The F Compiler and Tools

THE FORTRAN COMPANY

Library of Congress Catalog Card Number

9 8 7 6 5 4 3 2 1

ISBN

The Fortran Company
6025 North Wilmot Road
Tucson, Arizona 85750 USA
www.fortran.com

Composition by The Fortran Company

Contents

Preface

1 Installing the F Compiler 1-1

1.1 Introduction 1-1
1.2 Implementations Provided 1-1
1.3 Distribution Medium 1-1
1.4 Installation 1-2
1.5 Services from The Fortran Company 1-4

2 Using the F Compiler 2-1

2.1 Usage 2-1
2.2 Description 2-1
2.3 Options 2-2
2.4 Compilation Messages 2-6
2.5 Limits 2-6
2.6 Pre-connected Input/Output Information 2-7
2.7 Other Input/Output Information 2-7
2.8 Automatic File Preconnection 2-7
2.9 IEEE 754 Arithmetic Support 2-7
2.10 Random Number Algorithm 2-7
2.11 Runtime Garbage Collection 2-8
2.12 Modules 2-8
2.13 Data Types 2-8

3 Preprocessors 3-1

3.1 fppr 3-1
3.2 fpp 3-5
3.3 COCO 3-11

4 F Language Extensions 4-1

4.1 Allocatable Arrays 4-1
4.2 High Performance Fortran 4-4
4.3 USE Statement Changes 4-5

5 Calling Fortran and C 5-1

5.1 Calling Fortran 77 Procedures 5-1
5.2 Calling C Functions 5-1

6 The F Library 6-1

6.1 Kind Numbers 6-1
6.2 The F Input/Output Module 6-1

6.3 Math Module 6-5

7 The Slatec Library 7-1

7.1 Finding Roots in an Interval 7-1
7.2 Finding Roots of a Polynomial 7-2
7.3 Computing a Definite Integral 7-2
7.4 Special Functions 7-3
7.5 Solving Linear Equations 7-4
7.6 Differential Equations 7-4

8 Defined Data Types 8-1

8.1 Varying Length Strings 8-1
8.2 Big Integers 8-1
8.3 High Precision Reals 8-2
8.4 Rationals 8-3
8.5 Quaternions 8-3
8.6 Roman Numerals 8-4

9 HPF Library Module 9-1

9.1 Example 9-1
9.2 Elemental Functions 9-1
9.3 Reduction Functions 9-1
9.4 Scan Functions 9-1
9.5 Scatter Functions 9-2
9.6 Sorting Functions 9-3
9.7 Inquiry Functions 9-3

10 F API to POSIX Facilities 10-1

10.1 Examples 10-1
10.2 f90_unix_dir 10-2
10.3 f90_unix_dirent 10-3
10.4 f90_unix_env 10-4
10.5 f90_unix_errno 10-8
10.6 f90_unix_file 10-9
10.7 f90_unix_io 10-13
10.8 f90_unix_proc 10-13

A F Compiler Software License Agreement

Preface

This document describes facilities available with the F programming language compilation system from The Fortran Company.

Many of the features described in this document also are available with the Numerical Algorithms Group NAGWare f95 Fortran compiler, so F programs using these facilities also can be compiled and run on the NAG compiler.

F compilers and tools for several computing platforms are available from The Fortran Company. More information about F is available on the World Wide Web at http://www.fortran.com.

Walt Brainerd

2003 April

Installing the F Compiler 1

This Chapter describes how to install the F compiler and related software.

1.1 Introduction

Before installing the software, please read and abide by the Software Agreement in Appendix A.

1.2 Implementations Provided

This implementation is a compiled, tested, ready-to-use version of the Fortran Company / NAG F Compiler that is considered suitable for operation on the computer systems detailed below:

1.2.1 Linux

hardware	Intel 486 or later
operating system	Linux (built on Red Hat 8.0)
additional systems	The GNU C compiler (gcc) and run-time libraries must be installed

1.2.2 Solaris

hardware	SPARC
operating system	Solaris 2 and later (built on 2.5)
additional systems	The GNU C compiler (gcc) and run-time libraries must be installed.

1.2.3 Windows

hardware	Intel 486 and later
operating system	All versions of Windows (built on XP)
additional systems	The GNU C compiler (gcc) and run-time libraries must be installed.

This Windows version was compiled using Mingw (minimal GNU for Windows), available either with the F compiler by installing using the file `f_and_mingw_??????.zip` or from the web site:

```
http://www.mingw.org
```

Previous versions were compiled using the Cygwin tools available from the following site:

```
http://sources.redhat.com/cygwin
```

Things may work on Windows if the `gcc` compiler is installed anywhere on your execution path, but we recommend installing the Mingw tools (they are free) to avoid problems with library compatibility..

This version of F for Windows runs only from the command line in a DOS window.

1.3 Distribution Medium

1.3.1 Recording Details

The implementation is distributed in `tar` or `zip` compressed format.

1.3.2 Contents

1.3.2.1 Linux/Unix

The following shows the directory/file organization of the materials on the distribution media. A brief description of some of the files is given.

```
      |------- bin
      |
      |------- lib
      |
-----|------- doc
      |
      |------- examples
      |
      |------- src
```

`bin` contains the main executables `F` and `fpp`, the F preprocessor.

`lib` contains the files:

```
fcomp
dope.h            f95.h           hpf.h
libf96.a          libslatec.a    libg2c.a
```
(possibly other library files depending on platform)
several `.mod` files containing module information

`doc` contains the on-line help man pages, plain text copies of the installation instructions, the software license, and descriptions of some of the special F software in PDF or text format:

```
README         F.in
F.1     slatec.1     f90_unix_proc.3
```
(possibly other .3 man pages)
`F.pdf` (the man page in PDF format)
`F.ps` (the man page in PostScript format)

`examples` contains several examples, including calling a C function, calling a Fortran 77 program compiled with `g77`, using `fpp`, calling `system`, and getting command line arguments.

1.3.2.2 Windows

The single `F` folder contains the same folders as listed above for Linux and Unix: `bin`, `doc`, `lib`, and `examples`. It also contains the folder `tmp`, which is the default location to hold temporary files during compilation. If you get the distribution that includes Mingw, there is an additional `mingw` folder for it.

1.3.3 File Sizes

The F files require approximately 4MB of disc space. Mingw requires approximately 38MB.

1.4 Installation

1.4.1 Linux/Unix

Note: You may need to be root to install the software. If you are not, see below about the location of the library files and `gcc`.

First uncompress the file, e.g.,

```
gzip -d f_linux.tar.gz
```

Then untar the resulting tar file, e.g.,

```
tar xvf f_linux.tar
```

If the installation is being done as root, the ownership of the files should be altered, e.g.

```
chown -R root.bin *
```

The files in the `bin` subdirectory should be copied to a suitable location from where they can be accessed by users, e.g.

```
cp bin/* /usr/local/bin
```

This directory must have read and execute permission and must be in your execution path.
The files in the `lib` subdirectory should be copied to a directory such as `/usr/local/lib/F` or `/opt/F/lib`, e.g.

```
mkdir /usr/local/lib/F
cp -d -p lib/* /usr/local/lib/F
```

The `-d` (no dereference) option preserves the links. The `-p` option preserves permissions.

For Linux distributions, the default library directory is `/usr/local/lib/F`. For the Solaris distribution, the default library is `/opt/F/lib`. If you must use another directory to store the library files (for example, you may not have access to the `/usr` or `/opt` directory), you must compile with the `-Qpath` option:

```
F -Qpath /my_library_path/F my_prog.f95
```

Also, the F compiler assumes the `gcc` compiler is in `/usr/local/bin`; if it is not, someone with root privileges needs to create a link to that location or you need to compile with the `-wc=`*path* option.

The `man` pages in the doc directory should be copied to the appropriate man directory, e.g.

```
cp doc/*.1 /usr/man/man1
cp doc/*.3 /usr/man/man3
```

On some systems, it should be `/opt`, rather than `/usr`. You should be able to find the man pages wherever they are by modifying the environment variable `MANPATH`.

1.4.2 Windows

Select a location to install F.

```
mkdir C:\my_stuff
```

The easiest method if you have the CD distribution is to use Explorer (or DOS commands) to copy the two subfolders `F_??????` and `Mingw` to the installation directory (e.g., `C:\my_stuff`) and then rename `F_??????` as `F`. Alternatively, unzip the source file. This will produce the folder `F` and produce the subfolders `bin`, `doc`, `lib`, `examples`, and `tmp`. If you have the distribution that includes Mingw, it will create a second folder `Mingw`.

Make sure that the folder containing the executable `F.exe` (...`F\bin`) and the `gcc` compiler are in your path. A way to do this is to edit your `AUTOEXEC.BAT` file to add the line

```
PATH=C:\my_stuff\F\bin;C:\my_stuff\Mingw\bin;%PATH%
```

assuming `C:\my_stuff\F\bin` contains the executable F.exe and `C:\my_stuff\Mingw\bin` contains the `gcc` executable.

If it appears there are difficulties finding or creating files, try using the `-Qpath` and `-tempdir` compiler options as described in Section 2.3 and the F `man` page (`F.1`, `F.ps`, `F.pdf`).

On NT, click *Start* → *Settings* → *Control Panel* in the *SYSTEM* application. In *SYSTEM*, go to the *ENVIRONMENT* tab. Scroll down in the *System Variable* in the top window (note that the bottom window is per-user variables) until you get to *PATH*. Click on *PATH* and update the value in the line at the bottom of the screen by adding the appropriate paths, followed by a semicolon to separate them

from the paths already there. To ensure that this is in effect (you may need to log in again or reboot); type

```
echo %PATH%
```

If you are using the Cygwin tools, the `man` pages in the `doc` directory should be copied to the appropriate man directory, e.g.

```
cp doc/*.1 /usr/man/man1
```

1.4.2.1 Checking Accessibility

Installation may be checked by compiling and running the test program `examples/seven_11.f95`. To compile and link the program, copy the source file to your working directory and type:

```
F seven_11.f95
```

or simply

```
F seven_11
```

1.5 Services from The Fortran Company

1.5.1 Support

It may be possible for your local advisory service or your Site Contact to resolve queries concerning this document or the implementation in general. However, please do contact The Fortran Company at the supplied address, if you have any difficulties.

Technical queries should be addressed to

```
info@fortran.com
```

1.5.2 Other Sources of Information

Copies of the book *Programmers Guide to F*, by Brainerd, Goldberg, and Adams, *Key Features of F*, by Adams, Brainerd, Martin, and Smith, and other books about F are available:

```
http://www.fortran.com/books.html
```

or

The Fortran Company
6025 North Wilmot Road
Tucson, Arizona 85750 USA
+1-877-355-6640 (voice & fax)
+1-520-760-1397 (outside North America)
info@fortran.com

Using the F Compiler 2

2.1 Usage

Let us go through the steps to create, compile, and run a simple F program. Suppose we want to find the value of *sin*(0.5).

The first step is to use any editor to create a file with the suffix `.f95` that contains the F program to print this value. On Linux and Unix, the editor Emacs or Vi might be used as follows:

```
$ vi print_sin.f95
```

On Windows, Edit or Notepad might be used to create the file.

Suppose the file contains the following F program:

```
program print_sin
   print *, sin(0.5)
end program print_sin
```

A nice convention is to name the file the same as the name of the program, but with the `.f95` suffix.

The next step is to compiler the program. The F command has the following form:

`F` [*option*] . . . [*file*] . . .

so, on a Linux or Unix system, the command for our example is:

```
$ F print_sin.f95
```

and on Windows, you need to be in a DOS window and enter the same command:

```
C:\My_stuff> F print_sin.f95
```

On a Linux or Unix system, this produces the executable program named `a.out`, which can be executed by entering:

```
$ ./a.out
```

On Windows, the executable file is named `a.exe` and can be run by entering the command `a` or `a.exe`.

2.2 Description

F is the Fortran Company / Numerical Algorithms Group compiler for F programs. It translates programs written in F into executable programs, relocatable binary modules, assembler source files, or C source files. A `man` page is provided.

The suffix of a filename determines the action F performs upon it. Files with names ending in `.f90` or `.f95` are taken to be F source files. Files ending in `.F90` or `.F95` are taken to be F source files requiring preprocessing by `fpp` (Section 3.2). The file list may contain filenames of any form.

If a filename without a suffix is provided and there is no file of that name, F will look for a file with the suffix `.f95`, and if that does not exist, for a file with the suffix `.f90`.

Modules and include files are expected to exist in the current working directory or in a directory named by the `-I` option.

Options not recognized by F are passed to the link phase (`gcc`).

2.3 Options

`-c`

Compile only (produce `.o` file for each source file); do not link the `.o` files to produce an executable file.

`-C`

Compile code with all possible runtime checks. This option is a synonym for `-C=all`.

`-C=check`

Compile checking code according to the value of check, which must be one of: `all` (perform all checks), `array` (check array bounds), `calls` (check procedure references), `do` (check `do` loops for zero step values), `none` (do no checking: this is the default), `present` (check optional references), or `pointer` (check pointer references).

`-D`*name*

Defines name to `fpp` as a preprocessor variable. This is equivalent to passing the `-D` option directly to `fpp`, and affects only `.F90` and `.F95` files.

`-dryrun`

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

`-F`

Preprocess all `.F90` and `.F95` files producing `.f90` and `.f95` output; no compilation is done.

`-float-store`

Do not store floating-point variables in registers on machines with floating-point registers wider than 64 bits. This can avoid problems with excess precision.

`-g90`

Compile and link for debugging by `dbx90`, the F aware front-end to `dbx`. This produces a debug information (`.g90`) file for each F source file. This option must be specified for compilation and linking and may be unavailable on some implementations.

`-gc`

Enables automatic garbage collection of the executable program. This option must be specified for compilation and linking.

`-gline`

Compile code to produce line number information in runtime error messages (note that this affects most, but not all, runtime error messages). This option increases both executable file size and execution time.

`-hpf`

Accept the extensions to F as specified by the High Performance Fortran Forum in HPF 1.0. These consist of some intrinsic functions, the `extrinsic` keyword, and a number of compiler directives. The compiler directives are checked for correctness but have no effect on compilation.

`-I` *pathname*

Add pathname to the list of directories which are to be searched for module information (`.mod`) files and include files. The current working directory is always searched first, then any directories named in `-I` options, then the compiler's library directory (usually `/usr/local/lib/F` or `/opt/F/lib` on Linux and Unix systems).

`-ieee=`*mode*

Set the mode of IEEE arithmetic operation according to mode, which must be one of `full`, `nonstd`, or `stop`.

`-ieee=full` enables all IEEE arithmetic facilities including non-stop arithmetic.

`-ieee=nonstd` disables IEEE gradual underflow, producing zero instead of a denormalized number; the resulting program may run faster. Non-stop arithmetic is also disabled, terminating execution on floating overflow, divide by zero, or invalid operand.

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

The `-ieee` option must be specified when compiling the main program unit, and its effect is global. The default mode is `-ieee=stop`.

`-info`

Request output of information messages. The default is to suppress these messages.

`-kind=`*option*

Specify the kind numbering system to be used; option must be one of `byte` or `sequential`.
For `-kind=byte`, the kind numbers for integer, real, and logical will match the number of bytes of storage (e.g., default real is 4). Note that complex kind numbers are the same as its real components, and thus half of the total byte length in the entity.

For `-kind=sequential` (the default), the kind numbers for all data types are numbered sequentially from 1, increasing with precision (e.g., default real is 1 and the next higher precision is 2).

