sun®
microsystems

Sun™ FORTRAN Programmer's Guide

Part No: 800-2163-10
Revision A of 6 May 1988

sun®
microsystems

Sun™ FORTRAN Programmer's Guide

This FORTRAN software and documentation is a proprietary product of Sun Microsystems, Inc.

Some of the material in this manual is based on the Bell Laboratories document entitled *A Portable Fortran 77 Compiler*, by S.I. Feldman and P.J. Weinberger, dated 1 August 1978. Material on the I/O Library is derived from the paper entitled *Introduction to the f77 I/O Library*, by David L. Wasley, University of California, Berkeley, California 94720. Further work was done at Sun Microsystems.

Contents

Tables

Preface

Purpose and Audience

This guide provides information needed to write FORTRAN programs on the Sun™ workstation. It assumes you have a thorough understanding of FORTRAN and of some operating system — not necessarily either the SunOS™ or UNIX® operating system.

It does assume that you know certain SunOS commands and concepts: how to log on and off, how to find your way around the SunOS file system, and something about piping and redirection. If you need to refresh your memory on these topics, see Chapter 6 — "The File System and FORTRAN I/O", or refer to *Getting Started with SunOS: Beginner's Guide*, or see an introductory UNIX book.

This guide describes the extensions provided by FORTRAN 77 and Sun FORTRAN, but it does not assume you know either Sun FORTRAN or VAX™/VMS™ FORTRAN. For basic features of standard FORTRAN, refer to any standard FORTRAN text. This guide also introduces related extensions to the Sun debugger, and the source-code converter.

Conventions in Text

Note the following conventions we use in this manual to display information. After logging in, the SunOS system prompt looks something like this:

```
demo% █
```

The basic SunOS prompt is merely the percent sign (%). However, most Sun workstations have distinct host names and our examples are more easily distinguished if we use a symbol longer than a % sign. For this reason, examples in this manual use `demo%` to denote the system prompt.

The system prompts and replies are shown in `plain typewriter font`, shown here and in the example below. Text the user types is shown in **`boldface typewriter font`**. For example:

```
demo% echo hello
hello
demo% █
```

The `plain typewriter font` is also used to indicate FORTRAN statements and reserved words. *Italics* indicates general arguments or parameters that you should replace with the appropriate input. Italics are also used to indicate emphasis. Examples of coding are in white boxes, and screen displays are in gray boxes.

Organization

This manual consists of a *tutorial* part (the chapters), with exposition and examples, aimed at experienced programmers, and a *reference* part (the appendices), mostly tables:

`Chapter 1` is an introduction to Sun FORTRAN and SunOS.

`Chapter 2` is a minimal guide to running Sun FORTRAN.

`Chapter 3` introduces the compiler: syntax, options, etc.

`Chapter 4` describes extensions to data structures and expressions.

`Chapter 5` describes control structure extensions.

`Chapter 6` describes the SunOS file system as it relates to FORTRAN I/O.

`Chapter 7` describes Sun FORTRAN I/O statements.

`Chapter 8` introduces program development.

`Chapter 9` introduces the debuggers `dbx` and `dbxtool`.

`Chapter 10` describes the VMS extensions and the source-code converter.

`Chapter 11` describes the conventions for interfacing C and FORTRAN.

`Chapter 12` describes the Ratfor language.

`Appendix A` is a table of the ASCII character set.

`Appendix B` provides selected examples for all Sun FORTRAN statements.

`Appendix C` lists intrinsic functions for Sun FORTRAN.

`Appendix D` is a list of the FORTRAN-related runtime error messages.

`Appendix E` is a bibliography.

`Appendix F` contains the manual pages for Sun FORTRAN.

References

- *Programming in VAX FORTRAN*, September 1984, Software Version 4.0, Digital Equipment Corporation. [AA-D034D-TE]
- *Cray-1 Computer Systems FORTRAN (CFT) Reference Manual*, SR-0009, November 1982, Cray Research Inc.
- *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, April 1978, American National Standards Institute, Inc.
- *Applied FORTRAN 77 Featuring Structured Programming*, Roy Ageloff and Richard Mojena, 1981, Wadsworth, Inc.
- *FORTRAN-77 Featuring Structured Programming*, Loren P. Meissner, and Elliott I. Organick, 1980, Addison-Wesley.

- □ *Problem Solving with Structured FORTRAN 77*, D.M. Etter, 1984, Benjamin/Cummings Publishing Company, Inc.

See also the bibliography at the end of this manual.

1

Introduction

1

Introduction

Sun FORTRAN is an enhanced ANSI FORTRAN 77 development system for the Sun workstation. Sun FORTRAN is GSA certified, provides an IEEE standard floating-point package, and offers VAX VMS FORTRAN 4.0 extensions, plus recursion and pointers. For installations that use both Sun and VAX computers, you can write FORTRAN programs that can run the same on both systems.

Routines written in other Sun languages, such as C, Pascal, and Modula-2, may be combined with FORTRAN programs, since these Sun languages have common calling conventions.

Both global and peephole optimizations are available to create FORTRAN applications that are significantly faster and smaller. Benchmarks show that optimized applications are as much as 80% faster, with an average of 10% reduction in code size.

Sun FORTRAN comes with a complete set of UNIX system calls and FORTRAN support libraries.

Sun FORTRAN is highly integrated with powerful Sun development tools, including both SunView,™ and the SunPro™ tools: `dbxtool`, `make`, and SCCS. Tools that you may find useful are summarized here.

Text Editing The major text editor for source programs is `vi` (vee-eye), the visual display editor. It has considerable power because it offers the capabilities of both a line and a screen editor. `vi` also provides several commands specifically for editing programs. These are options you can set in the editor. Two examples are the `autoindent` option, which supplies white space at the beginning of a line, and the `showmatch` option, which shows matching parentheses. For more information, see the *Editing Text Files* manual section on `vi`.

Other editors are available for use, such as `ed`, `ex`, and `textedit` (available under *SunView* on Sun workstations).

FORTRAN Tools

`fpr` is a FORTRAN *output filter* for printing files that have FORTRAN carriagecontrol characters in column one. The UNIX implementation on the Sun system does not use carriagecontrol since UNIX provides no explicit printer files. Thus, you use `fpr` when you want to transform files formatted with FORTRAN carriagecontrol conventions into files formatted according to UNIX line-printer conventions. For more information on `fpr`, refer to Appendix F or the *SunOS Reference Manual*, or try the command **`man fpr`**.

`fsplit` splits a multi-routine FORTRAN file into one file per routine.

`Debug Aids`

There are three main debugging tools available on the Sun system:

`dbx` is an interactive symbolic debugger that understands Sun FORTRAN programs.

`dbxtool` is `dbx` with a mouse and windows.

`adb` is an interactive, general-purpose low-level debugger.

The online documentation consists of pages from the *FORTRAN Programmer's Guide* and the *SunOS Reference Manual* that are called 'man pages'. Some commonly used pages for FORTRAN are:

- `f77`(1)
- `f77cvt`(1)
- `fpr`(1)
- `fsplit`(1)
- `dbx`(1)
- `dbxtool`(1)
- ieee_handler (3M)
- matherr (3M)
- sigfpe (3)

The man pages that deal with error handling and exception processing are: `ieee_handler` `matherr`, and `sigfpe`. `ieee_handler` is the IEEE exception trap handler function, `sigfpe` is the signal handler function for specific SIGFPE error codes, and `matherr` is the math library exception-handling function.

f77 invokes the FORTRAN compiler; fpr and fsplit are FORTRAN tools briefly explained above. See the manual pages near the end of this guide for descriptions of other FORTRAN routines.

Other Sun manuals containing information on editing or using FORTRAN are:

- *Editing Text Files*
- *Programmer's Tools Manuals Minibox*
- *SunOS Reference Manual*

2

Getting Started

2

Getting Started

2.1. Overview

This chapter provides the knowledgable user with a bare minimum on how to compile and run Sun FORTRAN programs. It is meant for the user who knows FORTRAN thoroughly and needs to start writing FORTRAN programs on a Sun immediately. If that is not your style, skip to Chapter 3 — "Using the Compiler".

Using FORTRAN on a Sun workstation involves three steps:

- Write a program in FORTRAN using an editor, giving the filename a `.f` suffix.
- Compile this `.f` file using the `f77` command.
- Execute by typing the name of the executable file.

For example, here is a simple program that displays a message on the workstation screen.

```
demo% cat greetings.f
        PROGRAM  GREETINGS
        PRINT *, 'Real programmers hack FORTRAN!'
        END
demo% █
```

2.2. Compiling

Compile the program `greetings` using the `f77` command as follows:

```
demo% f77 greetings.f
greetings.f:
      MAIN greetings:
demo% █
```

Note that `f77` displays a message indicating the stage of the compilation. In this example, `f77` compiles `greetings.f` executable code is put onto the `a.out` file.

2.3. Running

You can then run the program by typing **a.out** on the command line:

```
demo% a.out
  Real programmers hack FORTRAN!
demo% █
```

2.4. Renaming the Executables

It is inconvenient to have the result of every compilation end up on a file called `a.out` since if such a file already exists, it is overwritten. You can avoid this in either of two ways:

□ Change the name of `a.out` after each compilation, using the `mv` command. For example:

```
demo% mv a.out greetings
demo% █
```

□ Use the compiler's **-o** option to tell it to rename the output executable file. For example:

```
demo% f77 -o greetings  greetings.f
greetings.f:
   MAIN greetings:
demo% █
```

places the executable code on the `greetings` file.

Either way (**mv** or **-o**), you can run the program by typing the name of the executable file, as follows:

```
demo% greetings
  Real programmers hack FORTRAN!
demo% █
```

At this stage, the impatient user may jump right in and start cutting code. Other more cautious types will at least browse through one or more of the next few chapters, if only to check out the compiler options and extended features.

If you are not familiar with the file system of UNIX or SunOS, you should read Chapter 6 — "The File System and FORTRAN I/O", or refer to *Getting Started with SunOS: Beginner's Guide*, or see an introductory UNIX book.

3

Using the Compiler

3

Using the Compiler

3.1. Overview

This chapter describes the following topics:

- Compiler command syntax
- Kinds of lines accepted
- Multiple routines per file
- Compiler options

Most details of the language are deferred to later chapters.

3.2. Compiler Command

The syntax of the compiler command is as follows:

`f77 [`*options*`]` *filename* ...

As an example with two files:

```
demo% f77 growth.f  fft.F
```

or with some options:

```
demo% f77 -g -u growth.f  fft.F
```

3.3. Source Lines

Various properties and kinds of source lines are described below:

The **INCLUDE** Statement

The statement:

```
   INCLUDE 'stuff'
```

is replaced by the contents of the file `stuff`.

Searchpath

If the name referred to by the `INCLUDE` statement begins with the character '/', it is taken by `f77` to mean the absolute pathname* of the include file.

* For more on pathnames, see Chapter 6 — *The File System and FORTRAN I/O*.

Otherwise, the compiler looks for the include file in the following places, in this order:

- □ The directory containing the source file with the `INCLUDE` statement.
- □ the current directory in which the `f77` command was issued.
- □ the `/usr/include` directory.

For example, if:

(1) your current working directory is `/usr/ftn`,
(2) your source file is `/usr/ftn/projA/myprg.f`,
(3) this source file has the line "`INCLUDE ver1/const.h`",

then `f77` will search for `const.h` in the following three paths, in order:

```
/usr/ftn/projA/ver1/const.h
/usr/ftn/ver1/const.h
/usr/include/ver1/const.h
```

Nesting

These `INCLUDE` s can be nested ten deep.

NOTE *Files included via the preprocessor* `#include` *directive may contain* `#defines` *and the like, while files included with the compiler* `INCLUDE` *statement must contain only FORTRAN statements.*

Comment Lines

A line with a `C` or `c` or an asterisk (*) in column one is a comment line.

Blank Lines

A totally blank line is a comment line.

Standard Source Lines

The standard source lines for FORTRAN are specified as follows:

- □ The first 72 columns of each line are scanned. (See "*Extended Lines*", below.)
- □ The first five columns must be blank or contain a numeric label.
- □ Continuation lines are identified by a nonblank or non-zero in column 6.
- □ Short lines are padded to 72 characters; long lines are truncated (See "*Extended Lines*", below.)

Tab-format Source Lines

The tab-format source lines are in the format used in the UNIX operating system:

- □ A tab in any of columns 1 though 5 marks the beginning of a tab-format source line. The text following the tab is scanned as if it started in column 7. The line may be up to 72 columns long. (See "*Extended Lines*", below.)
- □ Continuation lines are identified by an ampersand (&) in column 1.

Continuation Lines

The default maximum number of continuation lines is 19 (1 initial and 19 continuation). See the **N1*n*** option, below.

Extended Lines

The compiler includes an option to accept extended source lines, with up to 132 characters. By default, it ignores any characters after column 72. To specify the recognition of extended source lines, use the **-e** option, as in this example:

```
demo% f77 -e prog.f
```

Padding

Padding is significant in lines such as:

```
         1         2         3         4         5         6         7
C23456789012345678901234567890123456789012345678901234567890123456789012

      DATA SIXTYH/60H
     1         /
```

3.4. Upper and Lower Case

In the standard, there are only 26 letters — FORTRAN 77 is a one-case language. Consistent with ordinary UNIX system usage, this compiler accepts upper-case or lower-case input. That is, the program source file can be in either lower-case or upper-case, or any mixture, but note the following general rules concerning case:

- □ The normal action of the compiler is to maintain names of procedures and names of variables in lower-case. (The **–U** option prevents this. In this **–U** mode, it is possible to specify external names with upper-case letters in them, and to have distinct variables differing only in case.)
- □ The compiler does not translate characters inside character-string constants.
- □ The strings returned by `INQUIRE` are in upper-case.
- □ The debugger `dbxtool` does not convert to lower-case.

Case with `INQUIRE`

Since strings returned by the FORTRAN `INQUIRE` statement are in upper-case, use of `INQUIRE` needs some caution regarding case. For example:

```
demo% cat inq1.f
* inq1.f Inquire with UPPER and lower-case
        CHARACTER ANSWER*15
        INQUIRE ( 6, SEQUENTIAL=ANSWER )
        IF ( ANSWER .EQ. 'YES' ) PRINT *, 'CAPS MATCH'
        IF ( ANSWER .EQ. 'yes' ) PRINT *, 'lowers match'
        END
demo% f77 inq1.f
inq1.f:
   MAIN:
demo% a.out
  CAPS MATCH
demo% █
```

The match on *upper*-case is successful; the match on *lower*-case fails. The programmer should probably be alert to such distinctions.

Case with `dbxtool`

Use of the debugger `dbxtool` also requires some caution regarding case.* If your source file is in upper-case, then before you use `dbxtool` you should either tell `f77` to use upper-case, for example:

```
demo% f77 -U inq1.f
inq1.f:
   MAIN:
demo% █
```

or use the `tr` command to translate the source file from upper-case to lower-case or vice versa. For example, to read the upper-case source file `SBENCH.f` and write the lower-case source file `sbench.f`:

```
demo% tr A-Z a-z < SBENCH.f > sbench.f
```

If your programs have bugs, `dbxtool` is useful. Most who have tried it found it was more than worth the bother of recompiling with the **-U** option.

3.5. Routines per File

The scope of FORTRAN variables and routines (as compared with C) has nothing to do with the files they reside in, so a source file can contain any number of compilation units (main programs, functions, or subroutines). However, there are two good reasons to keep each compilation unit in a separate source file:

- □ Reduce the compilation overhead of changing one procedure.
- □ Minimize loading of unreferenced functions.

