran 95, BOZ literals were only allowed to initialize integer variables in DATA statements. Since Fortran 2003 BOZ literals are also allowed as argument of `REAL`, `DBLE`, `INT` and `CMPLX`; the result is the same as if the integer BOZ literal had been converted by `TRANSFER` to, respectively, `real`, `double precision`, `integer` or `complex`. As GNU Fortran extension the intrinsic procedures `FLOAT`, `DFLOAT`, `COMPLEX` and `DCMPLX` are treated alike.

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

Furthermore, GNU Fortran allows using BOZ literal constants outside DATA statements and the four intrinsic functions allowed by Fortran 2003. In DATA statements, in direct assignments, where the right-hand side only contains a BOZ literal constant, and for old-style initializers of the form `integer i /o'0173'/`, the constant is transferred as if `TRANSFER` had been used; for `COMPLEX` numbers, only the real part is initialized unless `CMPLX` is used. In all other cases, the BOZ literal constant is converted to an `INTEGER` value with the largest decimal representation. This value is then converted numerically to the type and kind of the variable in question. (For instance `real :: r = b'0000001' + 1` initializes `r` with `2.0`.) As different compilers implement the extension differently, one should be careful when doing bitwise initialization of non-integer variables.

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

5.1.9 Real array indices

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

5.1.10 Unary operators

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

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

5.1.11 Implicitly convert LOGICAL and INTEGER values

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

```
LOGICAL :: l
l = 1
INTEGER :: i
i = .TRUE.
```

However, there is no implicit conversion of `INTEGER` values in `if`-statements, nor of `LOGICAL` or `INTEGER` values in I/O operations.

5.1.12 Hollerith constants support

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

Examples of valid uses of Hollerith constants:

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

Invalid Hollerith constants examples:

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

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

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

5.1.13 Cray pointers

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

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

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

or,

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

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

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

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

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

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

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

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

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

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

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

The pointer can also be set by a call to the MALLOC intrinsic (see Section 6.141 [MALLOC], page 116).

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

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

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

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

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

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

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

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

5.1.14 CONVERT specifier

GNU Fortran allows the conversion of unformatted data between little- and big-endian representation to facilitate moving of data between different systems. The conversion can be indicated with the CONVERT specifier on the OPEN statement. See Section 3.12 [GFORTRAN_CONVERT_UNIT], page 20, for an alternative way of specifying the data format via an environment variable.

Valid values for CONVERT are:

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

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

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

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

Using the option could look like this:

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

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

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

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

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

5.1.15 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

GNU Fortran strives to be compatible to the OpenMP Application Program Interface v2.5.

To enable the processing of the OpenMP directive !$omp in free-form source code; the c$omp, *$omp and !$omp directives in fixed form; the !$ conditional compilation sentinels in free form; and the c$, *$ and !$ sentinels in fixed form, gfortran needs to be invoked with the '-fopenmp'. This also arranges for automatic linking of the GNU OpenMP runtime library Section "libgomp" in *GNU OpenMP runtime library*.

The OpenMP Fortran runtime library routines are provided both in a form of a Fortran 90 module named omp_lib and in a form of a Fortran include file named 'omp_lib.h'.

An example of a parallelized loop taken from Appendix A.1 of the OpenMP Application Program Interface v2.5:

```
SUBROUTINE A1(N, A, B)
  INTEGER I, N
  REAL B(N), A(N)
!$OMP PARALLEL DO !I is private by default
  DO I=2,N
    B(I) = (A(I) + A(I-1)) / 2.0
  ENDDO
!$OMP END PARALLEL DO
END SUBROUTINE A1
```

Please note:

- '-fopenmp' implies '-frecursive', i.e. all local arrays will be allocated on the stack. When porting existing code to OpenMP, this may lead to surprising results, especially to segmentation faults if the stacksize is limited.
- On glibc-based systems, OpenMP enabled applications can not be statically linked due to limitations of the underlying pthreads-implementation. It might be possible to get a working solution if `-Wl,--whole-archive -lpthread -Wl,--no-whole-archive` is added to the command line. However, this is not supported by `gcc` and thus not recommended.

5.1.16 Argument list functions %VAL, %REF and %LOC

GNU Fortran supports argument list functions `%VAL`, `%REF` and `%LOC` statements, for backward compatibility with g77. It is recommended that these should be used only for code that is accessing facilities outside of GNU Fortran, such as operating system or windowing facilities. It is best to constrain such uses to isolated portions of a program–portions that deal specifically and exclusively with low-level, system-dependent facilities. Such portions might well provide a portable interface for use by the program as a whole, but are themselves not portable, and should be thoroughly tested each time they are rebuilt using a new compiler or version of a compiler.

`%VAL` passes a scalar argument by value, `%REF` passes it by reference and `%LOC` passes its memory location. Since gfortran already passes scalar arguments by reference, `%REF` is in effect a do-nothing. `%LOC` has the same effect as a fortran pointer.

An example of passing an argument by value to a C subroutine foo.:

```
C
C prototype      void foo_ (float x);
C
      external foo
      real*4 x
      x = 3.14159
      call foo (%VAL (x))
      end
```

For details refer to the g77 manual http://gcc.gnu.org/onlinedocs/gcc-3.4.6/g77/index.html#Top.

Also, the gfortran testsuite c_by_val.f and its partner c_by_val.c are worth a look.

5.2 Extensions not implemented in GNU Fortran

The long history of the Fortran language, its wide use and broad userbase, the large number of different compiler vendors and the lack of some features crucial to users in the first standards have lead to the existence of an important number of extensions to the language. While some of the most useful or popular extensions are supported by the GNU Fortran compiler, not all existing extensions are supported. This section aims at listing these extensions and offering advice on how best make code that uses them running with the GNU Fortran compiler.

5.2.1 STRUCTURE and RECORD

Structures are user-defined aggregate data types; this functionality was standardized in Fortran 90 with an different syntax, under the name of "derived types". Here is an example of code using the non portable structure syntax:

```
! Declaring a structure named ‘‘item’’ and containing three fields:
! an integer ID, an description string and a floating-point price.
STRUCTURE /item/
  INTEGER id
  CHARACTER(LEN=200) description
  REAL price
END STRUCTURE

! Define two variables, an single record of type ‘‘item’’
! named ‘‘pear’’, and an array of items named ‘‘store_catalog’’
RECORD /item/ pear, store_catalog(100)

! We can directly access the fields of both variables
pear.id = 92316
pear.description = "juicy D’Anjou pear"
pear.price = 0.15
store_catalog(7).id = 7831
store_catalog(7).description = "milk bottle"
store_catalog(7).price = 1.2

! We can also manipulates the whole structure
store_catalog(12) = pear
print *, store_catalog(12)
```

This code can easily be rewritten in the Fortran 90 syntax as following:

```
! ‘‘STRUCTURE /name/ ... END STRUCTURE’’ becomes
! ‘‘TYPE name ... END TYPE’’
TYPE item
  INTEGER id
  CHARACTER(LEN=200) description
  REAL price
END TYPE

! ‘‘RECORD /name/ variable’’ becomes ‘‘TYPE(name) variable’’
TYPE(item) pear, store_catalog(100)

! Instead of using a dot (.) to access fields of a record, the
! standard syntax uses a percent sign (%)
pear%id = 92316
```

```
pear%description = "juicy D'Anjou pear"
pear%price = 0.15
store_catalog(7)%id = 7831
store_catalog(7)%description = "milk bottle"
store_catalog(7)%price = 1.2

! Assignments of a whole variable don't change
store_catalog(12) = pear
print *, store_catalog(12)
```

5.2.2 ENCODE and DECODE statements

GNU Fortran doesn't support the ENCODE and DECODE statements. These statements are best replaced by READ and WRITE statements involving internal files (CHARACTER variables and arrays), which have been part of the Fortran standard since Fortran 77. For example, replace a code fragment like

```
      INTEGER*1 LINE(80)
      REAL A, B, C
c     ... Code that sets LINE
      DECODE (80, 9000, LINE) A, B, C
 9000 FORMAT (1X, 3(F10.5))
```

with the following:

```
      CHARACTER(LEN=80) LINE
      REAL A, B, C
c     ... Code that sets LINE
      READ (UNIT=LINE, FMT=9000) A, B, C
 9000 FORMAT (1X, 3(F10.5))
```

Similarly, replace a code fragment like

```
      INTEGER*1 LINE(80)
      REAL A, B, C
c     ... Code that sets A, B and C
      ENCODE (80, 9000, LINE) A, B, C
 9000 FORMAT (1X, 'OUTPUT IS ', 3(F10.5))
```

with the following:

```
      INTEGER*1 LINE(80)
      REAL A, B, C
c     ... Code that sets A, B and C
      WRITE (UNIT=LINE, FMT=9000) A, B, C
 9000 FORMAT (1X, 'OUTPUT IS ', 3(F10.5))
```

6 Intrinsic Procedures

6.1 Introduction to intrinsic procedures

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

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

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

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

6.2 `ABORT` — Abort the program

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

Standard: GNU extension

Class: Subroutine

Syntax: `CALL ABORT`

Return value:
: Does not return.

Example:

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

See also: Section 6.66 [EXIT], page 77, Section 6.121 [KILL], page 106

6.3 ABS — Absolute value

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

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

Class: Elemental function

Syntax: RESULT = ABS(X)

Arguments:

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

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

Example:

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

Specific names:

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

6.4 ACCESS — Checks file access modes

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

Standard: GNU extension

Class: Inquiry function

Syntax: RESULT = ACCESS(NAME, MODE)

Arguments:

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

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

Example:

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

Specific names:
See also:

6.5 `ACHAR` — Character in ASCII collating sequence

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

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = ACHAR(I)`

Arguments:

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

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

Example:

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

Note: See Section 6.104 [ICHAR], page 97 for a discussion of converting between numerical values and formatted string representations.

See also: Section 6.38 [CHAR], page 59, Section 6.98 [IACHAR], page 94, Section 6.104 [ICHAR], page 97

6.6 ACOS — Arccosine function

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

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = ACOS(X)`

Arguments:

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

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

Example:

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

Specific names:

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

See also: Inverse function: Section 6.45 [COS], page 63

6.7 ACOSH — Hyperbolic arccosine function

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

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = ACOSH(X)`

Arguments:

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

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

Example:

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

Specific names:

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

See also: Inverse function: Section 6.46 [COSH], page 64

6.8 ADJUSTL — Left adjust a string

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

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = ADJUSTL(STR)`

Arguments:

STR	The type shall be `CHARACTER`.

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

Example:

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

See also: Section 6.9 [ADJUSTR], page 41, Section 6.213 [TRIM], page 157

6.9 ADJUSTR — Right adjust a string

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

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = ADJUSTR(STR)`

Arguments:

STR	The type shall be `CHARACTER`.

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

Example:

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

See also: Section 6.8 [ADJUSTL], page 41, Section 6.213 [TRIM], page 157

6.10 AIMAG — Imaginary part of complex number

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

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

Class: Elemental function

Syntax: RESULT = AIMAG(Z)

Arguments:

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

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

Example:

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

Specific names:

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

6.11 AINT — Truncate to a whole number

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

Standard: F77 and later

Class: Elemental function

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

Arguments:

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

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

Example:

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

Specific names:

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

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

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

Standard: GNU extension

Class: Subroutine

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

Arguments:

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

Example:

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

```
  call sleep(10)
end program test_alarm
```

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

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

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

Standard: F95 and later

Class: Transformational function

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

Arguments:

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

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

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

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

Example:

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

6.14 `ALLOCATED` — Status of an allocatable entity

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

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = ALLOCATED(X)`

Arguments:

X	The argument shall be an `ALLOCATABLE` array.

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

Example:

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

6.15 `AND` — Bitwise logical AND

Description:
Bitwise logical `AND`.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. For integer arguments, programmers should consider the use of the Section 6.99 [IAND], page 95 intrinsic defined by the Fortran standard.

Standard: GNU extension

Class: Function

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

Arguments:

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

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

Example:

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

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

See also: F95 elemental function: Section 6.99 [IAND], page 95

6.16 ANINT — Nearest whole number

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

Standard: F77 and later

Class: Elemental function

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

Arguments:

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

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

Example:

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

Specific names:

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

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

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

Standard: F95 and later

Class: Transformational function

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

Arguments:

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

Return value:

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

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

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

Example:

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

6.18 ASIN — Arcsine function

Description:

ASIN(X) computes the arcsine of its *X* (inverse of SIN(X)).

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = ASIN(X)

Arguments:

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

Return value:

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

Example:

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

Specific names:

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

See also: Inverse function: Section 6.191 [SIN], page 144

6.19 `ASINH` — Hyperbolic arcsine function

Description:
`ASINH(X)` computes the hyperbolic arcsine of *X* (inverse of `SINH(X)`).

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = ASINH(X)`

Arguments:

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

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

Example:

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

Specific names:

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

See also: Inverse function: Section 6.192 [SINH], page 145

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

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

Standard: F95 and later

Class: Inquiry function

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

Arguments:

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

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

Return value:

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

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

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

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

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

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

Example:

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

See also: Section 6.162 [NULL], page 128

6.21 `ATAN` — Arctangent function

Description:

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

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = ATAN(X)`

Arguments:

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

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

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

Specific names:

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

See also: Inverse function: Section 6.206 [TAN], page 153

6.22 `ATAN2` — Arctangent function

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

Standard: F77 and later

Class: Elemental function

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

Arguments:

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

Return value:
The return value has the same type and kind type parameter as *Y*. It is the principal value of the complex number $X + iY$. If *X* is nonzero, then it lies in the range $-\pi \leq \text{atan}(x) \leq \pi$. 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	Standard
`DATAN2(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

6.23 `ATANH` — Hyperbolic arctangent function

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

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = ATANH(X)`

Arguments:

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

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

Example:

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

Specific names:

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

See also: Inverse function: Section 6.207 [TANH], page 153

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

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

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = BESJ0(X)`

Arguments:

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

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

Example:

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

Specific names:

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

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

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

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = BESJ1(X)

Arguments:

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

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

Example:

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

Specific names:

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

6.26 BESJN — Bessel function of the first kind

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

If both arguments are arrays, their ranks and shapes shall conform.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = BESJN(N, X)

Arguments:

N	Shall be a scalar or an array of type INTEGER(*).
X	Shall be a scalar or an array of type REAL(*).

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

Example:

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

Specific names:

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

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

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

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = BESY0(X)

Arguments:

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

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

Example:

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

Specific names:

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

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

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

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = BESY1(X)

Arguments:

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

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

Example:

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

Specific names:

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

6.29 BESYN — Bessel function of the second kind

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

If both arguments are arrays, their ranks and shapes shall conform.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = BESYN(N, X)`

Arguments:

N	Shall be a scalar or an array of type `INTEGER(*)`.
X	Shall be a scalar or an array of type `REAL(*)`.

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

Example:

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

Specific names:

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

6.30 BIT_SIZE — Bit size inquiry function

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

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = BIT_SIZE(I)`

Arguments:

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

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

Example:

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

6.31 BTEST — Bit test function

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

Standard: F95 and later

Class: Elemental function

Syntax: RESULT = BTEST(I, POS)

Arguments:

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

Return value:
The return value is of type LOGICAL

Example:

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

6.32 C_ASSOCIATED — Status of a C pointer

Description:
C_ASSOICATED(c_prt1[, c_ptr2]) determines the status of the C pointer *c_ptr1* or if *c_ptr1* is associated with the target *c_ptr2*.

Standard: F2003 and later

Class: Inquiry function

Syntax: RESULT = C_ASSOICATED(c_prt1[, c_ptr2])

Arguments:

c_ptr1	Scalar of the type C_PTR or C_FUNPTR.
c_ptr2	(Optional) Scalar of the same type as *c_ptr1*.

Return value:
The return value is of type LOGICAL; it is .false. if either *c_ptr1* is a C NULL pointer or if *c_ptr1* and *c_ptr2* point to different addresses.

Example:

```
subroutine association_test(a,b)
  use iso_c_binding, only: c_associated, c_loc, c_ptr
  implicit none
  real, pointer :: a
  type(c_ptr) :: b
  if(c_associated(b, c_loc(a))) &
     stop 'b and a do not point to same target'
end subroutine association_test
```

See also: Section 6.36 [C_LOC], page 58, Section 6.33 [C_FUNLOC], page 56

6.33 C_FUNLOC — Obtain the C address of a procedure

Description:
`C_FUNLOC(x)` determines the C address of the argument.

Standard: F2003 and later

Class: Inquiry function

Syntax: `RESULT = C_FUNLOC(x)`

Arguments:

x	Interoperable function or pointer to such function.

Return value:
The return value is of type `C_FUNPTR` and contains the C address of the argument.

Example:

```
module x
  use iso_c_binding
  implicit none
contains
  subroutine sub(a) bind(c)
    real(c_float) :: a
    a = sqrt(a)+5.0
  end subroutine sub
end module x
program main
  use iso_c_binding
  use x
  implicit none
  interface
    subroutine my_routine(p) bind(c,name='myC_func')
      import :: c_funptr
      type(c_funptr), intent(in) :: p
    end subroutine
  end interface
  call my_routine(c_funloc(sub))
end program main
```

See also: Section 6.32 [C_ASSOCIATED], page 55, Section 6.36 [C_LOC], page 58, Section 6.35 [C_F_POINTER], page 57, Section 6.34 [C_F_PROCPOINTER], page 56

6.34 C_F_PROCPOINTER — Convert C into Fortran procedure pointer

Description:
`C_F_PROCPOINTER(cptr, fptr)` Assign the target of the C function pointer *cptr* to the Fortran procedure pointer *fptr*.

Note: Due to the currently lacking support of procedure pointers in GNU Fortran this function is not fully operable.

Standard: F2003 and later

Class: Subroutine

Syntax: `CALL C_F_PROCPOINTER(cptr, fptr)`

Arguments:

cptr	scalar of the type `C_FUNPTR`. It is `INTENT(IN)`.
fptr	procedure pointer interoperable with *cptr*. It is `INTENT(OUT)`.

Example:

```
program main
  use iso_c_binding
  implicit none
  abstract interface
    function func(a)
      import :: c_float
      real(c_float), intent(in) :: a
      real(c_float) :: func
    end function
  end interface
  interface
     function getIterFunc() bind(c,name="getIterFunc")
       import :: c_funptr
       type(c_funptr) :: getIterFunc
     end function
  end interface
  type(c_funptr) :: cfunptr
  procedure(func), pointer :: myFunc
  cfunptr = getIterFunc()
  call c_f_procpointer(cfunptr, myFunc)
end program main
```

See also: Section 6.36 [C_LOC], page 58, Section 6.35 [C_F_POINTER], page 57

6.35 `C_F_POINTER` — Convert C into Fortran pointer

Description:

`C_F_POINTER(cptr, fptr[, shape])` Assign the target the C pointer *cptr* to the Fortran pointer *fptr* and specify its shape.

Standard: F2003 and later

Class: Subroutine

Syntax: `CALL C_F_POINTER(cptr, fptr[, shape])`

Arguments:

cptr	scalar of the type `C_PTR`. It is `INTENT(IN)`.
fptr	pointer interoperable with *cptr*. It is `INTENT(OUT)`.
shape	(Optional) Rank-one array of type `INTEGER` with `INTENT(IN)`. It shall be present if and only if *fptr* is an array. The size must be equal to the rank of *fptr*.

Example:

```
program main
  use iso_c_binding
```

```
  implicit none
  interface
    subroutine my_routine(p) bind(c,name='myC_func')
      import :: c_ptr
      type(c_ptr), intent(out) :: p
    end subroutine
  end interface
  type(c_ptr) :: cptr
  real,pointer :: a(:)
  call my_routine(cptr)
  call c_f_pointer(cptr, a, [12])
end program main
```

See also: Section 6.36 [C_LOC], page 58, Section 6.34 [C_F_PROCPOINTER], page 56

6.36 `C_LOC` — Obtain the C address of an object

Description:
`C_LOC(x)` determines the C address of the argument.

Standard: F2003 and later

Class: Inquiry function

Syntax: `RESULT = C_LOC(x)`

Arguments:

x	Associated scalar pointer or interoperable scalar or allocated allocatable variable with `TARGET` attribute.

Return value:
The return value is of type `C_PTR` and contains the C address of the argument.