`-lx`

Load with library `libx.a`. The loader will search for this library in the directories specified by any `-L`*dir* options followed by the normal system directories (e.g., see the Unix `ld`(1) command).

`-L`*dir*

Add *dir* to the list of directories for library files.

`-M`

Produce module information files (`.mod` files) only.

`-mdir` *dir*

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

`-nomod`

Suppress module information (`.mod`) file production.

`-o` *output*

Name the output file output instead of `a.out` (`a.exe` on Windows). This may also be used to specify the name of the output file produced under the `-c` and `-S` options.

`-O`

Normal optimization, equivalent to `-O2`.

`-O`*N*

Set the optimization level to *N*. The optimization levels are:

`-O0`	No optimization. This is the default, and is the only optimization level compatible with debugging.
`-O1`	Minimal quick optimization
`-O2`	Normal optimization
`-O3`	Further optimization
`-O4`	Maximal optimization

`-Oassumed`

This is a synonym for `-Oassumed=contig`.

`-Oassumed=`*shape*

Optimizes assumed-shape array dummy arguments according to the value of *shape*, which must be one of

`always_contig`	Optimized for contiguous actual arguments. If the actual argument is not contiguous a run-time error will occur (the compiler is not standard-conforming under this option).
`contig`	Optimized for contiguous actual arguments; if the actual argument is not contiguous (i.e., it is an array section) a contiguous local copy is made. This may speed up array section accessing if a sufficiently large number of array element or array operations is performed (i.e., if the cost of making the local copy is less than the overhead of noncontiguous array accesses), but usually makes such accesses slower. Note that this option does not affect dummy arguments with the target attribute; these are always accessed via the dope vector.
`section`	Optimized for low/moderate accesses to array section (noncontiguous) actual arguments. This is the default.

Note that character arrays are not affected by these options.

`-Oblock` *N*

Specify the dimension of the blocks used for evaluating the `matmul` intrinsic. The default value (only when `-O` is used) is 30, i.e., the arguments are processed in 30×30 blocks.

`-Ounroll=`*N*

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

`-Ounsafe`

Perform possibly unsafe optimizations that may depend on the numerical stability of the program.

`-pg`

Compile code to generate profiling information which is written at run-time to an implementation-dependent file (normally `gmon.out` or `mon.out`}. An execution profile may then be generated using `gprof` (on Sun and IBM RISC System) or `prof` (on DECstation and SGI). This option must be specified for compilation and linking and may be unavailable in some implementations.

`-Qpath` *pathname*

Change the F compiler library path name from the default (usually `/usr/local/lib/F` or `/opt/F/lib` on Unix and Linux and something like `C:my_stuff\F\lib` on Windows) to *pathname*.

`-s`

Strip symbol table information from the executable file. This option is passed to the loader so has effect only during the link step.

`-S`

Produce assembler (actually C source code). The resulting .c file should be compiled with the F command, not with the C compiler directly.

`-target=`*machine*

Specify the machine for which code should be generated and optimized.

For Sun/SPARC, machine may be one of

V7 SPARCstation 1 et al.

V8 SPARCstation 2 et al.

super SuperSPARC

ultra UltraSPARC

The default is to compile for SPARC V7. Note that programs compiled for later versions of the architecture may not run, or may run much more slowly, on an earlier machine.

`-tempdir` *directory*

Set the directory for temporary files to *directory*. The default is to use the directory named by the `TMPDIR` environment variable, or if that is not set, `/tmp` (Linux and Unix) or `..\tmp`, relative to the location of `F.exe` (Windows).

`-time`

Report execution times for the various compilation phases.

`-unsharedf95`

Bind with the unshared (static) version of the F runtime library; this allows a dynamically linked executable to be run on systems where the F compiler is not installed. This option is effective only if specified during the link step.

`-v`

Verbose. Print the name of each pass as the compiler executes.

`-V`

Print version information about the compiler.

`-w`

Suppress all warning messages.

`-W` *options*

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

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

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

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

`-Wl,`*option* Pass *option* directly to the host C compiler when loading (producing the executable). Multiple options may be specified in one `-Wl` option by separating them with commas.

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

`-Wp,`*option* Pass option directly to `fpp` when preprocessing (only for `.F90` and `.F95` files).

`-xs`

(Sun/SPARC option only) Store symbol tables for `dbx` in the executable (avoids the need to keep the object files for debugging).

2.4 Compilation Messages

The diagnostics produced by F itself are intended to be self-explanatory. The loader, or more rarely the host C compiler, may produce additional diagnostics.

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

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

Warning Some questionable usage has been found in the user's code which may indicate a programming error.

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

Error The source code does not conform to the F language specification or does not make sense. Compilation continues after recovery.

Fatal A serious error in the user's program from which the compiler cannot recover; the compilation is aborted immediately.

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

2.5 Limits

Maximum `DO` loop nesting level = 199
Maximum `CASE` construct nesting level = 30
Maximum input/output implied `DO` loop nesting = 20
Maximum array constructor implied `DO` loop nesting = 20
Maximum number of dummy arguments = 32767

Maximum number of arguments to MIN and MAX = 20
Maximum unit number = 99

2.6 Pre-connected Input/Output Information

Standard error (stderr) unit number = 0
Default standard input (stdin) unit number = 5
Default standard output (stdout) unit number = 6

2.7 Other Input/Output Information

Default record length for formatted output = 1024

2.8 Automatic File Preconnection

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

A file preconnected in this manner is opened with ACCESS="SEQUENTIAL". If the initial READ or WRITE is an unformatted i/o statement, it is opened with FORM="UNFORMATTED" otherwise with FORM="FORMATTED". By default a formatted connection is opened with POSITION="REWIND".

Automatic preconnection applies only to the initial use of a logical unit; once closed, the unit will not be reconnected automatically but must be opened explicitly.

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

2.9 IEEE 754 Arithmetic Support

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

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

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

2.10 Random Number Algorithm

The random number generator supplied as the intrinsic subroutine RANDOM_NUMBER is the "good, minimal standard" generator described in "Random Number Generators: Good Ones Are Hard to Find" [*CACM* October 1988, Volume 31 Number 10, pp. 1192-1201]. This is a parametric multiplicative linear congruential algorithm with the following parameters:

modulus: 2^{31}-1 (2147483647)
multiplier: 16807

This is a full-period generator. The seed is obtained from the time-of-day clock; namely *time*×16807+1, where *time* is the number of seconds past midnight.

2.11 Runtime Garbage Collection

The `-gc` option enables use of the runtime garbage collector. It is necessary to use this option during the link phase for it to have effect; specifying it additionally during the compilation phase can result in improved performance. This option may not be available on all systems.

The collector used is based on version 4.11 of the publicly available general purpose garbage collecting storage allocator of Hans-J. Boehm, Alan J. Demers and Xerox Corporation, described in "Garbage Collection in an Uncooperative Environment" (H-Boehm and M Weiser, *Software Practice and Experience*, September 1988, pp. 807-820).

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

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

2.12 Modules

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

Compiling a module creates a `.mod` file and usually a `.o` or `.obj` file. The `.mod` file is used by the compiler at compile time to provide information about module contents; hence, it must available to the compiler whenever it is compiling a program that uses the module (see the `-I` option). The `.o` or `.obj` file (if generated) contains the code of any module procedures and must be specified when creating an executable file.

Note that the name of the `.mod` file will be the name of the module; the `.o` or `.obj` file will be named after the original source file.

When a pre-compiled module is used, the F compiler attempts to find its source file and, if that is successful, checks the modification times, producing a warning message if the `.mod` file is out of date.

2.13 Data Types

The table below lists the data types provided by the F system together with their kind numbers. There are two possibilities for the kind numbers: (1) the default mode of operation (which may be specified explicitly by the `-kind=sequential` option) and (2) the byte numbering scheme (specified by the option `-kind=byte`).

Type	Kind Number (Sequential)	Kind Number (Byte)	Description
Real	1	4	Single precision floating-point
Real	2	8	Double precision floating-point
Real[1]	3	16	Quadruple precision floating-point

Type	Kind Number (Sequential)	Kind Number (Byte)	Description
Complex	1	4	Single precision complex
Complex	2	8	Double precision complex
Complex[1]	3	16	Quadruple precision complex
Logical	1	1	Single byte logical
Logical	2	2	Double byte logical
Logical	3	4	Default logical
Logical[2]	4	8	Eight byte logical
Integer	1	1	8-bit integer
Integer	2	2	16-bit integer
Integer	3	4	32-bit (default) integer
Integer2	4	8	64-bit integer
Character	1	1	ASCII character

[1]Quadruple precision real and complex types are available only on a few machines.

[2]64-bit logical and integer types are available on most machines.

Preprocessors 3

Preprocessors are available with the F distribution. fpp is an extensive preprocessor similar to the C preprocessor cpp. fppr is a simpler preprocessor written by Michel Olagnon. COCO (conditional compilation) is an ancillary Fortran standard.

3.1 fppr

fppr is a preprocessor and "pretty printer" available on all F systems. Here is a simple example.

```
$define WINDOWS 0
$define FPPR_KWD_CASE FPPR_LOWER
$define FPPR_USR_CASE FPPR_LEAVE
$define FPPR_MAX_LINE 132
program test_fppr

$if WINDOWS
character(len=*), parameter :: slash = "\"
$else
character(len=*), parameter :: slash = "/"
$endif

character(len=*), parameter :: file_name = &
   "." // slash // "fppr.f95"
integer :: ios
character(len=99) :: line

open (file=file_name, unit=35, &
      iostat=ios, status="old", &
      action="read", position="rewind")
if (ios == 0) then
   print *, "Successfully opened ", file_name
   read (unit=35, fmt="(a)") line
   print *, "First line: ", trim(line)
else
   print *, "Couldn't open ", file_name
   print *, "IOSTAT = ", ios
end if

end program test_fppr
[walt@localhost Examples]$ fppr < fppr_in.F95 > fppr.f95
This is f90ppr: @(#) fppridnt.f90
     V-1.3 00/05/09 Michel Olagnon
( usage: f90ppr < file.F90  > file.f90 )
[walt@localhost Examples]$ F fppr.f95
[walt@localhost Examples]$ ./a.out
 Successfully opened ./fppr.f95
 First line: program test_fppr
```

Running fppr with input from fppr_in.F95 (shown above) produces an output file fppr.f95. fppr must be executed explicitly; it is not invoked by the F compiler based on the suffix .F95, the way fpp is. Because the fppr variable WINDOWS is not defined to be true, the generated code will include the parameter statement that sets the variable slash to the forward slash; if WINDOWS were true, it would be the backslash. Here is the output file fppr.f95.

```
program test_fppr
!
      character (len=*), parameter :: slash = "/"
!
      character (len=*), parameter :: file_name = &
          "." // slash // "fppr.f95"
      integer :: ios
      character (len=99) :: line
!
      open (file=file_name, unit=35, iostat=ios, &
         status="old", action="read", position="rewind")
      if (ios == 0) then
         print *, "Successfully opened ", file_name
         read (unit=35, fmt="(a)") line
         print *, "First line: ", trim (line)
      else
         print *, "Couldn't open ", file_name
         print *, "IOSTAT = ", ios
      end if
!
end program test_fppr
```

fppr does not make use of any command line argument, and the input and output files need thus to be specified with redirection, (they default to the standard input and the standard output).

3.1.1 Options

All options have to be specified through the use of directives.

3.1.2 Directives

All fppr directives start with a dollar symbol ($) as the first non-blank character in an instruction. The dollar sign was chosen because it is an element of the F character set, but has no special meaning or use. The question mark, which is also an element of the F character set with no special meaning, is used as a "vanishing" separator (see $define below)

`$define` *name token-string*

Replace subsequent instances of *name* with *token-string*. *name* must be identified as a token. In order to enable replacement of sub-strings embedded within tokens, ? is a special "vanishing" separator that is removed by the pre-processor.

`$define` *name* $"*token-string*"

Replace subsequent instances of *name* with *token-string* where *token-string* must not be analyzed since it may consist of multiple instructions, for instance.

`$eval` *name expression*

Replace subsequent instances of *name* with *value* where value is the result, presently of default real or integer kind, of the evaluation of *expression.*

`$undef` *name*

Remove any definition for the symbol name.

`$include` "*filename*"

Read in the contents of *filename* at this location. This data is processed by `fppr` as if it were part of the current file.

$if *constant-expression*

Subsequent lines up to the matching `$else`, `$elif`, or `$endif` directive, appear in the output only if *constant-expression* yields a nonzero value. All non-assignment F operators, including logical ones, are legal in *constant-expression*. The logical constants are taken as 0 when false, and as 1 when true. Many intrinsic functions are also legal in *constant-expression*. The precedence of the operators is the same as that for F. Logical, integer, real constants and `$defined` identifiers for such constants are allowed in *constant-expression*.

`$ifdef` *name*

Subsequent lines up to the matching `$else`, `$elif`, or `$endif` appear in the output only if name has been defined.

`$ifndef` *name*

Subsequent lines up to the matching `$else`, `$elif`, or `$endif` appear in the output only if name has not been defined, or if its definition has been removed with an `$undef` directive.

`$elif` *constant-expression*

Any number of `$elif` directives may appear between an `$if`, `$ifdef`, or `$ifndef` directive and a matching `$else` or `$endif` directive. The lines following the `$elif` directive appear in the output only if all of the following conditions hold:

- The *constant-expression* in the preceding `$if` directive evaluated to zero, the name in the preceding `$ifdef` is not defined, or the name in the preceding `$ifndef` directive was defined.
- The constant-expression in all intervening `$elif` directives evaluated to zero.
- The current constant-expression evaluates to non-zero.

If the constant-expression evaluates to non-zero, subsequent `$elif` and `$else` directives are ignored up to the matching `$endif`. Any constant-expression allowed in an `$if` directive is allowed in an `$elif` directive.

`$else`

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

`$endif`

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

`$macro` *name* `(` *argument* [`,` *argument*] . . . `)` *token-string*

Replace subsequent instances of *name*, followed by a parenthesized list of arguments, with *token-string*, where each occurrence of an argument in *token-string* is replaced by the corresponding token in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into

the expanded *token-string* unchanged. After the entire *token-string* has been expanded, `fppr` does not re-start its scan for names to expand at the beginning of the newly created *token-string*, the opposite of the C preprocessor.

3.1.3 Macros and Defines

Macro names are not recognized within character strings during the regular scan. Thus:

```
$define abc xyz
print *, “abc”
```

does not expand abc in the second line, since it is inside a quoted string.

Macros are not expanded while processing a `$define` or `$undef`. Thus:

```
$define abc zingo
$define xyz abc
$undef abc
xyz
```

produces `abc`. The token appearing immediately after an `$ifdef` or `$ifndef` is not expanded.

Macros are not expanded during the scan which determines the actual parameters to another macro call. Thus:

```
$macro reverse(first,second) second first
$define greeting hello
reverse(greeting,        &
$define greeting goodbye &
)
```

produces

```
$define greeting goodbye greeting.
```

3.1.4 Options

A few pre-defined keywords are provided to control some features of the output code:

`FPPR_FALSE_CMT` !*string*

Lines beginning with !*string* should not be considered as comments, but processed. For instance, one may define:

```
$define FPPR_FALSE_CMT !HPF$
```

in order to use HPF directives in one’s code.

`FPPR_MAX_LINE` *expression*

The current desirable maximum line length for deciding about splitting to a continuation line is set to the value resulting of evaluation of *expression*. If the value is out of the range 2-132, the directive has no effect.