* This debugger displays variable names so the users can *select* the variable they want displayed. It gets the variable names from the source file, so if the source has them in *upper* and the compiler has them in *lower*, then `dbxtool` cannot find the selected variable.

Note that this applies only to the .o modules in libraries. Files explicitly named in the link command are unconditionally loaded.

f77 produces one .o file for each .f file it processes. If any routine in the .o file is referenced, the linker ld copies in the entire .o file, loading all routines, referenced or not.

For example, suppose we have two files: subs.f and main.f:

File subs.f has routines a and b.
File main.f has a main program that calls a but not b.
The command:

```
demo% f77 main.f sub.f
```

produces an a.out file that contains the code for subroutine b even though b is not referenced.

The fsplit command can be used to break up multiple-routine source files into a series of files, one routine per file.

3.6. Other Files

The f77 command recognizes several other kinds of files. The table below summarizes the filename extensions that f77 understands.

Table 3-1 *Filename Suffixes Sun FORTRAN Understands*

Suffix	*Language*	*Action*
.f	FORTRAN	Compile FORTRAN source files, put object files in current directory, default name of object file is that of the source but with .o suffix.
.F	FORTRAN	Process FORTRAN source files by the C preprocessor before compiling by f77.
.c	C	C source files are compiled by the C compiler. The f77 and cc commands generate slightly different loading sequences, since FORTRAN programs need a few extra libraries and a different startup routine than do C programs.
.s	Assembler	Process assembly-language source files by the assembler as.
.il	In-line Expansion	Process in-line expansion code template files. These are used to expand calls to selected routines in-line when the -O option is used.
.o	Object Files	Pass object files through to the linker.

Note: Files with none of the above filename suffixes are passed to the linker.

Language Preprocessor

The cpp program is the C language preprocessor, which is invoked during the first pass of a FORTRAN compilation if the source filename has the .F extension. Its main uses here are for constant definitions and conditional compilation. See cpp (1), or the **-D***name* option in *Compiler Options*, in the next section.)

3.7. Compiler Options

The list below contains the options that f77 understands. Note that the compiler option **-help** displays essentially the same list, as does the **man f77** command. (See the *Manual Pages*, online or in the appendices.)

-66
Report non-FORTRAN 66 constructs as errors.

-a Insert code to count how many times each basic block is executed. Invokes a runtime recording mechanism that creates a .d file for every .f file (at normal termination). The .d file accumulates execution data for the corresponding source file. The tcov(1) utility can then be run on the source file to generate statistics about the program.

-align *_block_*
Cause the common block whose FORTRAN name is *block* to be page-aligned: its size is increased to a whole number of pages, and its first byte is placed at the beginning of a page. For example, the command "**f77 -align _BUFFO_ GROWTH.F**" causes BUFFO to be page-aligned. This option applies to *uninitialized* data only: if any variable of the common block is initialized in a DATA statement, then the block will not be aligned. This option is passed to the linker.

-ansi
Identify all non-ANSI extensions. Note that f77cvt provides an option to flag any Sun FORTRAN extensions that it uses during the conversion of a VMS FORTRAN source file. For more on f77cvt, see Section 10.4 — "The Source Code Converter."

-c Suppress linking and produce a .o file for each source file.

-C Compile code to check that subscripts are within declared array bounds.

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

-D*name=def*

-D*name*
Define *name* to the C preprocessor, as if by "#define". If no definition is given, the name is defined as "1" (.F files only).

-e Accept extended source lines, up to 132 characters long.

-f Align local data and COMMON blocks on 8-byte boundaries. Resulting code may not be standard and may not be portable.

-f*float_option*

This option applies only to the Sun-2™ or Sun-3™. See the *Sun Floating-Point Programmer's Guide* for more information.

–f68881
Generate code that assumes the presence of the Sun Floating-Point Accelerator (Sun-3 only).

–ffpa
Generate code that assumes the presence of the Sun-3 floating-point accelerator board (Sun-3 only).

–fsky
Generate code that assumes the presence of a Sky™ Floating-Point Processor board. Programs compiled with this option can only be run in systems that have a Sky board installed. (Sun-2 only).

–fsoft
Generate code that uses software floating-point calls (this is the default).

–fstore
Insure that expressions allocated to extended-precision registers are rounded to storage precision whenever an assignment occurs in the source code. Only has effect when **–f68881** is specified (Sun-3 only).

–fswitch
Run-time-switched floating-point calls. The compiled object code is linked at runtime to routines that support the FPA, MC68881, Sky floating-point board, or software-floating-point calls, depending on the system that is running the program (Sun-2 or Sun-3).

–F Apply the C preprocessor to relevant files and put the result in the file with the suffix changed to `.f`, but do not compile.

–g Produce additional symbol table information for `dbx` or `dbxtool`. Also, pass the `–lg` file to `ld`(1).

–help
Display an equivalent of this list of options.

–i2
Make the default size of integer and logical constants and variables short (2 bytes).

–i4
Make the default size of integer and logical constants and variables four bytes (this is the default).

–I*path*
Add *path* to the list of directories in which to search for '`#include`' files with *relative* pathnames (not beginning with /). Search first for '`#include`' files whose names do not begin with '/' in the directory containing the source file, then in directories named in **–I** options, and finally in directories on a standard list (`.F` suffix files only). Note that this does not affect FORTRAN's `INCLUDE` statement, only the C preprocessor's. For example, "**f77 -I/usr/applib growth.f**"

causes the compiler to search for '`#include`' files in the `/usr/applib` directory.

`-l`*x*

Link with object library `/lib/lib`*x*`.a`, where *x* is a string. If that does not exist, then `ld` tries `/usr/lib/lib`*x*`.a` (see `ld`(1)).

`-L`*dir*

Add *dir* to the list of directories containing object-library routines (for linking using `ld`(1)).

`-misalign`

Allow for misaligned data in memory. This option is *for* for the Sun-4 only. Use this option *only* if you get a warning that `COMMON` or `EQUIVALENCE` statements cause data to be misaligned. **WARNING:** With this option, the compiler will generate much *slower* code for references to dummy arguments. If you can, you should recode the indicated section instead of recompiling with this option. For example, the program

```
      INTEGER*2   I(4)
      REAL        R1, R2
      EQUIVALENCE (R1, I(1)), (R2, I(2))
      END
```

causes the error message

```
  "misalign.f", line 4: Error: bad alignment for "r2"
                        forced by equivalence
```

-N[**`cdlnqsx`**]*nnn*

Make static tables in the compiler bigger. `f77` complains if tables overflow and suggests you apply one or more of these flags. These flags have the following meanings:

`c` Maximum depth of nesting for control statements (for example, `DO` loops). The default is 20.

`d` Maximum depth of nesting for data structures and unions. The default is 20.

`l` Maximum number of continuation lines for a continued statement. The default is 19 (1 initial and 19 continuation).

`n` Maximum number of identifiers. The default is 1009.

`q` Maximum number of equivalenced variables. The default is 150.

`s` Maximum number of statement numbers. The default is 401.

`x` Maximum number of external names (common block names, subroutine and function names). The default is 200.

–o *output*

Name the final output file *output* instead of `a.out`.

–onetrip

Compile DO loops so that they are performed at least once if reached. Sun FORTRAN DO loops are not performed at all if the upper limit is smaller than the lower limit, unlike FORTRAN 66 DO loops.

–O*n*

Optimize the object code. If you use this with the **–g** option, then the `-O`*n* is ignored.

–O1

Peephole Optimization only. Do not use **–O1** unless **–O2** and **–O3** result in excessive compilation time, or running out of swap space.

–O2

Partial optimization. Does a restricted set of global optimizations. Do not use **–O2** unless **–O3** results in excessive compilation time, or running out of swap space.

–O3

Global Optimization. (same as **–O**)

Note:

If the optimizer runs out of swap space, try any of the following possibly corrective measures (listed in increasing order of difficulty):

- Change from **-O3** to **-O2**.
- Divide large, complicated routines into smaller, simpler ones.
- Increase the limit for the stacksize: insert the line "`limit stacksize 8 megabytes`" into your `.cshrc` file.
- Repartition you disk with two to four times as much swap space. Backup everything first. You may well need help from your system administrator to do this.

–p Prepare object files for profiling, see `prof` (1).

–pg

Produce counting code in the manner of **–p**, but invoke a runtime recording mechanism that keeps more extensive statistics and produces a `gmon.out` file at normal termination. An execution profile can then be generated by use of `gprof` (1).

–pipe

Use pipes, rather than intermediate files between compilation stages. Very cpu-intensive.

–P Partial optimization. (same as –O2)

–Qoption *prog opt*
Pass the option *opt* to the program *prog*. The option must be appropriate to that program and may begin with a minus sign. *prog* can be one of: `as`, `c2`, `cg`, `cpp`, `f77pass1`, `iropt`, `inline`, or `ld`.

–Qpath *pathname*
Insert the directory *pathname* into the compilation search path (to use alternate versions of programs invoked during compilation). This path will also be searched first for certain relocatable object files that are implicitly referenced by the compiler driver (such files as `*crt*.o` and `bb_link.o`).

–Qproduce *sourcetype*
Produce source code of the type *sourcetype*, where *sourcetype* is one of:
`.o` Object file from `as` (1).
`.s` Assembler source (from `f77pass1`, `inline`, `c2`, or `cg`.)

–S Compile the named programs, and leave the assembly-language output on corresponding files suffixed with `.s` (no `.o` file is created).

–temp=*dir*
Set directory for temporary files to be *dir*.

–time
Report execution times for the various compilation passes.

–u Make the default type of variables 'undefined', rather than using FORTRAN implicit typing.

–U Do not convert upper-case letters to lower-case, but leave them in the original case. The default is to convert to lower-case except within character-string constants.

–v Print the name of each pass as the compiler executes.

–w Suppress all warning messages.

–w66
Suppress only messages generated by programs using obsolete FORTRAN 66 features.

Unrecognized Arguments

Other arguments are taken to be either linker option arguments or names of `f77`-compatible object programs, typically produced by an earlier run, or perhaps libraries of `f77`-compatible routines. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program called (by default) `a.out` or with a filename specified by the **–o** option.

4

Data Structures and Expressions

4

Data Structures and Expressions

4.1. Overview

This chapter describes data structures and expressions that have been added to FORTRAN by FORTRAN 77 and Sun FORTRAN, It includes constants and parameters, names and types of variables, structured variables, and pointers.

4.2. Names

Names can have as many as 32 characters. They must begin with a letter, and they can consist of letters, digits, the dollar sign ($), and the underline character (_). This applies to names of variables, symbolic constants (parameters), labeled common, and names of programs and procedures (as in the PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA statements).

4.3. Data Types

Some new data types have been added since FORTRAN 66.

The BYTE Type

Sun FORTRAN has added a 1-byte integer data type, BYTE, which has the synonym LOGICAL*1. A BYTE variable can hold the logical values .TRUE. or .FALSE., or one character, or an integer between -128 and 127. An example showing some uses of these types is:

```
demo% cat byt1.f
* byt1.f The BYTE data type
    BYTE  BIT3, C1, COUNTER, SWITCH
    DATA  BIT3 / 8 /,  C1 / 'W' /,
 &            COUNTER / 0 /, SWITCH / .FALSE. /
    WRITE ( *, 1 ) BIT3, C1, COUNTER, SWITCH
    IF ( COUNTER .LT. 127 ) COUNTER = COUNTER + 1
    SWITCH = .NOT. SWITCH
    WRITE ( *, 1 ) BIT3, C1, COUNTER, SWITCH
 1  FORMAT ( 1X, O3, 1X, A1, 1X, I3, 1X, O3 )
    STOP
    END
demo% f77 byt1.f
byt1.f:
 MAIN:
demo% a.out
  10 W   0   0
  10 W   1   1
demo%
```

↑
column 2

The above example use the octal format specifier O. For more on this see Chapter 7 — "Input and Output".

The CHARACTER Type

FORTRAN 77 has the CHARACTER data type. Local and COMMON character variables are declared with a length denoted by a constant expression.
For example:

```
CHARACTER*17 A, B(3,4)
CHARACTER*(6+3) C
```

or:

```
CHARACTER A*17, B(3,4)*17
CHARACTER C*(6+3)
```

If the length is omitted, it is assumed equal to 1.

A dummy argument character string can have a constant length. For example:

```
SUBROUTINE SCHLEP ( A )
CHARACTER A*32
```

or the length can be declared to be the same as that of the corresponding actual argument at runtime. For example:

```
SUBROUTINE SCHLEP ( A )
CHARACTER A*(*)
```

There is an intrinsic function LEN that returns the actual declared length of a character string. For example, the program

```
CHARACTER  A*17
A = "xyz"
PRINT *, LEN( A )
STOP
END
```

will display 17, not 3.

Character arrays and common blocks containing character variables are packed: in an array of character variables, the first character of one element follows the last character of the preceding element, without holes.

See the section — Expressions — for more about character objects.

The **DOUBLE COMPLEX** Type

Sun FORTRAN adds the DOUBLE COMPLEX type. Each datum is a pair of double-precision real variables. A double complex version of each complex built-in function is provided. Generally the specific function names begin with Z or CD instead of C, except for the two functions DIMAG and DREAL, which return a real value. DREAL is a Sun synonym for DBLE.

Table 4-1 *Summary of Double Complex Functions*

Name	*Arg Type*	*Result Type*	*Meaning*
ZABS **CDABS**	COMPLEX*16	REAL*8	a → |a|
ZEXP **CDEXP**	COMPLEX*16	COMPLEX*16	a → e^a
ZLOG **CDLOG**	COMPLEX*16	COMPLEX*16	natural log
ZSQRT **CDSQRT**	COMPLEX*16	COMPLEX*16	square root
ZSIN **CDSIN**	COMPLEX*16	COMPLEX*16	sine
ZCOS **CDCOS**	COMPLEX*16	COMPLEX*16	cosine
DCMPLX	any numeric	COMPLEX*16	Expand to Double Complex
DCONJG	COMPLEX*16	COMPLEX*16	x + yi → x - yi
DIMAG	COMPLEX*16	REAL*8	x + yi → y
DREAL	any numeric	REAL*8	Expand to Double Precision

For more detail on these functions, see Appendix C — *Intrinsic Functions.*

Short Integer Type

Sun FORTRAN accepts declarations of type INTEGER*2. An expression involving only objects of type INTEGER*2 is of that type. Generic functions return short or long integers depending on the default integer type. If a procedure is compiled using the **-i2** flag, all integer constants that fit and all variables of type INTEGER (no explicit size) are of type INTEGER*2. If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one is chosen that returns the prevailing length (INTEGER*2 when the **-i2** command flag is in effect). When the **-i2** option is in effect, the default length of LOGICAL quantities is 2 bytes.

Ordinary integers follow the FORTRAN 77 rules about occupying the same space as a REAL variable; they are assumed to be equivalent to the C type long int, and half-word integers are of C type short int. These short integer and logical quantities do not obey the standard rules for storage association.

Storage Allocation

This section describes the way storage is allocated to *variables* of different types. In general, any *word* value (a value that occupies 16 bits) is always aligned on a word boundary. Anything larger than a word is also aligned on a word boundary (on a Sun-3, 32-bit or larger unequivalenced local variables are longword-aligned). Values that can fit into a single character are character-aligned.

`BYTE` or `LOGICAL*1`
: occupies one character (8 bits) of storage, aligned on a character boundary. A value of 0 represents `.FALSE.`, 1 represents `.TRUE.`, and any other value is 'undefined' as a logical value, but may be used as a character or small integer.

