G95

It's Free Crunch Time

http://www.g95.org

Key G95 Features

- Free Fortran 95 compliant compiler.
- Current (September 2006) g95 version is 0.91.
- GNU Open Source, GPL license.
- Operation of compiled programs can be modified by a large list of environment variables, documented in the compiled program itself.
- TR15581– Allocatable dummy arguments, derived type components.
- F2003 style procedure pointers, structure constructors, interoperability
- F2003 intrinsic procedures and modules.
- Dummy arguments of type `VALUE` in subroutine are passed by value.
- Comma option in OPEN, READ, and WRITE for denoting decimal point.
- Square brackets `[` and `]` may be used for array constructors.
- `IMPORT` statement, used in an interface body to enable access to entities of the host scoping unit.
- `MIN()` and `MAX()` for character as well as numeric types.
- `OPEN` for "Transparent" or stream I/O.
- Backwards compatibility with g77's Application Binary Interface (ABI).
- Default integers of 32 bits or 64 bits available.
- Invoke `SYSTEM()` command.
- Tabbed source allowed.
- Symbolic names with $ option.
- Hollerith data.
- DOUBLE COMPLEX extension.
- Varying length for named `COMMON`.
- Mix numeric and character in `COMMON` and `EQUIVALENCE`.
- `INTEGER` kinds: 1, 2, 4, 8.
- `LOGICAL` kinds: 1, 2, 4, 8.
- `REAL` kinds : 4, 8.
- `REAL(KIND=10)` for x86-compatible systems. 19 digits of precision, value range $10^{\pm 4931}$.
- List-formatted floating point output prints the minimal number of digits necessary to uniquely distinguish the number.
- VAX style debug (D) lines.
- C style string constants option (e.g. 'hello\nworld').
- \ and $ edit descriptors.
- VAX style system intrinsics (SECNDS etc.)
- Unix system extensions library (getenv, etime, stat, etc.)
- Detect non-conformant or non-allocated arrays at run-time - see Table IV at: `http://ftp.aset.psu.edu/pub/ger/fortran/test/results.txt`
- Detection of memory leaks - see Table V at: `http://ftp.aset.psu.edu/pub/ger/fortran/test/results.txt`
- Traceback of runtime errors.
- Smart compile feature prevents module compile cascades.
- F compatibility option. See `http://www.fortran.com/F`. G95 can be built as an F compiler.
- Program suspend/resume feature available for x86/Linux.
- Obsolete real or double precision loop index is DELETED.
- Quick response by developer on bug reports is typical.
- Builds with GCC 4.0.3 and 4.1.1 release versions.
- Available for Linux/x86, PowerPC, 64-bit Opteron, 64-bit Itanium, 64-bit Alpha.
- Available for Windows/Cygwin, MinGW, & Interix.
- Available for OSX on Power Mac G4, x86-OSX.
- Available for FreeBSD on x86, HP-UX 11, Sparc-Solaris, x86-Solaris, OpenBSD, NetBSD, AIX, IRIX, Tru64 UNIX on Alpha.
- Fink versions are also available.
- Binaries of 'stable' and current versions for most platforms are available at `http://ftp.g95.org`.

Every now and then, I get to meet someone that I've exchanged email with about g95. The most frequent comment that I get in these situations is what an extraordinary job that I am doing alone. I always laugh and point out that I've never done it alone. The number of people who have actively helped with g95 is probably close to a thousand or so. The assumption is that the person doing writing the code is doing all the work, when in reality people who distill crashes down to a dozen lines of code are in fact performing an extremely valuable service, one that is frequently overlooked. Writing something as complicated as a modern fortran compiler is not something you do by yourself. I know.

Like most things, g95 was born out of frustration. I wrote my PhD thesis code in fortran 77 using g77. Fortran is such a wonderful language for numerical computation– it is a quick and dirty language for people who care more about the answer than writing the program. My thesis code had a lot of fairly sophisticated data structures in it– linked lists, octrees, sparse matrices, supporting finite element grid generation, solving Poisson's equation, multipole expansions, conjugate gradient minimization and lots of computational geometry. Because I was using fortran 77, the code ended up very clunky and could have benefitted immensely from dynamic memory allocation and derived types. And my thesis was winding down and I needed a new challenge.

Beyond the convenience of more advanced language features, I've also been greatly inspired by the work of Bill Kahan. The thing I came away with after reading many of Bill's papers has been the idea that even though numerical calculations are tricky, ways can be found to do things such that errors are reduced to the point where no one cares about them any longer. The user is often at the mercy of the library author at this point.

Although the compiler is the cool part, it is the libraries that have always interested me more. The actions of the compiler are fairly strictly defined by the standard, and it is in the library that innovation and experimentation can roam free. Even when it was in a fairly primitive state, there were already more bells and whistles in the library compared to other vendors. The corefile resume feature is something I'd wanted for years before actually getting the chance to implement it.

It's been a lot of fun writing g95, and I look forward to maintaining it in the decades ahead.

Andy Vaught
Mesa, Arizona
October 2006

License

G95 itself is licensed under the GNU General Public License (GPL). For all the legal details, see `http://www.gnu.org/licenses/gpl.html`.

The runtime library is mostly GPL and contains an exception to the GPL that gives g95 users the right to link the g95 libraries to codes not covered under the GPL and to distribute linked combinations without causing the resulting programs to be covered by the GPL, or become affected by the GPL in any way.

Installation Notes

Unix (Linux/OSX/Solaris/Irix/etc.):
Open a console, and go to the directory in which you want to install g95. To download and install g95, run the following commands:

```
wget -O - http://ftp.g95.org/g95-x86-linux.tgz | tar xvfz -
ln -s $PWD/g95-install/bin/i686-pc-linux-gnu-g95 /usr/bin/g95
```

The following files and directories should be present:

```
./g95-install/
./g95-install/bin/
./g95-install/bin/i686-pc-linux-gnu-g95
./g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.1.1/
./g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.1.1/f951
./g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.1.1/crtendS.o
./g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.1.1/crtend.o
./g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.1.1/crtbeginT.o
./g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.1.1/crtbeginS.o
./g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.1.1/crtbegin.o
./g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.1.1/cc1
./g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.1.1/libf95.a
./g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.1.1/libgcc.a
./g95-install/INSTALL
./g95-install/G95Manual.pdf
```

The file `cc1` is a symbolic link to `f951` in the same directory.

Cygwin

The `-mno-cygwin` option allows the Cygwin version of g95 to build executables that do not require access to the file cygwin1.dll in order to work, and so can be easily run on other systems. Also the executables are free of restrictions attached to the GNU GPL license. To install a Cygwin version with a working -mno-cygwin option, you will need the mingw libraries installed, available from the Cygwin site at `http://www.cygwin.com`.
Download the binary from `http://ftp.g95.org/g95-x86-cygwin.tgz` to your root Cygwin directory (usually c:\Cygwin). Start a Cygwin session, and issue these commands:

```
cd /
tar -xvzf g95-x86-cygwin.tgz
```

This installs the g95 executable in the `/usr/local/bin` directory structure. Caution: Do not use Winzip to extract the files from the tarball or the necessary links may not be properly set up.