`FPPR_STP_INDENT` *expression*

The current indentation step is set to the value resulting of evaluation of *expression*. If the value is out of a reasonable range (0-60), the directive has no effect. Note that it is recommended to use this directive when current indentation is zero, otherwise unsymmetrical back-indents would occur.

`FPPR_NMBR_LINES` *expression*

If *expression* evaluates to true, or non-zero, or is omitted, line numbering information is output in the same form as with `cpp`. If *expression* evaluates to 0, line numbering information is no longer output.

`FPPR_FXD_IN` expression

If *expression* evaluates to true, or non-zero, or is omitted, the input treated as fixed-form. If *expression* evaluates to 0, the input reverts to free-form.

`FPPR_USE_SHARP` *expression*

If *expression* evaluates to true, or non-zero, or is omitted, the sharp sign (#) may be used as well as the dollar sign as the first character of pre-processing commands. If *expression* evaluates to 0, only commands starting with dollar are processed.

`FPPR_FXD_OUT` *expression*

If *expression* evaluates to true, or non-zero, or is omitted, the output code is intended to be fixed-form compatible. If *expression* evaluates to 0, the output code reverts to free-form.

`FPPR_KWD_CASE` *expression*

If *expression* evaluates to 1, or is the keyword `FPPR_UPPER`, F keywords are output in upper case. If *expression* evaluates to 0, or is the keyword `FPPR_LEAVE`, F keywords are output in mixed case. If *expression* evaluates to –1, or is the keyword `FPPR_LOWER`, F keywords are output in lower case.

`FPPR_USR_CASE` *expression*

If *expression* evaluates to 1, or is the keyword `FPPR_UPPER`, user-defined F identifiers are output in upper case. If *expression* evaluates to 0, or is the keyword `FPPR_LEAVE`, user-defined F identifiers are output in the same case as they were input. If *expression* evaluates to –1, or is the keyword `FPPR_LOWER`, user-defined F identifiers are output in lower case.

3.1.5 Output

Output consists of a copy of the input file, with modifications, formatted with indentation, and possibly changes in the case of the identifiers according to the current active options.

3.1.6 Diagnostics

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

3.1.7 Source Code

The source code is available in the `src` directory of the F distribution. It is provided by Michel Olagnon and more information about this program and others provided by Michel may be found at

`http://www.ifremer.fr/ifremer/ditigo/molagnon/`

3.2 fpp

`fpp` is an F language preprocessor. Currently, it is not available with the Windows distribution of F.

fpp may be invoked explicitly in command mode with the following form:

`fpp` [*options*] [*input-file* [*output-file*]]

If the input file and output file are not given, standard input and output are used. `fpp` also may be invoked implicitly by requesting that the F compiler process a file that ends with `.F90` or `.F95`.

3.2.1 A Simple Example

Consider the following F program with embedded `fpp` commands stored in the file `fpp.F95`:

```
program fpp_example
#if defined(X)
print *, 1.1
#else
print *, 2.2
#endif
#define FF 5
print *. FF.FF
end program fpp_example
```

If the command

```
fpp -DX fpp.f95
```

is entered, the following is produced in the standard output file:

```
# 1 “fpp.F95”
program fpp_example

print *, 1.1
# 6


print *, 5.5
end program fpp_example
```

If the command

```
F -DX fpp.F95
```

is entered and the resulting program is executed, the output will be

```
   1.1000000
   5.5000000
```

If the program is compiled and executed with X not defined

```
F fpp.F95
```

the output will be

```
   2.2000000
   5.5000000
```

3.2.2 Options

`-c_com={yes|no}`

By default, C style comments are recognized. Turn this off by specifying `-c_com=no`

`-D`*name*

Define the preprocessor variable *name* as 1 (one). This is the same as if a `-D`*name*`=1` option appeared on the `fpp` command line, or as if a

```
#define name 1
```

line appeared in the source file, which is processed by `fpp`.

`-D`*name*`=`*def*

Define name as if by a `#define` directive. This is the same as if a

`#define` *name def*

line appeared in the source file that `fpp` is processing. The `-D` option has lower precedence than the `-U` option. That is, if the same name is used in both a `-U` option and a `-D` option, the name will be undefined regardless of the order of the options.

`-I`*directory*

Insert *directory* into the search path for `#include` files with names not beginning with `/`. *directory* is inserted ahead of the standard list of "include" directories. Thus, `#include` files with names enclosed in quotes (") are searched for first in the directory of the file with the `#include` line, then in directories named with `-I` options, and lastly, in directories from the standard list. For `#include` files with names enclosed in angle-brackets (<>), the directory of the file with the `#include` line is not searched.

`-M`

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

`-macro=`{`yes`|`no_com`|`no`}

By default, macros are expanded everywhere. Turn off macro expansion in comments by specifying `-macro=no_com` and turn off macro expansion all together by specifying `-macro=no`.

`-P`

Do not put line numbering directives to the output file. Otherwise, this directive appears as

#line-number file-name

`-U`*name*

Remove any initial definition of name, where *name* is a `fpp` variable that is predefined by a particular preprocessor. Here is a partial list of symbols that may be predefined, depending upon the architecture of the system: `unix`, `__unix`, `__SVR4`, `sun`, `__sun`, `sparc`, and `__sparc`.

`-undef`

Remove initial definitions for all predefined symbols.

`-w`

Suppress warning messages.

`-w0`

Suppress warning messages.

`-Xu`

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

`-Y`*directory*

Use the specified directory in place of the standard list of directories when searching for files.

3.2.3 Usage

3.2.3.1 Tokens

A source file may contain `fpp` tokens. `fpp` tokens are close to those of F. They are:

- `fpp` directive names
- symbolic names including F keywords. `fpp` permits all symbols in names that F does.
- constants of type integer, real, and character
- comments. There are F comments and `fpp` comments
- special characters, such as space, tab, newline, etc.

3.2.3.2 Output

Output consists of a modified copy of the input, plus lines of the form:

#line-number file-name

inserted to indicate the original source line number and filename of the output line that follows. The option `-P` (see above) disables the generation of these lines.

3.2.3.3 Directives

All `fpp` directives start with the hash symbol (`#`) as the first character on a line. White space (space or tab characters) can appear after the initial `#` for proper indentation. The directives can be divided into the following groups:

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

3.2.3.4 Macro Definition

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

`#define` *name token-string*

This is the definition of a `fpp` variable. Wherever name appears in the source lines following the definition, token-string will be substituted for name.

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

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

In these definitions, spaces between the macro name and the symbol `(` are prohibited to prevent the directive being interpreted as a `fpp` variable definition with the rest of the line beginning with the symbol (being interpreted as a token-string.

`#undef` *name*

Remove any definition for name (produced by `-D` options, `#define` directives, or by default). No additional tokens are permitted on the directive line after the name.

3.2.3.5 Including External Files

There are two forms of file inclusion:

`#include` "*filename*"

`#include` <*filename*>

The contents of filename are read in at this location. The lines read in from the file are processed by `fpp` as if it were a part of the current file.

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

3.2.3.6 3.1.3.6 Line Control

#*line-number* "*filename*"

Generate line control information for the next pass of the compiler. The line number is interpreted as the line number of the next line and the filename is interpreted as the name of the file from which it comes. If "*filename*" is not given, the current filename is unchanged.

3.2.3.7 Conditional Selection of Source Text

The form of conditional selection of source text is

```
#if condition_1
  block_1
#elif condition_2
  block_2
#elif condition_3
  block_3
...
#else
  block_n
#endif
```

The `#elif` and `#else` parts are optional. There may be any number of `#elif` parts. Each condition is an expression involving `fpp` constants, macros, and intrinsic functions. Condition expressions are similar to `cpp` expressions, and may contain any `cpp` operations and operands with the exception of C long, octal, and hexadecimal constants. Additionally, `fpp` will accept and evaluate Fortran logical operations and the logical constant; however, the operator for not equal must be `!=` instead of `/=`.

3.2.4 Details

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

3.2.4.1 Scope of Macro or Variable Definitions

The scope of a definition begins from the place of its definition and encloses all the source lines (and source lines from included files) from that definition line to the end of the current file. However, there are the following exceptions:

- `fpp` and F comments;
- format specifications;
- numeric, typeless, and character constants.

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

3.2.4.2 End of Macro Definition

Macro definition can be of any length and is limited only by the newline symbol. A macro can be defined in multiple lines. A macro can be continued to the next line with the insertion of \. So, the occurrence of a newline without a macro continuation signifies the end of the macro definition. For example:

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

3.2.4.3 Function-Like Macro Definition

The number of macro call arguments should be the same as the number of arguments in the corresponding macro definition. An error is flagged if it isn't.

3.2.4.4 Cancelling Macro Definitions

`#undef` *name*

After this directive, name would not interpreted by `fpp` as a macro or variable name. If this name has not been defined earlier as a macro name, then the given directive has no effect.

3.2.4.5 Conditional Source Code Selection

The lines following an `#if` directive up to the matching `#else`, `#elif`, or `#endif` directive, appear in the output only if its condition yields a true value. The lines following an `#elif` directive appear in the output only if all of the following conditions hold:

- The condition in the preceding matching `#if` directive evaluated to false.
- The conditions in all intervening matching `#elif` directives evaluated to false.
- The current condition evaluates to true.

Subsequent `#elif` and `#else` directives are ignored up to the matching `#endif`.

The lines following an `#else` directive up to the matching `#endif` directive, appear in the output only if the condition in all preceding matching `#if` and `#elif` directives evaluate to false.

The following are allowed in the condition of an `#if` or `#elif` directive:

- integer constants
- names
- the `fpp` intrinsic function defined
- C language operations: <, >, ==, !=, >=, <=, +, -, /, *, %, <<, >>, &, ~, |, &&, ||. They are interpreted by `fpp` in accordance to the C language semantics (this facility is provided for compatibility with legacy Fortran programs using `cpp`)
- F language operations: `.AND.`, `.OR.`, `.NEQV.`, `.EQV.`, `.NOT.`, ** (power)
- F logical constants: `.TRUE.` and `.FALSE.`

Names that have not been defined with the help of the `-D` option, a `#define` directive, or by default, get 0 as the value. The C operation `!=` (not equal) can be used in and `#if` or `#elif` directive, but not in a `#define` directive, because the symbol `!` is considered to be an F comment symbol.

3.2.4.6 Including External Files

Files are searched in the following order:

- for `#include` "*file_name*":

  in the directory in which the processed file has been found
  in the directories specified by the `-I` option
  in the default directory

- for `#include` <*file_name*>

  in the directories specified by the `-I` option
  in the default directory

3.2.4.7 Comments

`fpp` permits comments of two kinds:

1. Fortran language comments. A source line containing one of the symbols `C`, `c`, `*`, `d`, or `D` in the first position, is considered as a comment line. Within such lines macro expansions are not performed. The F comment symbol `!` is interpreted as the beginning of a comment extending to the end of the line. The only exception is the case when this symbol occurs within a condition in an `#if` or `#elif` directives (see above).

2. `fpp` C comments enclosed in the `/*` and `*/` parasymbols. They are excluded from the output and macro expansions are not performed within these symbols. `fpp` comments can be nested and for each parasymbol `/*` there must be a corresponding parasymbol `*/`. `fpp` comments are suitable for excluding the compilation of large portions of source instead of commenting every line with F comment symbols.

3.2.4.8 Intrinsic Functions

The intrinsic function

`defined(` *name* `)`

returns

- true if name is defined as a macro
- false if name is not defined

3.2.4.9 Macro Expansion

If, during expansion of a macro, the column width of a line exceeds 132, `fpp` inserts appropriate continuation lines.

3.2.4.10 Diagnostics

There are three kinds of diagnostic messages:

- warnings; preprocessing of source code is continued and the return value remains 0
- errors; `fpp` continues preprocessing, but sets the return code to a nonzero value, namely number of errors.
- fatal error; `fpp` cancels preprocessing and returns a nonzero return value.

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

3.3 COCO

The program `coco` provides preprocessing as per Part 3 of the Fortran Standard (coco stands for "conditional compilation"). It implements the auxiliary third part of ISO/IEC 1539-1:1997 (better known as Fortran 95). (Part 2 is the ISO_VARYING_STRINGS standard, which is sometimes implemented as a module.) A restore program, similar to that described in the standard, is also available for download.

The Fortran source code for coco may be found at `http://users.erols.dnagle`. It is not F conformant, but is standard Fortran 95, except that it uses two extensions from Fortran 2003. You don't need the source code to use `coco` with an F program.

Generally, coco programs are interpreted line by line. A line is either a coco directive or a source line. The coco directives start with the characters `??` in columns 1 and 2. Lines are continued by placing `&` as the last character of the line to be continued. Except for the `??` characters in columns 1 and 2, coco lines follow the same rules as free format lines in Fortran source code. A coco comment is any text following a `!` following the `??` characters. A coco comment may not follow the `&`.

A description of coco may be found in the file `coco.html` in the `Docs` directory. Here is a simple example.

Statement of the problem to be solved: A Fortran program needs to use full path names for file names. The separator in the file names should be `/` if the system is not Windows and `\` if it is Windows. A file `slash.inc` contains the following, which indicates whether the system is Windows or not.

```
?? logical, parameter :: windows = .false.
```

Then the following program will produce the correct character.

```
module slash

?? include “slash.inc”

   character, parameter, public :: &
?? if (windows) then
      slash = “\”
?? else
      slash = “/”
?? end if

end module slash

program p

   use slash
   print *, “Path is usr” // slash // “local”

end program p
```

The COCO preprocessor is run with

```
coco < slash.f90 > new_slash.f95
```

which produces the file `new_slash.f95`:

```
module slash

!?>?? include “slash.inc”
!?>??! INCLUDE slash.inc
!?>?? logical, parameter :: WINDOWS = .false.
!?>??! END INCLUDE slash.inc

   character, parameter, public :: &
!?>?? if (windows) then
!?>      slash = “\”
!?>?? else
      slash = “/”
!?>?? end if

end module slash

program p

   use slash
   print *, “Path is usr” // slash // “local”

end program p
```

Compiling and running the program produces the output:

```
 Path is usr/local
```

F Language Extensions 4

This chapter describes extensions to the F programming language accepted by the F compiler. These are accepted because they are described in ancillary standards documents and are almost certainly to be included in the next Fortran standard (Fortran 2000). The F compiler reports the use of these features as extensions.

There are also several modules available (Chapters 6-10), which, in effect, extend the language facilities available to the F programmer.

4.1 Allocatable Arrays

These extensions allow the use of the ALLOCATABLE attribute for dummy arguments, function results, and structure components.

This feature is described in the technical report ISO/IEC TR 15581:1998 and is expected to be included in the next Fortran standard.

4.1.1 Allocatable Dummy Arrays

A dummy argument can be declared to be an allocatable array. Any actual argument that is passed to an allocatable dummy array must itself be an allocatable array; it must also have the same type, kind type parameters, and rank. The actual argument need not be allocated before calling the procedure, which may itself allocate or deallocate the argument. For example:

```
program alloc_dummy

   real, allocatable, dimension(:) :: a

   call allocate_and_fill(a, 100)

   print *, sum(a)

contains

subroutine allocate_and_fill(da, n)

   real, allocatable, dimension(:), intent(in out) :: da
   integer, intent(in) :: n

   allocate (da(n))
   call random_number(da)

end subroutine allocate_and_fill

end program alloc_dummy
```

4.1.2 Allocatable Function Results

The result of a function can be declared to be an allocatable array. On invoking the function, the result variable will be unallocated. It must be allocated before returning from the function. The result of an allocatable array is automatically deallocated after it has been used.