`CHARACTER` or `CHARACTER*1`
: occupies one character (8 bits) of storage, aligned on a character boundary.

`CHARACTER*`*n*
: occupies *n* characters (8 bits each) of storage, aligned on a character boundary. (Every character string *constant* is aligned on a *word* boundary, and if it does not appear in a `DATA` statement, it is followed by a null character to ease communication with C routines.)
There are no null (zero-length) character strings.

`COMPLEX`
: elements are represented by two `REAL` elements. The first element represents the real part and the second represents the imaginary part.

`DOUBLE COMPLEX` or `COMPLEX*16`
: elements are represented by two `DOUBLE PRECISION` elements. The first represents the real part and the second represents the imaginary part.

`DOUBLE PRECISION` or `REAL*8`
: occupies 64 bits (eight characters or four words), aligned on a word boundary. A `DOUBLE PRECISION` element has a sign bit, an 11-bit exponent and a 52-bit fraction. FORTRAN `DOUBLE PRECISION` elements conform to the IEEE standard for double-precision floating-point data as defined in [19]. The layout is shown in Table 4-3 .

`INTEGER*2`
: occupies 16 bits (two characters or one word), aligned on a word boundary.

`INTEGER` or `INTEGER*4`
: occupies 32 bits (four characters or two words), aligned on a word boundary. If the **-i2** compiler flag is set, then `INTEGER` (without any size specification) is the same as `INTEGER*2`.

`LOGICAL`
: occupies four characters (32 bits) of storage, aligned on a word boundary. The value 0 represents `.FALSE.`, 1 represents `.TRUE.`, and any other value is an 'undefined' logical value. If the **-i2** compiler flag is set, then `LOGICAL` (without any size specification) is the same as `LOGICAL*2`.

LOGICAL*1 or BYTE
See BYTE or LOGICAL*1, above.

LOGICAL*4
occupies four characters (32 bits) of storage, aligned on a word boundary. The value 0 represents the value .FALSE., 1 represents .TRUE., and any other value is an 'undefined' logical value.

REAL or REAL*4
occupies 32 bits (four characters or two words), aligned on a word boundary. A REAL element has a sign bit, an 8-bit exponent and a 23-bit fraction. FORTRAN REAL elements conform to the IEEE standard[1]. The layout is shown in Table 4-3 .

REAL*8 or DOUBLE PRECISION
occupies 64 bits (eight characters or four words), aligned on a word boundary. A DOUBLE PRECISION element has a sign bit, an 11-bit exponent and a 52-bit fraction. FORTRAN DOUBLE PRECISION elements conform to the IEEE standard for double-precision floating-point data as defined in [19]. The layout is shown in Table 4-3 .

Implicit Typing

Unless otherwise declared, a variable whose name begins with I, J, K, L, M, or N is of type INTEGER, otherwise it is of type REAL.

The IMPLICIT Statement

The general implicit typing rule may be overridden with an IMPLICIT statement. For example:

```
IMPLICIT REAL(A-C,G), COMPLEX(W-Z), CHARACTER*17 (S)
```

declares that variables whose names begin with an A, B, C, or G are REAL, those beginning with W, X, Y, or Z are COMPLEX, and those beginning with S are CHARACTER*17. It is still generally poor practice to depend on implicit typing.

Implicit Undefined

As an aid to good programming practice, the Sun FORTRAN compiler has the related modifier UNDEFINED. For example, the statement

```
IMPLICIT UNDEFINED(A-Z)
```

turns off the automatic data typing mechanism, and causes the compiler to issue a diagnostic for each variable that is used but does not appear in a type statement. This applies throughout the subprogram with the statement. Specifying the **−u** compiler flag on the command line is equivalent to beginning each procedure with this statement.

[1] See p.754 [19].

4.4. Expressions

Character Constants

In standard FORTRAN 77, character-string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it must be repeated. For example:

```
'abc'
'ain''t'
```

The Sun FORTRAN compiler and I/O system recognize both the apostrophe (') and the double-quote ("). If a string begins with one variety of quotation marks, the other can be embedded within it without using the repeated quote or backslash escapes (see next section). For example:

```
"abc"
"ain't"
```

Each character string constant appearing outside a DATA statement is followed by a null character to ease communication with C routines. There are no null (zero-length) character strings in FORTRAN 77.

Escape Sequences

For compatibility with C usage, the following backslash escapes are recognized:

Table 4-2 *Backslash Escape Sequences*

Character	*Meaning*
\n	newline
\t	tab
\b	backspace
\f	form feed
\v	vertical tab
\0	null
\'	apostrophe (does not terminate a string)
\"	quotation mark (does not terminate a string)
\\	\
\x	*x*, where *x* is any other character

Hollerith

ANSI standard FORTRAN 77 does not have the old Hollerith (*n* H) notation, although the ANSI standard recommends implementing the Hollerith feature in order to improve compatibility with old programs. In Sun FORTRAN, Hollerith data can be used in place of character-string constants, and can also be used to initialize noncharacter variables in DATA statements, though none of these are recommended. For example:

```
CHARACTER*2 CODE
INTEGER*2 TAG
DATA  TAG / 2Hok /
CODE = 2Hno
```

Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

Joining Strings

FORTRAN 77 has the // operator for joining character-strings. This is usually called the *concatenation* operator. The result of *concatenating* two strings is a third string containing the characters of the left operand followed immediately by the characters of the right operand. For example, the program

```
CHARACTER A*8, B*2, C*12
A = "join"
B = "ed"
C = A // B
PRINT *, C
STOP
END
```

will display "`joined`". Also, the program

```
IF ( ("ab" // "cd") .EQ.  "abcd" )  PRINT *, "equal"
STOP
END
```

will display "`equal`". That is, the single constant `"ab" // "cd"` and the constant `"abcd"` are equal.

Substrings

You can extract a substring of a character object by specifying the initial and last character positions, separated by the colon. For example, the expression:

```
S(I:L)
```

is the string of characters with initial character from the I^{th} character of `S` and with the last character from the L^{th} character of `S`.

Similarly, the expression:

```
A(J,K)(M:N)
```

is the string of characters with initial character from the M^{th} character of the array element `A(J,K)` and with the last character from the N^{th} character of that element. Note that there are *(L-M+1)* characters in the substring.

Rules and Restrictions for Substrings

- The first character position is numbered one (not zero).
- The initial and last character positions must be integer expressions.
- If the *first* expression is omitted, the substring begins with the *first* character of the string.
- If the *second* expression is omitted, the substring ends with the *last* character of the string.
- The result is undefined unless $0 < I \le L \le$ *the declared length.**
- Substrings may be used on the left and right sides of assignments and as procedure actual arguments.

Exponentiation

FORTRAN 77 allows raising real quantities to complex powers, or complex quantities to real or complex powers. The principal part of the logarithm is used. Also, multiple exponentiation is defined. For example:

```
A**B**C
```

is equivalent to:

```
A ** (B**C)
```

Binary, octal, and hexadecimal constants

These constant indicators are for use in DATA statements only.

The B, O, X, and Z constant indicators in a DATA statement are for binary, octal and hexadecimal constants. The B is for binary, the O is for octal, and the X or Z are for hexadecimal. These are *typeless* constants. For example:

```
      INTEGER*2 N1, N2, N3, N4
      DATA N1 /B'0011111'/, N2/O'37'/, N3/X'1f'/, N4/Z'1f'/
      WRITE ( *, 1 )   N1, N2, N3, N4
  1   FORMAT ( 1X, O4, O4, Z4, Z4 )
      STOP
      END
```

Each of the above integer constants has the value 31 decimal.

Restrictions on typeless constants

- These constants are typeless; they are stored in the variables without any conversion to match the type of the variable.

[1] *I* is the *initial* position and *L* is the *last* position.

□ If the receiving data type has *more* digits than are specified in the constant, zeros are *filled on the left.*

□ If the receiving data type has *fewer* digits than are specified in the constant, digits are *truncated on the left.* If nonzero digits are lost, an error message is displayed.

□ Specified leading zeros are ignored.

□ You can specify up to 8 bytes of data for any one variable.

□ For binary constants, each digit is 0 or 1.

□ For octal constants, each digit must be in the range 0 to 7.

□ For hexadecimal constants, each digit must be in the range 0 to 9 or in the range A to F, or a to f.

Relaxation of Restrictions

FORTRAN 77 has relaxed restrictions on mixed mode, constant expressions, and subscripts.

Mixed Mode

FORTRAN 77 has relaxed restrictions on mixed mode. For instance, it is permissible to combine integer and complex quantities in an expression.

Mixed INTEGER and LOGICAL

In Sun FORTRAN you can use a LOGICAL value anyplace where standard FORTRAN requires a numeric value; the compiler implicitly converts it to INTEGER. Also, logical operations are allowed on integers and, conversely, integer operations are allowed on logical variables. If you use these features, your program may not be portable.

The following example shows some combinations of integer and logical types:

```
demo% cat mix1.f
* mix1.f Mixed integer and logical
    INTEGER*2  I1, I2, I3
    LOGICAL    L1, L2, L3
    DATA  I1 / 8 /,  I2 / 'W' /,  I3 / 0 /
    DATA  L1 /.TRUE./,  L2 /.TRUE./,  L3 /.TRUE./

    WRITE (*,1) "      ","I1","I2","I3","L1","L2","L3"
 1  FORMAT ( 1X, A8, 6 ( 5X, A2 ) )
    WRITE ( *, 2 ) 'Before:', I1,I2,I3, L1,L2,L3
    L1 = L1 + 1
    I2 = .NOT. I2
    L2 = I1 .AND. I3
    L3 = I1 .OR. I2
    WRITE ( *, 2 ) 'After: ', I1,I2,I3, L1,L2,L3
 2  FORMAT ( 1X, A8, 6 ( 1X, O6 ) )
    STOP
    END
demo% f77 mix1.f
mix1.f:
 MAIN:
demo% a.out
              I1     I2     I3     L1     L2     L3
  Before:     10  53440      0      1      1      1
  After:      10 124337      0      2      0 124337
demo%
```

Resultant type

- □ If you use integer operands with a logical operator, the operation is done bit-by-bit on the internal value of the operands. The result of such an operation is an integer.
- □ If the operands are mixed integer and logical, then the logicals are converted to integers and the result is an integer.

Constant expressions

Constant expressions are permitted where a constant is allowed, except in DATA statements. (A constant expression is made up of explicit constants and parameters and the FORTRAN operators, except for exponentiation to a floating-point power). An adjustable dimension may now be an integer expression involving constants, arguments, and variables in common.

Subscripts

Subscripts may now be general integer expressions; the old *cv*±*c*′ rules have been removed. The DO loop bounds may be general integer, real, or double-precision expressions. Computed GO TO expressions and I/O unit numbers can be general integer expressions.

4.5. Parameters

FORTRAN 77 allows you to give a *symbolic name* to a constant. Once defined, such a *symbolic name* can be used anywhere a constant can be used, except in a FORMAT statement.

The **PARAMETER** Statement

The FORTRAN 77 PARAMETER statement defines a symbolic name. For example:

```
    CHARACTER  S*8
    REAL  X, Y, PI
    PARAMETER ( X=17, Y=X/3, PI=3.14159D0, S=’hello’ )
```

A variation of the PARAMETER statement is:

```
    PARAMETER (X=17), (Y=X/3), (PI=3.14159D0), (S=’hello’)
```

Rules and restrictions for symbolic names of constants

- The type of each name is governed by the same implicit and explicit type rules as for variables.
- These names are formed by the same rules as names of variables.
- Once defined, a symbolic name of a constant cannot be redefined in another statement. In particular, it cannot appear as the left-hand-side of an assignment statement.
- The PARAMETER statement must appear before all executable and DATA statements.
- The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters, and in Sun FORTRAN certain intrinsic functions, as listed in the next section.)

Intrinsic Functions

The PARAMETER statement allows clauses of the form s=e where s is a symbolic name and e is a compile-time constant expression. The expression can include the following intrinsic functions:

```
LOC, CHAR,
AND, OR, NOT, XOR, LSHIFT, RSHIFT, LGE, LGT, LLE, LLT,
MIN, MAX, ABS, MOD, ICHAR, NINT, DIM,
DPROD, CMPLX, CONJ, AIMAG
```

(The functions IAND, IOR, IEOR, and ISHFT are not available, but you can use the corresponding AND, OR, XOR, LSHIFT, or RSHIFT; or if you use the f77cvt program it will make these conversions for you. For more on f77cvt see Section 10.4 — “The Source Code Converter.”

Equivalence Statements

As a special and peculiar case, FORTRAN 66 permits an element of a multidimensional array to be represented by a singly-subscripted reference in EQUIVALENCE statements. FORTRAN 77 does not permit this usage, since subscript lower bounds may now be different from 1. (See Subsection *Array Declarations*, a little later in this section.) The Sun FORTRAN compiler permits single subscripts in EQUIVALENCE statements, under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

4.6. Static Variables

According to ANSI standard FORTRAN, local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is neither declared in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. The only exceptions are variables that have been defined in a DATA statement and never changed. These rules permit overlay and stack implementations for the affected variables.

The SAVE Statement

In FORTRAN 77 you can specify that certain variables and common blocks retain their values between invocations. For example, the declaration

Dummy arguments cannot appear in a SAVE statement.

```
SAVE A, /B/, C
```

leaves the values of the variables A and C and all of the contents of common block B unaffected by a RETURN. The simple declaration

```
SAVE
```

has this effect on all variables and common blocks in the procedure. A common block must be saved in every procedure in which it is declared if the desired effect is to occur.

Automatic/Static

In addition to the SAVE statement, Sun FORTRAN allows you to type variables as *static* or *automatic*. For *static* variables, there is exactly one copy of each datum, and its value is retained between calls. For *automatic* variables, there is one copy for each invocation of the procedure. For example:

```
STATIC A, B, C
STATIC REAL  P, D, Q
IMPLICIT AUTOMATIC (X-Z)
```

Rules and restrictions on static and automatic

- □ Local variables are static by default.
- □ Arguments and function values are automatic.

- Automatic variables cannot appear in EQUIVALENCE, DATA, or SAVE statements.

Array Declarations

In FORTRAN 77 arrays can have as many as seven dimensions. For example:

```
REAL TAO(2,2,3,4,5,6,10)
```

The lower bound of each dimension can be declared to be other than 1 by using a colon between the lower bound and the upper bound. If the lower bound and colon are omitted, the lower bound is assumed to be 1. For example:

```
REAL A(5:3, 7, 3:5), B(0:2)
```

Furthermore, an adjustable array bound can be an integer expression involving constants, arguments, and variables in common:

```
SUBROUTINE  POPUP ( A, B, N )
COMMON  / DEFS / M, L, K
REAL A(5:3, 7, M:N), B(N+1:2*N)
```

The upper bound on the last dimension of an array argument can be denoted by an asterisk to indicate that the upper bound is not specified. For example:

```
SUBROUTINE  PULLDOWN ( A, B, C )
INTEGER A(5, *),  B(*), C(0:1, 2:*)
```

4.7. Pointers

A *pointer*[*] is an integer variable that contains an address. The POINTER statement establishes pairs of variables and pointers, where each pointer contains the address of its paired variable. Any variable paired with a pointer in a POINTER statement is called a *pointer-based variable*, or just a *based variable.*

Pointer statement

The syntax for the POINTER statement is as follows:

```
POINTER (p1, v1) [, (p2, v2) ... ]
```

where *v1*, *v2* are pointer-based variables, and *p1*, *p2* are the corresponding pointers. A simple POINTER statement is shown in this example:

```
POINTER  ( P, V )
```

Here, V is a pointer-based variable, and P is its associated pointer.