MinGW

The g95 binaries for the MS-Windows environment are packaged as self-extracting installers. Two versions are currently available. Windows 98 users should use the g95 package built using gcc 4.0.3, at `http://ftp.g95.org/g95-MinGW.exe`. Windows NT, XP and 2000 users have the option to use either the same package or one built with gcc 4.1.1, available at `http://ftp.g95.org/g95-MinGW-41.exe`.

The free MinGW/Msys system provides the GNU GCC files needed by g95, which include `ld.exe` (the linker), and `as.exe` (the GNU assembler) from the binutils package, available at `http://www.mingw.org`.

The installer script handles two kinds of installation. If no MinGW is found, it installs g95 along with some essential MinGW binutils programs and libraries in a directory selected by the user. Include the install directory in your `PATH`, and set the environment variable `LIBRARY_PATH` to point to your install directory.

If MinGW is already installed on your system, installing g95 in the root MinGW directory, (generally `C:\mingw`) is recommended to avoid potential conflicts. If the installer detects MinGW, it attempts installing in the MinGW file system. Include the `MinGW\bin` directory in your `PATH`, and set the environment variable

`LIBRARY_PATH=`*path-to-MinGW*`/lib`

On Windows 98 and Windows ME this generally requires editing the system `autoexec.bat` file, and a reboot is needed for the changes to take effect.

Windows XP Users Note: MinGW currently allows a mere 8 megabytes for the heap. If your application requires access to more memory, try compiling with: `-Wl,--heap=0x01000000`. Use larger hexadecimal values for `--heap` until your program runs.

Running G95

G95 determines how an input file should be compiled based on its extension. Allowable file name extensions for Fortran source files are limited to `.f`, `.F`, `.for`, `.FOR`, `.f90`, `.F90`, `.f95`, `.F95`, `.f03` and `.F03`. The filename extension determines whether Fortran sources are to be treated as fixed form, or free format. Files ending in `.f`, `.F`, `.for`, and `.FOR` are assumed to be fixed form source compatible with old f77 files. Files ending in `.f90`, `.F90`, `.f95`, `.F95`, `.f03` and `.F03` are assumed to be free source form. Files ending in uppercase letters are pre-processed with the C preprocessor by default, files ending in lowercase letters are not pre-processed by default.

The basic options for compiling Fortran sources with g95 are:

`-c`	Compile only, do not run the linker.
`-v`	Show the actual programs invoked by g95 and their arguments. Particularly useful for tracking path problems.
`-o`	Specify the name of the output file, either an object file or the executable. An `.exe` extension is automatically added on Windows systems. If no output file is specified, the default output file is named `a.out` on unix, or `a.exe` on Windows systems.

Simple examples:

```
g95 -c hello.f90
```
Compiles `hello.f90` to an object file named `hello.o`.

```
g95 hello.f90
```
Compiles `hello.f90` and links it to produce an executable `a.out` (on unix), or `a.exe` (on MS Windows systems).

```
g95 -c h1.f90 h2.f90 h3.f90
```
Compiles multiple source files. If all goes well, object files `h1.o`, `h2.o` and `h3.o` are created.

```
g95 -o hello h1.f90 h2.f90 h3.f90
```
Compiles multiple source files and links them together to an executable file named `hello` on unix, or `hello.exe` on MS Windows systems.

Option Synopsis

`g95 [ -c | -S | -E ]`	Compile & assemble | Produce assembly code | List source
`[-g] [-pg]`	Debug options
`[-O[`*n*`] ]`	Optimization level, $n = 0, 1, 2, 3$
`[-s ]`	Strip debug info
`[-W`*warn*` ] [-pedantic]`	Warning switches
`[-I`*dir*` ]`	Include directory to search
`[-L`*dir*` ]`	Library directory to search

`[-D macro[=value]... ]`	Define macro
`[-U macro ]`	Undefine macro
`[-f option ...]`	General compile options
`[-m machine-option ...]`	Machine specific options. See GCC manual
`[-o outfile ]`	Name of outfile
`  infile`	

G95 Options

Usage: g95 [options] file...

`-pass-exit-codes`	Exit with highest error code from a phase.
`--help`	Display this information.
`--target-help`	Display target specific command line options. (Use '`-v --help`' to display command line options of sub-processes).
`-dumpspecs`	Display all of the built in spec strings.
`-dumpversion`	Display the version of the compiler.
`-dumpmachine`	Display the compiler's target processor.
`-print-search-dirs`	Display the directories in the compiler's search path.
`-print-libgcc-file-name`	Display the name of the compiler's companion library.
`-print-file-name=`*lib*	Display the full path to library *lib*.
`-print-prog-name=`*prog*	Display the full path to compiler component *prog*.
`-print-multi-directory`	Display the root directory for versions of libgcc.
`-print-multi-lib`	Display the mapping between command line options and multiple library search directories.
`-print-multi-os-directory`	Display the relative path to OS libraries.
`-Wa,`*options*	Pass comma-separated *options* on to the assembler.
`-Wp,`*options*	Pass comma-separated *options* on to the preprocessor.
`-Wl,`*options*	Pass comma-separated *options* on to the linker.
`-Xassembler` *arg*	Pass *arg* to the assembler.
`-Xpreprocessor` *arg*	Pass *arg* to the preprocessor.
`-Xlinker` *arg*	Pass *arg* to the linker.
`-save-temps`	Do not delete intermediate files.
`-pipe`	Use pipes rather than intermediate files.
`-time`	Time the execution of each subprocess. Unavailable on some platforms (MinGW, OSX).
`-specs=`*file*	Override built-in specs with the contents of *file*.
`-std=`*standard*	Assume that the input sources are for *standard*.
`-B` *directory*	Add *directory* to the compiler's search paths.
`-b` *machine*	Run gcc for target *machine*, if installed.
`-V` *version*	Run gcc version number *version*, if installed.
`-v`	Display the programs invoked by the compiler.
`-M`	Produce a Makefile dependency lines on standard output.
`-###`	Like `-v` but options quoted and commands not executed.
`-E`	Pre-process only; do not compile, assemble or link.
`-S`	Compile only; do not assemble or link.
`-c`	Compile and assemble, but do not link.
`-o` *file*	Place the output into *file*.
`-x` *language*	Specify the *language* of the following input files. Permissible languages include: c, c++, assembler, none; 'none' means revert to the default behavior of guessing the language based on the file's extension.

Options starting with `-g`, `-f`, `-m`, `-O`, `-W`, or `--param` are automatically passed on to the various sub-processes invoked by g95. In order to pass other options on to these processes the `-W`*letter* options must be used. For bug reporting instructions, please see: `http://www.g95.org`.

By default, programs compiled with g95 have no optimization. The *n* in `-O`*n* specifies the level optimization, from 0 to 3. Zero means no optimization, and higher numbers imply more aggressive optimization. Specifying optimization gives the compiler the license to change the code in order to make it faster. The results of calculations are often affected in subtle ways. Using `-O` is the same as `-O1`.