Here is an example:

```
module comp_module

public :: compress

contains

! The result of this function is the original argument
! with adjacent duplicate entries deleted
! (so if it was sorted, each element is unique).

function compress(array) result (ra)

   integer, allocatable, dimension(:) :: ra
   integer, intent(in), dimension(:) :: array
   integer :: i, n

   n = count(array(:size(array)-1) /= array(2:))
   allocate(ra(n))
   n = 1
   ra(1) = array(1)
   do i = 2, size(array)
      if (array(i) /= ra(n)) then
         n = n + 1
         ra(n) = array(i)
      end if
   end do

end function compress

end module comp_module


program test_comp

   use comp_module

   integer, dimension(9) :: &
      x = (/ 1, 2, 2, 2, 3, 4, 4, 5, 5 /)

   print *, compress(x)

end program test_comp
```

Running this program produces

```
 1 2 3 4 5
```

4.1.3 Allocatable Structure Components

A structure component can be declared to be allocatable. An allocatable array component is initially not allocated, just like allocatable array variables. On exit from a procedure containing variables with allocatable components, all the allocatable components are automatically deallocated. This is in contradistinction to pointer components, which are not automatically deallocated.

Deallocating a variable that has an allocatable array component deallocates the component first; this happens recursively so that all allocatable subobjects are deallocated with no memory leaks.

Any allocated allocatable components of a function result are automatically deallocated after the result has been used.

In a structure constructor for a type with an allocatable component, the expression corresponding to an allocatable array component can be

1. the null intrinsic, indicating an unallocated array
2. an allocatable array which may be allocated or unallocated
3. any other array expression, indicating an allocated array

Intrinsic assignment of such types does a "deep copy" of the allocatable array components; it is as if the allocatable array component were deallocated (if necessary), then if the component in the expression was allocated, the variable's component is allocated to the right size and the value copied.

The following example illustrates many of these ideas. In the module `list_module`, a derived type `list` consisting of one allocatable array component is defined. Assignment is redefined between lists so that adjacent duplicate elements are removed. The function compress shown above is called by the assignment subroutine.

```
module list_module

use comp_module
private

type, public :: list
   integer, dimension(:), allocatable :: value
end type list

public :: assignment(=)
private :: assign

! Modify assignment of lists
! so that compression occurs

interface assignment(=)
   module procedure assign
end interface

contains

subroutine assign(v, e)

   type(list), intent(in) :: e
   type(list), intent(out) :: v

   allocate(v%value(size(compress(e%value))))
   v%value = compress(e%value)

end subroutine assign

end module list_module


program alloc_struct

use list_module
```

```
type(list) :: b
integer :: n

b = list( (/ 1,1,1,2,2,2,2,3,4,4 /) )
n = size(b%value)
print *, n, ":", b%value

end program alloc_struct
```

Running this program produces the following output:

```
 4 : 1 2 3 4
```

4.2 High Performance Fortran

All of the High Performance Fortran 1.0 language can be used by specifying the `-hpf` option. When this option is used, the `EXTRINSIC` prefix is allowed, the `ILEN`, `NUMBER_OF_PROCESSORS`, and `PROCESSORS_SHAPE` intrinsics are recognized, HPF directives are checked, and the `HPF_LIBRARY` module is available. The main benefit of having these features is to permit HPF code to be compiled with the F compiler and to have the intrinsic functions in the `HPF_LIBRARY` module available.

HPF directives are comments beginning with `!HPF$`. For full details see the *HPF Language Specification*; this appeared as a special issue of the journal *Scientific Programming*, published by John Wiley & Sons (1993).

All HPF directives are checked for syntactic and static semantic correctness. However, note that as this is a serial (single processor) implementation, use of these directives has no effect on the code produced.

4.2.1 HPF Intrinsic Functions

Three new intrinsic functions are available with the `-hpf` option:

`ILEN`

This function is elemental with one argument of type integer, returning a value of the same type. It returns one less than the number of bits needed to store the value of its argument in twos-complement arithmetic. For positive numbers, this is one more than the highest bit set (according to the model in section 13.5.7 of the Fortran standard, the least significant bit is bit zero); for negative numbers, it is one more than the highest bit not set.

`NUMBER_OF_PROCESSORS`

This function is scalar of type default integer with no arguments and returns the value 1.

`PROCESSOR_SHAPE`

This function has no arguments and returns a zero-sized array of type default integer.

4.2.2 Extrinsic Procedures

Extrinsic procedure declarations are intended to facilitate calling non-HPF routines from an HPF program. The extrinsic clause is another prefix similar to `RECURSIVE` and `PURE`, but which can occur only within an interface block. Its syntax is:

extrinsic-prefix **is** `EXTRINSIC (` *extrinsic-kind-keyword* `)`
extrinsic-kind-keyword **is** `HPF` **or** `HPF_LOCAL`

Declaring a procedure to be of extrinsic kind `HPF` is the same as leaving the extrinsic clause out altogether. In F, declaring a procedure to be of extrinsic kind `HPF_LOCAL` has no effect.

4.3 USE Statement Changes

These changes are described in ISO/IEC TR 15580: 1998. The `INTRINSIC` or `NON_INTRINSIC` specifier may be used to indicate whether an intrinsic or non-intrinsic module is required. If these are not used, the compiler will pick an intrinsic module only if no user-defined module is found. For example:

```
USE, INTRINSIC :: ieee_exceptions
```

Note that the double colon `::` is required if either specifier is used.

Calling Fortran and C 5

F programs may call Fortran 77 programs compiled with `g77` and C programs compiled with `gcc`.

5.1 Calling Fortran 77 Procedures

If you have installed either the Mingw or Cygwin tools, the Fortran 77 compiler `g77` should be installed.

A Fortran 77 program to be compiled should have the suffix `.f` or `.for`. Compiling should be similar to compiling an F program. Suppose, for example, the file `f77sub.f` contains the following Fortran 77 subroutine:

```
subroutine f77sub(arg)
   integer arg
   print *, 'The value of arg is ', arg
end
```

The subroutine may be compiled by the following command; the option `-c` is used because the file does not contain a complete program.

```
g77 -c f77sub.f
```

The following F program calls `f77sub`.

```
program f_calls_f77
   integer, parameter :: n = 42
   call f77sub(n)
end program f_calls_f77
```

The program may be compiled and linked with the object file produced by `g77` and then executed producing the output shown.

```
F f77sub.o f_calls_f77.f95
./a.out

 The value of arg is 42
```

5.2 Calling C Functions

Calling a C function is more complicated because of the difference in data types, calling conventions, and other things. Here is simple example.

```
typedef struct { float r, i;} Complex;

void csub_ (i, d, a, s, c, slen)
int *i;
double *d;
float a[];
char *s;
Complex *c;
int slen;

{
printf ("The value of i is %d\n", *i);
```

```
printf (“The value of d is %f\n”, *d);
printf (“The value of a[3] is %f\n”, a[3]);
printf (“The value of s is %s\n”, s);
printf (“The value of slen is %d\n”, slen);
printf (“The value of c is (%f, %f)\n”, c->r, c->i);
}
```

This can be compiled with the command

```
gcc -c csub.c
```

An F program that calls csub is

```
program f_calls_c
   integer, parameter :: n = 42
   integer, parameter :: double = selected_real_kind(9)
   real(kind=double), pointer :: dp
   integer :: i
   real, dimension(0:9) :: ra = ( (/ (1.1*i, i=0,9) /) )
   character(len=3) :: s = “abc”
   complex :: c = (1.1, 2.2)

   allocate (dp)
   dp = 4.2_double
   call csub (n, dp, ra, s, c)
end program f_calls_c
```

The program can be compiled, linked, and executed by the commands

```
F csub.o f_calls_c.f95
./a.out
```

producing the output

```
The value of i is 42
The value of d is 4.200000
The value of a[3] is 3.300000
The value of s is abc
The value of slen is 3
The value of c is (1.100000, 2.200000)
```

Note that the name of the C function has an underscore (_) appended. Also, the real and complex dummy arguments are pointers to correspond to the addresses passed for the actual argument.

5.2.1 Data Types

The following table shows the correspondence between F and C data types.

F data type	C data type
integer (8 bits)	signed char
integer (16 bits)	short
default integer (32bits)	int
integer (64 bits)	long long
logical (8 bits)	char

F data type	C data type
logical (16 bits)	short
default logical (32 bits)	int
logical (64 bits)	long long
real (single)	float
real (double)	double
real (quadruple)	long double
complex (single)	Complex
complex (double)	DComplex
complex (quadruple)	QComplex
character	***

where the form of Complex, DComplex, and QComplex are given by

```
typedef struct { float re, im; } Complex;
typedef struct { double re, im; } DComplex;
typedef struct { long double re, im; } QComplex;
```

***For F character actual arguments, there must be two C dummy arguments: `char *` for the string and `int` for the length. The length arguments must be at the end of the dummy argument list in the correct order.

The F Library 6

Several useful modules that may be accessed with the `use` statement are described in this section.

6.1 Kind Numbers

This module contains definitions of integer parameters that can be used as kind numbers. Instead of defining your own parameters, use the parameters defined in this module. For example

```
use f90_kind
logical(kind=byte) :: logb1, logb2
```

The available kind parameters are shown below. Note that not all kinds are available in all implementations.

integer: `int8`, `int16`, `int32`, `int64` (default is `int32`)
real: `single`, `double`, `quad` (default is `single`)
complex: `single`, `double`, `quad` (default is `single`)
logical: `byte`, `twobyte`, `word` (default is `word`)
character: `ascii` (default is `ascii`)

This module also is available with the Numerical Algorithms Group Fortran compiler.

6.2 The F Input/Output Module

Some useful predefined modules are provided that are available for use when doing input and output. They are accessed with the statement

```
use f_io_module
```

6.2.1 An Available Unit Number

The subroutine `new_unit` returns the smallest number greater than 9 of a unit that is available for use. If no unit number is available, –1 is returned. If two unit numbers are needed, the first unit number returned by `new_unit` needs to be connected before getting the second; otherwise, the first number will be returned again. Here is an example.

```
program units

   use f_io_module

   integer :: u1, u2

   call new_unit(u1)
   call new_unit(u2)
   print *, u1, u2
   open (unit=u1, status="scratch", &
         action="readwrite")
   call new_unit(u2)
   print *, u1, u2
   close (unit=u1)
   call new_unit(u1)
   print *, u1, u2
```

```
end program units
```

Running this program produces

```
 10 10
 10 11
 10 11
```

6.2.2 IOSTAT Error Messages

The I/O module contains definitions of integer parameters for all the IOSTAT values that can be returned as a result of use of an input/output statement. An example of its use is

```
program iostat

use f_io_module
integer :: ios

open (status="old", unit=44, file="qwerty.typ", &
      position="rewind", action="read", &
      iostat = ios)
if (ios == IOERR_NO_OLD_FILE) then
   print *, "Can't find 'querty.typ'"
else
   print *, "Found 'querty.typ'"
end if

end program iostat
```

The following show the parameter names and their values. IOSTAT values between 1 and 99 are reserved for host system status returns.

```
IOERR_OK 0
IOERR_EOF -1
IOERR_EOR -2
IOERR_BUFFER_OVERFLOW 100 /* i.e. record too long on output */
IOERR_INTERNAL_FILE_OVERFLOW 101 /* i.e. too many records for it */
IOERR_BAD_SCALE 102
IOERR_BAD_EXPONENT 103  /* Exponent too large for Ew.d/Dw.d */
IOERR_INPUT_BUFFER_OVERFLOW 104
IOERR_ZERO_REPEAT 105
IOERR_BAD_INTEGER 106
IOERR_INTEGER1_TOO_BIG 107
IOERR_INTEGER2_TOO_BIG 108
IOERR_INTEGER_OVERFLOW_REPEAT 109
IOERR_INTEGER_TOO_BIG 110
IOERR_BAD_REAL 111
IOERR_BAD_LOGICAL 112
IOERR_BAD_COMPLEX 113
IOERR_BAD_CHAR 114
IOERR_FORMAT_NO_LPAREN 115
IOERR_FORMAT_NO_ENDING_RPAREN 116
IOERR_NO_DATA_EDIT_IN_REVERSION 117
IOERR_SUBFMT_TOO_DEEP 118
IOERR_UNEXPECTED_FORMAT_END 119
IOERR_EXPECTED_INTEGER_VALUE 120
```

IOERR_FORMAT_MBNZ 121
IOERR_EXPECTED_PERIOD 122
IOERR_EXPECTED_P 123
IOERR_BAD_BNBZ 124
IOERR_BAD_EDIT 125
IOERR_NO_EDIT_FOR_REPEAT 126
IOERR_REPEAT_FOR_SIGN 127
IOERR_REPEAT_FOR_BLANK_INTERP 128
IOERR_MISSING_HOLLERITH_LENGTH 129
IOERR_REPEAT_FOR_CHAR_EDIT 130
IOERR_NO_SPACING_FOR_X 131
IOERR_REPEAT_FOR_POSITION 132
IOERR_CHAR_EDIT_IN_READ 133
IOERR_BAD_EDIT_FOR_REAL 134
IOERR_BAD_EDIT_FOR_INTEGER 135
IOERR_BAD_EDIT_FOR_LOGICAL 136
IOERR_CHAR_OVERLAPS_END 137
IOERR_ONLY_SIGN_FOUND 138
IOERR_BAD_INPUT_EXPONENT 139
IOERR_BAD_INPUT_REAL 140
IOERR_BAD_INPUT_INTEGER 141
IOERR_BAD_BINARY 142
IOERR_BAD_OCTAL 143
IOERR_BAD_HEX 144
IOERR_BAD_EDIT_FOR_CHARACTER 145
IOERR_READ_AFTER_WRITE 146
IOERR_BAD_UNIT 147
IOERR_NOT_CONNECTED 148
IOERR_CANNOT_BACKSPACE 149
IOERR_NOT_SEQUENTIAL 150
IOERR_NOT_READ 151
IOERR_NOT_FORMATTED 152
IOERR_NOT_WRITE 153
IOERR_NOT_UNFORMATTED 154
IOERR_OLD_UNCONNECTED_NEED_FILE 155
IOERR_SCRATCH_NAMED 156
IOERR_DIFFERENT_STATUS 157
IOERR_DIFFERENT_ACCESS 158
IOERR_DIFFERENT_FORM 159
IOERR_DIFFERENT_RECL 160
IOERR_DIFFERENT_ACTION 161
IOERR_DIFFERENT_POSITION 162
IOERR_BAD_STATUS 163
IOERR_BAD_ACCESS 164
IOERR_BAD_FORM 165
IOERR_BAD_BLANKS 166
IOERR_BAD_POSITION 167
IOERR_BAD_ACTION 168
IOERR_BAD_DELIM 169
IOERR_BAD_PAD 170
IOERR_NO_RECL 171
IOERR_CANNOT_KEEP 172
IOERR_ENDFILE_TWICE 173
IOERR_NAME_TOO_LONG 174