[*] Sun FORTRAN pointers are compatible with Cray-1 Computer Systems FORTRAN (CFT).

Usage of pointers

Once you have defined a variable as based on a pointer, you *must* assign an address to that pointer before you can reference the pointer-based variable with standard FORTRAN. (Whenever your program references a pointer-based variable, that variable's address is taken from the associated pointer.) It is *your* responsibility to provide an address of a variable of the appropriate type and size.

Address assignment

Since no storage is allocated when a pointer-based variable is defined, you *must* provide a memory area of the right size and assign the address to a pointer, usually with the normal assignment statement or data statement. You can obtain the address from the intrinsic function `LOC()`, or you can obtain both the area of memory and the address from the function `MALLOC()`.

Address via **LOC()**

The following example uses the `LOC()` function to get an address:

```
* ptr1.f: Assign an address via LOC()
      POINTER    ( P, V )
      CHARACTER A*12, V*12
      DATA  A / "ABCDEFGHIJKL" /
      P = LOC( A )
      PRINT *, V(5:5)
      STOP
      END
```

In the above example, the `CHARACTER` statement allocates 12 bytes of storage for `A`, but *no* storage for `V`; it merely specifies the type of `V` because `V` is a pointer-based variable. Then we assign the address of `A` to `P` so now any use of `V` will refer to `A` via the pointer `P`. The program will print an `E`.

Address via **MALLOC()**

The following example uses `MALLOC()` to get the area of memory and its address:

```
* ptr2.f: Assign an address via MALLOC()
      POINTER    ( P, V )
      CHARACTER A*12, V*12
      DATA  A / "ABCDEFGHIJKL" /
      P = MALLOC(12)
      V = A
      PRINT *, V(5:5)
      STOP
      END
```

In this example, we get 12 bytes of memory and its address from the function `MALLOC()`, then assign the address of that block of memory to the pointer `P`. The program will print an `E`.

The Function **MALLOC ()**

The function MALLOC() allocates an area of memory and returns the address of the start of that area. The argument to the function is an integer specifying the amount of memory to be allocated, in bytes. If successful, it returns a pointer to the first element of the region, otherwise it returns an integer 0. The region of memory is not initialized in any way — assume it is garbage.

The following example uses MALLOC() to get the area of memory and its address:

```
      POINTER    ( P, V )
      CHARACTER V*12, Z*1
      P = MALLOC( 12 )
      WRITE ( 6, 1 ) P
    1 FORMAT ( 1X, Z )
      STOP
      END
```

In the above example, we obtain 12 bytes of memory from the function MALLOC() and assign the address of that block of memory to the pointer P.

The Subroutine **FREE ()**

The subroutine FREE() deallocates a region of memory previously allocated by MALLOC(). The argument given to FREE() must be a pointer previously returned by MALLOC(), but not already given to FREE(). The memory is returned to the memory manager, making it unavailable to the programmer.

For example:

```
      POINTER   ( P1, X ),   ( P2, Y ),   ( P3, Z )
      ...
      P1 = MALLOC ( 36 )
      ...
      CALL FREE ( P1 )
      ...
```

In the above example, we get 36 bytes of memory from MALLOC() and then after some other instructions, probably using that chunk of memory, we tell FREE() to return those same 36 bytes to the memory manager.

Pointer arithmetic

Since the value of a pointer is the address of the thing it points to, in bytes, you can do integer addition or subtraction with pointers. Thus, if you add 4 to PTR, that would start it at the fifth character in A, as in the following example, which also prints an E:

```
* ptr3.f: Arithmetic with pointers
  POINTER   ( PTR, V )
  CHARACTER A*12, V*12, Z*1
  DATA  A / "ABCDEFGHIJKL" /
  PTR = LOC(A)
  PTR = PTR + 4
  Z = V(1:1)
  PRINT *, Z
  STOP
  END
```

You can dimension these variables in a separate type declaration or DIMENSION statement, or you can specify the dimensions in the POINTER statement itself, as shown in this example:

```
  POINTER ( PTR, Y(4,100))
```

Rules and restrictions on pointers

- The pointers are of type *integer*, and are automatically typed that way by the compiler. You must *not* type them yourself.
- The pointer-based variables can be of any type, including structures.
- No storage is allocated when such a pointer-based variable is defined, even if there is a size specification in the type statement.
- You can't use a pointer-based variable as a dummy argument or in COMMON, EQUIVALENCE, DATA, or NAMELIST statements.
- A pointer-based variable cannot itself be a pointer.
- The dimension expressions for pointer-based variables must be constant expressions in main programs. In subroutines and functions, the same rules apply for pointer-based array variables as for dummy arguments: the expression can contain dummy arguments and variables in common. Any variables in the expressions must be defined with an integer value at the time the subroutine or function is called.

Optimization and pointers

Pointers have the annoying side effect of reducing the assumptions that the global optimizer can make. In particular, in the absence of pointers, when a subroutine or function is called, the optimizer previously could be assured that the call will change only variables in common or those passed as arguments to that call. This is no longer valid, since a routine can take the address of an argument and save it in a pointer in common for use in a subsequent call to itself or to another routine. Therefore, the optimizer must assume that a variable passed as an argument in a subroutine or function call can be changed by any other call.

Such an unrestricted use of pointers would degrade optimization for the vast majority of programs that *don't* use pointers. Thus, the compiler places the following restrictions on the use of pointers whenever optimization is selected:

- Subroutines and functions are not permitted to save the address of any of their arguments between calls; a function can't return the address of any of its arguments, although it can return the value of a pointer argument.
- Only those variables whose addresses are explicitly taken with the `LOC()` or `MALLOC()` functions can be referenced through a pointer.

The following is an example of one kind of code that could cause trouble when optimization is enabled:

```
COMMON A, B, C
POINTER ( P, V )
P = LOC(A) + 4          ← possible problems if optimized
```

The compiler will assume that a reference through `P` may change `A`, but not `B`; this assumption could produce incorrect code.

4.8. Structures

Sun FORTRAN has an extension that allows organizing data into *structures*. The structure declaration defines the form of a *record* by specifying the name, type, size, and order of the *fields* that constitute the record. Once a structure is defined and named, it can be used in RECORD statements, as explained below. The structure declaration has the following syntax:

Structure declaration

```
STRUCTURE [/structure-name/] [field-list]
     field-declaration
     [field-declaration]
          . . .
     [field-declaration]
END STRUCTURE
```

where *structure-name* is the name of the structure, *field-list* is a list of fields of the specified structure, each *field-declaration* defines a field of the record, and

Field declaration

each field declaration can be one of the following:

- A substructure (either another structure declaration, or a record that has been previously defined)
- A *union* declaration (described below)
- A FORTRAN type declaration

An example of a STRUCTURE declaration is:

```
STRUCTURE /PRODUCT/
     INTEGER*4     ID
     CHARACTER*16 NAME
     CHARACTER*8  MODEL
     REAL*4       COST
     REAL*4       PRICE
END STRUCTURE
```

In this example, a *structure* named PRODUCT is defined to consist of the five fields ID, NAME, MODEL, COST, and PRICE. For an example with a *field-list*, see "Structure within a structure" later in this section.

Rules and restrictions for structures

- The name is enclosed in slashes and is optional only in nested structures.
- If slashes are present, a name must be present.
- You can specify the *field-list* within nested structures only.
- There must be at least one *field-declaration.*
- Each *structure-name* must be unique among structures, although you can use structure names for fields in other structures or as variable names.

- The only statements allowed between the STRUCTURE statement and the END STRUCTURE statement are *field-declaration* statements and PARAMETER statements. A PARAMETER statement inside a structure declaration block is equivalent to one outside.

Rules and restrictions for fields

Fields that are type declarations use the identical syntax of normal FORTRAN type statements, and all Sun FORTRAN types are allowed, subject to the following rules and restrictions:

- Any dimensioning needed must be in the type statement. The DIMENSION statement has no effect on field names.
- You can specify the pseudo-name %FILL for a field name to align fields in a record.
- You must explicitly type all field names. The IMPLICIT statement does not apply to statements in a STRUCTURE declaration, nor do the implicit I,J,K,L,M,N rules apply.
- You can't use arrays with adjustable or assumed size in field declarations, nor can you include passed-length CHARACTER declarations.
- **Field offsets** — In a structure declaration, the offset of field *n* is the offset of the preceding field, plus the length of the preceding field, possibly corrected for any adjustments made to maintain alignment. For a summary of storage allocation, see the Subsection \(lqStorage Allocation\(rq in Section 4.3 — "Data Types."

Record declaration

The RECORD statement declares variables to be records with a specified structure, or declares arrays to be arrays of such records. The syntax of a RECORD statement is as follows:

```
RECORD /structure-name/ record-list
     [,/structure-name/ record-list]
              .
              .
     [,/structure-name/ record-list]
```

where *structure-name* is the name of a previously declared structure, and *record-list* is a list of variables, arrays, or arrays with dimensioning and index ranges, separated by commas.

For example, using the structure in the example above:

```
RECORD /PRODUCT/  CURRENT, PRIOR, NEXT, LINE(10)
```

Each of the three variables CURRENT, PRIOR, and NEXT is a record which has the PRODUCT structure, and LINE is an array of 10 such records.

Rules and restrictions for records

- Each record is allocated separately in memory.
- Initially, records have undefined values.
- Records, record fields, record arrays, and record-array elements are allowed as arguments and dummy arguments. When you pass records as arguments, their fields must match in type, order, and dimension. The record declarations in the calling and called procedures must match. Within a union declaration, the order of the map fields is not relevant — see "Unions and maps," later in this section.
- Records and record fields are allowed in COMMON and DIMENSION statements.
- Records and record fields are not allowed in DATA, EQUIVALENCE, NAMELIST, or SAVE statements.

Record and field reference

You can refer to a whole record, or to an individual field in a record, and since structures can be nested, a field can itself be a structure, so you can refer to fields within fields, within fields, etc. The syntax of record and field reference is as follows:

record-name [*.field-name*] ... [*.field-name*]

where *record-name* is the name of a previously defined record variable, and each *field-name* is the name of a field in the record immediately to the left.

Examples of references are given below, based on the structure and records of the above two examples:

```
RECORD /PRODUCT/  CURRENT, PRIOR, NEXT, LINE(10)
...
CURRENT = NEXT
LINE(1) = CURRENT
WRITE ( 9 )  CURRENT
NEXT.ID = 82
```

In this example, the first assignment statement copies one whole record (all five fields) to another record, the second assignment statement copies a whole record into the first element of an array of records, the WRITE statement writes a whole record, and the last statement sets the ID of one record to 82.

A complete sample program is listed below to show structure and record declarations, record and field assignments, and field output:

```
demo% cat str1.f
* str1.f Simple structure
    STRUCTURE  / S /
            INTEGER*4        I
            REAL*4           R
    END STRUCTURE

    RECORD / S / R1, R2

    R1.I = 82
    R1.R = 2.7182818
    R2 = R1
    WRITE ( *, * )  R2.I, R2.R
    STOP
    END
demo% f77 str1.f
str1.f:
 MAIN:
demo% a.out
   82   2.718280
demo%
```

Substructure declaration

A structure can have a field that is also a structure. Such a field is called a *substructure*. You can declare a substructure in either of two ways:

- A RECORD declaration within a structure declaration
- A structure declaration within a structure declaration (nesting)

Record within a structure

A nested structure declaration is one that is contained within either a structure declaration or a union declaration (see below). You can use a previously defined record within a a structure declaration. For example, using the previously defined record PRODUCT you can define a structure SALE:

```
    STRUCTURE /SALE/
        CHARACTER*32        BUYER
        INTEGER*2           QUANTITY
        RECORD   /PRODUCT/  ITEM
    END STRUCTURE
```

In the above example, the structure SALE contains three fields: BUYER, QUANTITY, and ITEM, where ITEM is a record with the structure /PRODUCT/.

Structure within a structure

You can nest a declaration within a declaration. For example, if /PRODUCT/ is *not* declared previously, then you can declare it within the declaration of SALE:

```
STRUCTURE /SALE/
    CHARACTER*32        BUYER
    INTEGER*2           QUANTITY
    STRUCTURE /PRODUCT/ ITEM
        INTEGER*4       ID
        CHARACTER*16    NAME
        CHARACTER*8     MODEL
        REAL*4          COST
        REAL*4          PRICE
    END STRUCTURE
END STRUCTURE
```

Here the structure SALE still contains the same three fields as in the prior example: BUYER, QUANTITY, and ITEM. The field ITEM is an example of a *field-list* (in this case, a single-element list), as defined under "Structure declaration."

The size and complexity of the various structures determine which style of substructure declaration is best to use in a given situation.

Field reference in substructures

You can refer to fields within substructures; for example, with PRODUCT and SALE in the current program unit:

```
...
RECORD /SALE/ JAPAN
...
N = JAPAN.QUANTITY
I = JAPAN.ITEM.ID
...
```

Rules and restrictions for substructures

- You must define at least one field name for any substructure.
- No two fields at the same nesting level can have the same name. Fields at different levels of a structure can have the same name (although doing so might be questionable programming practice).
- You can use the pseudo-name %FILL to align fields in a record. This makes an unnamed empty field.
- You must not include a structure as a substructure of itself, at any level of nesting.

Unions and maps

A *union* declaration defines groups of fields that share memory at runtime.

Union declaration

The syntax of a union declaration is as follows:

```
UNION
     map-declaration
     map-declaration
     [map-declaration]
          . . .
     [map-declaration]
END UNION
```

Map declaration

The syntax of a map declaration is as follows:

```
MAP
     field-declaration
     [field-declaration]
          . . .
     [field-declaration]
END MAP
```

A *map* declaration defines alternate groups of fields in a union. During execution, one map at a time is associated with a shared storage location. When you reference a field in a map, the fields in any previous map become undefined and are succeeded by the fields in the map of the newly referenced field. The amount of memory used by a union is that of its biggest map.

For example, you can declare the structure /STUDENT/ to contain either NAME, CLASS, and MAJOR — or NAME, CLASS, CREDITS, and GRAD_DATE:

```
STRUCTURE /STUDENT/
     CHARACTER*32        NAME
     INTEGER*2           CLASS
     UNION
          MAP
              CHARACTER*16 MAJOR
          END MAP

          MAP
              INTEGER*2    CREDITS
              CHARACTER*8  GRAD_DATE
          END MAP
     END UNION
END STRUCTURE
```

If you define the variable PERSON to have the structure /STUDENT/ from the above example, then PERSON.MAJOR references a field from the first map, and PERSON.CREDITS references a field from the second map. If the variables of the second map field are initialized and then the program references the variable PERSON.MAJOR, the first map becomes active and the variables of the second map become undefined.

Fields in a map

Each *field-declaration* in a *map* declaration can be one of the following:

□ a *structure* declaration

□ a *record*

□ a *union* declaration

□ a declaration of a typed data field

4.9. Data Representations

Whatever the size of the data element in question, the most significant bit of the data element is always in the lowest-numbered character of the character sequence required to represent that object.

Representation of REAL and DOUBLE PRECISION

Both REAL and DOUBLE PRECISION data elements are represented according to the IEEE standard:

Table 4-3 *Floating-Point Representation*

	Single-Precision	***Double-Precision***
Sign	bit 31	bit 63
Exponent	bits 30–23 bias 127	bits 62–52 bias 1023
Fraction	bits 22–0	bits 51–0
Range approx.	3.402823e+38 1.175494e-38	1.797693e+308 2.225074e-308

A REAL or DOUBLE PRECISION number is represented by the form:

$$(-1)^{sign} * 2^{exponent-bias} * 1.f$$

where *f* is the bits in the fraction.