Example:

```
subroutine association_test(a,b)
  use iso_c_binding, only: c_associated, c_loc, c_ptr
  implicit none
  real, pointer :: a
  type(c_ptr) :: b
  if(c_associated(b, c_loc(a))) &
     stop 'b and a do not point to same target'
end subroutine association_test
```

See also: Section 6.32 [C_ASSOCIATED], page 55, Section 6.33 [C_FUNLOC], page 56, Section 6.35 [C_F_POINTER], page 57, Section 6.34 [C_F_PROCPOINTER], page 56

6.37 `CEILING` — Integer ceiling function

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

Standard: F95 and later

Class: Elemental function

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

Arguments:

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

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

Example:

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

See also: Section 6.73 [FLOOR], page 81, Section 6.160 [NINT], page 127

6.38 `CHAR` — Character conversion function

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

Standard: F77 and later

Class: Elemental function

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

Arguments:

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

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

Example:

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

Note: See Section 6.104 [ICHAR], page 97 for a discussion of converting between numerical values and formatted string representations.

See also: Section 6.5 [ACHAR], page 39, Section 6.98 [IACHAR], page 94, Section 6.104 [ICHAR], page 97

6.39 `CHDIR` — Change working directory

Description:
Change current working directory to a specified path.

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

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

Arguments:

NAME The type shall be `CHARACTER(*)` and shall specify a valid path within the file system.

STATUS (Optional) `INTEGER` status flag of the default kind. Returns 0 on success, and a system specific and nonzero error code otherwise.

Example:

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

See also: Section 6.88 [GETCWD], page 90

6.40 CHMOD — Change access permissions of files

Description:

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

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

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

Arguments:

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

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

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

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

Example: `CHMOD` as subroutine

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

`CHMOD` as function:

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

6.41 `CMPLX` — Complex conversion function

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

Standard: F77 and later

Class: Elemental function

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

Arguments:

X	The type may be `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)`.
Y	(Optional; only allowed if *X* is not `COMPLEX(*)`.) May be `INTEGER(*)` or `REAL(*)`.
KIND	(Optional) An `INTEGER(*)` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of `COMPLEX` type, with a kind equal to *KIND* if it is specified. If *KIND* is not specified, the result is of the default `COMPLEX` kind, regardless of the kinds of *X* and *Y*.

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
```

See also: Section 6.43 [COMPLEX], page 62

6.42 COMMAND_ARGUMENT_COUNT — Get number of command line arguments

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

Standard: F2003

Class: Inquiry function

Syntax: RESULT = COMMAND_ARGUMENT_COUNT()

Arguments:
None

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

Example:

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

See also: Section 6.86 [GET_COMMAND], page 89, Section 6.87 [GET_COMMAND_ARGUMENT], page 89

6.43 COMPLEX — Complex conversion function

Description:
COMPLEX(X, Y) returns a complex number where *X* is converted to the real component and *Y* is converted to the imaginary component.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = COMPLEX(X, Y)

Arguments:

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

Return value:
If *X* and *Y* are both of INTEGER type, then the return value is of default COMPLEX type.

If *X* and *Y* are of REAL type, or one is of REAL type and one is of INTEGER type, then the return value is of COMPLEX type with a kind equal to that of the REAL argument with the highest precision.

Example:

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

See also: Section 6.41 [CMPLX], page 61

6.44 CONJG — Complex conjugate function

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

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

Class: Elemental function

Syntax: `Z = CONJG(Z)`

Arguments:

Z	The type shall be `COMPLEX(*)`.

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

Example:

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

Specific names:

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

6.45 COS — Cosine function

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

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

Class: Elemental function

Syntax: `RESULT = COS(X)`

Arguments:

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

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

Example:

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

Specific names:

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

See also: Inverse function: Section 6.6 [ACOS], page 40

6.46 COSH — Hyperbolic cosine function

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

Standard: F77 and later

Class: Elemental function

Syntax: `X = COSH(X)`

Arguments:

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

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

Example:

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

Specific names:

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

See also: Inverse function: Section 6.7 [ACOSH], page 40

6.47 COUNT — Count function

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

Standard: F95 and later

Class: Transformational function

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

Arguments:

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

Return value:

The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind. The result has a rank equal to that of *MASK*.

Example:

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

6.48 `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.

If a time source is available, time will be reported with microsecond resolution. If no time source is available, *TIME* is set to `-1.0`.

Note that *TIME* may contain a, system dependent, arbitrary offset and may not start with `0.0`. For `CPU_TIME`, the absolute value is meaningless, only differences between subsequent calls to this subroutine, as shown in the example below, should be used.

Standard: F95 and later

Class: Subroutine

Syntax: `CALL CPU_TIME(TIME)`

Arguments:

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

Return value:

None

Example:

```
program test_cpu_time
    real :: start, finish
```

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

See also: Section 6.205 [SYSTEM_CLOCK], page 152, Section 6.51 [DATE_AND_TIME], page 67

6.49 CSHIFT — Circular shift elements of an array

Description:

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

Standard: F95 and later

Class: Transformational function

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

Arguments:

ARRAY	Shall be an array of any type.
SHIFT	The type shall be INTEGER.
DIM	The type shall be INTEGER.

Return value:

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

Example:

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

6.50 CTIME — Convert a time into a string

Description:

CTIME converts a system time value, such as returned by TIME8(), to a string of the form 'Sat Aug 19 18:13:14 1995'.

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

`CALL CTIME(TIME, RESULT)`.
`RESULT = CTIME(TIME)`, (not recommended).

Arguments:

TIME	The type shall be of type `INTEGER(KIND=8)`.
RESULT	The type shall be of type `CHARACTER`.

Return value:
The converted date and time as a string.

Example:

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

    ! Do something, main part of the program

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

See Also: Section 6.95 [GMTIME], page 93, Section 6.140 [LTIME], page 116, Section 6.208 [TIME], page 154, Section 6.209 [TIME8], page 155

6.51 DATE_AND_TIME — Date and time subroutine

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

VALUES is `INTENT(OUT)` and provides the following:

`VALUE(1)`:	The year
`VALUE(2)`:	The month
`VALUE(3)`:	The day of the month
`VALUE(4)`:	Time difference with UTC in minutes
`VALUE(5)`:	The hour of the day
`VALUE(6)`:	The minutes of the hour
`VALUE(7)`:	The seconds of the minute
`VALUE(8)`:	The milliseconds of the second

Standard: F95 and later

Class: Subroutine

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

Arguments:

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

Return value:
None

Example:

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

See also: Section 6.48 [CPU_TIME], page 65, Section 6.205 [SYSTEM_CLOCK], page 152

6.52 `DBLE` — Double conversion function

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

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = DBLE(X)`

Arguments:

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

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

Example:

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

See also: Section 6.54 [DFLOAT], page 69, Section 6.70 [FLOAT], page 79, Section 6.175 [REAL], page 135

6.53 DCMPLX — Double complex conversion function

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

Standard: GNU extension

Class: Elemental function

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

Arguments:

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

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

Example:

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

6.54 DFLOAT — Double conversion function

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

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = DFLOAT(X)

Arguments:

X	The type shall be INTEGER(*).

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

Example:

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

See also: Section 6.52 [DBLE], page 68, Section 6.70 [FLOAT], page 79, Section 6.175 [REAL], page 135

6.55 DIGITS — Significant digits function

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

Standard: F95 and later

Class: Inquiry function

Syntax: RESULT = DIGITS(X)

Arguments:

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

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

Example:

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

6.56 DIM — Positive difference

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

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = DIM(X, Y)

Arguments:

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

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

Example:

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

Specific names:

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

6.57 DOT_PRODUCT — Dot product function

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

Standard: F95 and later

Class: Transformational function

Syntax: RESULT = DOT_PRODUCT(X, Y)

Arguments:

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

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

Example:

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

6.58 DPROD — Double product function

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

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = DPROD(X, Y)

Arguments:

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

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

Example:

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

6.59 DREAL — Double real part function

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

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = DREAL(Z)`

Arguments:

Z	The type shall be `COMPLEX(8)`.

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

Example:

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

See also: Section 6.10 [AIMAG], page 42

6.60 DTIME — Execution time subroutine (or function)

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

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

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

Please note, that this implementation is thread safe if used within OpenMP directives, i. e. its state will be consistent while called from multiple threads.

However, if `DTIME` is called from multiple threads, the result is still the time since the last invocation. This may not give the intended results. If possible, use `CPU_TIME` instead.

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

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

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

Standard: GNU extension

Class: Subroutine, function

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 last invocation or since the start of program execution if not called before.

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
```

See also: Section 6.48 [CPU_TIME], page 65

6.61 `EOSHIFT` — End-off shift elements of an array

Description:

`EOSHIFT(ARRAY, SHIFT[,BOUNDARY, DIM])` performs an end-off shift on elements of *ARRAY* along the dimension of *DIM*. If *DIM* is omitted it is taken to be `1`. *DIM* is a scaler of type `INTEGER` in the range of $1/leqDIM/leqn$) where n is the rank of *ARRAY*. If the rank of *ARRAY* is one, then all elements of

ARRAY are shifted by *SHIFT* places. If rank is greater than one, then all complete rank one sections of *ARRAY* along the given dimension are shifted. Elements shifted out one end of each rank one section are dropped. If *BOUNDARY* is present then the corresponding value of from *BOUNDARY* is copied back in the other end. If *BOUNDARY* is not present then the following are copied in depending on the type of *ARRAY*.

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

Standard: F95 and later

Class: Transformational function

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

Arguments:

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

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

Example:

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

6.62 `EPSILON` — Epsilon function

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

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = EPSILON(X)`

Arguments:

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

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

Example:

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

6.63 ERF — Error function

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

Standard: GNU Extension

Class: Elemental function

Syntax: RESULT = ERF(X)

Arguments:

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

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

Example:

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

Specific names:

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

6.64 ERFC — Error function

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

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = ERFC(X)

Arguments:

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

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

Example:

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

Specific names:

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

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

Description:

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

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

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

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

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

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

Arguments:

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

Return value:

Elapsed time in seconds since the start of program execution.

Example:

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

```
    print *, tarray(1)
    print *, tarray(2)
end program test_etime
```

See also: Section 6.48 [CPU_TIME], page 65

6.66 EXIT — Exit the program with status.

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

Standard: GNU extension

Class: Subroutine

Syntax: CALL EXIT([STATUS])

Arguments:

STATUS	Shall be an INTEGER of the default kind.

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

Example:

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

See also: Section 6.2 [ABORT], page 37, Section 6.121 [KILL], page 106

6.67 EXP — Exponential function

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

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

Class: Elemental function

Syntax: RESULT = EXP(X)

Arguments:

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

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

Example:

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

Specific names:

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

6.68 `EXPONENT` — Exponent function

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

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = EXPONENT(X)`

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

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

Example:

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

6.69 `FDATE` — Get the current time as a string

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

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

DATE is an `INTENT(OUT) CHARACTER` variable.

Standard: GNU extension

Class: Subroutine, function

Syntax:

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

Arguments:
DATE The type shall be of type `CHARACTER`.

Return value:
The current date as a string.

Example:

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

6.70 FLOAT — Convert integer to default real

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

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = FLOAT(I)`

Arguments:

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

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

Example:

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

See also: Section 6.52 [DBLE], page 68, Section 6.54 [DFLOAT], page 69, Section 6.175 [REAL], page 135

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

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

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

Note that the `FGET` intrinsic is provided for backwards compatibility with `g77`. GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability. See also Chapter 4 [Fortran 2003 status], page 25.

Standard: GNU extension

Class: Subroutine, function

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

Arguments:

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

Example:

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

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

See also: Section 6.72 [FGETC], page 80, Section 6.76 [FPUT], page 82, Section 6.77 [FPUTC], page 83

6.72 FGETC — Read a single character in stream mode

Description:

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

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

Note that the `FGET` intrinsic is provided for backwards compatibility with `g77`. GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability. See also Chapter 4 [Fortran 2003 status], page 25.

Standard: GNU extension

Class: Subroutine, function

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

Arguments:

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

Example:

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

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

See also: Section 6.71 [FGET], page 79, Section 6.76 [FPUT], page 82, Section 6.77 [FPUTC], page 83

6.73 FLOOR — Integer floor function

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

Standard: F95 and later

Class: Elemental function

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

Arguments:

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

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

Example:

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

See also: Section 6.37 [CEILING], page 58, Section 6.160 [NINT], page 127

6.74 FLUSH — Flush I/O unit(s)

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

Standard: GNU extension

Class: Subroutine

Syntax: CALL FLUSH(UNIT)

Arguments:

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

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

6.75 `FNUM` — File number function

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

Standard: GNU extension

Class: Function

Syntax: `RESULT = FNUM(UNIT)`

Arguments:

UNIT	The type shall be `INTEGER`.

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

Example:

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

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

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

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

Note that the `FGET` intrinsic is provided for backwards compatibility with `g77`. GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability. See also Chapter 4 [Fortran 2003 status], page 25.

Standard: GNU extension

Class: Subroutine, function

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

Arguments:

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

Example:

```
PROGRAM test_fput
  CHARACTER(len=10) :: str = "gfortran"
  INTEGER :: i
  DO i = 1, len_trim(str)
    CALL fput(str(i:i))
  END DO
END PROGRAM
```

See also: Section 6.77 [FPUTC], page 83, Section 6.71 [FGET], page 79, Section 6.72 [FGETC], page 80

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

Description:

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

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

Note that the `FGET` intrinsic is provided for backwards compatibility with `g77`. GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability. See also Chapter 4 [Fortran 2003 status], page 25.

Standard: GNU extension

Class: Subroutine, function

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

Arguments:

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

Example:

```
PROGRAM test_fputc
  CHARACTER(len=10) :: str = "gfortran"
  INTEGER :: fd = 42, i

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

```
  CLOSE(fd)
END PROGRAM
```

See also: Section 6.76 [FPUT], page 82, Section 6.71 [FGET], page 79, Section 6.72 [FGETC], page 80

6.78 FRACTION — Fractional part of the model representation

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

Standard: F95 and later

Class: Elemental function

Syntax: `Y = FRACTION(X)`

Arguments:

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

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

Example:

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

6.79 FREE — Frees memory

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

Standard: GNU extension

Class: Subroutine

Syntax: `CALL FREE(PTR)`

Arguments:

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

Return value:
None

Example: See `MALLOC` for an example.

See also: Section 6.141 [MALLOC], page 116

6.80 FSEEK — Low level file positioning subroutine

Description:

Moves *UNIT* to the specified *OFFSET*. If *WHENCE* is set to 0, the *OFFSET* is taken as an absolute value SEEK_SET, if set to 1, *OFFSET* is taken to be relative to the current position SEEK_CUR, and if set to 2 relative to the end of the file SEEK_END. On error, *STATUS* is set to a nonzero value. If *STATUS* the seek fails silently.

This intrinsic routine is not fully backwards compatible with g77. In g77, the FSEEK takes a statement label instead of a *STATUS* variable. If FSEEK is used in old code, change

```
CALL FSEEK(UNIT, OFFSET, WHENCE, *label)
```

to

```
INTEGER :: status
CALL FSEEK(UNIT, OFFSET, WHENCE, status)
IF (status /= 0) GOTO label
```

Please note that GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability. See also Chapter 4 [Fortran 2003 status], page 25.

Standard: GNU extension

Class: Subroutine

Syntax: CALL FSEEK(UNIT, OFFSET, WHENCE[, STATUS])

Arguments:

UNIT	Shall be a scalar of type INTEGER.
OFFSET	Shall be a scalar of type INTEGER.
WHENCE	Shall be a scalar of type INTEGER. Its value shall be either 0, 1 or 2.
STATUS	(Optional) shall be a scalar of type INTEGER(4).

Example:

```
PROGRAM test_fseek
  INTEGER, PARAMETER :: SEEK_SET = 0, SEEK_CUR = 1, SEEK_END = 2
  INTEGER :: fd, offset, ierr

  ierr   = 0
  offset = 5
  fd     = 10

  OPEN(UNIT=fd, FILE="fseek.test")
  CALL FSEEK(fd, offset, SEEK_SET, ierr)  ! move to OFFSET
  print *, FTELL(fd), ierr

  CALL FSEEK(fd, 0, SEEK_END, ierr)       ! move to end
  print *, FTELL(fd), ierr

  CALL FSEEK(fd, 0, SEEK_SET, ierr)       ! move to beginning
  print *, FTELL(fd), ierr

  CLOSE(UNIT=fd)
END PROGRAM
```

See also: Section 6.82 [FTELL], page 86

6.81 FSTAT — Get file status

Description:

FSTAT is identical to Section 6.201 [STAT], page 149, except that information about an already opened file is obtained.

The elements in BUFF are the same as described by Section 6.201 [STAT], page 149.

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

Standard: GNU extension

Class: Subroutine, function

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

Arguments:

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

Example: See Section 6.201 [STAT], page 149 for an example.

See also: To stat a link: Section 6.139 [LSTAT], page 115, to stat a file: Section 6.201 [STAT], page 149

6.82 FTELL — Current stream position

Description:

Retrieves the current position within an open file.

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

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

Arguments:

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

Return value:

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

Example:

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

See also: Section 6.80 [FSEEK], page 85

6.83 GAMMA — Gamma function

Description:
GAMMA(X) computes Gamma (Γ) of *X*. For positive, integer values of *X* the Gamma function simplifies to the factorial function $\Gamma(x) = (x-1)!$.

$$\Gamma(x) = \int_0^\infty t^{x-1} \mathrm{e}^{-t} \, \mathrm{d}t$$

Standard: GNU Extension

Class: Elemental function

Syntax: `X = GAMMA(X)`

Arguments:

X	Shall be of type `REAL` and neither zero nor a negative integer.

Return value:
The return value is of type `REAL` of the same kind as *X*.

Example:

```
program test_gamma
  real :: x = 1.0
  x = gamma(x) ! returns 1.0
end program test_gamma
```

Specific names:

Name	Argument	Return type	Standard
`GAMMA(X)`	`REAL(4) X`	`REAL(4)`	GNU Extension
`DGAMMA(X)`	`REAL(8) X`	`REAL(8)`	GNU Extension

See also: Logarithm of the Gamma function: Section 6.126 [LGAMMA], page 108

6.84 GERROR — Get last system error message

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

Standard: GNU extension

Class: Subroutine

Syntax: `CALL GERROR(RESULT)`

Arguments:

RESULT	Shall of type `CHARACTER(*)`.

Example:

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

See also: Section 6.107 [IERRNO], page 99, Section 6.165 [PERROR], page 130

6.85 GETARG — Get command line arguments

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

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 6.87 [GET_COMMAND_ARGUMENT], page 89 intrinsic defined by the Fortran 2003 standard.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL GETARG(POS, VALUE)`

Arguments:

POS	Shall be of type `INTEGER` and not wider than the default integer kind; $POS \geq 0$
VALUE	Shall be of type `CHARACTER(*)`.

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

Example:

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

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

See also: GNU Fortran 77 compatibility function: Section 6.100 [IARGC], page 96

F2003 functions and subroutines: Section 6.86 [GET_COMMAND], page 89, Section 6.87 [GET_COMMAND_ARGUMENT], page 89, Section 6.42 [COMMAND_ARGUMENT_COUNT], page 62

6.86 GET_COMMAND — Get the entire command line

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

Standard: F2003

Class: Subroutine

Syntax: `CALL GET_COMMAND(CMD)`

Arguments:

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

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

Example:

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

See also: Section 6.87 [GET_COMMAND_ARGUMENT], page 89, Section 6.42 [COMMAND_ARGUMENT_COUNT], page 62

6.87 GET_COMMAND_ARGUMENT — Get command line arguments

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

Standard: F2003

Class: Subroutine

Syntax: `CALL GET_COMMAND_ARGUMENT(N, ARG)`

Arguments:

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

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

Example:

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

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

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

See also: Section 6.86 [GET_COMMAND], page 89, Section 6.42 [COMMAND_ARGUMENT_COUNT], page 62

6.88 GETCWD — Get current working directory

Description:
Get current working directory.

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

Standard: GNU extension

Class: Subroutine, function

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

Arguments:

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

Example:

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

See also: Section 6.39 [CHDIR], page 59

6.89 GETENV — Get an environmental variable

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

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 6.90 [GET_ENVIRONMENT_VARIABLE], page 91 intrinsic defined by the Fortran 2003 standard.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL GETENV(ENVVAR, VALUE)`

Arguments:

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

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

Example:

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

See also: Section 6.90 [GET_ENVIRONMENT_VARIABLE], page 91

6.90 GET_ENVIRONMENT_VARIABLE — Get an environmental variable

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

Standard: F2003

Class: Subroutine

Syntax: `CALL GET_ENVIRONMENT_VARIABLE(ENVVAR, VALUE)`

Arguments:

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

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

Example:

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

6.91 GETGID — Group ID function

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

Standard: GNU extension

Class: Function

Syntax: `RESULT = GETGID()`

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

Example: See `GETPID` for an example.

See also: Section 6.93 [GETPID], page 92, Section 6.94 [GETUID], page 93

6.92 `GETLOG` — Get login name

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

Standard: GNU extension

Class: Subroutine

Syntax: `CALL GETLOG(LOGIN)`

Arguments:

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

Return value:
Stores the current user name in *LOGIN*. (On systems where POSIX functions `geteuid` and `getpwuid` are not available, and the `getlogin` function is not implemented either, this will return a blank string.)

Example:

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

See also: Section 6.94 [GETUID], page 93

6.93 `GETPID` — Process ID function

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

Standard: GNU extension

Class: Function

Syntax: `RESULT = GETPID()`

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

Example:

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

See also: Section 6.91 [GETGID], page 91, Section 6.94 [GETUID], page 93

6.94 GETUID — User ID function

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

Standard: GNU extension

Class: Function

Syntax: `RESULT = GETUID()`

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

Example: See `GETPID` for an example.

See also: Section 6.93 [GETPID], page 92, Section 6.92 [GETLOG], page 92

6.95 GMTIME — Convert time to GMT info

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

Standard: GNU extension

Class: Subroutine

Syntax: `CALL GMTIME(STIME, TARRAY)`

Arguments:

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

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

See also: Section 6.50 [CTIME], page 66, Section 6.140 [LTIME], page 116, Section 6.208 [TIME], page 154, Section 6.209 [TIME8], page 155

6.96 HOSTNM — Get system host name

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

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

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

Arguments:

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

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

6.97 HUGE — Largest number of a kind

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

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = HUGE(X)`

Arguments:

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

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

Example:

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

6.98 IACHAR — Code in ASCII collating sequence

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

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = IACHAR(C [, KIND])`

Arguments:

C	Shall be a scalar `CHARACTER`, with `INTENT(IN)`
KIND	(Optional) An `INTEGER` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind.

Example:

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

Note: See Section 6.104 [ICHAR], page 97 for a discussion of converting between numerical values and formatted string representations.

See also: Section 6.5 [ACHAR], page 39, Section 6.38 [CHAR], page 59, Section 6.104 [ICHAR], page 97

6.99 `IAND` — Bitwise logical and

Description:
Bitwise logical `AND`.

Standard: F95 and later

Class: Elemental function

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

Arguments:

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

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

Example:

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

See also: Section 6.112 [IOR], page 102, Section 6.106 [IEOR], page 99, Section 6.102 [IBITS], page 97, Section 6.103 [IBSET], page 97, Section 6.101 [IBCLR], page 96, Section 6.161 [NOT], page 128

6.100 IARGC — Get the number of command line arguments

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

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 6.42 [COMMAND_ARGUMENT_COUNT], page 62 intrinsic defined by the Fortran 2003 standard.

Standard: GNU extension

Class: Function

Syntax: `RESULT = IARGC()`

Arguments:
None.

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

Example: See Section 6.85 [GETARG], page 88

See also: GNU Fortran 77 compatibility subroutine: Section 6.85 [GETARG], page 88

F2003 functions and subroutines: Section 6.86 [GET_COMMAND], page 89, Section 6.87 [GET_COMMAND_ARGUMENT], page 89, Section 6.42 [COMMAND_ARGUMENT_COUNT], page 62

6.101 IBCLR — Clear bit

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

Standard: F95 and later

Class: Elemental function

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

Arguments:

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

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

See also: Section 6.102 [IBITS], page 97, Section 6.103 [IBSET], page 97, Section 6.99 [IAND], page 95, Section 6.112 [IOR], page 102, Section 6.106 [IEOR], page 99, Section 6.157 [MVBITS], page 126

6.102 IBITS — Bit extraction

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

Standard: F95 and later

Class: Elemental function

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

Arguments:

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

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

See also: Section 6.30 [BIT_SIZE], page 54, Section 6.101 [IBCLR], page 96, Section 6.103 [IBSET], page 97, Section 6.99 [IAND], page 95, Section 6.112 [IOR], page 102, Section 6.106 [IEOR], page 99

6.103 IBSET — Set bit

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

Standard: F95 and later

Class: Elemental function

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

Arguments:

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

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

See also: Section 6.101 [IBCLR], page 96, Section 6.102 [IBITS], page 97, Section 6.99 [IAND], page 95, Section 6.112 [IOR], page 102, Section 6.106 [IEOR], page 99, Section 6.157 [MVBITS], page 126

6.104 ICHAR — Character-to-integer conversion function

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

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = ICHAR(C [, KIND])`

Arguments:

C	Shall be a scalar `CHARACTER`, with `INTENT(IN)`
KIND	(Optional) An `INTEGER` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind.

Example:

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

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

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

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

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

See also: Section 6.5 [ACHAR], page 39, Section 6.38 [CHAR], page 59, Section 6.98 [IACHAR], page 94

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

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

Standard: GNU extension

Class: Subroutine

Syntax: `CALL IDATE(TARRAY)`

Arguments:

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

Return value:
Does not return.

Example:

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

6.106 `IEOR` — Bitwise logical exclusive or

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

Standard: F95 and later

Class: Elemental function

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

Arguments:

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

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

See also: Section 6.112 [IOR], page 102, Section 6.99 [IAND], page 95, Section 6.102 [IBITS], page 97, Section 6.103 [IBSET], page 97, Section 6.101 [IBCLR], page 96, Section 6.161 [NOT], page 128

6.107 `IERRNO` — Get the last system error number

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

Standard: GNU extension

Class: Function

Syntax: `RESULT = IERRNO()`

Arguments:
None.

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

See also: Section 6.165 [PERROR], page 130

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

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

Standard: F77 and later

Class: Elemental function

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

Arguments:

STRING	Shall be a scalar `CHARACTER(*)`, with `INTENT(IN)`
SUBSTRING	Shall be a scalar `CHARACTER(*)`, with `INTENT(IN)`
BACK	(Optional) Shall be a scalar `LOGICAL(*)`, with `INTENT(IN)`
KIND	(Optional) An `INTEGER` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind.

See also: Section 6.182 [SCAN], page 139, Section 6.219 [VERIFY], page 159

6.109 `INT` — Convert to integer type

Description:
Convert to integer type

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = INT(A [, KIND))`

Arguments:

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

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

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

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

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

Example:

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

Specific names:

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

6.110 INT2 — Convert to 16-bit integer type

Description:
Convert to a KIND=2 integer type. This is equivalent to the standard INT intrinsic with an optional argument of KIND=2, and is only included for backwards compatibility.

The SHORT intrinsic is equivalent to INT2.

Standard: GNU extension.

Class: Elemental function

Syntax: RESULT = INT2(A)

Arguments:

A	Shall be of type INTEGER(*), REAL(*), or COMPLEX(*).

Return value:
The return value is a INTEGER(2) variable.

See also: Section 6.109 [INT], page 100, Section 6.111 [INT8], page 101, Section 6.137 [LONG], page 114

6.111 INT8 — Convert to 64-bit integer type

Description:
Convert to a KIND=8 integer type. This is equivalent to the standard INT intrinsic with an optional argument of KIND=8, and is only included for backwards compatibility.

Standard: GNU extension.

Class: Elemental function

Syntax: RESULT = INT8(A)

Arguments:

A	Shall be of type INTEGER(*), REAL(*), or COMPLEX(*).

Return value:
The return value is a INTEGER(8) variable.

See also: Section 6.109 [INT], page 100, Section 6.110 [INT2], page 101, Section 6.137 [LONG], page 114

6.112 IOR — Bitwise logical or

Description:
`IOR` returns the bitwise boolean inclusive-OR of *I* and *J*.

Standard: F95 and later

Class: Elemental function

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

Arguments:

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

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

See also: Section 6.106 [IEOR], page 99, Section 6.99 [IAND], page 95, Section 6.102 [IBITS], page 97, Section 6.103 [IBSET], page 97, Section 6.101 [IBCLR], page 96, Section 6.161 [NOT], page 128

6.113 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`.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. It implements a simple modulo generator as provided by `g77`. For new code, one should consider the use of Section 6.172 [RANDOM_NUMBER], page 133 as it implements a superior algorithm.

Standard: GNU extension

Class: Function

Syntax: `RESULT = IRAND(FLAG)`

Arguments:

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

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

Example:

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

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

6.114 IS_IOSTAT_END — Test for end-of-file value

Description:
IS_IOSTAT_END tests whether an variable has the value of the I/O status "end of file". The function is equivalent to comparing the variable with the IOSTAT_END parameter of the intrinsic module ISO_FORTRAN_ENV.

Standard: Fortran 2003.

Class: Elemental function

Syntax: RESULT = IS_IOSTAT_END(I)

Arguments:

I	Shall be of the type INTEGER.

Return value:
Returns a LOGICAL of the default kind, which .TRUE. if *I* has the value which indicates an end of file condition for IOSTAT= specifiers, and is .FALSE. otherwise.

Example:

```
PROGRAM iostat
  IMPLICIT NONE
  INTEGER :: stat, i
  OPEN(88, FILE='test.dat')
  READ(88, *, IOSTAT=stat) i
  IF(IS_IOSTAT_END(stat)) STOP 'END OF FILE'
END PROGRAM
```

6.115 IS_IOSTAT_EOR — Test for end-of-record value

Description:
IS_IOSTAT_EOR tests whether an variable has the value of the I/O status "end of record". The function is equivalent to comparing the variable with the IOSTAT_EOR parameter of the intrinsic module ISO_FORTRAN_ENV.

Standard: Fortran 2003.

Class: Elemental function

Syntax: RESULT = IS_IOSTAT_EOR(I)

Arguments:

I	Shall be of the type INTEGER.

Return value:
Returns a LOGICAL of the default kind, which .TRUE. if *I* has the value which indicates an end of file condition for IOSTAT= specifiers, and is .FALSE. otherwise.

Example:

```
PROGRAM iostat
  IMPLICIT NONE
  INTEGER :: stat, i(50)
  OPEN(88, FILE='test.dat', FORM='UNFORMATTED')
```

```
  READ(88, IOSTAT=stat) i
  IF(IS_IOSTAT_EOR(stat)) STOP 'END OF RECORD'
END PROGRAM
```

6.116 ISATTY — Whether a unit is a terminal device.

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

Standard: GNU extension.

Class: Function

Syntax: `RESULT = ISATTY(UNIT)`

Arguments:

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

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

Example:

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

See also: Section 6.214 [TTYNAM], page 157

6.117 ISHFT — Shift bits

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

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = ISHFT(I, SHIFT)`

Arguments:

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

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

See also: Section 6.118 [ISHFTC], page 105

6.118 ISHFTC — Shift bits circularly

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

Standard: F95 and later

Class: Elemental function

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

Arguments:

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

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

See also: Section 6.117 [ISHFT], page 104

6.119 ISNAN — Test for a NaN

Description:
`ISNAN` tests whether a floating-point value is an IEEE Not-a-Number (NaN).

Standard: GNU extension

Class: Elemental function

Syntax: `ISNAN(X)`

Arguments:

X	Variable of the type `REAL`.

Return value:
Returns a default-kind `LOGICAL`. The returned value is `TRUE` if *X* is a NaN and `FALSE` otherwise.

Example:

```
program test_nan
  implicit none
  real :: x
  x = -1.0
  x = sqrt(x)
  if (isnan(x)) stop '"x" is a NaN'
end program test_nan
```

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

Description:
: `IDATE(TARRAY)` Fills *TARRAY* with the numerical values at the current local time. The hour (in the range 1-24), minute (in the range 1-60), and seconds (in the range 1-60) appear in elements 1, 2, and 3 of *TARRAY*, respectively.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL ITIME(TARRAY)`

Arguments:

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

Return value:
: Does not return.

Example:

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

6.121 KILL — Send a signal to a process

Description:
Standard: Sends the signal specified by *SIGNAL* to the process *PID*. See `kill(2)`.

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

Class: Subroutine, function

Syntax: `CALL KILL(PID, SIGNAL [, STATUS])`

Arguments:

PID	Shall be a scalar `INTEGER`, with `INTENT(IN)`
SIGNAL	Shall be a scalar `INTEGER`, with `INTENT(IN)`
STATUS	(Optional) status flag of type `INTEGER(4)` or `INTEGER(8)`. Returns 0 on success, or a system-specific error code otherwise.

See also: Section 6.2 [ABORT], page 37, Section 6.66 [EXIT], page 77

6.122 KIND — Kind of an entity

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

Standard: F95 and later

Class: Inquiry function

Syntax: `K = KIND(X)`

Arguments:

X	Shall be of type `LOGICAL`, `INTEGER`, `REAL`, `COMPLEX` or `CHARACTER`.

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

Example:

```
program test_kind
  integer,parameter :: kc = kind(' ')
  integer,parameter :: kl = kind(.true.)

  print *, "The default character kind is ", kc
  print *, "The default logical kind is ", kl
end program test_kind
```

6.123 `LBOUND` — Lower dimension bounds of an array

Description:
Returns the lower bounds of an array, or a single lower bound along the *DIM* dimension.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = LBOUND(ARRAY [, DIM [, KIND]])`

Arguments:

ARRAY	Shall be an array, of any type.
DIM	(Optional) Shall be a scalar `INTEGER(*)`.
KIND	(Optional) An `INTEGER` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind. If *DIM* is absent, the result is an array of the lower bounds of *ARRAY*. If *DIM* is present, the result is a scalar corresponding to the lower bound of the array along that dimension. If *ARRAY* is an expression rather than a whole array or array structure component, or if it has a zero extent along the relevant dimension, the lower bound is taken to be 1.

See also: Section 6.215 [UBOUND], page 158

6.124 `LEN` — Length of a character entity

Description:
Returns the length of a character string. If *STRING* is an array, the length of an element of *STRING* is returned. Note that *STRING* need not be defined when

this intrinsic is invoked, since only the length, not the content, of *STRING* is needed.

Standard: F77 and later

Class: Inquiry function

Syntax: `L = LEN(STRING [, KIND])`

Arguments:

STRING	Shall be a scalar or array of type `CHARACTER(*)`, with `INTENT(IN)`
KIND	(Optional) An `INTEGER` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind.

See also: Section 6.125 [LEN_TRIM], page 108, Section 6.8 [ADJUSTL], page 41, Section 6.9 [ADJUSTR], page 41

6.125 LEN_TRIM — Length of a character entity without trailing blank characters

Description:
Returns the length of a character string, ignoring any trailing blanks.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = LEN_TRIM(STRING [, KIND])`

Arguments:

STRING	Shall be a scalar of type `CHARACTER(*)`, with `INTENT(IN)`
KIND	(Optional) An `INTEGER` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind.

See also: Section 6.124 [LEN], page 107, Section 6.8 [ADJUSTL], page 41, Section 6.9 [ADJUSTR], page 41

6.126 LGAMMA — Logarithm of the Gamma function

Description:
`GAMMA(X)` computes the natural logrithm of the absolute value of the Gamma (Γ) function.

Standard: GNU Extension

Class: Elemental function

Syntax: `X = LGAMMA(X)`

Arguments:

X	Shall be of type `REAL` and neither zero nor a negative integer.

Return value:
The return value is of type `REAL` of the same kind as *X*.

Example:

```
program test_log_gamma
  real :: x = 1.0
  x = lgamma(x) ! returns 0.0
end program test_log_gamma
```

Specific names:

Name	Argument	Return type	Standard
`LGAMMA(X)`	`REAL(4) X`	`REAL(4)`	GNU Extension
`ALGAMA(X)`	`REAL(4) X`	`REAL(4)`	GNU Extension
`DLGAMA(X)`	`REAL(8) X`	`REAL(8)`	GNU Extension

See also: Gamma function: Section 6.83 [GAMMA], page 87

6.127 `LGE` — Lexical greater than or equal

Description:
Determines whether one string is lexically greater than or equal to another string, where the two strings are interpreted as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if spaces were appended to it to form a value that has the same length as the longer.

In general, the lexical comparison intrinsics `LGE`, `LGT`, `LLE`, and `LLT` differ from the corresponding intrinsic operators `.GE.`, `.GT.`, `.LE.`, and `.LT.`, in that the latter use the processor's character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = LGE(STRING_A, STRING_B)`

Arguments:

STRING_A	Shall be of default `CHARACTER` type.
STRING_B	Shall be of default `CHARACTER` type.

Return value:
Returns `.TRUE.` if `STRING_A >= STRING_B`, and `.FALSE.` otherwise, based on the ASCII ordering.

See also: Section 6.128 [LGT], page 110, Section 6.130 [LLE], page 111, Section 6.131 [LLT], page 111

6.128 LGT — Lexical greater than

Description:
Determines whether one string is lexically greater than another string, where the two strings are interpreted as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if spaces were appended to it to form a value that has the same length as the longer.

In general, the lexical comparison intrinsics `LGE`, `LGT`, `LLE`, and `LLT` differ from the corresponding intrinsic operators `.GE.`, `.GT.`, `.LE.`, and `.LT.`, in that the latter use the processor's character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = LGT(STRING_A, STRING_B)`

Arguments:

STRING_A	Shall be of default `CHARACTER` type.
STRING_B	Shall be of default `CHARACTER` type.

Return value:
Returns `.TRUE.` if `STRING_A > STRING_B`, and `.FALSE.` otherwise, based on the ASCII ordering.

See also: Section 6.127 [LGE], page 109, Section 6.130 [LLE], page 111, Section 6.131 [LLT], page 111

6.129 LINK — Create a hard link

Description:
Makes a (hard) link from file *PATH1* to *PATH2*. A null character (`CHAR(0)`) can be used to mark the end of the names in *PATH1* and *PATH2*; otherwise, trailing blanks in the file names are ignored. If the *STATUS* argument is supplied, it contains 0 on success or a nonzero error code upon return; see `link(2)`.

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

```
CALL LINK(PATH1, PATH2 [, STATUS])
STATUS = LINK(PATH1, PATH2)
```

Arguments:

PATH1	Shall be of default `CHARACTER` type.
PATH2	Shall be of default `CHARACTER` type.
STATUS	(Optional) Shall be of default `INTEGER` type.

See also: Section 6.203 [SYMLNK], page 151, Section 6.217 [UNLINK], page 158

6.130 LLE — Lexical less than or equal

Description:

Determines whether one string is lexically less than or equal to another string, where the two strings are interpreted as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if spaces were appended to it to form a value that has the same length as the longer.

In general, the lexical comparison intrinsics `LGE`, `LGT`, `LLE`, and `LLT` differ from the corresponding intrinsic operators `.GE.`, `.GT.`, `.LE.`, and `.LT.`, in that the latter use the processor's character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = LLE(STRING_A, STRING_B)`

Arguments:

STRING_A	Shall be of default `CHARACTER` type.
STRING_B	Shall be of default `CHARACTER` type.

Return value:

Returns `.TRUE.` if `STRING_A <= STRING_B`, and `.FALSE.` otherwise, based on the ASCII ordering.

See also: Section 6.127 [LGE], page 109, Section 6.128 [LGT], page 110, Section 6.131 [LLT], page 111

6.131 LLT — Lexical less than

Description:

Determines whether one string is lexically less than another string, where the two strings are interpreted as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if spaces were appended to it to form a value that has the same length as the longer.

In general, the lexical comparison intrinsics `LGE`, `LGT`, `LLE`, and `LLT` differ from the corresponding intrinsic operators `.GE.`, `.GT.`, `.LE.`, and `.LT.`, in that the latter use the processor's character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = LLT(STRING_A, STRING_B)`

Arguments:

STRING_A	Shall be of default `CHARACTER` type.
STRING_B	Shall be of default `CHARACTER` type.

Return value:
Returns `.TRUE.` if `STRING_A < STRING_B`, and `.FALSE.` otherwise, based on the ASCII ordering.

See also: Section 6.127 [LGE], page 109, Section 6.128 [LGT], page 110, Section 6.130 [LLE], page 111

6.132 `LNBLNK` — Index of the last non-blank character in a string

Description:
Returns the length of a character string, ignoring any trailing blanks. This is identical to the standard `LEN_TRIM` intrinsic, and is only included for backwards compatibility.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = LNBLNK(STRING)`

Arguments:

STRING	Shall be a scalar of type `CHARACTER(*)`, with `INTENT(IN)`

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

See also: Section 6.108 [INDEX intrinsic], page 100, Section 6.125 [LEN_TRIM], page 108

6.133 `LOC` — Returns the address of a variable

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

Standard: GNU extension

Class: Inquiry function

Syntax: `RESULT = LOC(X)`

Arguments:

X	Variable of any type.

Return value:
The return value is of type `INTEGER`, with a `KIND` corresponding to the size (in bytes) of a memory address on the target machine.

Example:

```
program test_loc
  integer :: i
  real :: r
  i = loc(r)
  print *, i
end program test_loc
```

6.134 LOG — Logarithm function

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

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = LOG(X)

Arguments:

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

Return value:
The return value is of type REAL(*) or COMPLEX(*). The kind type parameter is the same as *X*.

Example:

```
program test_log
  real(8) :: x = 1.0_8
  complex :: z = (1.0, 2.0)
  x = log(x)
  z = log(z)
end program test_log
```

Specific names:

Name	Argument	Return type	Standard
ALOG(X)	REAL(4) X	REAL(4)	f95, gnu
DLOG(X)	REAL(8) X	REAL(8)	f95, gnu
CLOG(X)	COMPLEX(4) X	COMPLEX(4)	f95, gnu
ZLOG(X)	COMPLEX(8) X	COMPLEX(8)	f95, gnu
CDLOG(X)	COMPLEX(8) X	COMPLEX(8)	f95, gnu

6.135 LOG10 — Base 10 logarithm function

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

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = LOG10(X)

Arguments:

X	The type shall be REAL(*).

Return value:
The return value is of type REAL(*) or COMPLEX(*). The kind type parameter is the same as *X*.

Example:

```
program test_log10
  real(8) :: x = 10.0_8
  x = log10(x)
end program test_log10
```

Specific names:

Name	Argument	Return type	Standard
`ALOG10(X)`	`REAL(4) X`	`REAL(4)`	F95 and later
`DLOG10(X)`	`REAL(8) X`	`REAL(8)`	F95 and later

6.136 `LOGICAL` — Convert to logical type

Description:
Converts one kind of `LOGICAL` variable to another.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = LOGICAL(L [, KIND])`

Arguments:

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

Return value:
The return value is a `LOGICAL` value equal to *L*, with a kind corresponding to *KIND*, or of the default logical kind if *KIND* is not given.

See also: Section 6.109 [INT], page 100, Section 6.175 [REAL], page 135, Section 6.41 [CMPLX], page 61

6.137 `LONG` — Convert to integer type

Description:
Convert to a `KIND=4` integer type, which is the same size as a C `long` integer. This is equivalent to the standard `INT` intrinsic with an optional argument of `KIND=4`, and is only included for backwards compatibility.

Standard: GNU extension.

Class: Elemental function

Syntax: `RESULT = LONG(A)`

Arguments:

A	Shall be of type `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)`.

Return value:
The return value is a `INTEGER(4)` variable.

See also: Section 6.109 [INT], page 100, Section 6.110 [INT2], page 101, Section 6.111 [INT8], page 101

6.138 LSHIFT — Left shift bits

Description:
: LSHIFT returns a value corresponding to *I* with all of the bits shifted left by *SHIFT* places. If the absolute value of *SHIFT* is greater than BIT_SIZE(I), the value is undefined. Bits shifted out from the left end are lost; zeros are shifted in from the opposite end.

  This function has been superseded by the ISHFT intrinsic, which is standard in Fortran 95 and later.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = LSHIFT(I, SHIFT)

Arguments:

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

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

See also: Section 6.117 [ISHFT], page 104, Section 6.118 [ISHFTC], page 105, Section 6.180 [RSHIFT], page 138

6.139 LSTAT — Get file status

Description:
: LSTAT is identical to Section 6.201 [STAT], page 149, except that if path is a symbolic link, then the link itself is statted, not the file that it refers to.

  The elements in BUFF are the same as described by Section 6.201 [STAT], page 149.

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

Standard: GNU extension

Class: Subroutine, function

Syntax: CALL LSTAT(FILE, BUFF [, STATUS])

Arguments:

FILE	The type shall be CHARACTER(*), a valid path within the file system.
BUFF	The type shall be INTEGER(4), DIMENSION(13).
STATUS	(Optional) status flag of type INTEGER(4). Returns 0 on success and a system specific error code otherwise.

Example: See Section 6.201 [STAT], page 149 for an example.

See also: To stat an open file: Section 6.81 [FSTAT], page 86, to stat a file: Section 6.201 [STAT], page 149

6.140 LTIME — Convert time to local time info

Description:
Given a system time value *STIME* (as provided by the `TIME8()` intrinsic), fills *TARRAY* with values extracted from it appropriate to the local time zone using `localtime(3)`.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL LTIME(STIME, TARRAY)`

Arguments:

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

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

See also: Section 6.50 [CTIME], page 66, Section 6.95 [GMTIME], page 93, Section 6.208 [TIME], page 154, Section 6.209 [TIME8], page 155

6.141 MALLOC — Allocate dynamic memory

Description:
`MALLOC(SIZE)` allocates *SIZE* bytes of dynamic memory and returns the address of the allocated memory. The `MALLOC` intrinsic is an extension intended to be used with Cray pointers, and is provided in GNU Fortran to allow the user to compile legacy code. For new code using Fortran 95 pointers, the memory allocation intrinsic is `ALLOCATE`.

Standard: GNU extension

Class: Function

Syntax: `PTR = MALLOC(SIZE)`

Arguments:

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

Return value:

The return value is of type `INTEGER(K)`, with *K* such that variables of type `INTEGER(K)` have the same size as C pointers (`sizeof(void *)`).

Example: The following example demonstrates the use of `MALLOC` and `FREE` with Cray pointers. This example is intended to run on 32-bit systems, where the default integer kind is suitable to store pointers; on 64-bit systems, ptr_x would need to be declared as `integer(kind=8)`.

```
program test_malloc
  integer i
  integer ptr_x
  real*8 x(*), z
  pointer(ptr_x,x)

  ptr_x = malloc(20*8)
  do i = 1, 20
    x(i) = sqrt(1.0d0 / i)
  end do
  z = 0
  do i = 1, 20
    z = z + x(i)
    print *, z
  end do
  call free(ptr_x)
end program test_malloc
```

See also: Section 6.79 [FREE], page 84

6.142 `MATMUL` — matrix multiplication

Description:

Performs a matrix multiplication on numeric or logical arguments.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = MATMUL(MATRIX_A, MATRIX_B)`

Arguments:

MATRIX_A	An array of `INTEGER(*)`, `REAL(*)`, `COMPLEX(*)`, or `LOGICAL(*)` type, with a rank of one or two.
MATRIX_B	An array of `INTEGER(*)`, `REAL(*)`, or `COMPLEX(*)` type if *MATRIX_A* is of a numeric type; otherwise, an array of `LOGICAL(*)` type. The rank shall be one or two, and the first (or only) dimension of *MATRIX_B* shall be equal to the last (or only) dimension of *MATRIX_A*.

Return value:

The matrix product of *MATRIX_A* and *MATRIX_B*. The type and kind of the result follow the usual type and kind promotion rules, as for the `*` or `.AND.` operators.

See also:

6.143 MAX — Maximum value of an argument list

Description:
Returns the argument with the largest (most positive) value.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = MAX(A1, A2 [, A3 [, ...]])`

Arguments:

A1	The type shall be `INTEGER(*)` or `REAL(*)`.
A2, *A3*, ...	An expression of the same type and kind as *A1*. (As a GNU extension, arguments of different kinds are permitted.)

Return value:
The return value corresponds to the maximum value among the arguments, and has the same type and kind as the first argument.

Specific names:

Name	Argument	Return type	Standard
`MAX0(I)`	`INTEGER(4) I`	`INTEGER(4)`	F77 and later
`AMAX0(I)`	`INTEGER(4) I`	`REAL(MAX(X))`	F77 and later
`MAX1(X)`	`REAL(*) X`	`INT(MAX(X))`	F77 and later
`AMAX1(X)`	`REAL(4) X`	`REAL(4)`	F77 and later
`DMAX1(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

See also: Section 6.145 [MAXLOC], page 119 Section 6.146 [MAXVAL], page 119, Section 6.150 [MIN], page 121

6.144 MAXEXPONENT — Maximum exponent of a real kind

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

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = MAXEXPONENT(X)`

Arguments:

X	Shall be of type `REAL`.

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

Example:

```
program exponents
  real(kind=4) :: x
  real(kind=8) :: y

  print *, minexponent(x), maxexponent(x)
  print *, minexponent(y), maxexponent(y)
end program exponents
```

6.145 `MAXLOC` — Location of the maximum value within an array

Description:
Determines the location of the element in the array with the maximum value, or, if the *DIM* argument is supplied, determines the locations of the maximum element along each row of the array in the *DIM* direction. If *MASK* is present, only the elements for which *MASK* is `.TRUE.` are considered. If more than one element in the array has the maximum value, the location returned is that of the first such element in array element order. If the array has zero size, or all of the elements of *MASK* are `.FALSE.`, then the result is an array of zeroes. Similarly, if *DIM* is supplied and all of the elements of *MASK* along a given row are zero, the result value for that row is zero.

Standard: F95 and later

Class: Transformational function

Syntax:

```
RESULT = MAXLOC(ARRAY, DIM [, MASK])
RESULT = MAXLOC(ARRAY [, MASK])
```

Arguments:

ARRAY	Shall be an array of type `INTEGER(*)`, `REAL(*)`, or `CHARACTER(*)`.
DIM	(Optional) Shall be a scalar of type `INTEGER(*)`, with a value between one and the rank of *ARRAY*, inclusive. It may not be an optional dummy argument.
MASK	Shall be an array of type `LOGICAL(*)`, and conformable with *ARRAY*.

Return value:
If *DIM* is absent, the result is a rank-one array with a length equal to the rank of *ARRAY*. If *DIM* is present, the result is an array with a rank one less than the rank of *ARRAY*, and a size corresponding to the size of *ARRAY* with the *DIM* dimension removed. If *DIM* is present and *ARRAY* has a rank of one, the result is a scalar. In all cases, the result is of default `INTEGER` type.

See also: Section 6.143 [MAX], page 118, Section 6.146 [MAXVAL], page 119

6.146 `MAXVAL` — Maximum value of an array

Description:
Determines the maximum value of the elements in an array value, or, if the *DIM* argument is supplied, determines the maximum value along each row of the array in the *DIM* direction. If *MASK* is present, only the elements for which *MASK* is `.TRUE.` are considered. If the array has zero size, or all of the elements of *MASK* are `.FALSE.`, then the result is the most negative number of the type and kind of *ARRAY* if *ARRAY* is numeric, or a string of nulls if *ARRAY* is of character type.

Standard: F95 and later

Class: Transformational function

Syntax:

```
RESULT = MAXVAL(ARRAY, DIM [, MASK])
RESULT = MAXVAL(ARRAY [, MASK])
```

Arguments:

ARRAY	Shall be an array of type `INTEGER(*)`, `REAL(*)`, or `CHARACTER(*)`.
DIM	(Optional) Shall be a scalar of type `INTEGER(*)`, with a value between one and the rank of *ARRAY*, inclusive. It may not be an optional dummy argument.
MASK	Shall be an array of type `LOGICAL(*)`, and conformable with *ARRAY*.

Return value:

If *DIM* is absent, or if *ARRAY* has a rank of one, the result is a scalar. If *DIM* is present, the result is an array with a rank one less than the rank of *ARRAY*, and a size corresponding to the size of *ARRAY* with the *DIM* dimension removed. In all cases, the result is of the same type and kind as *ARRAY*.

See also: Section 6.143 [MAX], page 118, Section 6.145 [MAXLOC], page 119

6.147 `MCLOCK` — Time function

Description:

Returns the number of clock ticks since the start of the process, based on the UNIX function `clock(3)`.

This intrinsic is not fully portable, such as to systems with 32-bit `INTEGER` types but supporting times wider than 32 bits. Therefore, the values returned by this intrinsic might be, or become, negative, or numerically less than previous values, during a single run of the compiled program.

Standard: GNU extension

Class: Function

Syntax: `RESULT = MCLOCK()`

Return value:

The return value is a scalar of type `INTEGER(4)`, equal to the number of clock ticks since the start of the process, or `-1` if the system does not support `clock(3)`.

See also: Section 6.50 [CTIME], page 66, Section 6.95 [GMTIME], page 93, Section 6.140 [LTIME], page 116, Section 6.147 [MCLOCK], page 120, Section 6.208 [TIME], page 154

6.148 `MCLOCK8` — Time function (64-bit)

Description:

Returns the number of clock ticks since the start of the process, based on the UNIX function `clock(3)`.

Warning: this intrinsic does not increase the range of the timing values over that returned by `clock(3)`. On a system with a 32-bit `clock(3)`, `MCLOCK8()` will return a 32-bit value, even though it is converted to a 64-bit `INTEGER(8)` value. That means overflows of the 32-bit value can still occur. Therefore, the values returned by this intrinsic might be or become negative or numerically less than previous values during a single run of the compiled program.

Standard: GNU extension

Class: Function

Syntax: `RESULT = MCLOCK8()`

Return value:
The return value is a scalar of type `INTEGER(8)`, equal to the number of clock ticks since the start of the process, or `-1` if the system does not support `clock(3)`.

See also: Section 6.50 [CTIME], page 66, Section 6.95 [GMTIME], page 93, Section 6.140 [LTIME], page 116, Section 6.147 [MCLOCK], page 120, Section 6.209 [TIME8], page 155

6.149 `MERGE` — Merge variables

Description:
Select values from two arrays according to a logical mask. The result is equal to *TSOURCE* if *MASK* is `.TRUE.`, or equal to *FSOURCE* if it is `.FALSE.`.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = MERGE(TSOURCE, FSOURCE, MASK)`

Arguments:

TSOURCE	May be of any type.
FSOURCE	Shall be of the same type and type parameters as *TSOURCE*.
MASK	Shall be of type `LOGICAL(*)`.

Return value:
The result is of the same type and type parameters as *TSOURCE*.

6.150 `MIN` — Minimum value of an argument list

Description:
Returns the argument with the smallest (most negative) value.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = MIN(A1, A2 [, A3, ...])`

Arguments:

A1	The type shall be `INTEGER(*)` or `REAL(*)`.
A2, *A3*, ...	An expression of the same type and kind as *A1*. (As a GNU extension, arguments of different kinds are permitted.)

Return value:
The return value corresponds to the maximum value among the arguments, and has the same type and kind as the first argument.

Specific names:

Name	Argument	Return type	Standard
`MIN0(I)`	`INTEGER(4) I`	`INTEGER(4)`	F77 and later
`AMIN0(I)`	`INTEGER(4) I`	`REAL(MIN(X))`	F77 and later
`MIN1(X)`	`REAL(*) X`	`INT(MIN(X))`	F77 and later
`AMIN1(X)`	`REAL(4) X`	`REAL(4)`	F77 and later
`DMIN1(X)`	`REAL(8) X`	`REAL(8)`	F77 and later

See also: Section 6.143 [MAX], page 118, Section 6.152 [MINLOC], page 122, Section 6.153 [MINVAL], page 123

6.151 `MINEXPONENT` — Minimum exponent of a real kind

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

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = MINEXPONENT(X)`

Arguments:
X Shall be of type `REAL`.

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

Example: See `MAXEXPONENT` for an example.

6.152 `MINLOC` — Location of the minimum value within an array

Description:
Determines the location of the element in the array with the minimum value, or, if the *DIM* argument is supplied, determines the locations of the minimum element along each row of the array in the *DIM* direction. If *MASK* is present, only the elements for which *MASK* is `.TRUE.` are considered. If more than one element in the array has the minimum value, the location returned is that of the first such element in array element order. If the array has zero size, or all of the elements of *MASK* are `.FALSE.`, then the result is an array of zeroes. Similarly, if *DIM* is supplied and all of the elements of *MASK* along a given row are zero, the result value for that row is zero.

Standard: F95 and later

Class: Transformational function

Syntax:

```
RESULT = MINLOC(ARRAY, DIM [, MASK])
RESULT = MINLOC(ARRAY [, MASK])
```

Arguments:

ARRAY	Shall be an array of type `INTEGER(*)`, `REAL(*)`, or `CHARACTER(*)`.
DIM	(Optional) Shall be a scalar of type `INTEGER(*)`, with a value between one and the rank of *ARRAY*, inclusive. It may not be an optional dummy argument.
MASK	Shall be an array of type `LOGICAL(*)`, and conformable with *ARRAY*.

Return value:

If *DIM* is absent, the result is a rank-one array with a length equal to the rank of *ARRAY*. If *DIM* is present, the result is an array with a rank one less than the rank of *ARRAY*, and a size corresponding to the size of *ARRAY* with the *DIM* dimension removed. If *DIM* is present and *ARRAY* has a rank of one, the result is a scalar. In all cases, the result is of default `INTEGER` type.

See also: Section 6.150 [MIN], page 121, Section 6.153 [MINVAL], page 123

6.153 `MINVAL` — Minimum value of an array

Description:

Determines the minimum value of the elements in an array value, or, if the *DIM* argument is supplied, determines the minimum value along each row of the array in the *DIM* direction. If *MASK* is present, only the elements for which *MASK* is `.TRUE.` are considered. If the array has zero size, or all of the elements of *MASK* are `.FALSE.`, then the result is `HUGE(ARRAY)` if *ARRAY* is numeric, or a string of `CHAR(255)` characters if *ARRAY* is of character type.

Standard: F95 and later

Class: Transformational function

Syntax:

```
RESULT = MINVAL(ARRAY, DIM [, MASK])
RESULT = MINVAL(ARRAY [, MASK])
```

Arguments:

ARRAY	Shall be an array of type `INTEGER(*)`, `REAL(*)`, or `CHARACTER(*)`.
DIM	(Optional) Shall be a scalar of type `INTEGER(*)`, with a value between one and the rank of *ARRAY*, inclusive. It may not be an optional dummy argument.
MASK	Shall be an array of type `LOGICAL(*)`, and conformable with *ARRAY*.

Return value:

If *DIM* is absent, or if *ARRAY* has a rank of one, the result is a scalar. If *DIM* is present, the result is an array with a rank one less than the rank of *ARRAY*, and a size corresponding to the size of *ARRAY* with the *DIM* dimension removed. In all cases, the result is of the same type and kind as *ARRAY*.

See also: Section 6.150 [MIN], page 121, Section 6.152 [MINLOC], page 122

6.154 MOD — Remainder function

Description:
`MOD(A,P)` computes the remainder of the division of A by P. It is calculated as `A - (INT(A/P) * P)`.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = MOD(A, P)`

Arguments:

A	Shall be a scalar of type `INTEGER` or `REAL`
P	Shall be a scalar of the same type as *A* and not equal to zero

Return value:
The kind of the return value is the result of cross-promoting the kinds of the arguments.

Example:

```
program test_mod
  print *, mod(17,3)
  print *, mod(17.5,5.5)
  print *, mod(17.5d0,5.5)
  print *, mod(17.5,5.5d0)

  print *, mod(-17,3)
  print *, mod(-17.5,5.5)
  print *, mod(-17.5d0,5.5)
  print *, mod(-17.5,5.5d0)

  print *, mod(17,-3)
  print *, mod(17.5,-5.5)
  print *, mod(17.5d0,-5.5)
  print *, mod(17.5,-5.5d0)
end program test_mod
```

Specific names:

Name	Arguments	Return type	Standard
`AMOD(A,P)`	`REAL(4)`	`REAL(4)`	F95 and later
`DMOD(A,P)`	`REAL(8)`	`REAL(8)`	F95 and later

6.155 MODULO — Modulo function

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

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = MODULO(A, P)`

Arguments:

A	Shall be a scalar of type `INTEGER` or `REAL`
P	Shall be a scalar of the same type and kind as *A*

Return value:
The type and kind of the result are those of the arguments.

If *A* and *P* are of type `INTEGER`:
`MODULO(A,P)` has the value *R* such that `A=Q*P+R`, where *Q* is an integer and *R* is between 0 (inclusive) and *P* (exclusive).

If *A* and *P* are of type `REAL`:
`MODULO(A,P)` has the value of `A - FLOOR (A / P) * P`.

In all cases, if *P* is zero the result is processor-dependent.

Example:

```
program test_modulo
  print *, modulo(17,3)
  print *, modulo(17.5,5.5)

  print *, modulo(-17,3)
  print *, modulo(-17.5,5.5)

  print *, modulo(17,-3)
  print *, modulo(17.5,-5.5)
end program
```

6.156 `MOVE_ALLOC` — Move allocation from one object to another

Description:
`MOVE_ALLOC(SRC, DEST)` moves the allocation from *SRC* to *DEST*. *SRC* will become deallocated in the process.

Standard: F2003 and later

Class: Subroutine

Syntax: `CALL MOVE_ALLOC(SRC, DEST)`

Arguments:

SRC	`ALLOCATABLE`, `INTENT(INOUT)`, may be of any type and kind.
DEST	`ALLOCATABLE`, `INTENT(OUT)`, shall be of the same type, kind and rank as *SRC*

Return value:
None

Example:

```
program test_move_alloc
    integer, allocatable :: a(:), b(:)

    allocate(a(3))
```

```
    a = [ 1, 2, 3 ]
    call move_alloc(a, b)
    print *, allocated(a), allocated(b)
    print *, b
end program test_move_alloc
```

6.157 MVBITS — Move bits from one integer to another

Description:
: Moves *LEN* bits from positions *FROMPOS* through `FROMPOS+LEN-1` of *FROM* to positions *TOPOS* through `TOPOS+LEN-1` of *TO*. The portion of argument *TO* not affected by the movement of bits is unchanged. The values of `FROMPOS+LEN-1` and `TOPOS+LEN-1` must be less than `BIT_SIZE(FROM)`.

Standard: F95 and later

Class: Elemental subroutine

Syntax: `CALL MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)`

Arguments:

FROM	The type shall be `INTEGER(*)`.
FROMPOS	The type shall be `INTEGER(*)`.
LEN	The type shall be `INTEGER(*)`.
TO	The type shall be `INTEGER(*)`, of the same kind as *FROM*.
TOPOS	The type shall be `INTEGER(*)`.

See also: Section 6.101 [IBCLR], page 96, Section 6.103 [IBSET], page 97, Section 6.102 [IBITS], page 97, Section 6.99 [IAND], page 95, Section 6.112 [IOR], page 102, Section 6.106 [IEOR], page 99

6.158 NEAREST — Nearest representable number

Description:
: `NEAREST(X, S)` returns the processor-representable number nearest to `X` in the direction indicated by the sign of `S`.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = NEAREST(X, S)`

Arguments:

X	Shall be of type `REAL`.
S	(Optional) shall be of type `REAL` and not equal to zero.

Return value:
: The return value is of the same type as `X`. If `S` is positive, `NEAREST` returns the processor-representable number greater than `X` and nearest to it. If `S` is negative, `NEAREST` returns the processor-representable number smaller than `X` and nearest to it.

Example:

```
program test_nearest
  real :: x, y
  x = nearest(42.0, 1.0)
  y = nearest(42.0, -1.0)
  write (*,"(3(G20.15))") x, y, x - y
end program test_nearest
```

6.159 NEW_LINE — New line character

Description:
NEW_LINE(C) returns the new-line character.

Standard: F2003 and later

Class: Inquiry function

Syntax: RESULT = NEW_LINE(C)

Arguments:

C	The argument shall be a scalar or array of the type CHARACTER.

Return value:
Returns a *CHARACTER* scalar of length one with the new-line character of the same kind as parameter *C*.

Example:

```
program newline
  implicit none
  write(*,'(A)') 'This is record 1.'//NEW_LINE('A')//'This is record 2.'
end program newline
```

6.160 NINT — Nearest whole number

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

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = NINT(X)

Arguments:

X	The type of the argument shall be REAL.

Return value:
Returns *A* with the fractional portion of its magnitude eliminated by rounding to the nearest whole number and with its sign preserved, converted to an INTEGER of the default kind.

Example:

```
program test_nint
  real(4) x4
  real(8) x8
  x4 = 1.234E0_4
```

```
    x8 = 4.321_8
    print *, nint(x4), idnint(x8)
  end program test_nint
```

Specific names:

Name	Argument	Standard
IDNINT(X)	REAL(8)	F95 and later

See also: Section 6.37 [CEILING], page 58, Section 6.73 [FLOOR], page 81

6.161 NOT — Logical negation

Description:
NOT returns the bitwise boolean inverse of *I*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = NOT(I)`

Arguments:

I	The type shall be INTEGER(*).

Return value:
The return type is INTEGER(*), of the same kind as the argument.

See also: Section 6.99 [IAND], page 95, Section 6.106 [IEOR], page 99, Section 6.112 [IOR], page 102, Section 6.102 [IBITS], page 97, Section 6.103 [IBSET], page 97, Section 6.101 [IBCLR], page 96

6.162 NULL — Function that returns an disassociated pointer

Description:
Returns a disassociated pointer.

If *MOLD* is present, a dissassociated pointer of the same type is returned, otherwise the type is determined by context.

In Fortran 95, *MOLD* is optional. Please note that F2003 includes cases where it is required.

Standard: F95 and later

Class: Transformational function

Syntax: `PTR => NULL([MOLD])`

Arguments:

MOLD	(Optional) shall be a pointer of any association status and of any type.

Return value:
A disassociated pointer.

Example:

```
REAL, POINTER, DIMENSION(:) :: VEC => NULL ()
```

See also: Section 6.20 [ASSOCIATED], page 48

6.163 OR — Bitwise logical OR

Description:

Bitwise logical OR.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. For integer arguments, programmers should consider the use of the Section 6.112 [IOR], page 102 intrinsic defined by the Fortran standard.

Standard: GNU extension

Class: Function

Syntax: RESULT = OR(X, Y)

Arguments:

X	The type shall be either INTEGER(*) or LOGICAL.
Y	The type shall be either INTEGER(*) or LOGICAL.

Return value:

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

Example:

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

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

See also: F95 elemental function: Section 6.112 [IOR], page 102

6.164 PACK — Pack an array into an array of rank one

Description:

Stores the elements of *ARRAY* in an array of rank one.

The beginning of the resulting array is made up of elements whose *MASK* equals TRUE. Afterwards, positions are filled with elements taken from *VECTOR*.

Standard: F95 and later

Class: Transformational function

Syntax: RESULT = PACK(ARRAY, MASK[,VECTOR]

Arguments:

ARRAY	Shall be an array of any type.
MASK	Shall be an array of type LOGICAL and of the same size as *ARRAY*. Alternatively, it may be a LOGICAL scalar.

VECTOR (Optional) shall be an array of the same type as *ARRAY* and of rank one. If present, the number of elements in *VECTOR* shall be equal to or greater than the number of true elements in *MASK*. If *MASK* is scalar, the number of elements in *VECTOR* shall be equal to or greater than the number of elements in *ARRAY*.

Return value:
The result is an array of rank one and the same type as that of *ARRAY*. If *VECTOR* is present, the result size is that of *VECTOR*, the number of `TRUE` values in *MASK* otherwise.

Example: Gathering nonzero elements from an array:

```
PROGRAM test_pack_1
  INTEGER :: m(6)
  m = (/ 1, 0, 0, 0, 5, 0 /)
  WRITE(*, FMT="(6(I0, ' '))") pack(m, m /= 0)  ! "1 5"
END PROGRAM
```

Gathering nonzero elements from an array and appending elements from *VECTOR*:

```
PROGRAM test_pack_2
  INTEGER :: m(4)
  m = (/ 1, 0, 0, 2 /)
  WRITE(*, FMT="(4(I0, ' '))") pack(m, m /= 0, (/ 0, 0, 3, 4 /))  ! "1 2 3 4"
END PROGRAM
```

See also: Section 6.218 [UNPACK], page 159

6.165 `PERROR` — Print system error message

Description:
Prints (on the C `stderr` stream) a newline-terminated error message corresponding to the last system error. This is prefixed by *STRING*, a colon and a space. See `perror(3)`.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL PERROR(STRING)`

Arguments:

STRING A scalar of default `CHARACTER` type.

See also: Section 6.107 [IERRNO], page 99

6.166 `PRECISION` — Decimal precision of a real kind

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

Standard: F95 and later

Class: Inquiry function

Syntax: RESULT = PRECISION(X)

Arguments:

X	Shall be of type REAL or COMPLEX.

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

Example:

```
program prec_and_range
  real(kind=4) :: x(2)
  complex(kind=8) :: y

  print *, precision(x), range(x)
  print *, precision(y), range(y)
end program prec_and_range
```

6.167 PRESENT — Determine whether an optional dummy argument is specified

Description:
Determines whether an optional dummy argument is present.

Standard: F95 and later

Class: Inquiry function

Syntax: RESULT = PRESENT(A)

Arguments:

A	May be of any type and may be a pointer, scalar or array value, or a dummy procedure. It shall be the name of an optional dummy argument accessible within the current subroutine or function.

Return value:
Returns either TRUE if the optional argument *A* is present, or FALSE otherwise.

Example:

```
PROGRAM test_present
  WRITE(*,*) f(), f(42)      ! "F T"
CONTAINS
  LOGICAL FUNCTION f(x)
    INTEGER, INTENT(IN), OPTIONAL :: x
    f = PRESENT(x)
  END FUNCTION
END PROGRAM
```

6.168 PRODUCT — Product of array elements

Description:
Multiplies the elements of *ARRAY* along dimension *DIM* if the corresponding element in *MASK* is TRUE.

Standard: F95 and later

Class: Transformational function

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

Arguments:

ARRAY	Shall be an array of type `INTEGER(*)`, `REAL(*)` or `COMPLEX(*)`.
DIM	(Optional) shall be a scalar of type `INTEGER` with a value in the range from 1 to n, where n equals the rank of *ARRAY*.
MASK	(Optional) shall be of type `LOGICAL` and either be a scalar or an array of the same shape as *ARRAY*.

Return value:

The result is of the same type as *ARRAY*.

If *DIM* is absent, a scalar with the product of all elements in *ARRAY* is returned. Otherwise, an array of rank n-1, where n equals the rank of *ARRAY*, and a shape similar to that of *ARRAY* with dimension *DIM* dropped is returned.

Example:

```
PROGRAM test_product
  INTEGER :: x(5) = (/ 1, 2, 3, 4 ,5 /)
  print *, PRODUCT(x)                    ! all elements, product = 120
  print *, PRODUCT(x, MASK=MOD(x, 2)==1) ! odd elements, product = 15
END PROGRAM
```

See also: Section 6.202 [SUM], page 151

6.169 `RADIX` — Base of a model number

Description:

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

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = RADIX(X)`

Arguments:

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

Return value:

The return value is a scalar of type `INTEGER` and of the default integer kind.

Example:

```
program test_radix
  print *, "The radix for the default integer kind is", radix(0)
  print *, "The radix for the default real kind is", radix(0.0)
end program test_radix
```

6.170 RAN — Real pseudo-random number

Description:
For compatibility with HP FORTRAN 77/iX, the RAN intrinsic is provided as an alias for RAND. See Section 6.171 [RAND], page 133 for complete documentation.

Standard: GNU extension

Class: Function

See also: Section 6.171 [RAND], page 133, Section 6.172 [RANDOM_NUMBER], page 133

6.171 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.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. It implements a simple modulo generator as provided by g77. For new code, one should consider the use of Section 6.172 [RANDOM_NUMBER], page 133 as it implements a superior algorithm.

Standard: GNU extension

Class: Function

Syntax: RESULT = RAND(FLAG)

Arguments:

FLAG	Shall be a scalar INTEGER of kind 4.

Return value:
The return value is of REAL type and the default kind.

Example:

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

  call srand(seed)
  print *, rand(), rand(), rand(), rand()
  print *, rand(seed), rand(), rand(), rand()
end program test_rand
```

See also: Section 6.200 [SRAND], page 149, Section 6.172 [RANDOM_NUMBER], page 133

6.172 RANDOM_NUMBER — Pseudo-random number

Description:
Returns a single pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 \leq x < 1$.

The runtime-library implements George Marsaglia's KISS (Keep It Simple Stupid) random number generator (RNG). This RNG combines:

1. The congruential generator $x(n) = 69069 \cdot x(n-1) + 1327217885$ with a period of 2^{32},
2. A 3-shift shift-register generator with a period of $2^{32} - 1$,
3. Two 16-bit multiply-with-carry generators with a period of $597273182964842497 > 2^{59}$.

The overall period exceeds 2^{123}.

Please note, this RNG is thread safe if used within OpenMP directives, i. e. its state will be consistent while called from multiple threads. However, the KISS generator does not create random numbers in parallel from multiple sources, but in sequence from a single source. If an OpenMP-enabled application heavily relies on random numbers, one should consider employing a dedicated parallel random number generator instead.

Standard: F95 and later

Class: Subroutine

Syntax: `RANDOM_NUMBER(HARVEST)`

Arguments:

HARVEST Shall be a scalar or an array of type `REAL(*)`.

Example:

```
program test_random_number
  REAL :: r(5,5)
  CALL init_random_seed()         ! see example of RANDOM_SEED
  CALL RANDOM_NUMBER(r)
end program
```

See also: Section 6.173 [RANDOM_SEED], page 134

6.173 `RANDOM_SEED` — Initialize a pseudo-random number sequence

Description:

Restarts or queries the state of the pseudorandom number generator used by `RANDOM_NUMBER`.

If `RANDOM_SEED` is called without arguments, it is initialized to a default state. The example below shows how to initialize the random seed based on the system's time.

Standard: F95 and later

Class: Subroutine

Syntax: `CALL RANDOM_SEED(SIZE, PUT, GET)`

Arguments:

SIZE (Optional) Shall be a scalar and of type default `INTEGER`, with `INTENT(OUT)`. It specifies the minimum size of the arrays used with the *PUT* and *GET* arguments.

PUT	(Optional) Shall be an array of type default `INTEGER` and rank one. It is `INTENT(IN)` and the size of the array must be larger than or equal to the number returned by the *SIZE* argument.
GET	(Optional) Shall be an array of type default `INTEGER` and rank one. It is `INTENT(OUT)` and the size of the array must be larger than or equal to the number returned by the *SIZE* argument.

Example:

```
SUBROUTINE init_random_seed()
  INTEGER :: i, n, clock
  INTEGER, DIMENSION(:), ALLOCATABLE :: seed

  CALL RANDOM_SEED(size = n)
  ALLOCATE(seed(n))

  CALL SYSTEM_CLOCK(COUNT=clock)

  seed = clock + 37 * (/ (i - 1, i = 1, n) /)
  CALL RANDOM_SEED(PUT = seed)

  DEALLOCATE(seed)
END SUBROUTINE
```

See also: Section 6.172 [RANDOM_NUMBER], page 133

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

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

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = RANGE(X)`

Arguments:

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

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

Example: See `PRECISION` for an example.

6.175 `REAL` — Convert to real type

Description:
`REAL(X [, KIND])` converts its argument *X* to a real type. The `REALPART(X)` function is provided for compatibility with `g77`, and its use is strongly discouraged.

Standard: F77 and later

Class: Elemental function

Syntax:

```
RESULT = REAL(X [, KIND])
RESULT = REALPART(Z)
```

Arguments:

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

Return value:

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

(A) `REAL(X)` is converted to a default real type if *X* is an integer or real variable.

(B) `REAL(X)` is converted to a real type with the kind type parameter of *X* if *X* is a complex variable.

(C) `REAL(X, KIND)` is converted to a real type with kind type parameter *KIND* if *X* is a complex, integer, or real variable.

Example:

```
program test_real
  complex :: x = (1.0, 2.0)
  print *, real(x), real(x,8), realpart(x)
end program test_real
```

See also: Section 6.52 [DBLE], page 68, Section 6.54 [DFLOAT], page 69, Section 6.70 [FLOAT], page 79

6.176 `RENAME` — Rename a file

Description:

Renames a file from file *PATH1* to *PATH2*. A null character (`CHAR(0)`) can be used to mark the end of the names in *PATH1* and *PATH2*; otherwise, trailing blanks in the file names are ignored. If the *STATUS* argument is supplied, it contains 0 on success or a nonzero error code upon return; see `rename(2)`.

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

```
CALL RENAME(PATH1, PATH2 [, STATUS])
STATUS = RENAME(PATH1, PATH2)
```

Arguments:

PATH1	Shall be of default `CHARACTER` type.
PATH2	Shall be of default `CHARACTER` type.
STATUS	(Optional) Shall be of default `INTEGER` type.

See also: Section 6.129 [LINK], page 110

6.177 REPEAT — Repeated string concatenation

Description:
Concatenates *NCOPIES* copies of a string.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = REPEAT(STRING, NCOPIES)`

Arguments:

STRING	Shall be scalar and of type `CHARACTER(*)`.
NCOPIES	Shall be scalar and of type `INTEGER(*)`.

Return value:
A new scalar of type `CHARACTER` built up from *NCOPIES* copies of *STRING*.

Example:

```
program test_repeat
  write(*,*) repeat("x", 5)   ! "xxxxx"
end program
```

6.178 RESHAPE — Function to reshape an array

Description:
Reshapes *SOURCE* to correspond to *SHAPE*. If necessary, the new array may be padded with elements from *PAD* or permuted as defined by *ORDER*.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = RESHAPE(SOURCE, SHAPE[, PAD, ORDER])`

Arguments:

SOURCE	Shall be an array of any type.
SHAPE	Shall be of type `INTEGER` and an array of rank one. Its values must be positive or zero.
PAD	(Optional) shall be an array of the same type as *SOURCE*.
ORDER	(Optional) shall be of type `INTEGER` and an array of the same shape as *SHAPE*. Its values shall be a permutation of the numbers from 1 to n, where n is the size of *SHAPE*. If *ORDER* is absent, the natural ordering shall be assumed.

Return value:
The result is an array of shape *SHAPE* with the same type as *SOURCE*.

Example:

```
PROGRAM test_reshape
  INTEGER, DIMENSION(4) :: x
  WRITE(*,*) SHAPE(x)                       ! prints "4"
  WRITE(*,*) SHAPE(RESHAPE(x, (/2, 2/)))    ! prints "2 2"
END PROGRAM
```

See also: Section 6.188 [SHAPE], page 142

6.179 `RRSPACING` — Reciprocal of the relative spacing

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

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = RRSPACING(X)`

Arguments:

X	Shall be of type `REAL`.

Return value:
The return value is of the same type and kind as *X*. The value returned is equal to `ABS(FRACTION(X)) * FLOAT(RADIX(X))**DIGITS(X)`.

See also: Section 6.197 [SPACING], page 147

6.180 `RSHIFT` — Right shift bits

Description:
`RSHIFT` returns a value corresponding to *I* with all of the bits shifted right by *SHIFT* places. If the absolute value of *SHIFT* is greater than `BIT_SIZE(I)`, the value is undefined. Bits shifted out from the left end are lost; zeros are shifted in from the opposite end.

This function has been superseded by the `ISHFT` intrinsic, which is standard in Fortran 95 and later.

Standard: GNU extension

Class: Elemental function

Syntax: `RESULT = RSHIFT(I, SHIFT)`

Arguments:

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

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

See also: Section 6.117 [ISHFT], page 104, Section 6.118 [ISHFTC], page 105, Section 6.138 [LSHIFT], page 115

6.181 `SCALE` — Scale a real value

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

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = SCALE(X, I)`

Arguments:

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

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

Example:

```
program test_scale
  real :: x = 178.1387e-4
  integer :: i = 5
  print *, scale(x,i), x*radix(x)**i
end program test_scale
```

6.182 SCAN — Scan a string for the presence of a set of characters

Description:
Scans a *STRING* for any of the characters in a *SET* of characters.

If *BACK* is either absent or equals `FALSE`, this function returns the position of the leftmost character of *STRING* that is in *SET*. If *BACK* equals `TRUE`, the rightmost position is returned. If no character of *SET* is found in *STRING*, the result is zero.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = SCAN(STRING, SET[, BACK [, KIND]])`

Arguments:

STRING	Shall be of type `CHARACTER(*)`.
SET	Shall be of type `CHARACTER(*)`.
BACK	(Optional) shall be of type `LOGICAL`.
KIND	(Optional) An `INTEGER` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind.

Example:

```
PROGRAM test_scan
  WRITE(*,*) SCAN("FORTRAN", "AO")          ! 2, found 'O'
  WRITE(*,*) SCAN("FORTRAN", "AO", .TRUE.)  ! 6, found 'A'
  WRITE(*,*) SCAN("FORTRAN", "C++")         ! 0, found none
END PROGRAM
```

See also: Section 6.108 [INDEX intrinsic], page 100, Section 6.219 [VERIFY], page 159

6.183 SECNDS — Time function

Description:
SECNDS(X) gets the time in seconds from the real-time system clock. *X* is a reference time, also in seconds. If this is zero, the time in seconds from midnight is returned. This function is non-standard and its use is discouraged.

Standard: GNU extension

Class: Function

Syntax: RESULT = SECNDS (X)

Arguments:

T	Shall be of type REAL(4).
X	Shall be of type REAL(4).