Significant speedups can be obtained specifying at least `-O2 -march=`*arch* where *arch* is your processor architecture, ie `pentium4`, `athlon`, `opteron`, etc. Further Fortran typical options are `-funroll-loops`, `-fomit-frame-pointer`, `-malign-double` and `-msse2`. For information on all the GCC options available when compiling with g95, see: `http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc`.

Preprocessor Options

G95 can handle files that contain C preprocessor constructs.

`-cpp`	Force the input files to be run through the C preprocessor
`-no-cpp`	Prevent the input files from being pre-processed
`-D name[=value]`	Define a preprocessor macro
`-U name`	Undefine a preprocessor macro
`-E`	Show pre-processed source only
`-I` *directory*	Append *directory* to the include and module files search path. Files are searched for in various directories in this order: Directory of the main source file, the current directory, directories specified by `-I`, directories specified in the `G95_INCLUDE_PATH` environment variable and finally the system directories.

Fortran Options

`-Wall`	Enable most warning messages.
`-Werror`	Change warnings into errors.
`-Werror=`*numbers*	Change the comma-separated list of warnings into errors.
`-Wextra`	Enable warnings not enabled by `-Wall`. These are `-Wobsolescent`, `-Wunused-module-vars`, `-Wunused-module-procs`, `-Wunused-internal-procs`, `-Wunused-parameter`, `-Wunused-types`, `-Wmissing-intent` and `-Wimplicit-interface`.
`-Wglobals`	Cross-check procedure use and definition within the same source file. On by default, use `-Wno-globals` to disable.
`-Wimplicit-none`	Same as `-fimplicit-none`.
`-Wimplicit-interface`	Warn about using an implicit interface.
`-Wline-truncation`	Warn about truncated source lines.
`-Wmissing-intent`	Warn about missing intents on format arguments.
`-Wobsolescent`	Warn about obsolescent constructs.
`-Wno=`*numbers*	Disable a comma separated list of warnings indicated by numbers.
`-Wuninitialized`	Warn about variables used before initialized. Requires `-O2`.
`-Wunused-internal-procs`	Warn if an internal procedure is never used.
`-Wunused-vars`	Warn about unused variables.
`-Wunused-types`	Warn about unused module types. Not implied by `-Wall`.
`-Wunset-vars`	Warn about unset variables.
`-Wunused-module-vars`	Warn about unused module variables. Useful for building `ONLY` clauses.
`-Wunused-module-procs`	Warn about unused module procedures. Useful for building `ONLY` clauses.
`-Wunused-parameter`	Warn about unused parameters. Not implied by `-Wall`.
`-Wprecision-loss`	Warn about precision loss in implicit type conversions.
`-fbackslash`	Interpret backslashes in character constants as escape codes. This option is on by default. Use the `-fno-backslash` to treat backslashes literally.
`-fc-binding`	Print C prototypes of procedures to standard output.
`-fd-comment`	Make D lines executable statements in fixed form.
`-fdollar-ok`	Allow dollar signs in entity names.

`-fendian=`*value*	Force the endian-ness of unformatted reads and writes. The *value* must be `big` or `little`. Overrides runtime environment variables.
`-ffixed-form`	Assume that the source file is fixed form.
`-ffixed-line-length-132`	132 character line width in fixed mode.
`-ffixed-line-length-80`	80 character line width in fixed mode.
`-ffree-form`	Assume that the source file is free form.
`-ffree-line-length-huge`	Allow very large source lines (10k).
`-fimplicit-none`	Specify that no implicit typing is allowed, unless overridden by explicit `IMPLICIT` statements.
`-fintrinsic-extensions`	Enable g95-specific intrinsic functions even in a `-std=` mode.
`-fintrinsic-extensions=`	Include selected intrinsic functions even in a `-std=` mode. The list is comma-separated and case insensitive.
`-fmod=`*directory*	Put module files in *directory*.
`-fmodule-private`	Set default accessibility of module-entities to `PRIVATE`.
`-fmultiple-save`	Allow the `SAVE` attribute to be specified multiple times.
`-fone-error`	Force compilation to stop after the first error.
`-ftr15581`	Enable the TR15581 allocatable array extensions even in `-std=F` or `-std=f95` modes.
`-std=F`	Warn about non-F features. See `http://www.fortran.com/F`.
`-std=f2003`	Strict Fortran 2003 checking.
`-std=f95`	Strict Fortran 95 checking.
`-i4`	Set kinds of integers without specification to kind=4 (32 bits).
`-i8`	Set kinds of integers without specification to kind=8 (64 bits).
`-r8`	Set kinds of reals without kind specifications to double precision.
`-d8`	Implies `-i8` and `-r8`.

Code Generation Options

`-fbounds-check`	Check array and substring bounds at runtime.
`-fcase-upper`	Make all public symbols uppercase.
`-fleading-underscore`	Add a leading underscore to public names.
`-fonetrip`	Execute `DO`-loops at least once. (Buggy `FORTRAN` 66).
`-fpack-derived`	Try to layout derived types as compactly as possible. Requires less memory, but may be slower.
`-fqkind=`*n*	Set the kind for a real with the 'q' exponent to *n*.
`-fsecond-underscore`	Append a second trailing underscore in names having an underscore (default). Use `-fno-second-underscore` to suppress.
`-fshort-circuit`	Cause the `.AND.` and `.OR.` operators to not compute the second operand if the value of the expression is known from the first operand.
`-fsloppy-char`	Suppress errors when writing non-character data to character descriptors, and allow comparisons between INTEGER and CHARACTER variables.
`-fstatic`	Put local variables in static memory where possible. This is not the same as linking things statically (`-static`).
`-ftrace=`	`-ftrace=frame` will insert code to allow stack tracebacks on abnormal end of program. This will slow down your program. `-ftrace=full` additionally allows finding the line number of arithmetic exceptions (slower). Default is `-ftrace=none`.
`-funderscoring`	Append a trailing underscore in global names. This option is on by default, use `-fno-underscoring` to suppress.
`-max-frame-size=`*n*	How large in bytes that a single stack frame will get before arrays are allocated dynamically.
`-finteger=`*n*	Initialize uninitialized scalar integer variables to *n*.

`-flogical=`*value*	Initialize uninitialized scalar logical variables. Legal *values* are `none`, `true` and `false`.
`-freal=`*value*	Initialize uninitialized scalar real and complex variables. Legal *values* are `none`, `zero`, `nan`, `inf`, `+inf` and `-inf`.
`-fpointer=`*value*	Initialize scalar pointers. Legal *values* are `none`, `null` and `invalid`.
`-fround=`*value*	Controls compile-time rounding. *value* can be `nearest`, `plus`, `minus` and `zero`. Default is round to nearest, `plus` is round to plus infinity, `minus` is minus infinity, `zero` is towards zero.
`-fzero`	Initialize numeric types to zero, logical values to false and pointers to null. The other initialization options override this one.