Representation with Extreme Exponents

zero (signed)
is represented by an exponent of zero and a fraction of zero.

subnormal number
The form of a subnormal number is

$$(-1)^{sign} * 2^{1-bias} * 0.\mathrm{f}$$

where *f* is the bits in the significand.

signed infinity
(that is, affine infinity) is represented by the largest value that the exponent can assume (all ones), and a zero fraction.

Not a Number (NaN)

is represented by the largest value that the exponent can assume (all ones), and a nonzero fraction.

Normalized `REAL` and `DOUBLE PRECISION` numbers have an implicit leading bit that provides one more bit of precision than usual.

Hexadecimal Representation of Selected Numbers

Table 4-4 *Hexadecimal Representation of Selected Numbers*

Value	*Single-Precision*	*Double-Precision*
+0	00000000	0000000000000000
-0	80000000	8000000000000000
+1.0	3F800000	3FF0000000000000
-1.0	BF800000	BFF0000000000000
+2.0	40000000	4000000000000000
+3.0	40400000	4008000000000000
+Infinity	7F800000	7FF0000000000000
-Infinity	FF800000	FFF0000000000000
NaN	7Fxxxxxx	7FFxxxxxxxxxxxxx

Arithmetic Operations on Extreme Values

This section describes the results of basic arithmetic operations performed on combinations of extremal and ordinary values. No traps or any other exception actions are taken. All inputs are assumed to be positive. Overflow and underflow are assumed not to happen. Table 4-5 summarizes the abbreviations used in the following tables:

Table 4-5 *Abbreviations for Numbers*

Abbreviation	*Meaning*
Sub	Subnormal Number
Num	Normalized Number
Inf	Infinity (positive or negative)
NaN	Not a Number
Uno	Unordered

Inf + Inf = Inf
Inf - Inf = NaN

Addition and Subtraction

Left Operand	*Right Operand*				
	0	Sub	Num	Inf	NaN
0	0	Sub	Num	Inf	NaN
Sub	Sub	Sub	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	See Note	NaN
NaN	NaN	NaN	NaN	NaN	NaN

NS means either Num or Sub result possible.

Multiplication

Left Operand	*Right Operand*				
	0	Sub	Num	Inf	NaN
0	0	0	0	NaN	NaN
Sub	0	0	NS	Inf	NaN
Num	0	NS	Num	Inf	NaN
Inf	NaN	Inf	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Division

Left Operand	*Right Operand*				
	0	Sub	Num	Inf	NaN
0	NaN	0	0	0	NaN
Sub	Inf	Num	Num	0	NaN
Num	Inf	Num	Num	0	NaN
Inf	Inf	Inf	Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN

If either X or Y is NaN, then
X .NE. Y is TRUE
and the others
(.EQ., .GT., .GE., .LT., .LE.)
are FALSE .
+0 compares equal to -0.

If any argument is NaN, then
the results of MAX or MIN
are undefined.

Comparison					
Left Operand	*Right Operand*				
	0	Sub	Num	Inf	NaN
0	=	<	<	<	Uno
Sub	>		<	<	Uno
Num	>	>		<	Uno
Inf	>	>	>	=	Uno
NaN	Uno	Uno	Uno	Uno	Uno

5

Control and Program Structures

5

Control and Program Structures

5.1. Overview

FORTRAN 77 and Sun FORTRAN have added various features which control which statements get executed next. This chapter describes such features, including DO loops, IF-THEN-ELSE, the INTRINSIC statement, alternate entries, alternate returns, recursion, and block data.

5.2. Alternate Returns

FORTRAN 77 allows you to return from a subroutine to a *specified labeled statement* in the calling routine. To do this, the corresponding arguments of the CALL and SUBROUTINE statements must indicate that the argument passed is a statement label, and the RETURN statement indicates which alternate return to use. The alternate return syntax for these three statements is illustrated in the examples below:

- In the CALL statement, the argument of an alternate return is a statement label preceded by an asterisk. For example:

```
CALL SHRED ( J, *90, M, *5 )
```

- In the SUBROUTINE or ENTRY statement, the argument of an alternate return is an asterisk. For example:

```
SUBROUTINE SHRED ( A, *, B, * )
```

- In the RETURN statement, you select a return by an optional integer expression. For example:

```
RETURN 2
```

In the above example, the "RETURN 2" causes a return to the 2^{nd} alternate return, in this case, statement number 5.

In general, "RETURN K" is a branch to the K^{th} statement label.

Restriction: For "RETURN K" we must have $1 \leq K \leq N$, where N is the number of alternate return (asterisk) arguments in the SUBROUTINE statement. If K is outside that range, then the usual return to the statement following the CALL is executed. Note that for two or more alternate return arguments, it is usually clearer to put them together at the end of the argument list.

5.3. Block Data Statements

Block data procedures can have a name. For example:

```
BLOCK DATA STUFF
```

According to the standard, only *one* unnamed block data procedure can appear in a program. The standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional linkers.

5.4. The DO Loop

Several extensions and refinements to the DO loop have been implemented since FORTRAN 66.

The DO Variables

Starting with FORTRAN 77, the DO variables and range parameters may be of INTEGER, REAL, or DOUBLE PRECISION types. However, the use of floating-point DO variables is dangerous because of the possibility of unexpected roundoff, and we strongly recommend against it. The action of the DO statement is defined for all values of the DO parameters. The statement

```
DO 10 I = L, U, D
```

performs $\max(0, \lfloor (U-L+D)/D \rfloor)$ iterations. The DO variable has a predictable value when exiting a loop — the value at the time a GO TO or RETURN terminates the loop; otherwise, it is the value that failed the limit test.

One-Trip DO Loops

In order to accommodate certain types of old programs, the **-onetrip** compiler flag makes f77 generate loops that are executed at least once. The FORTRAN 66 standard states that the effect of such a statement is undefined, but it is common practice that the range of a DO loop is performed at least once. Note that the FORTRAN 77 standard requires that the range of a DO loop not be performed if the initial value is already past the limit value, but this **-onetrip** flag allows you to override that requirement. For example:

```
DO 10 I = 2, 1
```

With **-onetrip** the above loop will be executed exactly once; without it, the loop will *not* be executed at all.

Unlabeled DO Loops

Sun FORTRAN extends the standard FORTRAN 77 loop by not requiring a label for the executable statement that terminates the loop. Compare:

```
      DO  3  I = 1, 10
      ...
 3    CONTINUE
```

and the following loop, which works exactly the same as the preceding example:

```
      DO  I = 1, 10
      ...
      END DO
```

Note that you cannot terminate nested loops with a single unlabeled END DO statement. For example, the following is *NOT* correct:

```
DO I = 1, 10
DO J = 5, 8
 ...
END DO                           ←invalid syntax
```

One acceptable alternative:

```
DO I = 1, 10
DO J = 5, 8
...
END DO
END DO
```

If a DO loop doesn't have a label, it *must* terminate with an END DO statement.

Indefinite DO WHILE Loops

Sun FORTRAN extends the standard FORTRAN 77 loop with the DO WHILE statement. This statement tests a logical expression before executing the statements inside the loop: as long as the expression's value is true, the loop repeats; as soon as the loop test finds the expression's value false, the loop terminates.

As with an unlabeled DO statement, a DO WHILE statement uses an END DO to terminate. For example:

```
CHARACTER*132 LINE
     ...
READ ( *, * )   LINE
L = 132
DO WHILE ( L .GT. 1 .and. LINE(L:L) .EQ. ' ' )
L = L-1
END DO
```

As an alternative, you can specify a label to terminate a DO WHILE statement. For example:

```
        DO 9 WHILE ( X .LE. 0.0 )
        ...
9        CONTINUE
```

Extended Range Removed

In FORTRAN 66, under a set of restrictive and rarely understood conditions, it is permissible to jump out of the range of a DO loop, then jump back into it. Extended range has been removed in the FORTRAN 77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss. You can transfer control out of a DO WHILE loop, but you can't transfer control *into* a loop from outside it.

5.5. The ENTRY Statement

FORTRAN 77 allows multiple entry points. Subroutine and function subprograms can have additional entry points, declared by an ENTRY statement with an optional argument list. For example:

```
SUBROUTINE FINAGLE( A, B, C )
INTEGER    A, B
CHARACTER  C*4
  ...
RETURN

ENTRY SCHLEP( A, B, C )
  ...
RETURN

ENTRY SHMOOZ
  ...
RETURN
END
```

In the above example, the subroutine FINAGLE has two entries; the entry SCHLEP has an argument list; the entry SHMOOZ has no argument list. In the calling routine we can call the above subroutine and entries as follows:

```
INTEGER    A, B
CHARACTER  C*4
  ...
CALL FINAGLE( A, B, C )
  ...

CALL SCHLEP( A, B, C )
  ...

CALL SHMOOZ
  ...
```

The order of the call statements need not match the order of the entry statements.

Rules and restrictions for ENTRY

- Execution begins at the first statement following the ENTRY line.
- All declarations of variables must precede all executable statements in the procedure.
- If the procedure begins with a SUBROUTINE statement, each entry point is a subroutine name.
- If it begins with a FUNCTION statement, each entry is a function name, with the type determined by the declared entry name's type.
- If any entry is a character-valued function, then all entries must be character-valued functions.
- In a function, an entry name of the same type as that where control entered must be assigned a value.
- Arguments do not retain their values between calls. The ancient trick of calling one entry point with a large number of arguments so that the procedure 'remembers' the locations of those arguments, then invoking an entry with just a few arguments for later calculation is still illegal. Furthermore, the trick doesn't work in this implementation, since arguments are not kept in static storage.

5.6. The IF-THEN-ELSE Statement

The IF-THEN-ELSE branching structure is standard with FORTRAN 77. It is often called a 'Block If'. A Block If begins with a statement of the form:

```
IF ( ... ) THEN
```

and ends with an

```
END IF
```

statement.

Two other new statements can appear in a Block If. There can be several

```
ELSE IF ( ... ) THEN
```

statements, followed by at most one

```
ELSE
```

statement.

End-of-line: There must be nothing more on the *same* line after the "IF (...) THEN" or after the"ELSE IF (...) THEN".

If the logical expression in the Block If statement is true, the statements following it up to the corresponding ELSE IF, ELSE, or END IF are executed. Otherwise, the next ELSE IF statement in the group is executed. If none of the ELSE IF conditions is true, control passes to the statements following the ELSE statement, if any. The ELSE must follow all ELSE IFs in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures.

One way to approximate a case construct is:

```
IF ( S .EQ. 'ab' ) THEN
  ...
ELSE IF ( S .EQ. 'cd' ) THEN
  ...
ELSE IF ( S .EQ. 'ef' ) THEN
  ...
ELSE
  ...
END IF
```

5.7. The INTRINSIC Statement

You can pass the name of a routine as an argument to another routine. If the name of an intrinsic function is to be passed to another routine, it must be declared INTRINSIC. Declaring it EXTERNAL passes a function other than the built-in one. All of the functions specified in the standard are in a single category, 'intrinsic functions,' rather than being divided into 'intrinsic' and 'basic external' functions. For example:

```
INTRINSIC SIN
  ...
CALL BENCH ( N, X, T, SIN )
```

5.8. Program Statement

FORTRAN 77 allows a main program to be begin with a statement that gives that program an external name. For example :

```
PROGRAM WORK
```

5.9. Recursion

Procedures can call themselves, directly or through a chain of other procedures. For example:

```
demo% cat rec1.f
* rec1.f -- Recursion: Factorial
    INTEGER  FACT, I, N
 1  WRITE( *, 2 )
 2  FORMAT( "N? ", $ )
    READ( *, * ) I
    IF ( I .LT. 0 .OR. I .GT. 16 ) STOP
    N = FACT( I )
    WRITE( *, * ) "N! = ", N
    GO TO 1
    END

    FUNCTION FACT( I )
    INTEGER  FACT
    IF ( I .LT. 2 ) THEN
        FACT =  1
    ELSE
        FACT = I * FACT( I - 1 )
    END IF
    RETURN
    END
demo% f77 rec1.f
rec1.f:
 MAIN recurs:
        fact:
"rec1.f", line 31: Warning: recursive call
demo% a.out
N? 3
  N! =    6
N? -1
demo% █
```

But note that a subroutine or function cannot pass its own name as a procedure parameter. To do so would require the name to appear in an EXTERNAL statement, which is prohibited by the ANSI standard. Note also that use of recursion may make FORTRAN programs nonportable.

6

The File System and FORTRAN I/O

6

The File System and FORTRAN I/O

This chapter is a basic introduction to the SunOS file system and how it relates to the FORTRAN I/O system. Topics covered include:

- Hierarchy
- Directories
- Filenames
- Pathnames
- Redirection
- Piping

If you understand these concepts then skip this chapter. For a more detailed discussion of the SunOS file system structure, refer to the *Getting Started with SunOS: Beginner's Guide.*

6.1. Hierarchy

The basic file system consists of a hierarchical file structure, established rules for filenames and pathnames, and various commands for moving around in the file system, showing your current location in the file system, and making, deleting or moving files or directories.

The system file structure of SunOS or of UNIX is analogous to an upside-down tree. The top of the file system is the `root` — directories, subdirectories, and files all branch *down* from the root. Directories and subdirectories are considered nodes on the directory tree, and can have subdirectories or ordinary files branching down from them. The only directory that is not a subdirectory is the root directory, so except for this instance, we do not usually make a distinction between directories and subdirectories.

A sequence of branching directory names and a filename in the file system tree describes a *path*. Files are at the ends of paths, and cannot have anything branching from them. When moving around in the file system, *down* means away from the root and *up* means toward the root. The figure below shows a diagram of the file system tree structure.