Return value:
None

Example:

```
program test_secnds
    integer :: i
    real(4) :: t1, t2
    print *, secnds (0.0)   ! seconds since midnight
    t1 = secnds (0.0)       ! reference time
    do i = 1, 10000000      ! do something
    end do
    t2 = secnds (t1)        ! elapsed time
    print *, "Something took ", t2, " seconds."
end program test_secnds
```

6.184 SECOND — CPU time function

Description:
Returns a REAL(4) value representing the elapsed CPU time in seconds. This provides the same functionality as the standard CPU_TIME intrinsic, and is only included for backwards compatibility.

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

CALL SECOND(TIME)
TIME = SECOND()

Arguments:

TIME	Shall be of type REAL(4).

Return value:
In either syntax, *TIME* is set to the process's current runtime in seconds.

See also: Section 6.48 [CPU_TIME], page 65

6.185 SELECTED_INT_KIND — Choose integer kind

Description:
`SELECTED_INT_KIND(I)` return the kind value of the smallest integer type that can represent all values ranging from -10^I (exclusive) to 10^I (exclusive). If there is no integer kind that accommodates this range, `SELECTED_INT_KIND` returns -1.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = SELECTED_INT_KIND(I)`

Arguments:

I	Shall be a scalar and of type `INTEGER`.

Example:

```
program large_integers
  integer,parameter :: k5 = selected_int_kind(5)
  integer,parameter :: k15 = selected_int_kind(15)
  integer(kind=k5) :: i5
  integer(kind=k15) :: i15

  print *, huge(i5), huge(i15)

  ! The following inequalities are always true
  print *, huge(i5) >= 10_k5**5-1
  print *, huge(i15) >= 10_k15**15-1
end program large_integers
```

6.186 SELECTED_REAL_KIND — Choose real kind

Description:
`SELECTED_REAL_KIND(P,R)` return the kind value of a real data type with decimal precision greater of at least `P` digits and exponent range greater at least `R`.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = SELECTED_REAL_KIND(P, R)`

Arguments:

P	(Optional) shall be a scalar and of type `INTEGER`.
R	(Optional) shall be a scalar and of type `INTEGER`.

At least one argument shall be present.

Return value:
`SELECTED_REAL_KIND` returns the value of the kind type parameter of a real data type with decimal precision of at least `P` digits and a decimal exponent range of at least `R`. If more than one real data type meet the criteria, the kind of the data type with the smallest decimal precision is returned. If no real data type matches the criteria, the result is

-1 if the processor does not support a real data type with a precision greater than or equal to `P`

-2 if the processor does not support a real type with an exponent range greater than or equal to `R`

-3 if neither is supported.

Example:

```
program real_kinds
  integer,parameter :: p6 = selected_real_kind(6)
  integer,parameter :: p10r100 = selected_real_kind(10,100)
  integer,parameter :: r400 = selected_real_kind(r=400)
  real(kind=p6) :: x
  real(kind=p10r100) :: y
  real(kind=r400) :: z

  print *, precision(x), range(x)
  print *, precision(y), range(y)
  print *, precision(z), range(z)
end program real_kinds
```

6.187 `SET_EXPONENT` — Set the exponent of the model

Description:
`SET_EXPONENT(X, I)` returns the real number whose fractional part is that that of *X* and whose exponent part is *I*.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = SET_EXPONENT(X, I)`

Arguments:

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

Return value:
The return value is of the same type and kind as *X*. The real number whose fractional part is that that of *X* and whose exponent part if *I* is returned; it is `FRACTION(X) * RADIX(X)**I`.

Example:

```
PROGRAM test_setexp
  REAL :: x = 178.1387e-4
  INTEGER :: i = 17
  PRINT *, SET_EXPONENT(x, i), FRACTION(x) * RADIX(x)**i
END PROGRAM
```

6.188 `SHAPE` — Determine the shape of an array

Description:
Determines the shape of an array.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = SHAPE(SOURCE)`

Arguments:

SOURCE	Shall be an array or scalar of any type. If *SOURCE* is a pointer it must be associated and allocatable arrays must be allocated.

Return value:

An `INTEGER` array of rank one with as many elements as *SOURCE* has dimensions. The elements of the resulting array correspond to the extend of *SOURCE* along the respective dimensions. If *SOURCE* is a scalar, the result is the rank one array of size zero.

Example:

```
PROGRAM test_shape
  INTEGER, DIMENSION(-1:1, -1:2) :: A
  WRITE(*,*) SHAPE(A)             ! (/ 3, 4 /)
  WRITE(*,*) SIZE(SHAPE(42))      ! (/ /)
END PROGRAM
```

See also: Section 6.178 [RESHAPE], page 137, Section 6.193 [SIZE], page 145

6.189 `SIGN` — Sign copying function

Description:

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

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = SIGN(A, B)`

Arguments:

A	Shall be of type `INTEGER` or `REAL`
B	Shall be of the same type and kind as *A*

Return value:

The kind of the return value is that of *A* and *B*. If $B \geq 0$ then the result is `ABS(A)`, else it is `-ABS(A)`.

Example:

```
program test_sign
  print *, sign(-12,1)
  print *, sign(-12,0)
  print *, sign(-12,-1)

  print *, sign(-12.,1.)
  print *, sign(-12.,0.)
  print *, sign(-12.,-1.)
end program test_sign
```

Specific names:

Name	Arguments	Return type	Standard
`ISIGN(A,P)`	`INTEGER(4)`	`INTEGER(4)`	f95, gnu
`DSIGN(A,P)`	`REAL(8)`	`REAL(8)`	f95, gnu

6.190 SIGNAL — Signal handling subroutine (or function)

Description:
SIGNAL(NUMBER, HANDLER [, STATUS]) causes external subroutine *HANDLER* to be executed with a single integer argument when signal *NUMBER* occurs. If *HANDLER* is an integer, it can be used to turn off handling of signal *NUMBER* or revert to its default action. See signal(2).

If SIGNAL is called as a subroutine and the *STATUS* argument is supplied, it is set to the value returned by signal(2).

Standard: GNU extension

Class: Subroutine, function

Syntax:

```
CALL SIGNAL(NUMBER, HANDLER [, STATUS])
STATUS = SIGNAL(NUMBER, HANDLER)
```

Arguments:

NUMBER	Shall be a scalar integer, with INTENT(IN)
HANDLER	Signal handler (INTEGER FUNCTION or SUBROUTINE) or dummy/global INTEGER scalar. INTEGER. It is INTENT(IN).
STATUS	(Optional) *STATUS* shall be a scalar integer. It has INTENT(OUT).

Return value:
The SIGNAL function returns the value returned by signal(2).

Example:

```
program test_signal
  intrinsic signal
  external handler_print

  call signal (12, handler_print)
  call signal (10, 1)

  call sleep (30)
end program test_signal
```

6.191 SIN — Sine function

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

Standard: F77 and later

Class: Elemental function

Syntax: RESULT = SIN(X)

Arguments:

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

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

Example:

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

Specific names:

Name	Argument	Return type	Standard
DSIN(X)	REAL(8) X	REAL(8)	f95, gnu
CSIN(X)	COMPLEX(4) X	COMPLEX(4)	f95, gnu
ZSIN(X)	COMPLEX(8) X	COMPLEX(8)	f95, gnu
CDSIN(X)	COMPLEX(8) X	COMPLEX(8)	f95, gnu

See also: Section 6.18 [ASIN], page 47

6.192 SINH — Hyperbolic sine function

Description:
SINH(X) computes the hyperbolic sine of *X*.

Standard: F95 and later

Class: Elemental function

Syntax: RESULT = SINH(X)

Arguments:

X	The type shall be REAL(*).

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

Example:

```
program test_sinh
  real(8) :: x = - 1.0_8
  x = sinh(x)
end program test_sinh
```

Specific names:

Name	Argument	Return type	Standard
DSINH(X)	REAL(8) X	REAL(8)	F95 and later

See also: Section 6.19 [ASINH], page 48

6.193 SIZE — Determine the size of an array

Description:
Determine the extent of *ARRAY* along a specified dimension *DIM*, or the total number of elements in *ARRAY* if *DIM* is absent.

Standard: F95 and later

Class: Inquiry function

Syntax: RESULT = SIZE(ARRAY[, DIM [, KIND]])

Arguments:

ARRAY	Shall be an array of any type. If *ARRAY* is a pointer it must be associated and allocatable arrays must be allocated.
DIM	(Optional) shall be a scalar of type `INTEGER` and its value shall be in the range from 1 to n, where n equals the rank of *ARRAY*.
KIND	(Optional) An `INTEGER` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind.

Example:

```
PROGRAM test_size
  WRITE(*,*) SIZE((/ 1, 2 /))    ! 2
END PROGRAM
```

See also: Section 6.188 [SHAPE], page 142, Section 6.178 [RESHAPE], page 137

6.194 `SIZEOF` — Size in bytes of an expression

Description:
`SIZEOF(X)` calculates the number of bytes of storage the expression `X` occupies.

Standard: GNU extension

Class: Intrinsic function

Syntax: `N = SIZEOF(X)`

Arguments:

X	The argument shall be of any type, rank or shape.

Return value:
The return value is of type integer and of the system-dependent kind *C_SIZE_T* (from the *ISO_C_BINDING* module). Its value is the number of bytes occupied by the argument. If the argument has the `POINTER` attribute, the number of bytes of the storage area pointed to is returned. If the argument is of a derived type with `POINTER` or `ALLOCATABLE` components, the return value doesn't account for the sizes of the data pointed to by these components.

Example:

```
integer :: i
real :: r, s(5)
print *, (sizeof(s)/sizeof(r) == 5)
end
```

The example will print `.TRUE.` unless you are using a platform where default `REAL` variables are unusually padded.

6.195 SLEEP — Sleep for the specified number of seconds

Description:
Calling this subroutine causes the process to pause for *SECONDS* seconds.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL SLEEP(SECONDS)`

Arguments:

SECONDS	The type shall be of default `INTEGER`.

Example:

```
program test_sleep
  call sleep(5)
end
```

6.196 SNGL — Convert double precision real to default real

Description:
`SNGL(A)` converts the double precision real *A* to a default real value. This is an archaic form of `REAL` that is specific to one type for *A*.

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = SNGL(A)`

Arguments:

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

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

See also: Section 6.52 [DBLE], page 68

6.197 SPACING — Smallest distance between two numbers of a given type

Description:
Determines the distance between the argument *X* and the nearest adjacent number of the same type.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = SPACING(X)`

Arguments:

X	Shall be of type `REAL(*)`.

Return value:
The result is of the same type as the input argument *X*.

Example:

```
PROGRAM test_spacing
  INTEGER, PARAMETER :: SGL = SELECTED_REAL_KIND(p=6, r=37)
  INTEGER, PARAMETER :: DBL = SELECTED_REAL_KIND(p=13, r=200)

  WRITE(*,*) spacing(1.0_SGL)      ! "1.1920929E-07"          on i686
  WRITE(*,*) spacing(1.0_DBL)      ! "2.220446049250313E-016" on i686
END PROGRAM
```

See also: Section 6.179 [RRSPACING], page 138

6.198 SPREAD — Add a dimension to an array

Description:
Replicates a *SOURCE* array *NCOPIES* times along a specified dimension *DIM*.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = SPREAD(SOURCE, DIM, NCOPIES)`

Arguments:

SOURCE	Shall be a scalar or an array of any type and a rank less than seven.
DIM	Shall be a scalar of type `INTEGER` with a value in the range from 1 to n+1, where n equals the rank of *SOURCE*.
NCOPIES	Shall be a scalar of type `INTEGER`.

Return value:
The result is an array of the same type as *SOURCE* and has rank n+1 where n equals the rank of *SOURCE*.

Example:

```
PROGRAM test_spread
  INTEGER :: a = 1, b(2) = (/ 1, 2 /)
  WRITE(*,*) SPREAD(A, 1, 2)            ! "1 1"
  WRITE(*,*) SPREAD(B, 1, 2)            ! "1 1 2 2"
END PROGRAM
```

See also: Section 6.218 [UNPACK], page 159

6.199 SQRT — Square-root function

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

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = SQRT(X)`

Arguments:

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

Return value:
The return value is of type `REAL(*)` or `COMPLEX(*)`. The kind type parameter is the same as *X*.

Example:

```
program test_sqrt
  real(8) :: x = 2.0_8
  complex :: z = (1.0, 2.0)
  x = sqrt(x)
  z = sqrt(z)
end program test_sqrt
```

Specific names:

Name	Argument	Return type	Standard
`DSQRT(X)`	`REAL(8) X`	`REAL(8)`	F95 and later
`CSQRT(X)`	`COMPLEX(4) X`	`COMPLEX(4)`	F95 and later
`ZSQRT(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	GNU extension
`CDSQRT(X)`	`COMPLEX(8) X`	`COMPLEX(8)`	GNU extension

6.200 `SRAND` — Reinitialize the random number generator

Description:
`SRAND` reinitializes the pseudo-random number generator called by `RAND` and `IRAND`. The new seed used by the generator is specified by the required argument *SEED*.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL SRAND(SEED)`

Arguments:

SEED	Shall be a scalar `INTEGER(kind=4)`.

Return value:
Does not return.

Example: See `RAND` and `IRAND` for examples.

Notes: The Fortran 2003 standard specifies the intrinsic `RANDOM_SEED` to initialize the pseudo-random numbers generator and `RANDOM_NUMBER` to generate pseudo-random numbers. Please note that in GNU Fortran, these two sets of intrinsics (`RAND`, `IRAND` and `SRAND` on the one hand, `RANDOM_NUMBER` and `RANDOM_SEED` on the other hand) access two independent pseudo-random number generators.

See also: Section 6.171 [RAND], page 133, Section 6.173 [RANDOM_SEED], page 134, Section 6.172 [RANDOM_NUMBER], page 133

6.201 `STAT` — Get file status

Description:
This function returns information about a file. No permissions are required on the file itself, but execute (search) permission is required on all of the directories in path that lead to the file.

The elements that are obtained and stored in the array `BUFF`:

`buff(1)`	Device ID
`buff(2)`	Inode number
`buff(3)`	File mode
`buff(4)`	Number of links
`buff(5)`	Owner's uid
`buff(6)`	Owner's gid
`buff(7)`	ID of device containing directory entry for file (0 if not available)
`buff(8)`	File size (bytes)
`buff(9)`	Last access time
`buff(10)`	Last modification time
`buff(11)`	Last file status change time
`buff(12)`	Preferred I/O block size (-1 if not available)
`buff(13)`	Number of blocks allocated (-1 if not available)

Not all these elements are relevant on all systems. If an element is not relevant, it is returned as 0.

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

Standard: GNU extension

Class: Subroutine, function

Syntax: `CALL STAT(FILE,BUFF[,STATUS])`

Arguments:

FILE	The type shall be `CHARACTER(*)`, a valid path within the file system.
BUFF	The type shall be `INTEGER(4), DIMENSION(13)`.
STATUS	(Optional) status flag of type `INTEGER(4)`. Returns 0 on success and a system specific error code otherwise.

Example:

```
PROGRAM test_stat
  INTEGER, DIMENSION(13) :: buff
  INTEGER :: status

  CALL STAT("/etc/passwd", buff, status)

  IF (status == 0) THEN
    WRITE (*, FMT="('Device ID:',               T30, I19)") buff(1)
    WRITE (*, FMT="('Inode number:',            T30, I19)") buff(2)
    WRITE (*, FMT="('File mode (octal):',       T30, O19)") buff(3)
    WRITE (*, FMT="('Number of links:',         T30, I19)") buff(4)
    WRITE (*, FMT="('Owner''s uid:',            T30, I19)") buff(5)
    WRITE (*, FMT="('Owner''s gid:',            T30, I19)") buff(6)
    WRITE (*, FMT="('Device where located:',    T30, I19)") buff(7)
    WRITE (*, FMT="('File size:',               T30, I19)") buff(8)
    WRITE (*, FMT="('Last access time:',        T30, A19)") CTIME(buff(9))
    WRITE (*, FMT="('Last modification time',   T30, A19)") CTIME(buff(10))
    WRITE (*, FMT="('Last status change time:', T30, A19)") CTIME(buff(11))
```

```
      WRITE (*, FMT="('Preferred block size:',    T30, I19)") buff(12)
      WRITE (*, FMT="('No. of blocks allocated:', T30, I19)") buff(13)
    END IF
  END PROGRAM
```

See also: To stat an open file: Section 6.81 [FSTAT], page 86, to stat a link: Section 6.139 [LSTAT], page 115

6.202 SUM — Sum of array elements

Description:
Adds the elements of *ARRAY* along dimension *DIM* if the corresponding element in *MASK* is TRUE.

Standard: F95 and later

Class: Transformational function

Syntax: RESULT = SUM(ARRAY[, MASK]) RESULT = SUM(ARRAY, DIM[, MASK])

Arguments:

ARRAY	Shall be an array of type INTEGER(*), REAL(*) or COMPLEX(*).
DIM	(Optional) shall be a scalar of type INTEGER with a value in the range from 1 to n, where n equals the rank of *ARRAY*.
MASK	(Optional) shall be of type LOGICAL and either be a scalar or an array of the same shape as *ARRAY*.

Return value:
The result is of the same type as *ARRAY*.

If *DIM* is absent, a scalar with the sum of all elements in *ARRAY* is returned. Otherwise, an array of rank n-1, where n equals the rank of *ARRAY*,and a shape similar to that of *ARRAY* with dimension *DIM* dropped is returned.

Example:

```
PROGRAM test_sum
  INTEGER :: x(5) = (/ 1, 2, 3, 4 ,5 /)
  print *, SUM(x)                        ! all elements, sum = 15
  print *, SUM(x, MASK=MOD(x, 2)==1)     ! odd elements, sum = 9
END PROGRAM
```

See also: Section 6.168 [PRODUCT], page 131

6.203 SYMLNK — Create a symbolic link

Description:
Makes a symbolic link from file *PATH1* to *PATH2*. A null character (CHAR(0)) can be used to mark the end of the names in *PATH1* and *PATH2*; otherwise, trailing blanks in the file names are ignored. If the *STATUS* argument is supplied, it contains 0 on success or a nonzero error code upon return; see symlink(2). If the system does not supply symlink(2), ENOSYS is returned.

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

```
CALL SYMLNK(PATH1, PATH2 [, STATUS])
STATUS = SYMLNK(PATH1, PATH2)
```

Arguments:

PATH1	Shall be of default `CHARACTER` type.
PATH2	Shall be of default `CHARACTER` type.
STATUS	(Optional) Shall be of default `INTEGER` type.

See also: Section 6.129 [LINK], page 110, Section 6.217 [UNLINK], page 158

6.204 `SYSTEM` — Execute a shell command

Description:

Passes the command *COMMAND* to a shell (see `system(3)`). If argument *STATUS* is present, it contains the value returned by `system(3)`, which is presumably 0 if the shell command succeeded. Note that which shell is used to invoke the command is system-dependent and environment-dependent.

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

```
CALL SYSTEM(COMMAND [, STATUS])
STATUS = SYSTEM(COMMAND)
```

Arguments:

COMMAND	Shall be of default `CHARACTER` type.
STATUS	(Optional) Shall be of default `INTEGER` type.

See also:

6.205 `SYSTEM_CLOCK` — Time function

Description:

Determines the *COUNT* of milliseconds of wall clock time since the Epoch (00:00:00 UTC, January 1, 1970) modulo *COUNT_MAX*, *COUNT_RATE* determines the number of clock ticks per second. *COUNT_RATE* and *COUNT_MAX* are constant and specific to `gfortran`.

If there is no clock, *COUNT* is set to `-HUGE(COUNT)`, and *COUNT_RATE* and *COUNT_MAX* are set to zero

Standard: F95 and later

Class: Subroutine

Syntax: `CALL SYSTEM_CLOCK([COUNT, COUNT_RATE, COUNT_MAX])`

Arguments:
Arguments:

COUNT	(Optional) shall be a scalar of type default `INTEGER` with `INTENT(OUT)`.
COUNT_RATE	(Optional) shall be a scalar of type default `INTEGER` with `INTENT(OUT)`.
COUNT_MAX	(Optional) shall be a scalar of type default `INTEGER` with `INTENT(OUT)`.

Example:

```
PROGRAM test_system_clock
  INTEGER :: count, count_rate, count_max
  CALL SYSTEM_CLOCK(count, count_rate, count_max)
  WRITE(*,*) count, count_rate, count_max
END PROGRAM
```

See also: Section 6.51 [DATE_AND_TIME], page 67, Section 6.48 [CPU_TIME], page 65

6.206 `TAN` — Tangent function

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

Standard: F77 and later

Class: Elemental function

Syntax: `RESULT = TAN(X)`

Arguments:

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

Return value:
The return value is of type `REAL(*)`. The kind type parameter is the same as *X*.