IOERR_NO_OLD_FILE 175
IOERR_NEW_FILE_EXISTS 176
IOERR_CANNOT_REWIND 177
IOERR_BACKSPACE_FAILED 178
IOERR_NOT_DIRECT 179
IOERR_BAD_REC 180
IOERR_BAD_INPUT_LOGICAL 181
IOERR_NO_INPUT_LOGICAL 182
IOERR_CANNOT_OPEN 183
IOERR_NO_AMPERSAND 184
IOERR_GROUP_NAME_TOO_LONG 185
IOERR_WRONG_NAMELIST 186
IOERR_NAMELIST_BAD_CHAR 187
IOERR_OBJECT_NAME_TOO_LONG 188
IOERR_EXPECTED_EQUALS 189
IOERR_UNKNOWN_OBJECT_NAME 190
IOERR_UNEXPECTED_SUBSCRIPT 191
IOERR_UNEXPECTED_COMPONENT 192
IOERR_COMPONENT_NAME_TOO_LONG 193
IOERR_UNKNOWN_COMPONENT 194
IOERR_ARRAY_OF_ARRAY 195
IOERR_BAD_INTEGER_LITERAL 196
IOERR_EXPECTED_COLON 197
IOERR_ZERO_LENGTH_INPUT 198
IOERR_SUBSTRING_OUT_OF_BOUNDS 199
IOERR_EXPECTED_COMMA 200
IOERR_EXPECTED_RPAREN 201
IOERR_SUBSCRIPT_OUT_OF_RANGE 202
IOERR_ZERO_SIZE_INPUT 203
IOERR_ZERO_STRIDE 204
IOERR_NO_NAMELIST_GROUP_NAME 205
IOERR_INPUT_LIST_TOO_BIG 206
IOERR_RECORD_TOO_SHORT 207
IOERR_CORRUPT_UNFORMATTED_FILE 208
IOERR_TRUNCATE_FAILED 209
IOERR_RW_AFTER_ENDFILE 210
IOERR_INTEGER64_TOO_BIG 211
IOERR_REPLACE_OR_NEW_NEED_FILE 212
IOERR_RECL_LE_ZERO 213
IOERR_END_OF_DIRECT_ACCESS 214
IOERR_REAL_INPUT_OVERFLOW 215
IOERR_DIRECT_POSITION_INCOMPATIBLE 216
IOERR_SCALE_FOLLOWED_BY_REPEAT 217
IOERR_NO_COMMA_SEPARATOR 218
IOERR_NO_COMMA_BEFORE_REPEAT 219
IOERR_SCRATCH_MUST_BE_READWRITE 220
IOERR_NEW_OR_REP_MUSTNT_BE_READ 221
IOERR_OLD_SEQ_MUST_BE_REW_OR_AP 222
IOERR_OPEN_OF_OPEN_FILE 223

6.3 Math Module

6.3.1 Math Constants

This module contains definitions of parameters for the constants π, e, ϕ, and γ. The names of the constants are `pi`, `e`, `phi`, `gamma`, `pi_double`, `e_double`, `phi_double`, and `gamma_double`. An example of its use is

```
program print_pi

   use math_module

   print *, pi_double

end program print_pi
```

6.3.2 The gcd Function

Also in the math module is the elemental function `gcd` that computes the greatest common divisor or two integers or two integer arrays.

```
program test_gcd
   use math_module
   print *, gcd((/432,16/),(/796,48/))
end program test_gcd
```

which prints

```
 4 16
```

The Slatec Library 7

The Slatec library is a collection of mathematical routines developed jointly by Sandia National Laboratories, Los Alamos National Laboratory, and the Air Force Phillips Laboratory, all in New Mexico.

They may be used in an F program as a "built-in" module. Invoke any of the procedures described below from any F program containing the following statement:

```
use slatec_module
```

7.1 Finding Roots in an Interval

```
find_root_in_interval(f, a, b, root, indicator)
```

is a subroutine that searches for a zero of a function *f(x)* between the given values *a* and *b*.

`f` is a function of one variable. `a` and `b` specify the interval in which to find a root of `f`. `root` is the computed root of `f` in the interval `a` to `b`. These are all type default real.

`indicator` is an optional default integer argument—if it is zero, the answer should be reliable; if it is negative, it is not.

Here is an example using the subroutine `find_root_in_interval`.

```
module function_module

   public :: f

contains

function f(x) result(r)

   real, intent(in) :: x
   real :: r

   r = x**2 - 2.0

end function f

end module function_module

program find_root

   use function_module
   use slatec_module
   real :: root
   integer :: indicator

   call find_root_in_interval&
        (f, 0.0, 2.0, root, indicator)

   if (indicator == 0) then
      print *, "A root is", root
   else
```

```
      print *, "Root not found"
   end if

end program find_root
```

Running this program produces

```
A root is   1.4142114
```

7.2 Finding Roots of a Polynomial

The subroutine

```
find_roots_of_polynomial &
      (coefficients, roots, indicator)
```

accepts the coefficients of a polynomial and finds its roots (values where the polynomial is zero).

`coefficients` is a default real array; the element with the smallest subscript is the constant term, followed by the first degree term, etc. Thus, a reasonable choice is to make the lower bound of `coefficients` 0 so that the subscript matches the power of the coefficient.

`roots` is a complex array with at least as many elements as the degree of the polynomial. The roots of the polynomial will be found in this array after calling `find_roots_of_polynomial`.

`indicator` is a default integer optional argument; if it is negative, the solution is not reliable. In particular, if indicator is –1, a solution was not found in 30 iterations, if it is –2, the high-order coefficient is 0, if it is –3 or –4, the argument array sizes are not appropriate; if it is –5, allocation of a work array was not successful.

Here is a simple example that computes the roots of $x^2 - 3x + 2 = 0$.

```
program poly_roots

   use slatec_module

   complex, dimension(2) :: roots
   integer :: ind

   call find_roots_of_polynomial &
         ( (/ 2.0, -3.0, 1.0 /), roots, ind)
   print *, "Indicator", ind
   print *, "Roots", roots

end program poly_roots
```

Running the program finds the roots 1 and 2.

```
 Indicator 0
 Roots (2.00000,0.00000E+00) (1.00000,0.00000E+00)
```

7.3 Computing a Definite Integral

```
integrate(f, a, b, value, tolerance, indicator)
```

is a general purpose subroutine for evaluation of one-dimensional integrals of user defined functions. `integrate` will pick its own points for evaluation of the integrand and these will vary from problem to problem. Thus, it is not designed to integrate over data sets.

`f` must be a function with a single argument. `a` and `b` are the limits of integration. tolerance is an optional requested error tolerance; if it is not present, 10^{-3} is used. value is the calculated integral. These are all type default real.

If the returned value of the optional default integer argument `indicator` is negative, the result is probably not correct. A positive value of `indicator` represents the number of integrand evaluations needed.

```
module sine_module

public :: sine

contains

function sine (x) result (sine_result)

   intrinsic :: sin
   real, intent (in) :: x
   real :: sine_result

   sine_result = sin (x)

end function sine

end module sine_module

program integration

use sine_module
use slatec_module
real :: answer
integer :: indicator

call integrate(sine, a=0.0, b=3.14159, &
               value=answer, tolerance=1.0e-5, &
               indicator=indicator)
print *, "Indicator is", indicator
print *, "Value of integral is", answer

end program integration
```

Running this program produces

```
 Indicator is 25
 Value of integral is   2.0000000
```

7.4 Special Functions

`ln_gamma(x)` is a function that returns the natural logarithm of the gamma function for positive real values of `x`. `asinh(x)`, `acosh(x)`, and `atanh(x)` return the inverse hyperbolic function values. The program

```
program test_gamma
use slatec_module
print *, "4! = ", exp(ln_gamma (5.0))
end program test_gamma
```

produces

```
 4! =   24.0000000
```

7.5 Solving Linear Equations

```
solve_linear_equation(a, x, b, indicator)
```

is a subroutine that solves a set of linear equations `ax = b`. `a` must be an $n \times n$ two-dimensional array of coefficients. `b` must be a size n array of constants. `x` must be a size n array to hold the solution. These all must be type default real.

`indicator` is an optional intent out default integer value—if indicator is –1, the arguments are not of the correct sizes; if it is –2, allocation of a work array was not successful; other negative values indicate that the solution is not reliable; a positive value indicates approximately the number of correct digits in the solution, except that a value of 75 indicates that the solution `x` is zero.

```
program solve_linear

use slatec_module
real, dimension(3,3) :: a
real, dimension(3) :: b, x
integer :: i, j, indicator

forall (i=1:3,j=1:3)
   a(i,j)=i+j
end forall
a(3,3) = -a(3,3)

b = (/20,26,-4/)

call solve_linear_equation(a, x, b, indicator)

print*, "Indicator", indicator
print*, "Solution", x

end program solve_linear
```

```
Indicator 6
Solution   1.0000000   2.0000000   3.0000000
```

7.6 Differential Equations

```
solve_ode(f, x0, xf, y0, yf, tolerance, indicator)
```

is a subroutine that solves an ordinary differential equation

$$\frac{du}{dx} = f(x, u)$$

using a fifth-order Runge-Kutta method.

`f` must be a function of two variables. `x0` is the initial value of x. `y0` is the initial value of y. `xf` is the final value of x. `yf` is the final solution value of `y`. `tolerance` is an optional requested tolerance; if not present 1.0^{-3} is used. All of these are type default real.

`indicator` is an optional default integer value—if it is negative, the solution is not reliable; a value of 2 indicates success.

Here is a simple example with $f(x, u) = -0.01y$, $x_0 = 0$, $y_0 = 100$, and $x_f = 100$.

```
module f_module

   public :: f

contains
```

```
function f(x, y) result(r)

   real, intent(in) :: x, y
   real :: r

   r = -0.01 * y

end function f

end module f_module

program test_ode

use slatec_module
use f_module

real :: x0 = 0.0, xf = 100.0, &
        y0 = 100.0, yf

call solve_ode (f, x0, xf, y0, yf)

print *, "Answer is", yf

end program test_ode
```

Running the program produces

```
 Answer is  36.7878761
```

Defined Data Types 8

There are several modules available to the F programmer that define new data types and a selection of operations on those types. The varying string and multiple precision modules are written in Fortran, but interfaces are provided to allow them to be used in F programs. The code for the big integers, the rationals, the quaternions, and the Roman numerals is all written in F; the source for each of these modules is available in the `src` directory to provide information about the modules and examples of how to build these abstract data types.

8.1 Varying Length Strings

The ISO varying string module provides the type `iso_varying_string` with the operations you would expect to have for character string manipulations (concatenation, input/output, character intrinsic functions). Unlike F character variables, a varying string variable has a length that changes as different values are assigned to the variable. Here is a simple program illustrating these features.

```
program string
   use iso_varying_string
   type(varying_string) :: s
   call get(string=s)
   s = s // s
   call put(string=s)
   print *, len(s)
end program string
```

The following lines show what happens when the program is compiled and run.

```
$ F string.f95
$./a.exe
A nice string.
A nice string.A nice string. 28
```

The current version of the source code is from Rich Townsend and has been modified slightly so that we have an F conformant version. This program is in the F source code directory.

8.2 Big Integers

The `big_integer` data type can represent very large nonnegative integers. The representation of a big integer is a structure with one component that is an array of ordinary F integers. In this version, the largest integer that can be represented is fixed, but the size is specified by a parameter that can be changed. The module may then be recompiled. The source for this module is in the `examples` directory of the F distribution. All of the intrinsic operations and functions for intrinsic F integers are available for big integers.

```
program factors
   use big_integer_module
   type(big_integer) :: b, n, s

   b = “9876543456789”

   n = 2
   call check_factor()
```

```
   s = sqrt(b)
   n = 3
   do
      if (n > s) exit
      call check_factor()
      n = n + 2
   end do
   if (b /= 1) then
      call print_big(b)
      print *
   end if


contains

   subroutine check_factor()
      do
         if (modulo(b, n) == 0) then
            call print_big(n)
            print *
            b = b / n
            s = sqrt(b)
         else
            exit
         end if
      end do
   end subroutine check_factor

end program factors
```

Running the program produces

```
3
3
3
3
17
97
1697
43573
```

8.3 High Precision Reals

8.3.1 The MP Module

This module provides the capability of computing with large precision real values. It was written by David Bailey of Lawrence Berkeley National Laboratory. A description of the module is in the files `mp.ps` and `mp.pdf` in the `doc` directory. More information may be found at `http://www.nersc.gov/~dhbailey/mpdist/mpdist.html`. Here is a simple example of its use.

```
program mp
   use mp_module
   type(mp_real) :: pi

   call mpinit()
   pi = 4.0 * atan(mpreal(1.0))
```

```
   call mpwrite(6, pi)
end program mp
```

The result printed consists of quite a few digits of π.

```
10 ^      0 x  3.14159265358979323846264338327950288419716939937510582097,
```

8.3.2 The XP Module

This module also provides the capability of computing with large precision real values. It was written by David Smith. A description of the module is in the file `xp.txt` in the `doc` directory. Here is a simple example of its use.

```
program test_xp
   use xp_real_module
   type (xp_real) :: x, y
   x = 1.0
   y = 4.0
   call xp_print(y*atan(x))
end program test_xp
```

```
       3.14159265358979323846264338279502884197E+0
```

8.4 Rationals

A module to compute with rational numbers is provided by Dan Nagle of Purple Sage Computing Solutions, Inc. Some details are provided in the file `rationals.txt` in the `doc` directory. Here is a simple example.

```
program test_rationals
   use rationals_module
   type(rational) :: r1, r2

   r1 = (/3, 4/)
   r2 = (/5, 6/)
   r1 = r1 + r2
   print *, real(r1)
end program test_rationals
```

```
   1.5833333333333333
```

8.5 Quaternions

The quaternions module was written by David Arnold of the College of the Redwoods. The only documentation is the source file `quaternions_module.f95` for the in the `src` directory. There is some information about quaternions in the file `quaternions.pdf` in the `doc` directory and the original article about quaternions presented by William Hamilton in 1843 can be found at `http://www.maths.tcd.ie/pub/HistMath/People/Hamilton/Quatern2/Quatern2.html`. Here is an example.

```
program Quaternions
  use Quaternions_module
  type(quaternion) :: u, v
  u=quaternion(1,2,3,4)
  v=quaternion(5,6,7,8)
  call quaternion_print(u+v)
  print *, 3+4
  print *
  call quaternion_print(u-v)
  print *, 3-4
```

```
  print *
  call quaternion_print(3.0*u)
  call quaternion_print(u*v)
  print *, 3*4
  print *
  call quaternion_print(conjg(u))
  print *, conjg((3,4))
  print *
  print *, (abs(u))
  print *, abs((3,4))
end program Quaternions
```

```
(    6.000000    8.000000   10.000000   12.000000)
 7

(   -4.000000   -4.000000   -4.000000   -4.000000)
 -1

(    3.000000    6.000000    9.000000   12.000000)
(  -60.000000   12.000000   38.000000   24.000000)
 12

(    1.000000   -2.000000   -3.000000   -4.000000)
 (3.0000000,-4.0000000)

   5.4772258
   5.0000000
```

8.6 Roman Numerals

This module to compute with Roman numbers was written by Jeanne Martin, former convenor of the international Fortran standards committee and an author of *The Fortran 95 Handbook*. The only documentation available is in the source file in the `src` directory.

```
program test_roman
use roman_numerals_module
implicit none

type(roman) :: r
integer :: i

write (unit=*, fmt="(a)") "Integer  Roman Number"
do i = 1900, 2000
  r = i
  write (unit=*, fmt="(/, tr4, i4, tr2)", advance = "NO") i
  call print_roman (r)
end do
write (unit=*, fmt="(/)")

end program test_roman
```

Here is the result of running the program.

```
 Integer  Roman Number

    1900  MCM
```

```
1901  MCMI
1902  MCMII
1903  MCMIII
1904  MCMIV
1905  MCMV
1906  MCMVI
 . . .
1998  MCMXCVIII
1999  MCMXCIX
2000  MM
```

HPF Library Module 9

This section contains a brief description of the HPF_LIBRARY module provided by High Performance Fortran.