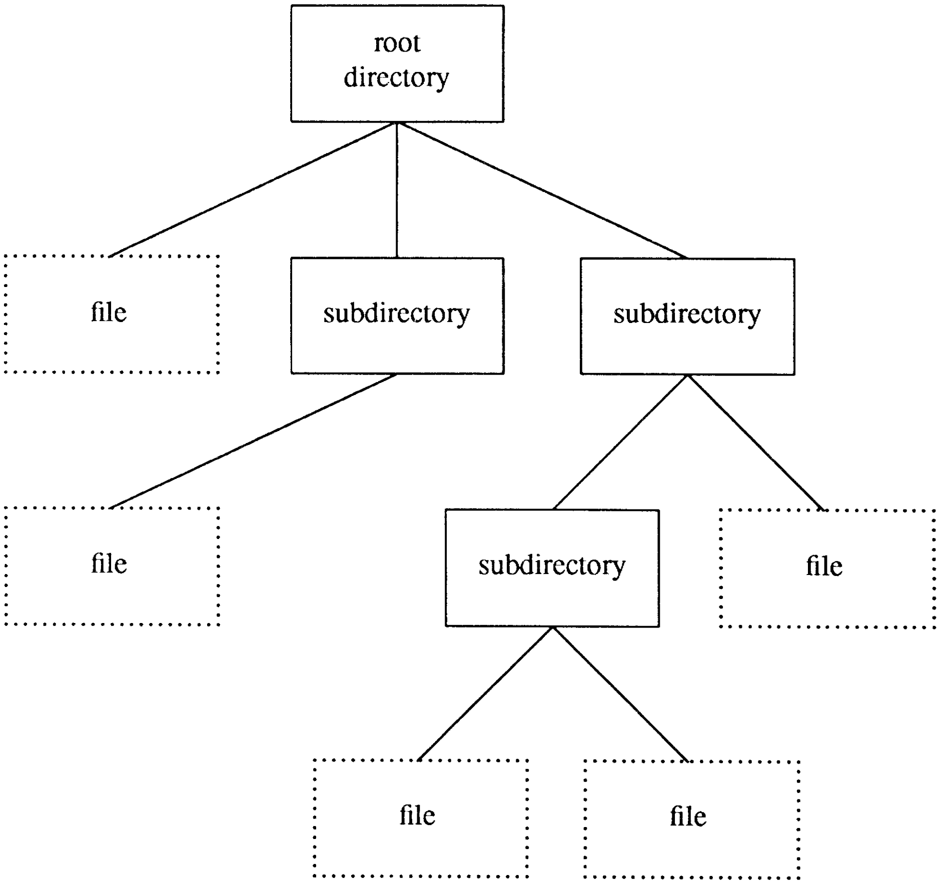

Figure 6-1 *Diagram showing a sample file system structure*

6.2. Directories

All files branch from directories except the root directory. Directories are just files with special properties. While you are logged on to a Sun system, you are said to be *in a directory*. When you first log on, you are usually in your *home* directory. At any time, wherever you are, the directory you are in is called your *current working directory*. It is often useful to list your current working directory. The `pwd` command *prints* the current working directory name and the `GETCWD` routine *gets* (returns)the current working directory name. You can change your current working directory simply by moving to another directory. The `cd` shell command and the `CHDIR` routine change to a different directory. Additional explanations of the file system organization and relevant shell commands are located in the *Getting Started with SunOS: Beginner's Guide.*

6.3. Filenames

All files have names, and you can use almost any character in a filename. The name can be up to 1024 characters long, but individual components can be only 512 characters long. However, to prevent the shell from misinterpreting certain special punctuation characters, you should restrict your use of punctuation in filenames to the dot (.), underscore (_), comma (,), plus (+), and minus (-). The slash (/) character has a specific meaning in a filename, and is only used to separate components of the pathname (as described below). Also, you should avoid using blanks in filenames. Directories are just files with special properties and follow the same naming rules as files. The only exception is the root directory, which is named slash (/).

6.4. Pathnames

To describe a file anywhere in the file system, you can list the sequence of names for the directory, subdirectory, etc., and file, separated by slash characters, down to the file you want to describe. If you show *all* the directories, starting at the *root*, that's called an *absolute* pathname. If you show only the directories below the current directory, that's called a *relative* pathname.

Relative pathnames

From anywhere in the directory structure, you can describe a *relative pathname* of a file. Relative pathnames start with the directory you are in (the current directory) instead of the root. For example, if you are in the directory "`/usr/you`", and you use the *relative* pathname:

```
mail/record
```

that is equivalent to using the *absolute* pathname:

```
/usr/you/mail/record
```

This is illustrated in the diagram below:

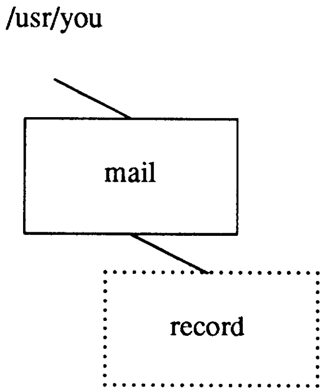

Figure 6-2 *Relative Path Name*

Absolute pathnames

A list of directories and a filename, separated by slash characters, from the root to the file you want to describe, is called an *absolute pathname.* It is also called the *complete file specification* or the *complete pathname.*

A complete file specification has the general form:

```
/directory/directory/.../directory/file
```

There can be any number of directory names between the root (/) and the file at the end of the path as long as the total number of characters in a given pathname is less than or equal to 1024.

An absolute pathname is illustrated in the diagram below:

/usr/you/mail/record

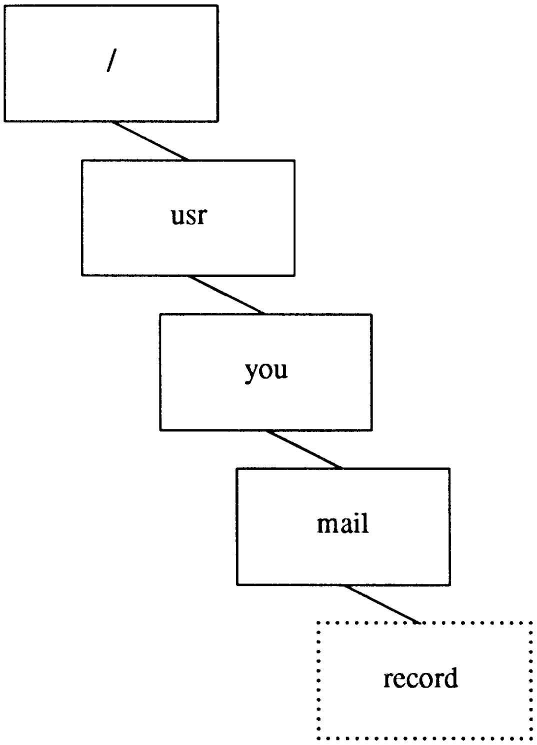

Figure 6-3 *Absolute Path Name*

6.5. Redirection

Redirection is a way of changing the files that a program uses without passing a filename to the program. Both input to and output from a program can be redirected. The symbol for redirecting standard input is the 'less than' sign (<), and for standard output is the 'greater than' sign (>).

File redirection is a function performed by the command interpreter or *shell* when a program is invoked by it. The shell command line:

```
demo% myprog < mydata
```

causes the file `mydata` (which must already exist) to be connected to the standard input of the program `myprog` when it is run. This means that if `myprog` is a FORTRAN program and reads from unit 5, it reads from the `mydata` file. Similarly, the shell command line:

```
demo% myprog > myoutput
```

causes the file `myoutput` (which is created if it does not exist, or rewound and truncated if it does) to be connected to the standard output of the program `myprog` when it is run. So if the FORTRAN program `myprog` writes to unit 6, it writes to the file `myoutput`.

Both standard input and standard output may be redirected to and from different files on the same command line. Standard error may also be redirected so it does not appear on your workstation's display. In general, this is not a good idea, since you usually want to see error messages from the program immediately, rather than sending them to a file.

The shell syntax to redirect standard error varies, depending on whether you are using the Bourne shell or the C shell. Refer to the *Beginner's Guide to the Sun Workstation* for more information on redirecting standard error.

6.6. Piping

You can connect the standard output of one program directly to the standard input of another without using an intervening temporary file. The mechanism to accomplish this is called a *pipe*. A shell command line using a pipe looks like this:

```
demo% firstprog | secondprog
```

This causes the standard output (unit 6) of `firstprog` to be piped to the standard input (unit 5) of `secondprog`. Piping and file redirection can be combined in the same command line. A simple example is:

```
demo% myprog < mydata | wc > datacount
```

in which the program `myprog` takes its standard input from the file `mydata`, and has its standard output piped into the standard input of the `wc` command, the standard output of which is redirected into the file `datacount`.

7

Input and Output

7

Input and Output

7.1. Overview

This chapter describes I/O features added by FORTRAN 77 and Sun FORTRAN. Topics covered include:

- General concepts of FORTRAN I/O
- FORTRAN I/O statements
- Formats
- Magnetic tape I/O

7.2. General Concepts of FORTRAN I/O

SunOS is not as format-oriented as FORTRAN. It treats files as sequences of characters instead of collections of records. The FORTRAN runtime system keeps track of file formats and access modes. It also provides the file facilities, including the FORTRAN libraries and the standard I/O library.

Logical Units

The maximum number of logical units that a program can have open at one time is the same as the SunOS system limit, currently 64.

The standard logical units 0, 5, and 6 are named internally `stderr`, `stdin`, and `stdout`, respectively. These are not actual filenames and cannot be used for opening these units. `INQUIRE` does not return these names and indicates that the above units are not named unless they have been opened to real files. However, these units can be redefined with an `OPEN` statement.

The names `stderr`, `stdin`, and `stdout` are meant to make error reporting more meaningful. To preserve error reporting, it is an error to close logical unit 0, although it can be reopened to another file.

If you want to open the default filename for any preconnected logical unit, remember to close the unit first. Redefining the standard units may impair normal console I/O. An alternative is to use shell redirection to externally redefine the above units.

To redefine default blank control or the format of the standard input or output files, use the `OPEN` statement specifying the unit number and no filename (see below).

I/O Errors

If the user's program does not trap I/O errors an appropriate error message is written to stderr before aborting. An error number is printed in square brackets, [ ], along with a brief error message showing the logical unit and I/O state. Error numbers < 100 refer to SunOS errors; these are described in *intro*(2) in the *SunOS Reference Manual*. Error numbers ≥ 100 come from the I/O library, and are described further in Appendix D of this manual. For external I/O, part of the current record will be displayed if the error was caused during reading from a file that can backspace. For internal I/O, part of the string is printed with a vertical bar (|) at the current position in the string.

Forms of I/O

The four forms of FORTRAN I/O are *formatted*, *unformatted*, *list-directed*, and *namelist.*

There are two modes of access to files: *sequential* and *direct.* When a file is opened, sequential or direct access is set, explicitly or by default.

There are two types of files: *external* and *internal.* Most files are external files; they reside on physical peripheral devices. The FORTRAN 77 language allows five types of external (peripheral device) files: sequential formatted, sequential unformatted, direct formatted, direct unformatted, and list-directed sequential.

An internal file is a location in memory that can be read from and written to like a peripheral device.

See **Table** 7-1 for a summary of FORTRAN I/O.

I/O Execution

Direct-access, list-directed I/O is not allowed. Direct-access, namelist I/O is not allowed. Namelist I/O on internal files is not allowed. Unformatted, internal I/O is not allowed. All other flavors of I/O are allowed, although some are not part of the ANSI standard. Any error detected during I/O processing will cause the program to abort unless alternative action has been provided specifically in the program. Any I/O statement may include an ERR= clause (and IOSTAT= clause) to specify an alternative branch to be taken on errors (and return the specific error code). Read statements may include END=*n* to branch on end-of-file. File position and the value of I/O list items are undefined following an error.

Summary of Sun FORTRAN Input and Output

Table 7-1 *Summary of Sun FORTRAN Input and Output*

Type of file		Access mode	
		Sequential	Direct
Formatted	internal	file is a character variable, array element, array, or substring	file is a character array; each record is one array element
	external	only formatted records of same or variable length	only formatted records, all the same length
Unformatted	internal	(not allowed)	(not allowed)
	external	contains only unformatted records	`READ`: one logical record at a time. `WRITE`: unfilled part of record undefined.
List-directed	internal	`READ`: reads characters until eof or I/O list is satisfied; `WRITE`: writes records until list is satisfied. *	(not allowed)
	external	values input or output depend on types in list.	(not allowed)
Namelist	internal	(not allowed)	(not allowed)
	external	`READ`: scans records (cols 2-80) for " `$`*groupname*", then for names in that group, and stores data in those variables. Scans until a "`$`", or eof; `WRITE`: writes records showing *groupname* and each variable name with value.	(not allowed)

* Avoid list-directed internal writes: the number of lines and items per line vary with the values of items. See "List-Directed I/O", later in this chapter.

Print Files

The ANSI standard is ambiguous regarding the definition of a 'print' file. Since SunOS has no default 'print' file, an additional FORM specifier is now recognized in the OPEN statement. Specifying FORM='PRINT' implies formatted output and enables vertical format control for that logical unit. Vertical format control is interpreted only on sequential formatted writes to a 'print' file (see "Vertical Format Control", later in this chapter).

The INQUIRE statement returns 'PRINT' in the FORM variable for logical units opened as 'print' files. It returns -1 for the unit number of an unopened file.

If a logical unit is already open, an OPEN statement including the FORM option or the BLANK option does nothing but redefine those options. This instance of the OPEN statement need not include the filename, and must not include a filename if UNIT refers to standard input or output. Therefore, to redefine the standard output as a 'print' file, use:

```
OPEN( UNIT=6, FORM='PRINT')
```

Scratch Files

To prevent a temporary file from disappearing after execution is completed, you must execute a CLOSE statement with STATUS='KEEP' . It is the default for all other files. Remember to get the scratch file's real name, using INQUIRE, if you want to reopen it later.

Carriagecontrol on All Files

Traditional FORTRAN environments usually assume carriage control on all logical units. They usually interpret blank spaces on input as zeroes and often provide attachment of global filenames to logical units at runtime. There are several routines in the I/O library to provide these functions.

If a program reads and writes only units 5 and 6, then including the **-lI66** flag in the f77 command causes carriage control to be interpreted on output and cause blanks to be read as zeroes on input without further modification of the program. If this is not adequate, the routine IOINIT (3F) can be called to specify control parameters separately, including whether files should be positioned at their beginning or end upon opening.

Logical Unit Preattachment

The IOINIT routine can also be used to attach logical units to specific files at runtime. It looks in the environment for names of a user-specified form, and then it opens the corresponding logical unit for sequential formatted I/O. Names must be of the general form *PREFIXnn*, where the particular *PREFIX* is specified in the call to IOINIT, and *nn* is the logical unit to be opened. Unit numbers less than 10 must include the leading '0'. For details, see the man page IOINIT(3F), online or in this guide.

For example, to attach external files test.inp and test.out to units 1 and 2:

```
demo% setenv TST01 test.inp
demo% setenv TST02 test.out

demo% cat ini1.f
        CHARACTER  PRFX*8
        LOGICAL CCTL, BZRO, APND, VRBOSE
        DATA  CCTL,    BZRO,     APND,     PRFX,   VRBOSE
&           /.TRUE., .FALSE., .FALSE., 'TST', .FALSE. /
C
        CALL  IOINIT( CCTL, BZRO, APND, PRFX, VRBOSE )
        READ( 1, * ) I, B, N
        WRITE( *, * )  'I = ', I, ' B = ', B, ' N = ', N
        WRITE( 2, * ) I, B, N
        END
demo% f77 ini1.f -lI77
ini1.f:
 MAIN:
demo% a.out
  I =   12   B =     3.14159012   N =   6
demo% █
```

IOINIT should prove adequate for most programs as written. However, it is written in FORTRAN specifically so that it may serve as an example for similar user-supplied routines. A copy may be retrieved by issuing the command:

```
demo% ar x /usr/lib/libI77.a ioinit.f
```

7.3. FORTRAN I/O Statements

The I/O extensions added by FORTRAN 77 and Sun FORTRAN are explained below:

The OPEN Statement

The OPEN statement connects a file with a unit, or alters some property of the connection. It has the following format:

```
OPEN( KEYWORD1=value1, KEYWORD2=value2, ... )
```

where *KEYWORDn* is a valid keyword specifier, as listed below.

All of the properties and options are explained in full below, but a few simple examples are presented first, just to show the kinds of statements we are talking about.

For example, either of the following forms of the OPEN statement will open the file projectA/data.test and connect it to FORTRAN unit 8:

```
OPEN( UNIT=8, FILE="projectA/data.test" )
OPEN( 8, FILE="projectA/data.test" )
```

In the above example, the following properties are established by *default*: sequential access, formatted file, and (unwisely) no allowance for error during file I/O. You can also explicitly specify such properties:

```
      OPEN( UNIT=8, FILE="projectA/data.test",
&           ACCESS='SEQUENTIAL', FORM='FORMATTED' )
```

For an even simpler example, either of the following will open the scratch file `fort.8` and connect it to unit 8:

```
    OPEN( UNIT=8 )
    OPEN( 8 )
```

As with the previous example: sequential access, formatted file, and no allowance for error during file I/O. If the file `fort.8` does not exist before execution, it is created.