Example:

```
program test_tan
  real(8) :: x = 0.165_8
  x = tan(x)
end program test_tan
```

Specific names:

Name	Argument	Return type	Standard
`DTAN(X)`	`REAL(8) X`	`REAL(8)`	F95 and later

See also: Section 6.21 [ATAN], page 49

6.207 `TANH` — Hyperbolic tangent function

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

Standard: F77 and later

Class: Elemental function

Syntax: `X = TANH(X)`

Arguments:

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

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

Example:

```
program test_tanh
  real(8) :: x = 2.1_8
  x = tanh(x)
end program test_tanh
```

Specific names:

Name	Argument	Return type	Standard
`DTANH(X)`	`REAL(8) X`	`REAL(8)`	F95 and later

See also: Section 6.23 [ATANH], page 50

6.208 `TIME` — Time function

Description:
Returns the current time encoded as an integer (in the manner of the UNIX function `time(3)`). This value is suitable for passing to `CTIME()`, `GMTIME()`, and `LTIME()`.

This intrinsic is not fully portable, such as to systems with 32-bit `INTEGER` types but supporting times wider than 32 bits. Therefore, the values returned by this intrinsic might be, or become, negative, or numerically less than previous values, during a single run of the compiled program.

See Section 6.209 [TIME8], page 155, for information on a similar intrinsic that might be portable to more GNU Fortran implementations, though to fewer Fortran compilers.

Standard: GNU extension

Class: Function

Syntax: `RESULT = TIME()`

Return value:
The return value is a scalar of type `INTEGER(4)`.

See also: Section 6.50 [CTIME], page 66, Section 6.95 [GMTIME], page 93, Section 6.140 [LTIME], page 116, Section 6.147 [MCLOCK], page 120, Section 6.209 [TIME8], page 155

6.209 TIME8 — Time function (64-bit)

Description:
Returns the current time encoded as an integer (in the manner of the UNIX function `time(3)`). This value is suitable for passing to `CTIME()`, `GMTIME()`, and `LTIME()`.

Warning: this intrinsic does not increase the range of the timing values over that returned by `time(3)`. On a system with a 32-bit `time(3)`, `TIME8()` will return a 32-bit value, even though it is converted to a 64-bit `INTEGER(8)` value. That means overflows of the 32-bit value can still occur. Therefore, the values returned by this intrinsic might be or become negative or numerically less than previous values during a single run of the compiled program.

Standard: GNU extension

Class: Function

Syntax: `RESULT = TIME8()`

Return value:
The return value is a scalar of type `INTEGER(8)`.

See also: Section 6.50 [CTIME], page 66, Section 6.95 [GMTIME], page 93, Section 6.140 [LTIME], page 116, Section 6.148 [MCLOCK8], page 120, Section 6.208 [TIME], page 154

6.210 TINY — Smallest positive number of a real kind

Description:
`TINY(X)` returns the smallest positive (non zero) number in the model of the type of `X`.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = TINY(X)`

Arguments:

X	Shall be of type `REAL`.

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

Example: See `HUGE` for an example.

6.211 TRANSFER — Transfer bit patterns

Description:
Interprets the bitwise representation of *SOURCE* in memory as if it is the representation of a variable or array of the same type and type parameters as *MOLD*.

This is approximately equivalent to the C concept of *casting* one type to another.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = TRANSFER(SOURCE, MOLD[, SIZE])`

Arguments:

SOURCE	Shall be a scalar or an array of any type.
MOLD	Shall be a scalar or an array of any type.
SIZE	(Optional) shall be a scalar of type `INTEGER`.

Return value:

The result has the same type as *MOLD*, with the bit level representation of *SOURCE*. If *SIZE* is present, the result is a one-dimensional array of length *SIZE*. If *SIZE* is absent but *MOLD* is an array (of any size or shape), the result is a one- dimensional array of the minimum length needed to contain the entirety of the bitwise representation of *SOURCE*. If *SIZE* is absent and *MOLD* is a scalar, the result is a scalar.

If the bitwise representation of the result is longer than that of *SOURCE*, then the leading bits of the result correspond to those of *SOURCE* and any trailing bits are filled arbitrarily.

When the resulting bit representation does not correspond to a valid representation of a variable of the same type as *MOLD*, the results are undefined, and subsequent operations on the result cannot be guaranteed to produce sensible behavior. For example, it is possible to create `LOGICAL` variables for which *`VAR`* and `.NOT.`*`VAR`* both appear to be true.

Example:

```
PROGRAM test_transfer
  integer :: x = 2143289344
  print *, transfer(x, 1.0)    ! prints "NaN" on i686
END PROGRAM
```

6.212 `TRANSPOSE` — Transpose an array of rank two

Description:

Transpose an array of rank two. Element (i, j) of the result has the value `MATRIX(j, i)`, for all i, j.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = TRANSPOSE(MATRIX)`

Arguments:

MATRIX	Shall be an array of any type and have a rank of two.

Return value:

The result has the the same type as *MATRIX*, and has shape `(/ m, n /)` if *MATRIX* has shape `(/ n, m /)`.

6.213 TRIM — Remove trailing blank characters of a string

Description:
Removes trailing blank characters of a string.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = TRIM(STRING)`

Arguments:

STRING	Shall be a scalar of type `CHARACTER(*)`.

Return value:
A scalar of type `CHARACTER(*)` which length is that of *STRING* less the number of trailing blanks.

Example:

```
PROGRAM test_trim
  CHARACTER(len=10), PARAMETER :: s = "GFORTRAN  "
  WRITE(*,*) LEN(s), LEN(TRIM(s))  ! "10 8", with/without trailing blanks
END PROGRAM
```

See also: Section 6.8 [ADJUSTL], page 41, Section 6.9 [ADJUSTR], page 41

6.214 TTYNAM — Get the name of a terminal device.

Description:
Get the name of a terminal device. For more information, see `ttyname(3)`.

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

```
CALL TTYNAM(UNIT, NAME)
NAME = TTYNAM(UNIT)
```

Arguments:

UNIT	Shall be a scalar `INTEGER(*)`.
NAME	Shall be of type `CHARACTER(*)`.

Example:

```
PROGRAM test_ttynam
  INTEGER :: unit
  DO unit = 1, 10
    IF (isatty(unit=unit)) write(*,*) ttynam(unit)
  END DO
END PROGRAM
```

See also: Section 6.116 [ISATTY], page 104

6.215 UBOUND — Upper dimension bounds of an array

Description:
Returns the upper bounds of an array, or a single upper bound along the *DIM* dimension.

Standard: F95 and later

Class: Inquiry function

Syntax: `RESULT = UBOUND(ARRAY [, DIM [, KIND]])`

Arguments:

ARRAY	Shall be an array, of any type.
DIM	(Optional) Shall be a scalar `INTEGER(*)`.
KIND	(Optional) An `INTEGER` initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind. If *DIM* is absent, the result is an array of the upper bounds of *ARRAY*. If *DIM* is present, the result is a scalar corresponding to the upper bound of the array along that dimension. If *ARRAY* is an expression rather than a whole array or array structure component, or if it has a zero extent along the relevant dimension, the upper bound is taken to be the number of elements along the relevant dimension.

See also: Section 6.123 [LBOUND], page 107

6.216 UMASK — Set the file creation mask

Description:
Sets the file creation mask to *MASK* and returns the old value in argument *OLD* if it is supplied. See `umask(2)`.

Standard: GNU extension

Class: Subroutine

Syntax: `CALL UMASK(MASK [, OLD])`

Arguments:

MASK	Shall be a scalar of type `INTEGER(*)`.
MASK	(Optional) Shall be a scalar of type `INTEGER(*)`.

6.217 UNLINK — Remove a file from the file system

Description:
Unlinks the file *PATH*. A null character (`CHAR(0)`) can be used to mark the end of the name in *PATH*; otherwise, trailing blanks in the file name are ignored. If the *STATUS* argument is supplied, it contains 0 on success or a nonzero error code upon return; see `unlink(2)`.

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

Standard: GNU extension

Class: Subroutine, function

Syntax:

```
CALL UNLINK(PATH [, STATUS])
STATUS = UNLINK(PATH)
```

Arguments:

PATH	Shall be of default `CHARACTER` type.
STATUS	(Optional) Shall be of default `INTEGER` type.

See also: Section 6.129 [LINK], page 110, Section 6.203 [SYMLNK], page 151

6.218 `UNPACK` — Unpack an array of rank one into an array

Description:
Store the elements of *VECTOR* in an array of higher rank.

Standard: F95 and later

Class: Transformational function

Syntax: `RESULT = UNPACK(VECTOR, MASK, FIELD)`

Arguments:

VECTOR	Shall be an array of any type and rank one. It shall have at least as many elements as *MASK* has `TRUE` values.
MASK	Shall be an array of type `LOGICAL`.
FIELD	Shall be of the sam type as *VECTOR* and have the same shape as *MASK*.

Return value:
The resulting array corresponds to *FIELD* with `TRUE` elements of *MASK* replaced by values from *VECTOR* in array element order.

Example:

```
PROGRAM test_unpack
  integer :: vector(2)  = (/1,1/)
  logical :: mask(4)  = (/ .TRUE., .FALSE., .FALSE., .TRUE. /)
  integer :: field(2,2) = 0, unity(2,2)

  ! result: unity matrix
  unity = unpack(vector, reshape(mask, (/2,2/)), field)
END PROGRAM
```

See also: Section 6.164 [PACK], page 129, Section 6.198 [SPREAD], page 148

6.219 `VERIFY` — Scan a string for the absence of a set of characters

Description:
Verifies that all the characters in a *SET* are present in a *STRING*.

If *BACK* is either absent or equals `FALSE`, this function returns the position of the leftmost character of *STRING* that is not in *SET*. If *BACK* equals `TRUE`, the rightmost position is returned. If all characters of *SET* are found in *STRING*, the result is zero.

Standard: F95 and later

Class: Elemental function

Syntax: `RESULT = VERIFY(STRING, SET[, BACK [, KIND]])`

Arguments:

STRING	Shall be of type `CHARACTER(*)`.
SET	Shall be of type `CHARACTER(*)`.
BACK	(Optional) shall be of type `LOGICAL`.
KIND	(Optional) An `INTEGER` initialization expression indicating the kind parameter of the result.

Return value:

The return value is of type `INTEGER` and of kind *KIND*. If *KIND* is absent, the return value is of default integer kind.

Example:

```
PROGRAM test_verify
  WRITE(*,*) VERIFY("FORTRAN", "AO")           ! 1, found 'F'
  WRITE(*,*) VERIFY("FORTRAN", "FOO")          ! 3, found 'R'
  WRITE(*,*) VERIFY("FORTRAN", "C++")          ! 1, found 'F'
  WRITE(*,*) VERIFY("FORTRAN", "C++", .TRUE.)  ! 7, found 'N'
  WRITE(*,*) VERIFY("FORTRAN", "FORTRAN")      ! 0' found none
END PROGRAM
```

See also: Section 6.182 [SCAN], page 139, Section 6.108 [INDEX intrinsic], page 100

6.220 `XOR` — Bitwise logical exclusive OR

Description:

Bitwise logical exclusive or.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. For integer arguments, programmers should consider the use of the Section 6.106 [IEOR], page 99 intrinsic defined by the Fortran standard.

Standard: GNU extension

Class: Function

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

Arguments:

X	The type shall be either `INTEGER(*)` or `LOGICAL`.
Y	The type shall be either `INTEGER(*)` or `LOGICAL`.

Return value:

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

Example:

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

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

See also: F95 elemental function: Section 6.106 [IEOR], page 99

7 Intrinsic Modules

7.1 ISO_FORTRAN_ENV

Standard: Fortran 2003

The ISO_FORTRAN_ENV module provides the following scalar default-integer named constants:

CHARACTER_STORAGE_SIZE:
: Size in bits of the character storage unit.

ERROR_UNIT:
: Indentifies the preconnected unit used for error reporting.

FILE_STORAGE_SIZE:
: Size in bits of the file-storage unit.

INPUT_UNIT:
: Indentifies the preconnected unit indentified by the asterisk (*) in READ statement.

IOSTAT_END:
: The value assigned to the variable passed to the IOSTAT= specifier of an input/output statement if an end-of-file condition occurred.

IOSTAT_EOR:
: The value assigned to the variable passed to the IOSTAT= specifier of an input/output statement if an end-of-record condition occurred.

NUMERIC_STORAGE_SIZE:
: The size in bits of the numeric storage unit.

OUTPUT_UNIT:
: Indentifies the preconnected unit indentified by the asterisk (*) in WRITE statement.

7.2 ISO_C_BINDING

Standard: Fortran 2003

The following intrinsic procedures are provided by the module; their definition can be found in the section Intrinsic Procedures of this manual.

C_ASSOCIATED
C_F_POINTER
C_F_PROCPOINTER
C_FUNLOC

C_LOC

The ISO_C_BINDING module provides the following named constants of the type integer, which can be used as KIND type parameter. Note that GNU Fortran currently does not support the C_INT_FAST... KIND type parameters (marked by an asterix (*) in the list below). The C_INT_FAST... parameters have therefore the value −2 and cannot be used as KIND type parameter of the INTEGER type.

Fortran Type	Named constant	C type
INTEGER	C_INT	int
INTEGER	C_SHORT	short int
INTEGER	C_LONG	long int
INTEGER	C_LONG_LONG	long long int
INTEGER	C_SIGNED_CHAR	signed char/unsigned char
INTEGER	C_SIZE_T	size_t
INTEGER	C_INT8_T	int8_t
INTEGER	C_INT16_T	int16_t
INTEGER	C_INT32_T	int32_t
INTEGER	C_INT64_T	int64_t
INTEGER	C_INT_LEAST8_T	int_least8_t
INTEGER	C_INT_LEAST16_T	int_least16_t
INTEGER	C_INT_LEAST32_T	int_least32_t
INTEGER	C_INT_LEAST64_T	int_least64_t
INTEGER	C_INT_FAST8_T*	int_fast8_t
INTEGER	C_INT_FAST16_T*	int_fast16_t
INTEGER	C_INT_FAST32_T*	int_fast32_t
INTEGER	C_INT_FAST64_T*	int_fast64_t
INTEGER	C_INTMAX_T	intmax_t
INTEGER	C_INTPTR_T	intptr_t
REAL	C_FLOAT	float
REAL	C_DOUBLE	double
REAL	C_LONG_DOUBLE	long double
COMPLEX	C_FLOAT_COMPLEX	float _Complex
COMPLEX	C_DOUBLE_COMPLEX	double _Complex
COMPLEX	C_LONG_DOUBLE_COMPLEX	long double _Complex
LOGICAL	C_BOOL	_Bool
CHARACTER	C_CHAR	char

Additionally, the following (CHARACTER(KIND=C_CHAR) are defined.

Name	C definition	Value
C_NULL_CHAR	null character	'\0'
C_ALERT	alert	'\a'
C_BACKSPACE	backspace	'\b'
C_FORM_FEED	form feed	'\f'
C_NEW_LINE	new line	'\n'
C_CARRIAGE_RETURN	carriage return	'\r'
C_HORIZONTAL_TAB	horizontal tab	'\t'
C_VERTICAL_TAB	vertical tab	'\v'

7.3 OpenMP Modules OMP_LIB and OMP_LIB_KINDS

Standard: OpenMP Application Program Interface v2.5

The OpenMP Fortran runtime library routines are provided both in a form of two Fortran 90 modules, named OMP_LIB and OMP_LIB_KINDS, and in a form of a Fortran include file named 'omp_lib.h'. The procedures provided by OMP_LIB can be found in the Section "Introduction" in *GNU OpenMP runtime library* manual, the named constants defined in the OMP_LIB_KINDS module are listed below.

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

OMP_LIB_KINDS provides the following scalar default-integer named constants:

```
omp_integer_kind
omp_logical_kind
omp_lock_kind
omp_nest_lock_kind
```

Contributing

Free software is only possible if people contribute to efforts to create it. We're always in need of more people helping out with ideas and comments, writing documentation and contributing code.

If you want to contribute to GNU Fortran, have a look at the long lists of projects you can take on. Some of these projects are small, some of them are large; some are completely orthogonal to the rest of what is happening on GNU Fortran, but others are "mainstream" projects in need of enthusiastic hackers. All of these projects are important! We'll eventually get around to the things here, but they are also things doable by someone who is willing and able.

Contributors to GNU Fortran

Most of the parser was hand-crafted by *Andy Vaught*, who is also the initiator of the whole project. Thanks Andy! Most of the interface with GCC was written by *Paul Brook*.

The following individuals have contributed code and/or ideas and significant help to the GNU Fortran project (in alphabetical order):

- Janne Blomqvist
- Steven Bosscher
- Paul Brook
- Tobias Burnus
- François-Xavier Coudert
- Bud Davis
- Jerry DeLisle
- Erik Edelmann
- Bernhard Fischer
- Daniel Franke
- Richard Guenther
- Richard Henderson
- Katherine Holcomb
- Jakub Jelinek
- Niels Kristian Bech Jensen
- Steven Johnson
- Steven G. Kargl
- Thomas Koenig
- Asher Langton
- H. J. Lu
- Toon Moene
- Brooks Moses
- Andrew Pinski
- Tim Prince

- Christopher D. Rickett
- Richard Sandiford
- Tobias Schlüter
- Roger Sayle
- Paul Thomas
- Andy Vaught
- Feng Wang
- Janus Weil

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

- Bill Clodius
- Dominique d'Humières
- Kate Hedstrom
- Erik Schnetter

Many other individuals have helped debug, test and improve the GNU Fortran compiler over the past few years, and we welcome you to do the same! If you already have done so, and you would like to see your name listed in the list above, please contact us.

Projects

Help build the test suite
: Solicit more code for donation to the test suite: the more extensive the testsuite, the smaller the risk of breaking things in the future! 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. Going through the bugzilla database at http://gcc.gnu.org/bugzilla/ to reduce testcases posted there and add more information (for example, for which version does the testcase work, for which versions does it fail?) is also very helpful.

Proposed Extensions

Here's a list of proposed extensions for the GNU Fortran compiler, in no particular order. Most of these are necessary to be fully compatible with existing Fortran compilers, but they are not part of the official J3 Fortran 95 standard.

Compiler extensions:

- User-specified alignment rules for structures.
- Flag to generate `Makefile` info.
- Automatically extend single precision constants to double.

- Compile code that conserves memory by dynamically allocating common and module storage either on stack or heap.
- Compile flag to generate code for array conformance checking (suggest -CC).
- User control of symbol names (underscores, etc).
- Compile setting for maximum size of stack frame size before spilling parts to static or heap.
- Flag to force local variables into static space.
- Flag to force local variables onto stack.

Environment Options

- Pluggable library modules for random numbers, linear algebra. LA should use BLAS calling conventions.
- Environment variables controlling actions on arithmetic exceptions like overflow, underflow, precision loss—Generate NaN, abort, default. action.
- Set precision for fp units that support it (i387).
- Variable for setting fp rounding mode.
- Variable to fill uninitialized variables with a user-defined bit pattern.
- Environment variable controlling filename that is opened for that unit number.
- Environment variable to clear/trash memory being freed.
- Environment variable to control tracing of allocations and frees.
- Environment variable to display allocated memory at normal program end.
- Environment variable for filename for * IO-unit.
- Environment variable for temporary file directory.
- Environment variable forcing standard output to be line buffered (unix).

GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. 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 them 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 prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. 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.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey 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; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
   b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
   c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or
e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others’ Freedom.

    If 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 convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU 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 that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    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.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS 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.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

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 state 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 3 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, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program 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, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU 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 Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

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.

Option Index

`gfortran`'s command line options are indexed here without any initial '-' or '--'. Where an option has both positive and negative forms (such as -foption and -fno-option), relevant entries in the manual are indexed under the most appropriate form; it may sometimes be useful to look up both forms.