Directory Options

`-I` *directory*	Append *directory* to the include and module files search path.
`-L`*directory*	Append *directory* to the library search path.
`-fmod=`*directory*	Put module files in *directory*

Environment Variables

The g95 runtime environment provides many options for tweaking the behavior of your program once it runs. These are controllable through environment variables. Running a g95-compiled program with the `--g95` option will dump all of these options to standard output. The values of the various variables are always strings, but the strings are interpreted as integers or boolean truth values. Only the first character of a boolean is examined and must be '`t`', '`f`', '`y`', '`n`', '`1`' or '`0`' (uppercase OK too). If a value is bad, no error is issued and the default is used. For GCC environment variables used by g95, such as `LIBRARY_PATH`, see the GCC documentation.

`G95_STDIN_UNIT`	Integer	Unit number that will be pre-connected to standard input. No pre-connection if negative, default is 5.
`G95_STDOUT_UNIT`	Integer	Unit number that will be pre-connected to standard output. No pre-connection if negative, default is 6.
`G95_STDERR_UNIT`	Integer	Unit number that will be pre-connected to standard error. No pre-connection if negative, default is 0.
`G95_USE_STDERR`	Boolean	Sends library output to standard error instead of standard output. Default is Yes.
`G95_ENDIAN`	String	Endian format to use for I/O of unformatted data. Values are `BIG`, `LITTLE` or `NATIVE`. Default is `NATIVE`.
`G95_CR`	Boolean	Output carriage returns for formatted sequential records. Default TRUE on non-Cygwin/Windows, FALSE elsewhere.
`G95_INPUT_CR`	Boolean	Treat a carriage return-linefeed as a record marker instead of just a linefeed. Default TRUE.
`G95_IGNORE_ENDFILE`	Boolean	Ignore attempts to read past the ENDFILE record in sequential access mode. Default FALSE.
`G95_TMPDIR`	String	Directory for scratch files. Overrides the `TMP` environment variable. If `TMP` is not set `/var/tmp` is used. No default.
`G95_UNBUFFERED_ALL`	Boolean	If TRUE, all output is unbuffered. This will slow down large writes but can be useful for forcing data to be displayed immediately. Default is FALSE.
`G95_SHOW_LOCUS`	Boolean	If TRUE, print filename and line number where runtime errors happen. Default is TRUE.
`G95_STOP_CODE`	Boolean	If TRUE, stop codes are propagated to system exit codes. Default TRUE.

G95_OPTIONAL_PLUS	Boolean	Print optional plus signs in numbers where permitted. Default FALSE.
G95_DEFAULT_RECL	Integer	Default maximum record length for sequential files. Most useful for adjusting line length of pre-connected units. Default is 50000000.
G95_LIST_SEPARATOR	String	Separator to use when writing list output. May contain any number of spaces and at most one comma. Default is a single space.
G95_LIST_EXP	Integer	Last power of ten which does not use exponential format for list output. Default 6.
G95_COMMA	Boolean	Use a comma character as the default decimal point for I/O. Default FALSE.
G95_EXPAND_UNPRINTABLE	Boolean	For formatted output, print otherwise unprintable characters with \-sequences. Default FALSE.
G95_QUIET	Boolean	Suppress bell characters (\a) in formatted output. Default FALSE.
G95_SYSTEM_CLOCK	Integer	Number of ticks per second reported by the `SYSTEM_CLOCK()` intrinsic. Zero disables the clock. Default 100000.
G95_SEED_RNG	Boolean	If TRUE, seeds the random number generator with a new seed when the program is run. Default FALSE.
G95_MINUS_ZERO	Boolean	If TRUE, prints zero values without a minus sign in formatted (non-list) output, even if the internal value is negative or minus zero. This is the traditional but nonstandard way of printing zeros. Default FALSE.
G95_ABORT	Boolean	If TRUE, dumps core on abnormal program end. Useful for finding the locus of the problem. Default FALSE.
G95_MEM_INIT	String	How to initialize allocated memory. Default value is NONE for no initialization (faster), NAN for a Not-a-Number with the mantissa 0x00f95 or a custom hexadecimal value.
G95_MEM_SEGMENTS	Integer	Maximum number of still-allocated memory segments to display when program ends. 0 means show none, less than 0 means show all. Default 25.
G95_MEM_MAXALLOC	Boolean	If TRUE, shows the maximum number of bytes allocated in user memory during the program run. Default FALSE.
G95_MEM_MXFAST	Integer	Maximum request size for handing requests in from fastbins. Fastbins are quicker but fragment more easily. Default 64 bytes.
G95_MEM_TRIM_THRESHOLD	Integer	Amount of top-most memory to keep around until it is returned to the operating system. -1 prevents returning memory to the system. Useful in long-lived programs. Default 262144.
G95_MEM_TOP_PAD	Integer	Extra space to allocate when getting memory from the OS. Can speed up future requests. Default 0.
G95_SIGHUP	String	Whether the program will `IGNORE`, `ABORT`, `DUMP` or `DUMP-QUIT` on SIGHUP. Default `ABORT`. Unix only.
G95_SIGINT	String	Whether the program will `IGNORE`, `ABORT`, `DUMP` or `DUMP-QUIT` on SIGINT. Default `ABORT`. Unix only.
G95_SIGQUIT	String	Whether the program will `IGNORE`, `ABORT`, `DUMP` or `DUMP-QUIT` on SIGQUIT. Default `ABORT`. Unix only.
G95_CHECKPOINT	Integer	On x86 Linux, the number of seconds between checkpoint corefile dumps, with zero meaning no dumps.
G95_CHECKPOINT_MSG	Boolean	If TRUE, print a message to stderr when process is checkpointed. Default TRUE.
G95_FPU_ROUND	String	Set floating point rounding mode. Values can be `NEAREST`, `UP`, `DOWN`, `ZERO`. Default is `NEAREST`.
G95_FPU_PRECISION	String	Precision of intermediate results. Value can be 24, 53 and 64. Default 64. Only available on x86 and compatibles.

`G95_FPU_DENORMAL`	Boolean	Raise a floating point exception when denormal numbers are encountered. Default FALSE.
`G95_FPU_INVALID`	Boolean	Raise a floating point exception on an invalid operation. Default FALSE.
`G95_FPU_ZERODIV`	Boolean	Raise a floating point exception when dividing by zero. Default FALSE.
`G95_FPU_OVERFLOW`	Boolean	Raise a floating point exception on overflow. Default FALSE.
`G95_FPU_UNDERFLOW`	Boolean	Raise a floating point exception on underflow. Default FALSE.
`G95_FPU_INEXACT`	Boolean	Raise a floating point exception on precision loss. Default FALSE.
`G95_FPU_EXCEPTIONS`	Boolean	Whether masked floating point exceptions should be shown after the program ends. Default FALSE.
G95_UNIT_*x*	String	Overrides the default unit name for unit *x*. Default is `fort.`*x*
G95_UNBUFFERED_*x*	Boolean	If TRUE, unit *x* is unbuffered. Default FALSE.

Runtime Error Codes

Running a g95-compiled program with the `--g95` option will dump this list of error codes to standard output.