For an example of allowing for I/O errors:

```
    OPEN( UNIT=8, FILE="projectA/data.test", ERR=99 )
```

which branches to statement label `99` if an error occurs during the `OPEN`.

The `OPEN` statement determines the type of file named, whether the connection specified is legal for the file type (for instance, `DIRECT` access is illegal for tape and tty devices), and allocates buffers for the connection if the file is on tape or if the subparameter `FILEOPT='BUFFER=`*n*`'` is specified. The default buffer size for tape is 64K characters. Existing files are never truncated on opening.

Valid specifiers for `OPEN` are as follows:

`UNIT` A required nonnegative integer that specifies the FORTRAN unit number to connect to. If the unit is first in the parameter list, then "`UNIT=`" can be omitted.

`FILE` An optional character expression naming the file to open. An `OPEN` statement need not specify a filename. If not specified, a default filename is created.

If you open a unit that's already open without specifying a filename (or with the previous filename), FORTRAN thinks you are reopening the file to change parameters. The only parameters you are allowed to change are `BLANK` (`NULL` or `ZERO`) and `FORM` (`FORMATTED` or `PRINT`). To change any other parameters, you must close, then reopen the file.

If `STATUS='SCRATCH'` is specified, a temporary file with a name of the form `tmp.`*FAAAxnnnnn* is opened, and (by default) deleted when closed or during termination of program execution.

Any other `STATUS` specifier without an associated filename results in opening a file named '`fort.`***n***', where ***n*** is the specified logical unit number. See below for a general description of the `STATUS` parameter.

ACCESS An optional character expression. The options are APPEND, DIRECT, or SEQUENTIAL. If not specified, SEQUENTIAL is assumed.

If ACCESS='APPEND' is specified:

- SEQUENTIAL and FILEOPT='EOF' are assumed. This is for opening a file to append records to an existing sequential-access file. This is a Sun FORTRAN extension.

If ACCESS='DIRECT' is specified:

- RECL must also be given, since all I/O transfers are done in multiples of fixed-size records.
- Only directly accessible files are allowed; thus, tty, pipes, and magnetic tape are not allowed.
- If FORM is not specified, unformatted transfer is assumed.
- If FORM='UNFORMATTED', the size of each transfer depends upon the data transferred.

If ACCESS='SEQUENTIAL':

- RECL is prohibited since records are of varying size.
- No padding of records is done.
- Files don't have to be randomly accessible; thus tty, pipes, and tapes can be used.
- If FORM is not specified, formatted transfer is assumed.
- If FORM='FORMATTED', each record is terminated with a newline (\n) character. This means that each record actually has one extra character.
- If FORM='PRINT', the file acts like a FORM='FORMATTED' file, except for the interpretation of column-1 characters on output (0 = double space, 1 = form feed, and blank = single space).
- If FORM='UNFORMATTED', each record is preceded and terminated with an INTEGER*4 count, making each record 8 characters longer than normal. This convention is not shared with other SunOS programs, so is useful only for communicating between FORTRAN programs.

FORM An optional character expression. The options are 'FORMATTED', 'UNFORMATTED', or 'PRINT'.
If not specified, 'FORMATTED' is assumed.
Interacts with ACCESS.

RECL "RECL=*n*" specifies a record length of *n* characters.
Required if ACCESS='DIRECT' .
Prohibited if ACCESS='SEQUENTIAL'.

Each `WRITE` defines one record and each `READ` reads one record (unread characters are flushed).

`ERR` An optional clause, with an integer statement label to branch to if an error occurs during the `OPEN`.

`IOSTAT` An optional clause, with an integer variable that receives the error status from an `OPEN`.

Note: If you want to avoid aborting the program when an error occurs on an `OPEN`, then include an `ERR=`*label* o an `IOSTAT=`*name* .

`BLANK` An optional character expression that indicates how blanks are treated. For formatted input only; the options are `'ZERO'` (blanks treated as zeroes), and `'NULL'` (blanks ignored during numeric conversion). If not specified, `'NULL'` is assumed.

`STATUS` An optional character expression. Possible values are:

- □ `'OLD'` — the file already exists (nonexistence is an error). For example: `STATUS='OLD'`
- □ `'NEW'` — the file doesn't exist (existence is an error) **Note:** `'FILE=`*name*`'` is required.
- □ `'UNKNOWN'` — existence is unknown (the default).
- □ `'SCRATCH'` — In general, if you open a file with `STATUS='SCRATCH'`, then the file will be removed when it is closed.
  Note: The standard prohibits opening a named file as scratch, that is if the `OPEN` statement has a `FILE=`*name* option, then it cannot have a `STATUS='SCRATCH'` option. Sun FORTRAN allows opening named files as scratch, but such files will be removed when closed or at program termination unless there is an explicit `CLOSE` statement with the option `STATUS='KEEP'`.

`FILEOPT` An optional character expression. The options are:

- □ `'NOPAD'` — don't extend records with blanks if you read past the end-of-record (formatted input only). That is, a *short* record causes an abort with an error message, rather than just filling with trailing blanks and continuing.
- □ `'BUFFER=`*n*`'` — This suboption is for magnetic tape only. It sets the size of the I/O buffer to use. It is necessary only when writing, since the I/O system defaults to 64K-character buffers for tape, allowing reads to anything smaller than that. **WARNING:** It must be at least 8 characters greater than the largest record you write to avoid spanning tape blocks.
- □ `'EOF'` — opens a file at end-of-file rather than at the beginning (useful for appending data to the file).

For example: FILEOPT='EOF'
See ACCESS='APPEND'.

The CLOSE Statement

The CLOSE statement severs the connection between a unit and a file. The unit number must be given. The optional parameters are IOSTAT, ERR (see OPEN for meanings), and STATUS. The values for STATUS are 'KEEP' or 'DELETE', where KEEP is the default (except for scratch files) and DELETE means that the file will be removed after it is closed.

A simple example of a CLOSE statement is:

```
CLOSE( 3, ERR=17 )
```

Sequentially accessed, external files are truncated to the current file position on CLOSE only if the last access to the file was a WRITE.

The INQUIRE Statement

The INQUIRE statement returns information about a unit or a file. You can determine such things as whether it exists, is opened, is connected for sequential I/O. If you do use INQUIRE, you must inquire either by unit or by file (but not by both in the same INQUIRE statement).

An inquire by *unit* has the general form:

```
INQUIRE( UNIT=unit_number, parameter list )
```

An example of an inquire by *unit*:

```
LOGICAL  OK
INQUIRE( UNIT=3, OPENED=OK )
IF  ( OK )  CALL GETSTD  ( 3, STDS )
```

You can, of course, ask for more than one answer in an INQUIRE statement. For example:

```
CHARACTER FN*32
LOGICAL   HASNAME, OK
INQUIRE ( UNIT=3, OPENED=OK, NAMED=HASNAME, NAME=FN )
IF ( OK .AND. HASNAME ) PRINT *, "Filename = '", FN, "'"
```

An inquire by *file* has the general form:

```
INQUIRE( FILE=filename, parameter list )
```

An example of an inquire by *file*:

```
LOGICAL  THERE
INQUIRE( FILE='.profile', EXIST=THERE )
IF ( THERE )  CALL GETPROFILE( FC, PROFILE )
```

The options to INQUIRE are as follows:

FILE a character variable specifies which file the INQUIRE is about. Trailing blanks in the filename are ignored. Files have the properties of name, existence (or nonexistence), and the ability to be connected in certain ways (FORMATTED, UNFORMATTED, SEQUENTIAL, or DIRECT). The file need not be connected to a unit in the current program.

UNIT a positive integer variable that refers to files after they are opened. Exactly one of FILE or UNIT must be used.

IOSTAT as in the OPEN statement.

ERR as in the OPEN statement.

EXIST a logical variable that is set to .TRUE. if the file or unit exists and .FALSE. otherwise.

OPENED a logical variable that is set to .TRUE. if the file is connected to a unit or the unit is connected to a file, and .FALSE. otherwise.

NUMBER an integer variable that is assigned the number of the unit connected to the file, if any. If no file is connected, the variable is unchanged.

NAMED a logical variable that is assigned .TRUE. if the file has a name, .FALSE. otherwise.

NAME a character variable that is assigned the name of the file connected to the unit. If you do an inquire-by-unit, the name parameter is undefined unless both the OPENED and NAMED variable's values are .TRUE. If you do an inquire-by-file, the name parameter is returned, even though standard FORTRAN 77 leaves it undefined.

ACCESS a character variable that is assigned the value 'SEQUENTIAL' if the connection is for sequential I/O and 'DIRECT' if the connection is for direct I/O. The value is undefined if there is no connection.

SEQUENTIAL

a character variable that is assigned the value 'YES' if the file could be connected for sequential I/O, 'NO' if the file could not be connected for sequential I/O, and 'UNKNOWN' if the system can't tell.

DIRECT a character variable that is assigned the value 'YES' if the file could be connected for direct I/O, 'NO' if the file could not be connected for direct I/O, and 'UNKNOWN' if the system can't tell.

FORM a character variable which is assigned the value 'FORMATTED' if the file is connected for formatted I/O and 'UNFORMATTED' if the file is connected for unformatted I/O.

FORMATTED
a character variable that is assigned the value 'YES' if the file could be connected for formatted I/O, 'NO' if the file could not be connected for formatted I/O, and 'UNKNOWN' if the system can't tell.

UNFORMATTED
a character variable that is assigned the value 'YES' if the file could be connected for unformatted I/O, 'NO' if the file could not be connected for unformatted I/O, and 'UNKNOWN' if the system can't tell.

RECL an integer variable that is assigned the record length of the records in the file if the file is connected for direct access.

NEXTREC an integer variable that is assigned one more than the number of the the last record read from a file connected for direct access.

BLANK a character variable that is assigned the value 'NULL' if null blank control is in effect for the file connected for formatted I/O and 'ZERO' if blanks are being converted to zeros and the file is connected for formatted I/O.

Using defaults

Here is an example, in which declarations are omitted:

```
    OPEN( 1, FILE='/dev/console' )
```

On a Sun system this statement opens the console for formatted sequential I/O. An INQUIRE for either unit 1 or file /dev/console would reveal that the file:

□ exists

□ is connected to unit 1

□ has the name /dev/console

□ is opened for sequential I/O

□ could be connected for sequential I/O

□ can't be connected for direct I/O (can't seek)

□ is connected for formatted I/O

□ can be connected for formatted I/O

□ can't be connected for unformatted I/O (can't seek)

□ has neither a record length nor a next record number

□ is ignoring blanks in numeric fields.

Permissions

In the Sun system environment, the only way to discover what permissions you have for a file is to use the ACCESS (3F) function. The INQUIRE statement does not determine permissions.

Table 7-2 *Summary of INQUIRE Options*

Form: *SPECIFIER=variable*		
SPECIFIER	*Value of the variable for inquire*	*Data type for the variable*
ACCESS	'DIRECT' 'SEQUENTIAL'	CHARACTER
BLANK	'NULL' 'ZERO'	CHARACTER
DIRECT *	'YES' 'NO' 'UNKNOWN'	CHARACTER
ERR	statement number	INTEGER
EXIST	.TRUE. .FALSE.	LOGICAL
FORM	'FORMATTED' 'UNFORMATTED'	CHARACTER
FORMATTED *	'YES' 'NO' 'UNKNOWN'	CHARACTER
IOSTAT	error number	INTEGER
NAME †	name of the file	CHARACTER
NAMED †	.TRUE. .FALSE.	LOGICAL
NEXTREC	next record number	INTEGER
NUMBER *	unit number	INTEGER
OPENED	.TRUE. .FALSE.	LOGICAL
RECL	record length	INTEGER
SEQUENTIAL *	'YES' 'NO' 'UNKNOWN'	CHARACTER
UNFORMATTED *	'YES' 'NO' 'UNKNOWN'	CHARACTER

Notes for the above table:

* Returned value is undefined for inquire-by-unit in standard FORTRAN 77 but is defined in Sun FORTRAN.

† Returned value is undefined for inquire-by-file in standard FORTRAN 77 but is defined in Sun FORTRAN.

General

- If a file is scratch, then NAMED and NUMBER are not returned.
- If there is no file with the specified name, then these are not returned: DIRECT, FORMATTED, NAME, NAMED, SEQUENTIAL, and UNFORMATTED.
- If OPENED=.FALSE., then these are not returned: ACCESS, BLANK, FORM, NEXTREC, and RECL.
- If no file is connected to the specified unit, then these are not returned: ACCESS, BLANK, DIRECT, FORM, FORMATTED, NAME, NAMED, NEXTREC, NUMBER, RECL, SEQUENTIAL, and UNFORMATTED.
- If ACCESS='SEQUENTIAL', then these are not returned: RECL and NEXTREC.
- If FORM='UNFORMATTED', then BLANK is not returned.

The BACKSPACE Statement

The BACKSPACE statement does one of two things, depending on what the device is, and whether or not the end-of-file has been reached. If it has, then it backs up over the endfile record — on a disk file this does nothing; but on a tape it corresponds to backing up over the tape mark, and positioning the tape after the last data record of the file, but before the endfile record. Otherwise, it backs up over the last data record read or written (i.e., the last FORTRAN logical record, which may involve reading one or more physical records). For FORMATTED records, it will search backwards looking for the record separator (\n or ^J); for UNFORMATTED records, it uses the character-count trailer that is part of the record.

Sequentially accessed, external files are truncated to the current file position on BACKSPACE only if the last access to the file was a WRITE.

The BACKSPACE options are:

UNIT A required nonnegative integer that specifies the FORTRAN unit number to connect to. If the unit number is first in the parameter list, then "UNIT=" can be omitted.

ERR An optional clause, with an integer statement reference (for example, ERR=9) to branch to if an error occurs during the BACKSPACE.

IOSTAT An optional clause, with an integer variable that receives the error status value from a BACKSPACE. For example, IOSTAT=STATUS, where STATUS is a user variable of type INTEGER.

The REWIND Statement

The REWIND statement positions you at the beginning of the current file on the specified unit. When writing a sequential file (such as one on tape), it does an implicit ENDFILE action first. If you are reading the endfile record, REWIND backspaces over that and all the data records preceding.

The REWIND does not necessarily rewind a tape to its beginning. If you are reading the second file on a tape, then it rewinds to the beginning of the second file. To fully rewind a tape, use the mt(1) utility program, which can be invoked from a FORTRAN program by calling the SYSTEM(3F) routine.

The options related to REWIND are:

UNIT — A required nonnegative integer that specifies the FORTRAN unit number to connect to. If the unit number is first in the parameter list, then "UNIT=" can be omitted.

ERR — An optional clause, with an integer statement reference (for example, ERR=9) to branch to if an error occurs during the REWIND.

IOSTAT — An optional clause, with an integer variable that receives the error status from a REWIND. For example, IOSTAT=STATUS, where STATUS is a user variable of type INTEGER.

The ENDFILE Statement

When writing to a SunOS disk file, ENDFILE truncates the file at the current position. This is because in disk files, the endfile record is represented by the end of the file.

Two endfile records signify the end-of-tape mark. When writing to a tape file, ENDFILE writes two endfile records, then the tape backspaces over the second one. If the file is closed at this point, both end-of-file and end-of-tape are marked. If more records are written at this point (either by continued write statements or by another program if you are using no-rewind magnetic tape), the first tape mark stands (endfile record), and is followed by another data file, then by more tape marks, and so on.