In the following tables, arguments with names in italics are optional.

For full information on these functions refer to the HPF Language Specification.

Note that use hpf_library is needed to access any of these functions and the program must be compiled with the -hpf option.

9.1 Example

```
program p
use hpf_library
print *, sum_prefix((/2,4,3,9/))
end program p
```

9.2 Elemental Functions

Function (Arguments)	Description
LEADZ(I)	Number of leading zero bits in the internal representation.
POPCN(I)	Number of one bits in the internal representation.
POPPAR(I)	1 if POPCNT(I) is odd, 0 otherwise

9.3 Reduction Functions

These are analogous to the SUM, PRODUCT, et al intrinsic functions.

Function (Arguments)	Reduction Operation
IALL(ARRAY, *DIM, MASK*)	IAND intrinsic
IANY(ARRAY, *DIM, MASK*)	IOR intrinsic
IPARITY(ARRAY, *DIM, MASK*)	IEOR intrinsic
PARITY(MASK, *DIM*)	.NEQV.

9.4 Scan Functions

Each scan function produces an array of the same shape as its first argument—MASK argument for logical functions, the ARRAY argument for the others). Scanning combines the "current" value with each array element of this argument according to the scan function; e.g., SUM_PREFIX and SUM_SUFFIX add each array element. There are two scan functions for each operation; one forwards (XXX_PREFIX) and one backwards (XXX_SUFFIX).

Note that by definition, XXX_SUFFIX(A(1:N)) is equal to XXX_PREFIX(A(N:1:-1)). Note also that the initial "current" value depends on the function: it is 0 for COUNT/IANY/IPARITY/SUM scans, 1 for

PRODUCT scan, –1 for IALL scan, true for ALL, false for ANY/PARITY, -HUGE(ARRAY) for MAXVAL, HUGE(ARRAY) for MINVAL and the value of the first element of the array for COPY.

E.g., SUM_PREFIX((/ 1,2,3,4 /)) is equal to [1,3,6,10].

If the DIM argument is not present, the whole array is scanned in array element order. With a DIM argument, the array is scanned in that dimension in parallel.

If the SEGMENT argument is present, the "current" value is reset each time the element of SEGMENT corresponding to the scan changes value (from the last element of SEGMENT considered).

If the MASK optional argument is present (for non-logical-valued functions), only those elements of ARRAY for which the corresponding element of MASK is true contribute to the value of the scan.

If the EXCLUSIVE argument is present and true, the scan combining operation is performed after deciding the value for each element, thus shifting the value of the scan function one place and inserting, at the beginning, the initial "current" value. E.g., SUM_PREFIX((/1,2,3/), EXCLUSIVE=.TRUE.) is equal to [0,1,3].

Function (Arguments)	Combining Operation
ALL_PREFIX(MASK,*DIM*,*SEGMENT*,*EXCLUSIVE*)	.AND.
ANY_PREFIX(MASK,*DIM*,*SEGMENT*,*EXCLUSIVE*)	.OR.
COPY_PREFIX(ARRAY,*DIM*,*SEGMENT*)	Copy
COUNT_PREFIX(MASK,*DIM*,*SEGMENT*,*EXCLUSIVE*)	Count true values
IALL_PREFIX(ARRAY,*DIM*,*MASK*,*SEGMENT*,*IEXCLUSIVE*)	IAND intrinsic
IANY_PREFIX(ARRAY,*DIM*,*MASK*,*SEGMENT*,*EXCLUSIVE*)	IOR intrinsic
IPARITY_PREFIX(ARRAY,*DIM*,*MASK*,*SEGMENT*,*EXCLUSIVE*)	IOER intrinsic
MAXVAL_PREFIX(ARRAY,*DIM*,*MASK*,*SEGMENT*,*EXCLUSIVE*)	MAXVAL intrinsic
MINVAL_PREFIX(ARRAY,*DIM*,*MASK*,*SEGMENT*,*EXCLUSIVE*)	MINVAL intrinsic
PARITY_PREFIX(MASK,*DIM*,*SEGMENT*,*EXCLUSIVE*)	.NEQV.
PRODUCT_PREFIX(ARRAY,*DIM*,*MASK*,*SEGMENT*,*EXCLUSIVE*)	Multiplication
SUM_PREFIX(ARRAY,*DIM*,*MASK*,*SEGMENT*,*EXCLUSIVE*)	Addition

9.5 Scatter Functions

Each scatter function produces an array of the same shape as the BASE argument. The default value of each element of the result is the same as that of the corresponding element of BASE.

The elements of ARRAY (or MASK, for logical functions) are scattered to selected elements of the result, where they are combined using the specified operation (see the relevant scan or reduction functions).

Each element is scattered by taking its index values and using the values from the corresponding INDXn arguments to index into the result. Note that the number of INDXn arguments must be the same as the rank of ARRAY (or MASK). The shape of ARRAY (or MASK) need not be the same as BASE.

If the optional MASK is present (only for non-logical functions), only those elements of ARRAY for which the corresponding elements of MASK are true are so scattered.

Function (Arguments)
ALL_SCATTER (MASK, BASE, INDX1, ..., INDXn)
ANY_SCATTER (MASK, BASE, INDX1, ..., INDXn)

```
COPY_SCATTER (ARRAY, BASE, INDX1, ..., INDXn, MASK)
COUNT_SCATTER (MASK, BASE, INDX1, ..., INDXn)
IALL_SCATTER (ARRAY, BASE, INDX1, ..., INDXn, MASK)
IANY_SCATTER (ARRAY, BASE, INDX1, ..., INDXn, MASK)
IPARITY_SCATTER (ARRAY, BASE, INDX1, ..., INDXn, MASK)
MAXVAL_SCATTER (ARRAY, BASE, INDX1, ..., INDXn, MASK)
MINVAL_SCATTER (ARRAY, BASE, INDX1, ..., INDXn, MASK)
PARITY_SCATTER (MASK, BASE, INDX1, ..., INDXn)
PRODUCT_SCATTER (ARRAY, BASE, INDX1, ..., INDXn, MASK)
SUM_SCATTER (ARRAY, BASE, INDX1, ..., INDXn, MASK)
```

9.6 Sorting Functions

These functions produce permutation indices which can be used to index an array in a sorted sequence.

Function (Arguments)	Description
GRADE_DOWN(ARRAY, *DIM*)	Descending sort as array indices
GRADE_UP(ARRAY, *DIM*)	Ascending sort as array indices

9.7 Inquiry Functions

These functions inquire about the actual alignment, distribution and template values used at runtime in a multi-processor environment. Since F is a single-processor implementation of HPF, these functions do not return very useful information.

Function	(Arguments)
HPF_ALIGNMENT	(ALIGNEE, *LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES*)
HPF_DISTRIBUTION	(DISTRIBUTEE, *AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE*)
HPF_TEMPLATE	(ALIGNEE, *TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED, DYNAMIC*)

F API to POSIX Facilities 10

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

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

Some of the code and descriptions of these facilities involve Fortran syntax and features that are not in F. The code is compiled with the NAGWare f95 Fortran compiler and made available in the F system as libraries and module files, which can be linked and accessed with F programs.

In spite of the names of these modules, many work on Windows as well as Linux and Unix; however, not all facilities work on all systems.

10.1 Examples

Before providing the details, here are some examples of the use of these features.

10.1.1 Calling system and sleep

```
program p
  use f90_unix_proc
  call system("ls -l")
  call sleep(10)
  print *,"goodnight"
end program p
```

10.1.2 Command-Line Arguments and Environment Variables

```
program test_args

use f90_unix_env

integer :: n_args, n
character(len=99) :: arg, shell

call getenv("SHELL",shell)
print *, trim(shell)

n_args = iargc()
do n = 0, n_args
   call getarg(n, arg)
   print *, n, trim(arg)
end do

end program test_args
```

Here is a sample execution of the program.

```
[walt@sonora Examples]$ F args.f95

[walt@sonora Examples]$ ./a.out aaaa bbbbb ccc
 /bin/bash
 0 ./a.out
 1 aaaa
 2 bbbbb
 3 ccc
```

10.2 f90_unix_dir

Interface to directory-related facilities in ISO/IEC 9945-1:1990, sections 5.2: Working Directory, 5.3.3 Set File Creation Mask, 5.3.4 Link to a File, 5.4 Special File Creation and 5.5 File Removal.

Error handling is described in f90_unix_errno. Note that for procedures with an optional ERRNO argument, if an error occurs and ERRNO is not present, the program will be terminated.

10.2.1 Parameters

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

This is the integer kind used to represent file permissions. Masks and values for these permissions are defined in module f90_unix_file.

10.2.2 Procedures

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

Sets the current working directory to PATH. If ERRNO is present it receives the error status.

Possible error conditions include EACCES, ENAMETOOLONG, ENOTDIR ENOENT.

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

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

If neither PATH nor LENPATH is present, error EINVAL is raised. If the path to current working directory cannot be searched, error EACCES is raised. If PATH is present and LENPATH is not present, and PATH is shorter than the current working directory name, error ERANGE is raised (see f90_unix_errno).

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

Creates a new link (named NEW) for an existing file (named EXISTING).

Possible errors include EACCES, EEXIST, EMLINK, ENAMETOOLONG, ENOENT, ENOSPC, ENOTDIR, EPERM, EROFS, EXDEV (see f90_unix_errno).

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

Creates a new directory with name PATH and mode MODE.

Possible errors include EACCES, EEXIST, EMLINK, ENAMETOOLONG, ENOENT, ENOSPC, ENOTDIR, EROFS (see f90_unix_errno).

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

Creates a new FIFO special file with name PATH and mode MODE.

Possible errors include EACCES, EEXIST, ENAMETOOLONG, ENOENT, ENOSPC, ENOTDIR, EROFS (see f90_unix_errno).

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

Changes the name of the file OLD to NEW. Any existing file NEW is first removed.

Possible errors include EACCES, EBUSY, EEXIST, ENOTEMPTY, EINVAL, EISDIR, ENAMETOOLONG, EMLINK, ENOENT, ENOSPC, ENOTDIR, EROFS, EXDEV (see f90_unix_errno).

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

Removes the directory PATH.

Possible errors include EACCES, EBUSY, EEXIST, ENOTEMPTY, ENAMETOOLONG, ENOENT, ENOTDIR, EROFS (see f90_unix_errno).

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

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

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

Deletes the file PATH.

Possible errors include EACCES, EBUSY, ENAMETOOLONG, ENOENT, ENOTDIR, EPERM, EROFS (see f90_unix_errno).

10.3 f90_unix_dirent

Interface to directory-reading facilities from ISO/IEC 9945-1:1990 section 5.1.2: Directory Operations.

Error handling is described in f90_unix_errno. Note that for procedures with an optional ERRNO argument, if an error occurs and ERRNO is not present, the program will be terminated.

10.3.1 Procedures

```
SUBROUTINE CLOSEDIR(DIRUNIT,ERRNO)
INTEGER,INTENT(IN) :: DIRUNIT
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Closes a directory stream that was opened by OPENDIR.

If DIRUNIT does not refer to an open directory stream, error EBADF (see f90_unix_errno) is raised.

```
SUBROUTINE OPENDIR(DIRNAME,DIRUNIT,ERRNO)
CHARACTER*(*),INTENT(IN) :: DIRNAME
INTEGER,INTENT(OUT) :: DIRUNIT
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Opens a directory stream, returning a handle to it in DIRUNIT.

Possible errors include EACCES, ENAMETOOLONG, ENOENT, ENOTDIR, EMFILE and ENFILE (see f90_unix_errno).

```
SUBROUTINE READDIR(DIRUNIT,NAME,LENNAME,ERRNO)
INTEGER,INTENT(IN) :: DIRUNIT
CHARACTER*(*),INTENT(OUT) :: NAME
INTEGER,INTENT(OUT) :: LENNAME
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Reads the first/next directory entry. The name of the file is placed into NAME, blank-padded or truncated as appropriate if the length of the file name differs from LEN(NAME). The length of the file name is placed in LENNAME. Note: The maximum file name length is available from SYSCONF; inquiry SC_NAME_MAX.

If there are no more directory entries, NAME is undefined and LENNAME is negative.

If DIRUNIT is not a directory stream handle produced by OPENDIR, or has been closed by CLOSEDIR, error EBADF (see f90_unix_errno) is raised.

```
SUBROUTINE REWINDDIR(DIRUNIT,ERRNO)
INTEGER,INTENT(IN) :: DIRUNIT
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Rewinds the directory stream so that the next call to READDIR on that stream will return the name of the first file in the directory.

10.4 f90_unix_env

Interface to the facilities from ISO/IEC 9945-1:1990 section 4: Process Environment, plus gethostname from 4.3BSD.

Error handling is described in f90_unix_errno. Note that for procedures with an optional ERRNO argument, if an error occurs and ERRNO is not present, the program will be terminated.

10.4.1 Parameters

```
INTEGER,PARAMETER :: CLOCK_TICK_KIND
```

The integer kind used for clock ticks (see TIMES).

```
INTEGER,PARAMETER :: LONG_KIND
```

The integer kind corresponding to the C type long. This is used for the type of one of the arguments of SYSCONF.

```
INTEGER,PARAMETER :: SC_STDIN_UNIT, SC_STDOUT_UNIT, &
   SC_STDERR_UNIT, SC_ARG_MAX, SC_CHILD_MAX, &
   SC_CLK_TCK, SC_JOB_CONTROL, SC_OPEN_MAX, &
   SC_NGROUPS_MAX, SC_SAVED_IDS, &
   SC_STREAM_MAX, SC_TZNAME_MAX, SC_VERSION
```

Values used as arguments to SYSCONF. The following table describes the returned information from SYSCONF; this is not the value of the SC_* constant.

SC_STDIN_UNIT — The logical unit number for standard input (READ with no io-unit, READ(*,...)).

SC_STDOUT_UNIT	The logical unit number for standard output (PRINT, WRITE(*,...)).
SC_STDERR_UNIT	The logical unit number on which errors are reported.
SC_ARG_MAX	Maximum length of arguments for the EXEC functions, in bytes, including environment data.
SC_CHILD_MAX	Maximum number of simultaneous processes for a single user.
SC_CLK_TCK	Number of clock ticks per second. (This is the same value returned by the CLK_TCK function.)
SC_JOB_CONTROL	Value available only if job control is supported by the operating system.
SC_NGROUPS_MAX	Maximum number of simultaneous supplementary group IDs per process.
SC_OPEN_MAX	Maximum number of files open simultaneously by a single process.
SC_SAVED_IDS	Value available only if each process has a saved set-uid and set-gid.
SC_STREAM_MAX	Maximum number of logical units that can be simultaneously open. Not always available.
SC_TZNAME_MAX	Maximum number of characters for the name of a time zone.
SC_VERSION	Posix version number. This will be 199009 if the underlying operating system's C interface conforms to ISO/IEC 9945-1:1990.

```
INTEGER,PARAMETER :: TIME_KIND
```

The integer kind used for holding date/time values (see TIME).

10.4.2 Derived Types

```
TYPE TMS
   INTEGER(CLOCK_TICK_KIND) UTIME,STIME,CUTIME,CSTIME
END TYPE
```

Derived type holding CPU usage time in clock ticks. UTIME and STIME contain CPU time information for a process, CUTIME and CSTIME contain CPU time information for its terminated child processes. In each case this is divided into user time (UTIME, CUTIME) and system time (STIME, CSTIME).

```
TYPE UTSNAME
   CHARACTER*(...) SYSNAME, NODENAME, RELEASE, &
                   VERSION, MACHINE
END TYPE
```

Derived type holding data returned by UNAME. Note that the character length of each component is fixed, but may be different on different systems. The values in these components are blank-padded (if short) or truncated (if long). For further information see ISO/IEC 9945-1:1990.