-2	End of record
-1	End of file
0	Successful return
	Operating system errno codes (1 - 199)
200	Conflicting statement options
201	Bad statement option
202	Missing statement option
203	File already opened in another unit
204	Unattached unit
205	FORMAT error
206	Incorrect ACTION specified
207	Read past ENDFILE record
208	Bad value during read
209	Numeric overflow on read
210	Out of memory
211	Array already allocated
212	Deallocated a bad pointer
214	Corrupt record in unformatted sequential-access file
215	Reading more data than the record size (RECL)
216	Writing more data than the record size (RECL)

Fortran 2003 Features

G95 implements several features of Fortran 2003. For a discussion of all the new features of Fortran 2003, see: `http://www.kcl.ac.uk/kis/support/cit/fortran/john_reid_new_2003.pdf`.

- The following intrinsic procedures are available: `COMMAND_ARGUMENT_COUNT()`, `GET_COMMAND_ARGUMENT()`, `GET_COMMAND()` and `GET_ENVIRONMENT_VARIABLE()`
- Real and double precision `DO` loop index variables are not implemented in g95.
- Square brackets `[` and `]` may be used as an alternative to `(/` and `/)` for array constructors.
- TR 15581 - allocatable derived types. Allows the use of the `ALLOCATABLE` attribute on dummy arguments, function results, and structure components.
- Stream I/O - F2003 stream access allows a Fortran program to read and write binary files without worrying about record structures. Clive Page has written some documentation on this feature, available at: `http://www.star.le.ac.uk/~cgp/streamIO.html`.
- `IMPORT` statement. Used in an interface body to enable access to entities of the host scoping unit.

- European convention for real numbers– a `DECIMAL='COMMA'` tag in `OPEN`, `READ` and `WRITE` statements allows replacement of the decimal point in real numbers with a comma.
- `MIN()` and `MAX()` work with character as well as numeric types.
- A type declaration attribute of `VALUE` for the dummy argument of a subprogram causes the actual argument to be passed by value.
- F2003 style structure constructors are supported.
- F2003 style procedure pointers are supported.
- F2003's `BIND(C)` construct, `ISO_C_BINDING` module providing easier C interoperability.

Interfacing with G95 Programs

While g95 produces stand-alone executables, it is occasionally desirable to interface with other programs, usually C. The first difficulty that a multi-language program will face is the names of the public symbols. G95 follows the f2c convention of adding an underscore to public names, or two underscores if the name contains an underscore. The `-fno-second-underscore` and `-fno-underscoring` options can be useful to force g95 to produce names compatible with your C compiler. Use the `nm` program to look at the `.o` files being produced by both compilers. G95 folds public names to lowercase as well, unless `-fupper-case` is given, in which case everything will be upper case. Module names are represented as *module-name*`_MP_`*entity-name*.

After linking, there are two main cases: Fortran calling C subroutines and C calling fortran subroutines. For C calling Fortran subroutines, the Fortran subroutines will often call Fortran library subroutines that expect the heap to be initialized in some way. To force a manual initialization from C, call `g95_runtime_start()` to initialize the fortran library and `g95_runtime_stop()` when done. The prototype of `g95_runtime_start()` is:

```
void g95_runtime_start(int argc, char *argv[]);
```

The library has to be able to process command-line options. If this is awkward to do and your program doesn't have a need for command-line arguments, pass `argc=0` and `argv=NULL`. On OSX, include `-lSystemStubs` when using g95 to run the linker and linking objects files compiled by GCC.

F2003 provides a number of features that allow easier interfacing with C. The `BIND(C)` attribute allows fortran symbols to be created that are more easily referenced from C (or other languages). For example:

```
SUBROUTINE foo(a) BIND(C)
```

This form creates a symbol named `foo` without any underscore name-mangling. All characters are forced to lowercase. A similar form is:

```
SUBROUTINE foo(a) BIND(C, name='Foo1')
```

This causes the name of the symbol to be `Foo1`. Within fortran, the subroutine is still referenced by the usual `foo`, `FOO` or any other case combination.

C programs pass arguments by value, where fortran passes them by reference. F2003 provides the `VALUE` attribute to specify dummy arguments that are passed by value. An example would be:

```
SUBROUTINE foo(a)
    INTEGER, VALUE :: a
    ...
```

A subroutine defined like this is still callable from fortran as well with the restriction that dummy arguments are no longer associated with actual arguments, and changing a dummy argument will no longer change an actual argument.

Global variables can similarly be accessed. The following subroutine prints out the value of the `VAR` variable, which would otherwise be inaccessible to fortran:

```
SUBROUTINE print_it
    INTEGER, BIND(C, name='VAR') :: v
    PRINT *, v
END SUBROUTINE
```

Where fortran considers types to have different kinds, C defines everything as distinct types. In order to specify the same object, F2003 provides an intrinsic module `ISO_C_BINDING` which contains mappings from fortran kinds to C types. When `USE`d, the following `PARAMETER`s are defined:

`c_int`	Integer kind for C's `int`
`c_short`	Integer kind for C's `short`
`c_long`	Integer kind for C's `long`
`c_long_long`	Integer kind for C's `long long`
`c_signed_char`	Integer kind for C's `char`
`c_size_t`	Integer kind for C's `size_t`
`c_intptr_t`	Integer kind of the same size as C pointers
`c_float`	Real kind for C's `float`
`c_double`	Real kind for C's `double`

There are many other things in `ISO_C_BINDING` as well. Using this module, one can write a program:

```
SUBROUTINE foo
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER(KIND=C_INT) :: int_var
    INTEGER(KIND=C_LONG_LONG) :: big_integer
    REAL(KIND=C_FLOAT) :: float_var
    ...
```

Using the Random Number Generator

```
REAL INTENT(OUT)::  harvest CALL random_number(harvest)
```

Returns a `REAL` scalar or an array of `REAL` random numbers in `harvest`, $0 \leq$ `harvest` < 1.
Seeding the random number generator:

```
INTEGER, OPTIONAL, INTENT(OUT) ::  sz
INTEGER, OPTIONAL, INTENT(IN) ::  pt(n1)
INTEGER, OPTIONAL, INTENT(OUT) ::  gt(n2)
CALL random_seed(sz,pt,gt)
```

`sz` is the minimum number of default integers required to hold the value of the seed; g95 returns four. Argument `pt` is an array of default integers with size `n1` $\geq$ `sz`, containing user provided seed values. Argument `gt` is an array of default integers with size `n2` $\geq$ `sz`, containing the current seed.

Calling `RANDOM_SEED()` without arguments initializes the seed to a value determined by the current time. This can be used to generate random sequences that are different for each invocation of the program. The seed is also initialized to a time-based value on program start if the `G95_SEED_RNG` environment variable is set to TRUE. If neither of these conditions are true, `RANDOM_NUMBER()` will always generate the same sequence.

The underlying generator is the xor-shift generator developed by George Marsaglia.