B

backslash 8

F

fall-intrinsics 8
fbacktrace 13
fblas-matmul-limit 17
fbounds-check 16
fconvert=*conversion* 14
fcray-pointer 9
fd-lines-as-code 8
fd-lines-as-comments 8
fdefault-double-8 8
fdefault-integer-8 8
fdefault-real-8 8
fdollar-ok 8
fdump-core 13
fdump-parse-tree 12
fexternal-blas 17
ff2c 14
ffixed-line-length-*n* 9
ffpe-trap=*list* 12
ffree-form 8
ffree-line-length-*n* 9
fimplicit-none 9
finit-character 17
finit-integer 17
finit-local-zero 17
finit-logical 17
finit-real 17
fintrinsic-modules-path *dir* 13
fmax-errors=*n* 10
fmax-identifier-length=*n* 9
fmax-stack-var-size 16
fmax-subrecord-length=*length* 14
fmodule-private 9
fno-automatic 14
fno-fixed-form 8
fno-underscoring 15
fopenmp 9
fpack-derived 17
frange-check 9
frecord-marker=*length* 14
frecursive 17
frepack-arrays 17
fsecond-underscore 16
fshort-enums 17, 25
fsign-zero 14
fsyntax-only 10

I

I*dir* 13

J

J*dir* 13

M

M*dir* 13

P

pedantic 10
pedantic-errors 11

S

static-libgfortran 13
std=*std* option 10

W

Waliasing 11
Wall 11
Wampersand 11
Wcharacter-truncation 11
Wconversion 11
Werror 12
Wimplicit-interface 11
Wnonstd-intrinsics 11
Wsurprising 11
Wtabs 12
Wunderflow 12
Wunused-parameter 12

Keyword Index

$

$ . . . 8

%

%LOC . . . 34
%REF . . . 34
%VAL . . . 34

&

& . . . 11

[

[...] . . . 25

A

ABORT . . . 37
ABS . . . 38
absolute value . . . 38
ACCESS . . . 38
ACCESS='STREAM' I/O . . . 25
ACHAR . . . 39
ACOS . . . 40
ACOSH . . . 40
adjust string . . . 41
ADJUSTL . . . 41
ADJUSTR . . . 41
AIMAG . . . 42
AINT . . . 42
ALARM . . . 43
ALGAMA . . . 108
aliasing . . . 11
ALL . . . 44
all warnings . . . 11
ALLOCATABLE components of derived types . . . 25
ALLOCATABLE dummy arguments . . . 25
ALLOCATABLE function results . . . 25
ALLOCATED . . . 45
allocation, moving . . . 125
allocation, status . . . 45
ALOG . . . 113
ALOG10 . . . 113
AMAX0 . . . 118
AMAX1 . . . 118
AMIN0 . . . 121
AMIN1 . . . 121
AMOD . . . 124
AND . . . 45
ANINT . . . 46
ANY . . . 46
area hyperbolic cosine . . . 40
area hyperbolic sine . . . 48
area hyperbolic tangent . . . 50
argument list functions . . . 34
arguments, to program . . . 62, 88, 89, 96
array, add elements . . . 151
array, apply condition . . . 44, 46
array, bounds checking . . . 16
array, change dimensions . . . 137
array, combine arrays . . . 121
array, condition testing . . . 44, 46
array, conditionally add elements . . . 151
array, conditionally count elements . . . 64
array, conditionally multiply elements . . . 131
array, constructors . . . 25
array, count elements . . . 145
array, duplicate dimensions . . . 148
array, duplicate elements . . . 148
array, element counting . . . 64
array, gather elements . . . 129
array, increase dimension . . . 148, 159
array, indices of type real . . . 30
array, location of maximum element . . . 119
array, location of minimum element . . . 122
array, lower bound . . . 107
array, maximum value . . . 119
array, merge arrays . . . 121
array, minimum value . . . 123
array, multiply elements . . . 131
array, number of elements . . . 64, 145
array, packing . . . 129
array, permutation . . . 66
array, product . . . 131
array, reduce dimension . . . 129
array, rotate . . . 66
array, scatter elements . . . 159
array, shape . . . 142
array, shift . . . 73
array, shift circularly . . . 66
array, size . . . 145
array, sum . . . 151
array, transmogrify . . . 137
array, transpose . . . 156
array, unpacking . . . 159
array, upper bound . . . 158
ASCII collating sequence . . . 39, 94
ASIN . . . 47
ASINH . . . 48, 50
ASSOCIATED . . . 48
association status . . . 48
association status, C pointer . . . 55
ATAN . . . 49
ATAN2 . . . 50
Authors . . . 167

B

backslash ... 8
backtrace ... 13
BESJ0 ... 51
BESJ1 ... 52
BESJN ... 52
Bessel function, first kind ... 51, 52
Bessel function, second kind ... 53, 54
BESY0 ... 53
BESY1 ... 53
BESYN ... 54
BIT_SIZE ... 54
bits, clear ... 96
bits, extract ... 97
bits, get ... 97
bits, move ... 126, 155
bits, negate ... 128
bits, number of ... 54
bits, set ... 97
bits, shift ... 104
bits, shift circular ... 105
bits, shift left ... 115
bits, shift right ... 138
bits, testing ... 55
bits, unset ... 96
bitwise logical and ... 45, 95
bitwise logical exclusive or ... 99, 160
bitwise logical not ... 128
bitwise logical or ... 102, 129
bounds checking ... 16
BOZ literal constants ... 29
BTEST ... 55

C

C_ASSOCIATED ... 55
C_F_POINTER ... 57
C_F_PROCPOINTER ... 56
C_FUNLOC ... 56
C_LOC ... 58
CABS ... 38
calling convention ... 14
CCOS ... 63
CDABS ... 38
CDCOS ... 63
CDEXP ... 77
CDLOG ... 113
CDSIN ... 144
CDSQRT ... 148
ceiling ... 46, 58
CEILING ... 58
CEXP ... 77
CHAR ... 59
character set ... 8
CHDIR ... 59
checking subscripts ... 16
CHMOD ... 60
clock ticks ... 120, 152
CLOG ... 113
CMPLX ... 61
code generation, conventions ... 14
collating sequence, ASCII ... 39, 94
command options ... 7
command-line arguments ... 62, 88, 89, 96
command-line arguments, number of ... 62, 96
COMMAND_ARGUMENT_COUNT ... 62
COMPLEX ... 62
complex conjugate ... 63
complex numbers, conversion to ... 61, 62, 69
complex numbers, imaginary part ... 42
complex numbers, real part ... 72, 135
Conditional compilation ... 2
CONJG ... 63
Contributing ... 167
Contributors ... 167
conversion ... 11
conversion, to character ... 59
conversion, to complex ... 61, 62, 69
conversion, to integer ... 30, 94, 97, 100, 101, 114
conversion, to logical ... 30, 114
conversion, to real ... 68, 69, 79, 135, 147
conversion, to string ... 66
CONVERT specifier ... 33
core, dump ... 13, 37
COS ... 63
COSH ... 64
cosine ... 63
cosine, hyperbolic ... 64
cosine, hyperbolic, inverse ... 40
cosine, inverse ... 40
COUNT ... 64
CPP ... 2
CPU_TIME ... 65
Credits ... 167
CSHIFT ... 66
CSIN ... 144
CSQRT ... 148
CTIME ... 66
current date ... 67, 78, 98
current time ... 67, 78, 106, 154, 155

D

DABS ... 38
DACOS ... 40
DACOSH ... 40
DASIN ... 47
DASINH ... 48, 50
DATAN ... 49
DATAN2 ... 50
date, current ... 67, 78, 98
DATE_AND_TIME ... 67
DBESJ0 ... 51
DBESJ1 ... 52
DBESJN ... 52
DBESY0 ... 53

DBESY1 . . . 53
DBESYN . . . 54
DBLE . . . 68
DCMPLX . . . 69
DCONJG . . . 63
DCOS . . . 63
DCOSH . . . 64
DDIM . . . 70
debugging information options . . . 12
DECODE . . . 36
delayed execution . . . 43, 147
DEXP . . . 77
DFLOAT . . . 69
DGAMMA . . . 87
dialect options . . . 8
DIGITS . . . 70
DIM . . . 70
DIMAG . . . 42
DINT . . . 42
directive, INCLUDE . . . 13
directory, options . . . 13
directory, search paths for inclusion . . . 13
division, modulo . . . 124
division, remainder . . . 124
DLGAMA . . . 108
DLOG . . . 113
DLOG10 . . . 113
DMAX1 . . . 118
DMIN1 . . . 121
DMOD . . . 124
DNINT . . . 46
dot product . . . 71
DOT_PRODUCT . . . 71
DPROD . . . 71
DREAL . . . 72
DSIGN . . . 143
DSIN . . . 144
DSINH . . . 145
DSQRT . . . 148
DTAN . . . 153
DTANH . . . 153
DTIME . . . 72

E

elapsed time . . . 72, 140
ENCODE . . . 36
ENUM statement . . . 25
ENUMERATOR statement . . . 25
environment variable . . . 18, 19, 90, 91
EOSHIFT . . . 73
EPSILON . . . 74
ERF . . . 75
ERFC . . . 75
error function . . . 75
error function, complementary . . . 75
errors, limiting . . . 10
escape characters . . . 8
ETIME . . . 76
EXIT . . . 77
EXP . . . 77
EXPONENT . . . 78
exponential function . . . 77
exponential function, inverse . . . 113
expression size . . . 146
extensions . . . 27
extensions, implemented . . . 27
extensions, not implemented . . . 34

F

f2c calling convention . . . 14, 16
Factorial function . . . 87
FDATE . . . 78
FDL, GNU Free Documentation License . . . 183
FGET . . . 79
FGETC . . . 80
file format, fixed . . . 8, 9
file format, free . . . 8, 9
file operation, file number . . . 82
file operation, flush . . . 81
file operation, position . . . 85, 86
file operation, read character . . . 79, 80
file operation, seek . . . 85
file operation, write character . . . 82, 83
file system, access mode . . . 38
file system, change access mode . . . 60
file system, create link . . . 110, 151
file system, file creation mask . . . 158
file system, file status . . . 86, 115, 149
file system, hard link . . . 110
file system, remove file . . . 158
file system, rename file . . . 136
file system, soft link . . . 151
FLOAT . . . 79
floating point, exponent . . . 78
floating point, fraction . . . 84
floating point, nearest different . . . 126
floating point, relative spacing . . . 138, 147
floating point, scale . . . 138
floating point, set exponent . . . 142
floor . . . 42, 81
FLOOR . . . 81
FLUSH . . . 81
FLUSH statement . . . 25
FNUM . . . 82
Fortran 77 . . . 3
FPP . . . 2
FPUT . . . 82
FPUTC . . . 83
FRACTION . . . 84
FREE . . . 84
FSEEK . . . 85
FSTAT . . . 86
FTELL . . . 86

G

g77 ... 3
g77 calling convention ... 14, 16
GAMMA ... 87, 108
Gamma function ... 87
Gamma function, logarithm of ... 108
GCC ... 2
GERROR ... 87
GET_COMMAND ... 89
GET_COMMAND_ARGUMENT ... 89
GET_ENVIRONMENT_VARIABLE ... 91
GETARG ... 88
GETCWD ... 90
GETENV ... 90
GETGID ... 91
GETLOG ... 92
GETPID ... 92
GETUID ... 93
GMTIME ... 93
GNU Compiler Collection ... 2
GNU Fortran command options ... 7

H

Hollerith constants ... 30
HOSTNM ... 94
HUGE ... 94
hyperbolic arccosine ... 40
hyperbolic arcsine ... 48
hyperbolic arctangent ... 50
hyperbolic cosine ... 64
hyperbolic function, cosine ... 64
hyperbolic function, cosine, inverse ... 40
hyperbolic function, sine ... 145
hyperbolic function, sine, inverse ... 48
hyperbolic function, tangent ... 153
hyperbolic function, tangent, inverse ... 50
hyperbolic sine ... 145
hyperbolic tangent ... 153

I

I/O item lists ... 29
IABS ... 38
IACHAR ... 94
IAND ... 95
IARGC ... 96
IBCLR ... 96
IBITS ... 97
IBSET ... 97
ICHAR ... 97
IDATE ... 98
IDIM ... 70
IDINT ... 100
IDNINT ... 127
IEEE, ISNAN ... 105
IEOR ... 99
IERRNO ... 99
IFIX ... 100
IMAG ... 42
IMAGPART ... 42
IMPORT statement ... 25
INCLUDE directive ... 13
inclusion, directory search paths for ... 13
INDEX ... 100
INT ... 100
INT2 ... 101
INT8 ... 101
integer kind ... 141
intrinsic Modules ... 163
intrinsic procedures ... 37
IOMSG= specifier ... 25
IOR ... 102
IOSTAT, end of file ... 103
IOSTAT, end of record ... 103
IRAND ... 102
IS_IOSTAT_END ... 103
IS_IOSTAT_EOR ... 103
ISATTY ... 104
ISHFT ... 104
ISHFTC ... 105
ISIGN ... 143
ISNAN ... 105
ISO C Bindings ... 25
ISO_FORTRAN_ENV statement ... 25
ITIME ... 106

K

KILL ... 106
kind ... 106
KIND ... 106
kind, integer ... 141
kind, old-style ... 27
kind, real ... 141

L

language, dialect options ... 8
LBOUND ... 107
LEN ... 107
LEN_TRIM ... 108
lexical comparison of strings ... 109, 110, 111
LGE ... 109
LGT ... 110
libf2c calling convention ... 14, 16
limits, largest number ... 94
limits, smallest number ... 155
LINK ... 110
linking, static ... 13
LLE ... 111
LLT ... 111
LNBLNK ... 112
LOC ... 112
location of a variable in memory ... 112
LOG ... 113

LOG10 . . . 113
logarithmic function . . . 113
logarithmic function, inverse . . . 77
LOGICAL . . . 114
logical and, bitwise . . . 45, 95
logical exclusive or, bitwise . . . 99, 160
logical not, bitwise . . . 128
logical or, bitwise . . . 102, 129
login name . . . 92
LONG . . . 114
LSHIFT . . . 115
LSTAT . . . 115
LTIME . . . 116

M

MALLOC . . . 116
MATMUL . . . 117
matrix multiplication . . . 117
matrix, transpose . . . 156
MAX . . . 118
MAX0 . . . 118
MAX1 . . . 118
MAXEXPONENT . . . 118
maximum value . . . 118, 119
MAXLOC . . . 119
MAXVAL . . . 119
MCLOCK . . . 120
MCLOCK8 . . . 120
MERGE . . . 121
messages, error . . . 10
messages, warning . . . 10
MIN . . . 121
MIN0 . . . 121
MIN1 . . . 121
MINEXPONENT . . . 122
minimum value . . . 121, 123
MINLOC . . . 122
MINVAL . . . 123
MOD . . . 124
model representation, base . . . 132
model representation, epsilon . . . 74
model representation, largest number . . . 94
model representation, maximum exponent . . . 118
model representation, minimum exponent . . . 122
model representation, precision . . . 130
model representation, radix . . . 132
model representation, range . . . 135
model representation, significant digits . . . 70
model representation, smallest number . . . 155
module entities . . . 9
module search path . . . 13
modulo . . . 124
MODULO . . . 124
MOVE_ALLOC . . . 125
moving allocation . . . 125
multiply array elements . . . 131
MVBITS . . . 126

N

Namelist . . . 28
NEAREST . . . 126
NEW_LINE . . . 127
newline . . . 127
NINT . . . 127
NOT . . . 128
NULL . . . 128

O

OpenMP . . . 9, 33
operators, unary . . . 30
options, code generation . . . 14
options, debugging . . . 12
options, dialect . . . 8
options, directory search . . . 13
options, errors . . . 10
options, fortran dialect . . . 8
options, gfortran command . . . 7
options, linking . . . 13
options, negative forms . . . 7
options, run-time . . . 14
options, runtime . . . 14
options, warnings . . . 10
OR . . . 129
output, newline . . . 127

P

PACK . . . 129
paths, search . . . 13
PERROR . . . 130
pointer, C address of pointers . . . 56
pointer, C address of procedures . . . 56
pointer, C association status . . . 55
pointer, convert C to Fortran . . . 57
pointer, cray . . . 84, 116
pointer, Cray . . . 31
pointer, disassociated . . . 128
pointer, status . . . 48, 128
positive difference . . . 70
PRECISION . . . 130
Preprocessing . . . 2
PRESENT . . . 131
private . . . 9
procedure pointer, convert C to Fortran . . . 58
process id . . . 92
PRODUCT . . . 131
product, double-precision . . . 71
product, matrix . . . 117
product, vector . . . 71
program termination . . . 77
program termination, with core dump . . . 37
PROTECTED statement . . . 25

R

RADIX 132
RAN 133
RAND 133
random number generation 102, 133
random number generation, seeding 134, 149
RANDOM_NUMBER 133
RANDOM_SEED 134
RANGE 135
range checking 16
read character, stream mode 79, 80
REAL 135
real kind 141
real number, exponent 78
real number, fraction 84
real number, nearest different 126
real number, relative spacing 138, 147
real number, scale 138
real number, set exponent 142
REALPART 135
RECORD 35
remainder 124
RENAME 136
repacking arrays 17
REPEAT 137
RESHAPE 137
root 148
rounding, ceiling 46, 58
rounding, floor 42, 81
rounding, nearest whole number 127
RRSPACING 138
RSHIFT 138

S

SAVE statement 14
SCALE 138
SCAN 139
search path 13
search paths, for included files 13
SECNDS 140
SECOND 140
seeding a random number generator 134, 149
SELECTED_INT_KIND 141
SELECTED_REAL_KIND 141
SET_EXPONENT 142
SHAPE 142
SHORT 101
SIGN 143
sign copying 143
SIGNAL 144
SIN 144
sine 144
sine, hyperbolic 145
sine, hyperbolic, inverse 48
sine, inverse 47
SINH 145
SIZE 145
size of a variable, in bits 54
size of an expression 146
SIZEOF 146
SLEEP 147
SNGL 147
SPACING 147
SPREAD 148
SQRT 148
square-root 148
SRAND 149
Standards 4
STAT 149
statement, ENUM 25
statement, ENUMERATOR 25
statement, FLUSH 25
statement, IMPORT 25
statement, ISO_FORTRAN_ENV 25
statement, PROTECTED 25
statement, SAVE 14
statement, USE, INTRINSIC 25
statement, VALUE 25
statement, VOLATILE 25
STREAM I/O 25
stream mode, read character 79, 80
stream mode, write character 82, 83
string, adjust left 41
string, adjust right 41
string, comparison 109, 110, 111
string, concatenate 137
string, find missing set 159
string, find non-blank character 112
string, find subset 139
string, find substring 100
string, length 107
string, length, without trailing whitespace 108
string, remove trailing whitespace 157
string, repeat 137
STRUCTURE 35
structure packing 17
subscript checking 16
substring position 100
SUM 151
sum array elements 151
suppressing warnings 10
symbol names 8
symbol names, transforming 15, 16
symbol names, underscores 15, 16
SYMLNK 151
syntax checking 10
SYSTEM 152
system, error handling 87, 99, 130
system, group id 91
system, host name 94
system, login name 92
system, process id 92
system, signal handling 144
system, system call 152
system, terminal 104, 157

system, user id 93
system, working directory 59, 90
SYSTEM_CLOCK 152

T

tabulators 12
TAN 153
tangent 153
tangent, hyperbolic 153
tangent, hyperbolic, inverse 50
tangent, inverse 49, 50
TANH 153
terminate program 77
terminate program, with core dump 37
TIME 154
time, clock ticks 120, 152
time, conversion to GMT info 93
time, conversion to local time info 116
time, conversion to string 66
time, current 67, 78, 106, 154, 155
time, elapsed 65, 72, 76, 140
TIME8 155
TINY 155
TR 15581 25
trace 13
TRANSFER 155
transforming symbol names 15, 16
transpose 156
TRANSPOSE 156
trigonometric function, cosine 63
trigonometric function, cosine, inverse 40
trigonometric function, sine 144
trigonometric function, sine, inverse 47
trigonometric function, tangent 153
trigonometric function, tangent, inverse 49, 50
TRIM 157
TTYNAM 157
type cast 155

U

UBOUND 158
UMASK 158
underflow 12
underscore 15, 16
UNLINK 158
UNPACK 159
unused parameter 12
USE, INTRINSIC statement 25
user id 93

V

VALUE statement 25
vector product 71
VERIFY 159
VOLATILE statement 25

W

warnings, aliasing 11
warnings, all 11
warnings, ampersand 11
warnings, character truncation 11
warnings, conversion 11
warnings, implicit interface 11
warnings, non-standard intrinsics 11
warnings, suppressing 10
warnings, suspicious code 11
warnings, tabs 12
warnings, to errors 12
warnings, underflow 12
warnings, unused parameter 12
write character, stream mode 82, 83

X

XOR 160

Z

ZABS 38
ZCOS 63
ZEXP 77
ZLOG 113
ZSIN 144
ZSQRT 148