10.4.3 Procedures

```
PURE INTEGER (KIND=CLOCK_TICK_KIND) FUNCTION CLK_TCK()
```

Returns the number of clock ticks in one second of CPU time (see TIMES).

```
PURE SUBROUTINE CTERMID(S,LENS)
CHARACTER*(*),OPTIONAL,INTENT(OUT) :: S
INTEGER,OPTIONAL,INTENT(OUT) :: LENS
```

If present, LENS is set to the length of the filename of the controlling terminal. If present, S is set to the filename of the controlling terminal. If S is longer than the filename of the controlling terminal it is padded with blanks. If S is shorter it is truncated; it is the user's responsibility to check the value of LENS to detect such truncation.

If the filename of the controlling terminal cannot be determined for any reason LENS (if present) will be set to zero and S (if present) will be blank.

```
SUBROUTINE GETARG(K,ARG,LENARG,ERRNO)
INTEGER,INTENT(IN) :: K
```

```
CHARACTER*(*),OPTIONAL,INTENT(OUT) :: ARG
INTEGER,OPTIONAL,INTENT(OUT) :: LENARG, ERRNO
```

Accesses command-line argument number K, where argument zero is the program name. If ARG is present, it receives the argument text (blank-padded or truncated as appropriate if the length of the argument differs from that of ARG). If ARGLEN is present, it receives the length of the argument. If ERRNO is present, it receives the error status.

Note that if K is less than zero or greater than the number of arguments (returned by IARGC) error EINVAL (see f90_unix_errno) is raised.

```
PURE INTEGER FUNCTION GETEGID()
```

Returns the effective group number of the calling process.

```
SUBROUTINE GETENV(NAME,VALUE,LENVALUE,ERRNO)
CHARACTER*(*),INTENT(IN) :: NAME
CHARACTER*(*),OPTIONAL,INTENT(OUT) :: VALUE
INTEGER,OPTIONAL,INTENT(OUT) :: LENVALUE, ERRNO
```

Accesses the environment variable named by NAME. If VALUE is present, it receives the text value of the variable (blank-padded or truncated as appropriate if the length of the value differs from that of VALUE). If LENVALUE is present, it receives the length of the value. If ERRNO is present, it receives the error status.

If there is no such variable, error EINVAL (see f90_unix_errno) is raised. Other possible errors include ENOMEM.

```
PURE INTEGER FUNCTION GETEUID()
```

Returns the effective user number of the calling process.

```
PURE INTEGER FUNCTION GETGID()
```

Returns the group number of the calling process.

```
SUBROUTINE GETGROUPS(GROUPLIST,NGROUPS,ERRNO)
INTEGER,OPTIONAL,INTENT(OUT) :: GROUPLIST(:), &
      NGROUPS, ERRNO
```

Retrieves supplementary group number information for the calling process. If GROUPLIST is present, it is filled with the supplementary group numbers. If NGROUPS is present, it receives the number of supplementary group numbers. If ERRNO is present, it receives the error status.

If GROUPLIST is too small to contain the complete list of supplementary group numbers, error EINVAL (see f90_unix_errno) is raised. The maximum number of supplementary group numbers can be found using SYSCONF (inquiry SC_NGROUPS_MAX); alternatively, CALL GETGROUPS(NGROUPS=N) will reliably return the actual number in use.

```
PURE SUBROUTINE GETHOSTNAME(NAME,LENNAME)
CHARACTER*(*),OPTIONAL,INTENT(OUT) :: NAME
INTEGER,OPTIONAL,INTENT(OUT) :: LENNAME
```

This provides the functionality of 4.3BSD's gethostname. If NAME is present it receives the text of the standard host name for the current processor, blank-padded or truncated if appropriate. If LENNAME is present it receives the length of the host name. If no host name is available LENNAME will be zero.

```
PURE SUBROUTINE GETLOGIN(S,LENS)
CHARACTER*(*),OPTIONAL,INTENT(OUT) :: S
INTEGER,OPTIONAL,INTENT(OUT) :: LENS
```

Accesses the user name (login name) associated with the calling process. If S is present, it receives the text of the name (blank-padded or truncated as appropriate if the length of the login name differs from that of S). If LENS is present, it receives the length of the login name.

```
PURE INTEGER FUNCTION GETPGRP()
```

Returns the process group number of the calling process.

```
PURE INTEGER FUNCTION GETPID()
```

Returns the process number of the calling process.

```
PURE INTEGER FUNCTION GETPPID()
```

Returns the process number of the parent of the calling process.

```
PURE INTEGER FUNCTION GETUID()
```

Returns the user number of the calling process.

```
PURE INTEGER FUNCTION IARGC()
```

Returns the number of command-line arguments. This will be –1 if even the program name is unavailable.

```
SUBROUTINE ISATTY(LUNIT,ANSWER,ERRNO)
INTEGER,INTENT(IN) :: LUNIT
LOGICAL,INTENT(OUT) :: ANSWER
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

ANSWER receives the value true if and only if logical unit LUNIT is connected to a terminal.

If LUNIT is not a valid unit number or is not connected to any file, error EBADF (see f90_unix_errno) is raised.

```
SUBROUTINE SETGID(GID,ERRNO)
INTEGER,INTENT(IN) :: GID
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Sets the group number of the calling process to GID. For full details refer to section 4.2.2 of ISO/IEC 9945-1:1990.

If GID is not a valid group number, error EINVAL (see f90_unix_errno) is raised. If the process is not allowed to set the group number to GID, error EPERM is raised.

```
SUBROUTINE SETPGID(PID,PGID,ERRNO)
INTEGER,INTENT(IN) :: PID, PGID
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Sets the process group number of process PID (or, if PID is zero, the calling process) to PGID. For full details refer to section 4.3.3 of ISO/IEC 9945-1:1990.

Possible errors include EACCES, EINVAL, ENOSYS, EPERM, ESRCH (see f90_unix_errno).

```
SUBROUTINE SETSID(SID,ERRNO)
INTEGER,INTENT(IN) :: SID
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Creates a session and sets the process group number of the calling process. For full details refer to section 4.3.2 of ISO/IEC 9945-1:1990.

Possible errors include EPERM (see f90_unix_errno).

```
SUBROUTINE SETUID(UID,ERRNO)
INTEGER,INTENT(IN) :: UID
```

```
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Sets the user number of the calling process to UID. For full details refer to section 4.2.2 of ISO/IEC 9945-1:1990. If UID is not a valid group number, error EINVAL (see f90_unix_errno) is raised. If the process is not allowed to set the user number to UID, error EPERM is raised.

```
SUBROUTINE SYSCONF(NAME,VAL,ERRNO)
INTEGER,INTENT(IN) :: NAME
INTEGER(LONG_KIND),INTENT(OUT) :: VAL
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Returns the value of a system configuration variable. The variables are named by integer parameters defined in this module, and are described in the Parameter section.

If NAME is not a valid configuration variable name, error EINVAL (see f90_unix_errno) is raised.

```
SUBROUTINE TIME(ITIME,ERRNO)
INTEGER(TIME_KIND),INTENT(OUT) :: ITIME
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

ITIME receives the operating system date/time in seconds since the epoch.

```
INTEGER(KIND=CLOCK_TICK_KIND) FUNCTION TIMES(BUFFER)
TYPE(TMS),INTENT(OUT) :: BUFFER
```

This function returns the elapsed real time in clock ticks since an arbitrary point in the past, or –1 if the function is unavailable. BUFFER is filled in with CPU time information for the calling process and any terminated child processes.

If this function returns zero the values in BUFFER will still be correct but the elapsed-time timer was not available.

```
SUBROUTINE TTYNAME(LUNIT,S,LENS,ERRNO)
INTEGER,INTENT(IN) :: LUNIT
CHARACTER*(*),OPTIONAL,INTENT(OUT) :: S
INTEGER,OPTIONAL,INTENT(OUT) :: LENS, ERRNO
```

Accesses the name of the terminal connected to logical unit LUNIT. If S is present, it receives the text of the terminal name (blank-padded or truncated as appropriate, if the length of the terminal name differs from that of S). If LENS is present, it receives the length of the terminal name. If ERRNO is present, it receives the error status.

If LUNIT is not a valid logical unit number, or is not connected, error EBADF (see f90_unix_errno) is raised.

```
SUBROUTINE UNAME(NAME,ERRNO)
TYPE(UTSNAME),INTENT(OUT) :: NAME
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Returns information about the operating system in NAME.

10.5 f90_unix_errno

The facilities in this module interface to ISO/IEC 9945-1:1990 section 2.4: Error Numbers.

10.5.1 Overview

Many procedures provided by the f90_unix_* modules have an optional ERRNO argument which is declared:

```
INTEGER,OPTIONAL,INTENT(OUT)
```

If this argument is provided it receives the error status from the procedure; zero indicates successful completion, otherwise it will be a non-zero error code, usually one of the ones listed in this module.

If the ERRNO argument is omitted and an error condition is raised, the program will be terminated with an informative error message.

If a procedure has no ERRNO argument it indicates that no error condition is possible-the procedure always succeeds.

10.5.2 Parameters

All parameters are of type default integer. The following table lists the error message typically associated with each error code; for full details see ISO/IEC 9945-1:1990, either section 2.4 or the appropriate section for the function raising the error.

E2BIG	Arg list too long
EACCES	Permission denied
EAGAIN	Resource temporarily unavailable
EBADF	Bad file descriptor
EBUSY	Resource busy
ECHIL	No child process
EDEADLK	Resource deadlock avoided
EDOM	Domain error
EEXIST	File exists
EFAULT	Bad address
EFBIG	File too large
EINTR	Interrupted function call
EINVAL	Invalid argument
EIO	Input/Output error
EISDIR	Is a directory
EMFILE	Too many open files
EMLINK	Too many links
ENAMETOOLONG	Filename too long
ENFILE	Too many open files in system
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOLCK	No locks available
ENOMEM	Not enough space
ENOSPC	No space left on device
ENOSYS	Function not implemented
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTTY	Inappropriate I/O control operation
ENXIO	No such device or address
EPERM	Operation not permitted
EPIPE	Broken pipe
ERANGE	Result too large
EROFS	Read-only file system
ESPIPE	Invalid seek
ESRCH	No such process
EXDEV	Improper link

10.6 f90_unix_file

Interface to facilities from ISO/IEC 9945-1:1990 section 5.6: File Characteristics.

Error handling is described in f90_unix_errno. Note that for procedures with an optional ERRNO argument, if an error occurs and ERRNO is not present, the program will be terminated.

10.6.1 Parameters

```
INTEGER,PARAMETER :: F_OK
```

Flag for requesting file existence check (see ACCESS).

```
USE F90_UNIX_DIR,ONLY:MODE_KIND
```

See f90_unix_errno for a description of this parameter.

```
INTEGER,PARAMETER :: R_OK
```

Flag for requesting file readability check (see ACCESS).

```
INTEGER(MODE_KIND),PARAMETER :: S_IRGRP
```

File mode bit indicating group read permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IROTH
```

File mode bit indicating other read permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IRUSR
```

File mode bit indicating user read permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IRWXG
```

Mask to select the group accessibility bits from a file mode (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IRWXO
```

Mask to select the other accessibility bits from a file mode (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IRWXU
```

Mask to select the user accessibility bits from a file mode (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_ISGID
```

File mode bit indicating that the file is set-group-ID (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_ISUID
```

File mode bit indicating that the file is set-user-ID (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IWGRP
```

File mode bit indicating group write permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IWOTH
```

File mode bit indicating other write permission (see STAT_T).

```
INTEGER,PARAMETER :: S_IWUSR
```

File mode bit indicating user write permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IXGRP
```

File mode bit indicating group execute permission (see STAT_T).

```
INTEGER(MODE_KIND),PARAMETER :: S_IXOTH
```

File mode bit indicating other execute permission (see STAT_T).

```
INTEGER,PARAMETER :: S_IXUSR
```

File mode bit indicating user execute permission (see STAT_T).

```
USE F90_UNIX_ENV,ONLY :: TIME_KIND
```

See f90_unix_errno for a description of this parameter.

```
INTEGER,PARAMETER :: W_OK
```

Flag for requesting file writability check (see ACCESS).

```
INTEGER,PARAMETER :: X_OK
```

Flag for requesting file executability check (see ACCESS).

10.6.2 Derived Types

```
TYPE STAT_T
   INTEGER(MODE_KIND) ST_MODE
   INTEGER(...) ST_INO
   INTEGER(...) ST_DEV
   INTEGER(...) ST_NLINK
   INTEGER(...) ST_UID
   INTEGER(...) ST_GID
   INTEGER(...) ST_SIZE
   INTEGER(TIME_KIND) ST_ATIME, ST_MTIME, ST_CTIME
END TYPE
```

Derived type holding file characteristics. The kind of the integer components indicated by ". . ." (for ST_INO et al) is operating-system dependent.

ST_MODE	File mode (read/write/execute permission for user/group/other, plus set-group-ID and set-user-ID bits).
ST_INO	File serial number.
ST_DEV	ID for the device on which the file resides.
ST_NLINK	The number of links (see f90_unix_dir, LINK operation) to the file.
ST_UID	User number of the file's owner.
ST_GID	Group number of the file.
ST_SIZE	File size in bytes (regular files only).
ST_ATIME	Time of last access.
ST_MTIME	Time of last modification.
ST_CTIME	Time of last file status change.

```
TYPE UTIMBUF
   INTEGER(TIME_KIND) ACTIME, MODTIME
END TYPE
```

Data type holding time values for communication to UTIME. ACTIME is the new value for ST_ATIME, MODTIME is the new value for ST_MTIME.

10.6.3 Procedures

```
PURE SUBROUTINE ACCESS(PATH,AMODE,ERRNO)
CHARACTER*(*),INTENT(IN) :: PATH
INTEGER,INTENT(IN) :: AMODE
INTEGER,INTENT(OUT) :: ERRNO
```

Checks file accessibility according to the value of AMODE; this should be F_OK or a combination of R_OK, W_OK and X_OK. In the latter case the values may be combined by addition or the IOR intrinsic.

The result of the accessibility check is returned in ERRNO, which receives zero for success (i.e., the file exists for F_OK, or all the accesses requested by the R_OK et al combination are allowed) or an error code indicating the reason for access rejection. Possible rejection codes include EACCES, ENAMETOOLONG, ENOENT, ENOTDIR, EROFS (see f90_unix_errno).

If the value of AMODE is invalid, error EINVAL is returned.

Note that most ACCESS inquires are equivalent to an INQUIRE statement; in particular:

```
CALL ACCESS(PATH,F_OK,ERRNO)
```

returns success (ERRNO = 0) if and only if

```
INQUIRE(FILE=PATH,EXIST=LVAR)
```

would set LVAR to true,

```
CALL ACCESS(PATH,R_OK,ERRNO)
```

returns success (ERRNO = 0) if and only if

```
INQUIRE(FILE=PATH,READ=CHVAR)
```

would set CHVAR to YES,

```
CALL ACCESS(PATH,W_OK,ERRNO)
```

returns success (ERRNO = 0) if and only if

```
INQUIRE(FILE=PATH,WRITE=CHVAR)
```

would set CHVAR to YES, and

```
CALL ACCESS(PATH,IOR(W_OK,R_OK),ERRNO)
```

returns success (ERRNO = 0) if and only if

```
INQUIRE(FILE=PATH,READWRITE=CHVAR)
```

would set CHVAR to YES.