Predefined Preprocessor Macros

The macros that are always defined are:

```
__G95__ 0
__G95_MINOR__ 91
__FORTRAN__ 95
__GNUC__ 4
```

The conditional macros are:

```
unix windows hpux linux solaris irix aix netbsd freebsd openbsd cygwin
```

Corefile Resume Feature

On x86 Linux systems, the execution of a g95-compiled program can be suspended and resumed. If you interrupt a program by sending it the QUIT signal, which is usually bound to control-backslash, the program will write an executable file named `dump` to the current directory. Running this file causes the execution of your program to resume from when the dump was written. The following session illustrates this:

```
andy@fulcrum:~/g95/g95 % cat tst.f90
  b = 0.0
  do i=1, 10
      do j=1, 3000000
          call random_number(a)
          a = 2.0*a - 1.0
          b = b + sin(sin(sin(a)))
      enddo
      print *, i, b
  enddo
  end
andy@fulcrum:~/g95/g95 % g95 tst.f90
andy@fulcrum:~/g95/g95 % a.out
 1 70.01749
 2 830.63153
 3 987.717
 4 316.48703
 5 -426.53815
 6 25.407673     (control-\ hit)
Process dumped
 7 -694.2718
 8 -425.95465
 9 -413.81763
 10 -882.66223
andy@fulcrum:~/g95/g95 % ./dump
Restarting
............Jumping
 7 -694.2718
 8 -425.95465
 9 -413.81763
 10 -882.66223
andy@fulcrum:~/g95/g95 %
```

Any open files must be present and in the same places as in the original process. If you link against other languages, this may not work. While the main use is allowing you to preserve the state of a run across a reboot, other possibilities include pushing a long job through a short queue or moving a running process to another machine. Automatic checkpointing of your program can be done by setting the environment variable `G95_CHECKPOINT` with the number of seconds to wait between dumps. A value of zero means no dumps. New checkpoint files overwrite old checkpoint files.

Smart Compiling

Consider a module `foo` whose source code resides in a file `foo.f95`. We can distinguish between two types of changes to `foo.f95`:

1. Changes that alter the usage of `foo`, e.g., by changing the interface to a procedure;
2. Changes that do not alter the usage of `foo`, but only its implementation, e.g., by fixing a bug in the body of a procedure.

Both kinds of changes will generally affect the contents of the object file `foo.o`, but only the first type of change can alter the contents of foo.mod. When it recompiles a module, g95 is smart enough to detect whether the `.mod` file needs updating: after changes of type 2, the old `.mod` file is retained.

This feature of g95 prevents unnecessary compilation cascades when building a large program. Indeed, suppose that many different source files depend on `foo.mod`, either directly (because of a `USE FOO` statement) or indirectly (by using a module that uses `foo`, or by using a module that uses a module that uses `foo`, etc). A change of type 1 to `foo.f95` will trigger a recompile of all dependant source files; fortunately, such changes are likely to be infrequent. The more common changes of type 2 cause a recompile only of `foo.f95` itself, after which the new object file `foo.o` can be immediately linked with the other existing object files to create the updated executable program.

G95 Intrinsic Function Extensions

ACCESS
```
    INTEGER FUNCTION access(filename, mode)
        CHARACTER(LEN=*) :: filename
        CHARACTER(LEN=*) :: mode
    END FUNCTION access
```
Checks whether the file filename can be accessed with the specified mode, where mode is one or more of the letters `rwxRWX`. Returns zero if the permissions are OK, nonzero if something is wrong.

ALGAMA
```
    REAL FUNCTION algama(x)
        REAL, INTENT(IN) :: x
    END FUNCTION algama
```
Returns the natural logarithm of $\Gamma(x)$. `ALGAMA` is a generic function that takes any real kind.

BESJ0
```
    REAL FUNCTION besj0(x)
        REAL, INTENT(IN) :: x
    END FUNCTION besj0
```
Returns the zeroth order Bessel function of the first kind. This function is generic.

BESJ1
```
    REAL FUNCTION besj1(x)
        REAL, INTENT(IN) :: x
    END FUNCTION besj1
```
Returns the first order Bessel function of the first kind. This function is generic.

BESJN
```
    REAL FUNCTION besjn(n,x)
        INTEGER, INTENT(IN) :: n
        REAL, INTENT(IN) :: x
    END FUNCTION besjn
```
Returns the nth order Bessel function of the first kind. This function is generic.

BESY0
```
    REAL FUNCTION besy0(x)
        REAL, INTENT(IN) :: x
    END FUNCTION besy0
```
Returns the zeroth order Bessel function of the second kind. This function is generic.

BESY1
```
    REAL FUNCTION besy1(x)
        REAL, INTENT(IN) :: x
    END FUNCTION besy1
```
Returns the first order Bessel function of the second kind. This function is generic.

BESYN

```
    REAL FUNCTION besyn(n,x)
        INTEGER, INTENT(IN) :: n
        REAL, INTENT(IN) :: x
    END FUNCTION besyn
```

Returns the nth order Bessel function of the second kind. This function is generic.

CHMOD

```
    INTEGER FUNCTION chmod(file,mode)
        CHARACTER(LEN=*), INTENT(IN) :: file
        INTEGER, INTENT(IN) :: mode
    END FUNCTION chmod
```

Change unix permissions for a file. Returns nonzero if an error occurs.

DBESJ0

```
    DOUBLE PRECISION FUNCTION dbesj0(x)
        DOUBLE PRECISION, INTENT(IN) :: x
    END FUNCTION dbesj0
```

Returns the zeroth order Bessel function of the first kind.

DBESJ1

```
    DOUBLE PRECISION FUNCTION dbesj1(x)
        DOUBLE PRECISION, INTENT(IN) :: x
    END FUNCTION dbesj1
```

Returns the first order Bessel function of the first kind.

DBESJN

```
    DOUBLE PRECISION FUNCTION dbesjn(n,x)
        INTEGER, INTENT(IN) :: n
        DOUBLE PRECISION, INTENT(IN) :: x
    END FUNCTION dbesjn
```

Returns the nth order Bessel function of the first kind.

DBESY0

```
    DOUBLE PRECISION FUNCTION dbesy0(x)
        DOUBLE PRECISION, INTENT(IN) :: x
    END FUNCTION debsy0
```

Returns the zeroth order Bessel function of the second kind.

DBESY1

```
    DOUBLE PRECISION FUNCTION dbesy1(x)
        DOUBLE PRECISION, INTENT(IN) :: x
    END FUNCTION dbesy1
```

Returns the first order Bessel function of the second kind.

DBESYN

```
    DOUBLE PRECISION FUNCTION dbesyn(n,x)
        INTEGER, INTENT(IN) :: n
        REAL, INTENT(IN) :: x
    END FUNCTION dbesyn
```

Returns the nth order Bessel function of the second kind.

DCMPLX

```
    DOUBLE COMPLEX FUNCTION dcmplx(x,y)
    END FUNCTION dcmplx
```

Double precision `CMPLX`, `x` and `y` may be any numeric type or kind.

DERF

```
    DOUBLE PRECISION FUNCTION derf(x)
        DOUBLE PRECISION, INTENT(IN) :: x
    END FUNCTION derf
```

Returns the double precision error function of `x`.

DERFC

```
    DOUBLE PRECISION FUNCTION derfc(x)
        DOUBLE PRECISION, INTENT(IN) :: x
    END FUNCTION derfc
```

Returns the double precision complementary error function of `x`.

DFLOAT

```
    DOUBLE PRECISION FUNCTION dfloat(x)
    END FUNCTION dfloat
```

Convert a numeric `x` to double precision. Alias for the `DBLE` intrinsic.

DGAMMA

```
    DOUBLE PRECISION FUNCTION dgamma(x)
        DOUBLE PRECISION, INTENT(IN) :: x
    END FUNCTION dgamma
```

Returns an approximation for $\Gamma(x)$.

DLGAMA

```
    DOUBLE PRECISION FUNCTION dlgama(x)
        DOUBLE PRECISION, INTENT(IN) :: x
    END FUNCTION dlgama
```

Returns the natural logarithm of $\Gamma(x)$.

DREAL

```
    DOUBLE PRECISION FUNCTION dreal(x)
    END FUNCTION dreal
```

Convert a numeric `x` to double precision. Alias for the `DBLE` intrinsic.

DTIME

```
    REAL FUNCTION dtime(tarray)
        REAL, OPTIONAL, INTENT(OUT) :: tarray(2)
    END FUNCTION dtime
```

Sets `tarray(1)` to the number of elapsed seconds of user time in the current process since `DTIME` was last invoked. Sets `tarray(2)` to the number of elapsed seconds of system time in the current process since `DTIME` was last invoked. Returns the sum of the two times.

ERF

```
    REAL FUNCTION erf(x)
        REAL, INTENT(IN) :: x
    END FUNCTION erf
```

Returns the error function of `x`. This function is generic.

ERFC

```
    REAL FUNCTION erfc(x)
        REAL, INTENT(IN) :: x
    END FUNCTION erfc
```

Returns the complementary error function of `x`. This function is generic.

ETIME
```
    REAL FUNCTION etime(tarray)
        REAL, OPTIONAL, INTENT(OUT) :: tarray(2)
    END FUNCTION etime
```
Sets `tarray(1)` to the number of elapsed seconds of user time in the current process. Sets `tarray(2)` to the number of elapsed seconds of system time in the current process. Returns the sum of the two times.

FNUM
```
    INTEGER FUNCTION fnum(unit)
        INTEGER, INTENT(IN) :: unit
    END FUNCTION fnum
```
Returns the file descriptor number corresponding to `unit`. Returns -1 if the unit is not connected.

FSTAT
```
    INTEGER FUNCTION fstat(unit, sarray)
        INTEGER, INTENT(IN) :: unit
        INTEGER, INTENT(OUT) :: sarray(13)
    END FUNCTION fstat
```
Obtains data about the file open on Fortran I/O `unit` and places them in the array `sarray()`. The values in this array are extracted from the stat structure as returned by fstat(2) q.v., as follows: `sarray(1)` Device number, `sarray(2)` Inode number, `sarray(3)` file mode, `sarray(4)` number of links, `sarray(5)` Owner uid, `sarray(6)` Owner gid, `sarray(7)` device type, `sarray(8)` file size, `sarray(9)` Access time, `sarray(10)` Modification time, `sarray(11)` Change time, `sarray(12)` Block size, `sarray(13)` Allocated blocks.

FDATE
```
    CHARACTER(LEN=*) FUNCTION fdate()
    END FUNCTION fdate
```
Returns the current date and time as: Day Mon dd hh:mm:ss yyyy.

FTELL
```
    INTEGER FUNCTION ftell(unit)
        INTEGER, INTENT(IN) :: unit
    END FUNCTION ftell
```
Returns the current offset of Fortran file `unit` or -1 if `unit` is not open.

GAMMA
```
    REAL FUNCTION gamma(x)
        REAL, INTENT(IN) :: x
    END FUNCTION gamma
```
Returns an approximation for $\Gamma(x)$. `GAMMA` is a generic function that takes any real kind.

GETCWD
```
    INTEGER FUNCTION getcwd(name)
        CHARACTER(LEN=*), INTENT(OUT) :: name
    END FUNCTION
```
Returns the current working directory in `name`. Returns nonzero if there is an error.

GETGID
```
    INTEGER FUNCTION getgid()
    END FUNCTION getgid
```
Returns the group id for the current process.

GETPID
```
    INTEGER FUNCTION getpid()
    END FUNCTION getpid
```
Returns the process id for the current process.

GETUID

```
    INTEGER FUNCTION getuid()
    END FUNCTION getuid
```

Returns the user's id.

HOSTNM

```
    INTEGER FUNCTION hostnm(name)
        CHARACTER(LEN=*), INTENT(OUT) :: name
    END FUNCTION hostnm
```

Sets `name` with the system's host name. Returns nonzero on error.

IARGC

```
    INTEGER FUNCTION iargc()
    END FUNCTION iargc
```

Returns the number of command-line arguments (not including the program name itself).

ISATTY

```
    LOGICAL FUNCTION isatty(unit)
        INTEGER, INTENT(IN) :: unit
    END FUNCTION isatty
```

Returns `.true.` if and only if the Fortran I/O unit specified by `unit` is connected to a terminal device.

ISNAN

```
    LOGICAL FUNCTION isnan(x)
        REAL, INTENT(IN) :: x
    END FUNCTION isnan
```

Returns `.true.` if `x` is a Not-a-Number (NaN). This function is generic.

LINK

```
    INTEGER FUNCTION link(path1, path2)
        CHARACTER(LEN=*), INTENT(IN) :: path1, path2
    END FUNCTION link
```

Makes a (hard) link from `path1` to `path2`.

LNBLNK

```
    INTEGER FUNCTION lnblnk(string)
        CHARACTER(LEN=*), INTENT(IN) :: string
    END FUNCTION lnblnk
```

Alias for the standard `len_trim` function. Returns the index of the last non-blank character in string.

LSTAT

```
    INTEGER FUNCTION LSTAT(file, sarray)
        CHARACTER(LEN=*), INTENT(IN) :: file
        INTEGER, INTENT(OUT) :: sarray(13)
    END FUNCTION LSTAT
```

If `file` is a symbolic link it returns data on the link itself. See the `FSTAT()` function for further details. Returns nonzero on error.

RAND

```
    REAL FUNCTION rand(x)
        INTEGER, OPTIONAL, INTENT(IN) :: x
    END FUNCTION rand
```

Returns a uniform pseudo-random number such that $0 \leq$ `rand` < 1. If `x` is 0, the next number in sequence is returned. If `x` is 1, the generator is restarted by calling `srand(0)`. If `x` has any other value, it is used as a new seed with srand.

SECNDS
```
    INTEGER FUNCTION secnds(t)
        REAL, INTENT(IN) :: t
    END FUNCTION secnds
```
Returns the local time in seconds since midnight minus the value `t`. This function is generic.

SIGNAL
```
    FUNCTION signal(signal, handler)
        INTEGER, INTENT(IN) :: signal
        PROCEDURE, INTENT(IN) :: handler
    END FUNCTION signal
```
Interface to the unix `signal` call. Return nonzero on error.

SIZEOF
```
    INTEGER FUNCTION sizeof(object)
    END FUNCTION sizeof
```
The argument `object` is the name of an expression or type. Returns the size of `object` in bytes.