The only differences are that ACCESS returns a reason for rejection, and can test file executability.

```
SUBROUTINE CHMOD(PATH,MODE,ERRNO)
CHARACTER*(*),INTENT(IN) :: PATH
INTEGER(MODE_KIND),INTENT(IN) :: MODE
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Sets the file mode (ST_MODE) to MODE.

Possible errors include EACCES, ENAMETOOLONG, ENOTDIR, EPERM, EROFS (see f90_unix_errno).

```
SUBROUTINE CHOWN(PATH,OWNER,GROUP,ERRNO)
CHARACTER*(*),INTENT(IN) :: PATH
INTEGER,INTENT(IN) :: OWNER, GROUP
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Changes the owner (ST_UID) of file PATH to OWNER, and the group number (ST_GID) of the file to GROUP.

Possible errors include EACCES, EINVAL, ENAMETOOLONG, ENOTDIR, ENOENT, EPERM, EROFS (see f90_unix_errno).

```
SUBROUTINE FSTAT(LUNIT,BUF,ERRNO)
INTEGER,INTENT(IN) :: LUNIT
TYPE(STAT_T),INTENT(OUT) :: BUF
```

```
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

BUF receives the characteristics of the file connected to logical unit LUNIT.

If LUNIT is not a valid logical unit number or is not connected to a file, error EBADF is raised (see f90_unix_errno).

```
SUBROUTINE STAT(PATH,BUF,ERRNO)
CHARACTER*(*),INTENT(IN) :: PATH
TYPE(STAT_T),INTENT(OUT) :: BUF
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

BUF receives the characteristics of the file PATH.

Possible errors include EACCES, ENAMETOOLONG, ENOENT, ENOTDIR (see f90_unix_errno).

10.7 f90_unix_io

This module contains part of a Fortran API to input/output facilities detailed in ISO/IEC 9945-1:1990.

Currently, only the FLUSH procedure is supported. Users are advised to use the ONLY clause when using this module, as it will have additional names added in later releases.

Error handling is described in f90_unix_errno. Note that for procedures with an optional ERRNO argument, if an error occurs and ERRNO is not present, the program will be terminated.

10.7.1 Procedures

```
SUBROUTINE FLUSH(LUNIT,ERRNO)
INTEGER,INTENT(IN) :: LUNIT
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Flushes the output buffer of logical unit LUNIT.

If LUNIT is not a valid unit number or is not connected to a file, error EBADF is raised (see f90_unix_errno).

10.8 f90_unix_proc

Interface to process-related facilities from ISO/IEC 9945-1:1990, section 3: Process Primitives, excluding 3.3 Signals. Facilities corresponding to the C language functions ABORT, ATEXIT, EXIT and SYSTEM are also provided by this module.

Error handling is described in f90_unix_errno. Note that for procedures with an optional ERRNO argument, if an error occurs and ERRNO is not present, the program will be terminated.

10.8.1 Parameters

```
INTEGER,PARAMETER :: PID_KIND
```

Integer kind for representing process IDs.

```
INTEGER,PARAMETER :: WNOHANG
```

Option bit for WAITPID indicating that the calling process should not wait for the child process to stop or exit.

```
INTEGER,PARAMETER :: WUNTRACED
```

Option bit for WAITPID indicating that status should be returned for stopped processes as well as terminated ones.

10.8.2 Procedures

```
SUBROUTINE ABORT(message)
CHARACTER*(*), OPTIONAL :: message
```

ABORT cleans up the i/o buffers and then terminates execution, producing a core dump on Unix systems. If MESSAGE is given it is written to logical unit 0 (zero) preceded by "abort:".

```
SUBROUTINE ALARM(SECONDS,SUBROUTINE,SECLEFT)
INTEGER,INTENT(IN) :: SECONDS
INTERFACE
   SUBROUTINE SUBROUTINE()
   END
END INTERFACE
OPTIONAL SUBROUTINE
INTEGER,OPTIONAL,INTENT(OUT) :: SECLEFT
```

Establishes an "alarm" call to the procedure SUBROUTINE to occur after SECONDS seconds have passed, or cancels an existing alarm if SECONDS = 0. If SUBROUTINE is not present, any previous association of a subroutine with the alarm signal is left unchanged. If SECLEFT is present, it receives the number of seconds that were left on the preceding alarm or zero if there were no existing alarm.

If an alarm call is established with no handler (i.e., SUBROUTINE was not present on the first call) the process may be terminated when the alarm goes off.

```
SUBROUTINE ATEXIT(SUBROUTINE,ERRNO)
INTERFACE
   SUBROUTINE SUBROUTINE()
   END
END INTERFACE
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Registers an argumentless subroutine for execution on normal termination of the program. If the program terminates using the f90_unix_proc procedure EXIT, all subroutines registered with ATEXIT will be invoked in reverse order of their registration.

If the program terminates using the f90_unix_proc procedure FASTEXIT, these subroutines will not be invoked.

If the program terminates due to an error, or by executing an F STOP statement or main program END statement, whether these subroutines will be invoked is undefined.

```
SUBROUTINE EXECV(PATH,ARGV,LENARGV,ERRNO)
CHARACTER*(*),INTENT(IN) :: PATH
CHARACTER*(*),INTENT(IN) :: ARGV(:)
INTEGER,INTENT(IN) :: LENARGV(:)
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Executes the file PATH in place of the current process image; for full details see ISO/IEC 9945-1:1990 section 3.1.2. ARGV is the array of argument strings, LENARGV containing the desired length of each argument. If ARGV is not zero-sized, ARGV(1)(:LENARGV(1)) is passed as argument zero (i.e., the program name). If LENARGV is not the same shape as ARGV, error EINVAL is raised (see f90_unix_errno). Other possible errors include E2BIG, EACCES, ENAMETOOLONG, ENOENT, ENOTDIR, ENOEXEC, ENOMEM.

```
SUBROUTINE EXECVE(PATH,ARGV,LENARGV,ENV,LENENV,ERRNO)
CHARACTER*(*),INTENT(IN) :: PATH
CHARACTER*(*),INTENT(IN) :: ARGV(:)
INTEGER,INTENT(IN) :: LENARGV(:)
CHARACTER*(*),INTENT(IN) :: ENV(:)
INTEGER,INTENT(IN) :: LENENV(:)
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Similar to EXECV, with the environment strings specified by ENV and LENENV being passed to the new program; for full details see ISO/IEC 9945-1:1990 section 3.1.2.

If LENARGV is not the same shape as ARGV or LENENV is not the same shape as LENENV, error EINVAL is raised (see f90_unix_errno). Other errors are the same as for EXECV.

```
SUBROUTINE EXECVP(FILE,ARGV,LENARGV,ERRNO)
CHARACTER*(*),INTENT(IN) :: FILE
CHARACTER*(*),INTENT(IN) :: ARGV(:)
INTEGER,INTENT(IN) :: LENARGV(:)
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

The same as EXECV except that the program to be executed, FILE, is searched for using the PATH environment variable (unless it contains a slash character, in which case EXECVP is identical in effect to EXECV).

Errors are the same as for EXECV.

```
SUBROUTINE EXIT(STATUS)
INTEGER,OPTIONAL :: STATUS
```

Terminate execution as if executing the END statement of the main program (or an unadorned STOP statement). If STATUS is given it is returned to the operating system (where applicable) as the execution status code.

```
SUBROUTINE FASTEXIT(STATUS)
INTEGER,OPTIONAL :: STATUS
```

This provides the functionality of ISO/IEC 9945-1:1990 function _exit (section 3.2.2). There are two main differences between FASTEXIT and EXIT:

1. When EXIT is called all open logical units are closed (as if by a CLOSE statement). With FASTEXIT this is not done, nor are any file buffers flushed, thus the contents and status of any file connected at the time of calling FASTEXIT are undefined.
2. Subroutines registered with ATEXIT are not executed.

```
SUBROUTINE FORK(PID,ERRNO)
INTEGER(PID_KIND),INTENT(OUT) :: PID
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Creates a new process which is an exact copy of the calling process. In the new process, the value returned in PID is zero; in the calling process the value returned in PID is the process ID of the new (child) process.

Possible errors include EAGAIN and ENOMEM (see f90_unix_errno).

```
SUBROUTINE PAUSE(ERRNO)
INTEGER,INTENT(OUT) :: ERRNO
```

Suspends process execution until a signal is raised. If the action of the signal was to terminate the process, the process is terminated without returning from PAUSE. If the action of the signal was to invoke a signal handler (e.g., via ALARM), process execution continues after return from the signal handler.

If process execution is continued after a signal, ERRNO is set to EINTR.

```
PURE SUBROUTINE SLEEP(SECONDS,SECLEFT)
INTEGER,INTENT(IN) :: SECONDS
INTEGER,OPTIONAL,INTENT(OUT) :: SECLEFT
```

Suspends process execution for SECONDS seconds, or until a signal has been delivered. If SECLEFT is present, it receives the number of seconds remaining in the sleep time (zero unless the sleep was interrupted by a signal).

```
SUBROUTINE SYSTEM(STRING,STATUS,ERRNO)
CHARACTER*(*),INTENT(IN) :: STRING
INTEGER,OPTIONAL,INTENT(OUT) :: STATUS,ERRNO
```

Passes STRING to the command processor for execution. If STATUS is present it receives the completion status-this is the same status returned by WAIT and can be decoded with WIFEXITED et al. If ERRNO is present it receives the error status from the SYSTEM call itself.

Possible errors are those from FORK or EXECV.

```
SUBROUTINE WAIT(STATUS,RETPID,ERRNO)
INTEGER,OPTIONAL,INTENT(OUT) :: STATUS
INTEGER(PID_KIND),OPTIONAL,INTENT(OUT) :: RETPID
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Wait for any child process to terminate (returns immediately if one has already terminated). See ISO/IEC 9945-1:1990 section 3.2.1 for full details.

If STATUS is present it receives the termination status of the child process. If RETPID is present it receives the process number of the child process.

Possible errors include ECHILD, EINTR (see f90_unix_errno).

```
SUBROUTINE WAITPID(PID,STATUS,OPTIONS,RETPID,ERRNO)
INTEGER(PID_KIND),INTENT(IN) :: PID
INTEGER,OPTIONAL,INTENT(OUT) :: STATUS, RETPID, ERRNO
INTEGER,OPTIONAL,INTENT(IN) :: OPTIONS
```

Wait for a particular child process to terminate (or for any one if PID = –1). If OPTIONS is not present it is as if it were present with a value of 0. See ISO/IEC 9945-1:1990 section 3.2.1 for full details.

Possible errors include ECHILD, EINTR, EINVAL (see f90_unix_errno).

```
PURE INTEGER FUNCTION WEXITSTATUS(STAT_VAL)
INTEGER,INTENT(IN) :: STAT_VAL
```

If WIFEXITED(STAT_VAL) is true, this function returns the low-order 8 bits of the status value supplied to EXIT or FASTEXIT by the child process. If the child process executed a STOP statement or main program END statement, the value will be zero. If WIFEXITED(STAT_VAL) is false, the function value is undefined.

```
PURE LOGICAL FUNCTION WIFEXITED(STAT_VAL)
INTEGER,INTENT(IN) :: STAT_VAL
```

Returns true if and only if the child process terminated by calling FASTEXIT, EXIT, or by executing a STOP statement or main program END statement.

```
PURE LOGICAL FUNCTION WIFSIGNALED(STAT_VAL)
INTEGER,INTENT(IN) :: STAT_VAL
```

Returns true if and only if the child process terminated by receiving a signal that was not caught.

```
PURE LOGICAL FUNCTION WIFSTOPPED(STAT_VAL)
INTEGER,INTENT(IN) :: STAT_VAL
```

Returns true if and only if the child process is stopped (and not terminated). Note that WAITPID must have been used with the WUNTRACED option to receive such a status value.

```
PURE INTEGER FUNCTION WSTOPSIG(STAT_VAL)
INTEGER,INTENT(IN) :: STAT_VAL
```

If WIFSTOPPED(STAT_VAL) is true, this function returns the signal number that caused the child process to stop. If WIFSTOPPED(STAT_VAL) is false, the function value is undefined.

```
PURE INTEGER FUNCTION WTERMSIG(STAT_VAL)
INTEGER,INTENT(IN) :: STAT_VAL
```

If WIFSIGNALED(STAT_VAL) is true, this function returns the signal number that caused the child process to terminate. If WIFSIGNALED(STAT_VAL) is false, the function value is undefined.

10.8.3 Generic Procedures

```
SUBROUTINE EXECL(PATH,ARG0...,ERRNO)
CHARACTER*(*),INTENT(IN) :: PATH
CHARACTER*(*),INTENT(IN) :: ARG0...
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Arguments are named ARG0, ARG1, etc., up to ARG20 (additional arguments may be provided in later releases).

This function is the same as EXECV except that the arguments are provided individually instead of via an array; and because they are provided individually, there is no need to provide the lengths (the lengths being taken from each argument itself).

```
Errors are the same as for EXECV.
SUBROUTINE EXECLP(FILE,ARG0...,ERRNO)
CHARACTER*(*),INTENT(IN) :: FILE
CHARACTER*(*),INTENT(IN) :: ARG0...
INTEGER,OPTIONAL,INTENT(OUT) :: ERRNO
```

Arguments are named ARG0, ARG1, etc., up to ARG20 (additional arguments may be provided in later releases).

This function is the same as EXECL except that determination of the program to be executed follows the same rules as EXECVP.

Errors are the same as for EXECV

F Compiler Software License Agreement A

Read the terms and conditions of this license agreement carefully before installing the Software on your system.

By installing the Software you are accepting the terms of this Agreement between you and The Fortran Company. If you do not agree to these terms, promptly destroy all files and other materials related to the Software.

"Software" means the F compiler and associated computer programs.

The Fortran Company grants to you a nonexclusive license to use the Software with the following terms and conditions:

The Fortran Company retains title and ownership of the Software. This Agreement is a license only and is not a transfer of ownership of the Software.

The Software is copyrighted. You may copy the software provided that you include a copy of this license.

You may adapt and modify any source programs, but may not reverse engineer any object or executable files.

You may not sell, rent, or lease the software.

This license is effective until terminated by The Fortran Company. It will terminate automatically without notice if you fail to comply with any provision of this license. Upon termination, you must destroy all copies of the Software.

F is a trademark of The Fortran Company. No right, license, or interest to such trademark is granted under this Agreement, and you agree that you will assert no such right, license, or interest with respect to such trademark.

The failure of either party to enforce any rights granted under this Agreement or to take action against the other party in the event of any breach of this Agreement will not be deemed a waiver by that party as to subsequent enforcement of rights or subsequent action in the event of future breaches. If applicable statute or rule of law invalidates any provision of this Agreement, the remainder of the Agreement will remain in binding effect.

THE FORTRAN COMPANY MAKES TO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR PERFORMANCE OR ACCURACY. THE FORTRAN COMPANY SHALL IN NO EVENT BE LIABLE TO THE LICENSEE FOR ANY DAMAGES (EITHER INCIDENTAL OR CONSEQUENTIAL), EXPENSE, CLAIM, LIABILITY, OR LOSS, WHETHER DIRECT OR INDIRECT, ARISING FROM THE USE OF THE SOFTWARE.