STAT
```
    INTEGER FUNCTION stat(file, sarray)
        CHARACTER(LEN=*), INTENT(IN) :: file
        INTEGER, INTENT(OUT) :: sarray(13), status
    END FUNCTION stat
```
Obtains data about the given `file` and places it in the array `sarray`. See the `fstat()` function for details. Returns nonzero on error.

SYSTEM
```
    INTEGER FUNCTION system(cmd)
        CHARACTER(LEN=*), INTENT(IN) :: cmd
    END FUNCTION system
```
Invoke an external command in the `cmd` string. Returns the system exit code.

TIME
```
    INTEGER FUNCTION time()
    END FUNCTION time
```
Returns the current time encoded as an integer in the manner of the UNIX function `time`.

UNLINK
```
    INTEGER FUNCTION unlink(file)
        CHARACTER(LEN=*), INTENT(IN) :: file
    END FUNCTION unlink
```
Unlink (delete) the file `file`. Returns nonzero on error.

%VAL()
When applied to a variable in a formal argument list, causes the variable to be passed by value. This pseudo-function is not recommended, and is only implemented for compatibility. The F2003 `VALUE` attribute is the standard mechanism for accomplishing this.

%REF()
When applied to a variable in a formal argument list, causes the variable to be passed by reference.

G95 Intrinsic Subroutine Extensions

ABORT
```
    SUBROUTINE abort()
    END SUBROUTINE abort
```
Causes the program to quit with a core dump by sending a SIGABORT to itself (unix).

CHDIR

```
    SUBROUTINE chdir(dir)
        CHARACTER(LEN=*), INTENT(IN) :: dir
    END SUBROUTINE
```

Sets the current working directory to `dir`.

DTIME

```
    SUBROUTINE dtime(tarray, result)
        REAL, OPTIONAL, INTENT(OUT) :: tarray(2), result
    END SUBROUTINE dtime
```

Sets `tarray(1)` to the number of elapsed seconds of user time in the current process since `DTIME` was last invoked. Sets `tarray(2)` to the number of elapsed seconds of system time in the current process since `DTIME` was last invoked. Sets `result` to the sum of the two times.

ETIME

```
    SUBROUTINE etime(tarray, result)
        REAL, OPTIONAL, INTENT(OUT) :: tarray(2), result
    END SUBROUTINE etime
```

Sets `tarray(1)` to the number of elapsed seconds of user time in the current process. Sets `tarray(2)` to the number of elapsed seconds of system time in the current process. Sets `result` to the sum of the two times.

EXIT

```
    SUBROUTINE exit(code)
        INTEGER, OPTIONAL, INTENT(IN) :: code
    END SUBROUTINE exit
```

Exit a program with status `code` after closing open Fortran I/O units. This subroutine is generic.

FDATE

```
    SUBROUTINE fdate(date)
        CHARACTER(LEN=*), INTENT(OUT) :: date
    END SUBROUTINE fdate
```

Sets `date` to the current date and time as: Day Mon dd hh:mm:ss yyyy.

FLUSH

```
    SUBROUTINE flush(unit)
        INTEGER, INTENT(IN) :: unit
    END SUBROUTINE flush
```

Flushes the Fortran file `unit` currently open for output.

FSTAT

```
    SUBROUTINE FSTAT(unit, sarray, status)
        INTEGER, INTENT(IN) :: unit
        INTEGER, INTENT(OUT) :: sarray(13), status
    END SUBROUTINE fstat
```

Obtains data about the file open on Fortran I/O `unit` and places them in the array `sarray()`. Sets status to nonzero on error. See the `fstat` function for information on how `sarray` is set.

GETARG

```
    SUBROUTINE getarg(pos, value)
        INTEGER, INTENT(IN) :: pos
        CHARACTER(LEN=*), INTENT(OUT) :: value
    END SUBROUTINE
```

Sets `value` to the `pos`th command-line argument.

GETENV

```
    SUBROUTINE getenv(variable, value)
        CHARACTER(LEN=*), INTENT(IN) :: variable
        CHARACTER(LEN=*), INTENT(OUT) :: value
    END SUBROUTINE getenv
```

Retrieves the environment variable `variable`, and sets `value` to its value.

GETLOG

```
    SUBROUTINE getlog(name)
        CHARACTER(LEN=*), INTENT(OUT) :: name
    END SUBROUTINE getlog
```

Returns the login name for the process in `name`.

IDATE

```
    SUBROUTINE idate(m, d, y)
        INTEGER :: m, d, y
    END SUBROUTINE idate
```

Sets `m` to the current month, `d` to the current day of the month and `y` to the current year. This subroutine is not very portable across implementations. Use the standard `DATE_AND_TIME` subroutine for new code.

LSTAT

```
    SUBROUTINE lstat(file,sarray,status)
        CHARACTER(LEN=*), INTENT(IN) :: file
        INTEGER, INTENT(OUT) :: sarray(13), status
    END SUBROUTINE lstat
```

If `file` is a symbolic link it returns data on the link itself. see `fstat()` for further details.

RENAME

```
    SUBROUTINE rename(path1, path2, status)
        CHARACTER(LEN=*), INTENT(IN) :: path1, path2
        INTEGER, OPTIONAL, INTENT(OUT) :: status
    END SUBROUTINE rename
```

Renames the file `path1` to `path2`. If the `status` argument is supplied, it is set to nonzero on error.

SIGNAL

```
    SUBROUTINE signal(signal, handler, status)
        INTEGER, INTENT(IN) :: signal
        PROCEDURE, INTENT(IN) :: handler
        INTEGER, INTENT(OUT) :: status
    END SUBROUTINE signal
```

Interface to the unix `signal` system call. Sets `status` to nonzero on error.

SLEEP

```
    SUBROUTINE sleep(seconds)
        INTEGER, INTENT(IN) :: seconds
    END SUBROUTINE sleep
```

Causes the process to pause for `seconds` seconds.

SRAND

```
    SUBROUTINE srand(seed)
        INTEGER, INTENT(IN) :: seed
    END SUBROUTINE srand
```

Re-initializes the random number generator. See the `srand()` function for details.

STAT
```
    SUBROUTINE stat(file, sarray, status)
        CHARACTER(LEN=*), INTENT(IN) :: file
        INTEGER, INTENT(OUT) :: sarray(13), status
    END SUBROUTINE
```
Obtains data about the given file and places it in the array `sarray`. See `fstat()` for details. Sets `status` to nonzero on error.

SYSTEM
```
    SUBROUTINE system(cmd, result)
        CHARACTER(LEN=*), INTENT(IN) :: cmd
        INTEGER, OPTIONAL, INTENT(OUT) :: result
    END SUBROUTINE system
```
Passes the command `cmd` to a shell. If `result` is supplied, it is set to the system exit code of `cmd`.

UNLINK
```
    SUBROUTINE unlink(file, status)
        CHARACTER(LEN=*), INTENT(IN) :: file
        INTEGER, INTENT(OUT) :: status
    END SUBROUTINE unlink
```
Unlink (delete) the file `file`. On error, `status` is set to nonzero.