The options related to ENDFILE are:

UNIT — A required nonnegative integer that specifies the FORTRAN unit number to connect to. If the unit number is first in the parameter list, then "UNIT=" can be omitted.

ERR — An optional clause, with an integer statement reference (for example, ERR=1000) to branch to if an error occurs during the ENDFILE operation.

IOSTAT — An optional clause, with an integer variable that receives the error status from ENDFILE. For example, IOSTAT=STATUS, where STATUS is a user variable of type INTEGER.

Direct I/O

Random access to files is also called direct access. A direct-access file contains a number of records that are written to or read from by referring to the record number. This record number is specified when the record is written. In a direct-access file, records must be all the same length and all the same type.

A logical record in a direct access, external file is a string of bytes of a length specified when the file is opened. Read and write statements must not specify logical records longer than the original record size definition. Shorter logical records are allowed. Unformatted, direct writes leave the unfilled part of the record undefined. Formatted, direct writes cause the unfilled record to be padded with blanks.

In using direct unformatted I/O, you should be careful with the number of values your program expects to read. Each READ operation acts on exactly *one* record; the number of values that the input list requires must be *less than or equal to* the number of values in that record.

Direct access READ and WRITE statements have an extra argument, REC=*n*, which gives the record number to be read or written. For example, with direct-access, *unformatted*:

```
      OPEN( 2, FILE='data.db', ACCESS='DIRECT', RECL=20,
     &           FORM='UNFORMATTED, ERR=90 )
      READ( 2, REC=13, ERR=30 ) X, Y
```

This opens a file for direct-access, unformatted I/O, with a record length of 20 characters, then reads the thirteenth record as is.

And with direct-access, *formatted*:

```
      OPEN( 2, FILE='inven.db', ACCESS='DIRECT', RECL=20,
     &           FORM='FORMATTED, ERR=90 )
      READ( 2, FMT="(I10,F10.0)", REC=13, ERR=30 ) A, B
```

This opens a file for direct-access, formatted I/O, with a record length of 20 characters, then reads the thirteenth record and converts it according to the format "(I10,F10.3)".

Internal Files

Internal files are character-string objects such as variables or substrings, or arrays of type character. In the former case, there is only a single record in the file but in the latter case, each array element is a record. The ANSI standard includes only sequential formatted I/O on internal files. (I/O is not a precise term to use here, but internal files are dealt with using READ and WRITE statements.) Internal files are used by giving the name of the character object in place of the unit number. For example:

```
     CHARACTER X*80
     READ( 5, '(A)' ) X
     READ( X, '(I3,I4)' ) N1, N2
```

reads a card image into X and then reads two integers from the front of it. A sequential READ or WRITE always starts at the beginning of an internal file.

Sun FORTRAN extends direct I/O to internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed. In this case, a record is a single element of an array of character strings. For example:

```
demo% cat int2.f
        CHARACTER  CARD(4)*16
*                        12341234
        DATA  CARD(1) / " 81  81 " /
        DATA  CARD(2) / " 82  82 " /
        DATA  CARD(3) / " 83  83 " /
        DATA  CARD(4) / " 84  84 " /
        READ ( CARD, FMT=20, REC=3 )  M, N
 20     FORMAT( I4, I4 )
        PRINT *, M, N
        STOP
        END
demo% f77 int2.f
int2.f:
 MAIN:
demo% a.out
  83  83
demo%
```

This does a direct-access read of the third record of the internal file CARD.

Formatted I/O

Formatted WRITE converts the record according to the instructions in the associated format from internal form to a form suitable for the external media involved. For example:

```
   WRITE( 6, 10 )  A, B
10 FORMAT( F8.3,  F6.2 )
```

Formatted records are terminated with *newline* characters. Formatted, sequential access causes one or more logical records to be read or written. For formatted write statements, logical record length is determined by the format statement interacting with the list of input or output variables (I/O list) at execution time.

Unformatted I/O

Unformatted I/O is used to transfer binary information to or from memory locations without changing its internal representation. Each execution of an *unformatted* I/O statement causes a single logical record to be read or written. Since internal representation varies with different machines, unformatted I/O is limited in its portability.

Unformatted I/O can be used to write data out temporarily, or to write data out quickly for subsequent input to another program on the same machine.

Logical record length for unformatted, sequential files is determined by the number of bytes required by the items in the I/O list. The requirements of this form of I/O cause the external physical record size to be somewhat larger than the logical record size. For example:

```
   WRITE( 8 )  A, B
```

The FORTRAN runtime system embeds the record boundaries in the data by inserting an `INTEGER*4` byte count at the beginning and end of each unformatted sequential record during an unformatted sequential `WRITE`. The trailing byte count enables `BACKSPACE` to operate on records. The result is that FORTRAN programs can use an unformatted sequential `READ` only on data that was written by an unformatted sequential `WRITE` operation. Any attempt to read such a record as formatted would have unpredictable results. If the first byte of the record were a "4", an unformatted sequential `READ` would interpret it as the first byte of a count, and conclude the record had at least 0x4000000 bytes!

NOTE *Avoid using the unformatted sequential* `READ` *unless your file was written that way. If you want to use unformatted I/O, try using the unformatted direct* `READ` *whenever possible. Open the file with* `RECL=1` *if your input lists are not all the same length.*

List-Directed I/O

List-directed I/O is a kind of *free-form* I/O for sequential access devices. It is invoked by using an asterisk as the format identifier, as in:

```
READ( 6, * ) A, B, C
```

Input Format

On input, values are separated by strings of blanks and (possibly) a comma. Values, except for character strings, cannot contain blanks. Character strings can be quoted strings, using pairs of quotes ("), or pairs of apostrophes ('), or unquoted strings (see below), but NOT hollerith (*n*`Hxyz`) strings. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means that the corresponding variable in the I/O list is not changed. Input data items can be preceded by repetition counts, as in

```
4*(3.,2.)  2*, 4*'hello'
```

which stands for 4 complex constants, 2 null input fields, and 4 string constants.

A slash (/) in the input list will terminate assignment of values to the input list during *list-directed* input and the remainder of the current input line is skipped. Text following the slash is ignored and may be used to comment the data file.

Output format

List-directed output provides a quick-and-easy way to print output without fussing with format details. A suitable simple format is automatically chosen for each item, and where a conflict exists between complete accuracy and simple output form, the simple form is chosen. For example, such a conflict occurs with numbers like 1.4, which has no exact binary representation:

```
demo% cat lis5.f
        READ ( 5, * ) X
        WRITE( 6, * ) X, " beauty"
        WRITE( 6, 1 ) X
 1      FORMAT( 1X, F13.8, "   truth" )
        STOP
        END
demo% f77 lis5.f
lis5.f:
 MAIN:
demo% a.out
1.4
    1.40000000   beauty
    1.39999998   truth
demo% █
```

If you need accuracy, specify the format.

In formatting list-directed output, the I/O system tries to prevent output lines longer than 80 characters. List-directed output of `COMPLEX` values includes an appropriate comma. List-directed output distinguishes between `REAL` and `DOUBLE PRECISION` values and formats them differently. A '\n' in a character string is output as a carriage return.

The values of character strings are printed as is; they are not enclosed in quotes, so only certain forms of strings can be read back using list-directed input; these forms are described in the next section.

Unquoted strings

Sun FORTRAN extends list-directed I/O to allow reading of a string not enclosed in quotes. The string must not start with a digit, and cannot contain separators (commas or slashes (/)) or whitespace (spaces or tabs). A newline terminates the string unless escaped with a backslash (\). Any string not meeting the above restrictions must be enclosed in single or double quotes.

Internal I/O

Sun FORTRAN extends list-directed I/O to allow internal I/O. During internal, list-directed reads, characters are consumed until the input list is satisfied or the end-of-file is reached. During internal, list-directed writes, records are filled until the output list is satisfied. The length of an internal array element should be at least 20 characters to avoid logical record overflow when writing double-precision values. Internal, list-directed read was implemented to make command line decoding easier. Internal, list-directed output should be avoided.

Namelist I/O

Namelist I/O lets you do format-free input or output of whole groups of variables, or input of selected items in a group of variables. The NAMELIST statement defines a group of variables or arrays: it specifies a *group-name*, and it lists the variables and arrays of that group.

The syntax of the NAMELIST statement is:

NAMELIST /*group-name*/*namelist* [[,] /*group-name*/*namelist*] . . .

where *group-name* is an identifier, and *namelist* is a list of variables or arrays, separated by commas. For example:

```
        CHARACTER*16   SAMPLE
        LOGICAL*4      NEW
        REAL*4         DELTA
        NAMELIST /CASE/ SAMPLE, NEW, DELTA
```

Rules and restrictions for namelist

- □ The group name can appear in only the NAMELIST, READ, or WRITE statements, and must be unique for the program.
- □ The list cannot include any dummy arguments, array elements, structures, substrings, records, record fields, pointers or pointer-based variables.* However, the *input data* can include array elements or substrings.
- □ A variable or array can be listed in more than one namelist group.

Namelist output

Namelist output uses a special form of the WRITE statement. This makes a report showing the group name, and for each variable of the group, it shows the name and current value in memory. It formats each value according to the type of each variable, and it writes the report so that namelist READ can read it.

The syntax of namelist WRITE is:

WRITE (*extu*, *namelist-specifier* [, *iostat*] [, *err*])

where *namelist-specifier* has the form:

[NML=]*group-name*

and *group-name* has been previously defined in a NAMELIST statement.

The namelist WRITE statement writes values of all variables in the group, in the same order as in the NAMELIST statement.

The following is an example of namelist output:

* For more on pointer-based variables, see Section 4.7 — "Pointers."

```
demo% cat naml.f
* naml.f Namelist output
        CHARACTER*8    SAMPLE
        LOGICAL*4      NEW
        REAL*4         DELTA
        NAMELIST /CASE/ SAMPLE, NEW, DELTA
        DATA SAMPLE /"Demo"/, NEW /.TRUE./, DELTA /0.1/
        WRITE ( *, CASE )
        STOP
        END
demo% f77 naml.f
f77 naml.f
naml.f:
 MAIN:
demo% a.out
 &case  sample=  Demo    , new=  T, delta=   0.100000
 &end
demo%
```

↑
column 2

Note that if you do omit the keyword `NML` then you must also omit the keyword `UNIT` and the unit parameter must be first, the namelist-specifier must be second, and there must *not* be a format specifier.

Alternate — The `WRITE` can have the following form:

```
    WRITE ( UNIT=6, NML=CASE )
```

Namelist input

The namelist READ statement reads the next external record, skipping over column one, and looking for the symbol "$" in column two or beyond, followed by the group name specified in the READ statement. The records are input and values assigned by matching names in the data with names in the group, using the data types of the variables in the group. Variables in the group that are not found in the input data are unaltered.

The syntax of namelist READ is:

READ (*extu*, *namelist-specifier* [, *iostat*] [, *err*] [, *end*])

The *namelist-specifier* has the form:

[NML=]*group-name*

where *group-name* has been previously defined in a NAMELIST statement.

For example:

```
      CHARACTER*16   SAMPLE
      LOGICAL*4      NEW
      REAL*4         DELTA
      NAMELIST /CASE/ SAMPLE, NEW, DELTA
      READ ( 1, CASE )
```

In this example, the group CASE consists of the three variables SAMPLE, NEW, and DELTA. If you do omit the keyword NML, then you must also omit the keyword UNIT, and the unit parameter must be first, the namelist-specifier must be second, and there must *not* be a format specifier.

Alternate — The READ can have the following form:

```
   READ ( UNIT=1, NML=CASE )
```

Namelist data

The first record of namelist input data has the special symbol "$" (dollar sign) in column two or beyond, followed by the namelist group name. This is followed by a series of assignment statements, starting in or after column two, on the same or subsequent records, each assigning a value to a variable (or one or more values to array elements) of the specified group. The input data is terminated with another "$", in or after column two, as in the pattern:

$*group-name variable=value* [, *variable=value*, . . .] $[END]

You can alternatively use an ampersand (&) in place of each dollar sign, but the beginning and ending delimiters must match. The END is an optional part of the last delimiter.

The input data assignment statements must be in one of the following forms:

variable=value

If an array is subscripted, it must be subscripted with the appropriate number of subscripts: 1, 2, 3, ...

array=value1[, value2,] ...

array(subscript)=value1[, value2,] ...

array(subscript,subscript)=value1[, value2,] ...

Use " or ' to delimit character constants. For more on character constants, see "Syntax rules for namelist data," below.

variable=character constant

variable(index:index)=character constant

The following is sample data to be read by the program segment above:

```
 $case    delta=0.05,   sample='Demo'     $
```

↑
column 2

or the data could be on several records:

```
 $case
    delta=0.05
     sample='Demo'
   $
```

↑
column 2

Here `NEW` was not input, and the order is not the same as in the `NAMELIST` statement.

Syntax rules for namelist data

The following syntax rules apply for input data to be read by namelist:

- The variables of the named group can be in any order, and any can be omitted.
- The data must start in or after column two. Column one is totally ignored.
- There must be at least one comma, space, or tab between variables, and one or more spaces or tabs are the same as a single space.* Consecutive commas are not permitted before a variable name. Spaces before or after a comma have no effect.
- No spaces or tabs are allowed inside a *group* name or a *variable* name, except around the punctuation of a subscript or substring. No name can be split over two records.
- The end of a record acts like a space character.

  Exception: In a character constant, it is ignored, and the character constant is continued with the next record. The last character of the current record is immediately followed by the second character of the next record. The first character of each record is ignored.

- The equal sign of the assignment statement can have zero or more blanks or tabs on each side of it.

- Only *constant* values can be used for subscripts, range indicators of substrings, and the values assigned to variables or arrays. You can *not* use a symbolic constant (`PARAMETER`).

  - Hollerith, octal, and hexadecimal constants are not permitted.

  - Each constant assigned has the same form as the corresponding FORTRAN constant.

  - There must be at least one comma, space, or tab between constants, and zero or more spaces or tabs are the same as a single space.* (You can enter `1,2,3` or `1 2 3` or `1, 2, 3` etc.)

  - A character constant is delimited by apostrophes (') or quotes ("), but if you start with one of those, you must finish that character constant with the same one. If you use the apostrophe as the delimiter, then to get an apostrophe in a string, use two consecutive apostrophes. For example:

```
sample='use "$" in 2'   {goes in as: use "$" in 2 }
sample='don''t'         {goes in as: don't }
sample="don''t"         {goes in as: don''t }
sample="don't"          {goes in as: don't }
```

  - A complex constant is a pair of real or integer constants separated by a comma and enclosed in parentheses. Spaces can occur only around the punctuation.

  - A logical constant is any form of true or false value, such as `.TRUE.` or `.FALSE.`, or any value beginning with `.T`, `.F`, etc.

- A null data item is denoted by two consecutive commas, and it means the corresponding array element or complex variable value is not to be changed. Null data item can be used with array elements or complex variables only. One null data item represents an entire complex constant, although you can't use it for either part of a complex constant. For example, the program:

```
* nam2.f  Namelist "r*c" and  consecutive commas
    REAL  ARRAY(4,4)
    NAMELIST  /GRID/ ARRAY
    WRITE ( *, * ) "Input?"
    READ ( *, GRID )
    WRITE ( *, GRID )
    STOP
    END
```

* Inside a character constant, consecutive spaces or tabs are preserved, not compressed.

with the data:

```
 $GRID ARRAY = 9,9,9,9,,,,,8,8,8,8 $
 ↑                    ↑
```

column 2 5 consecutive commas

loads 9's into row 1, skips 4 elements, and loads 8's into row 3 of `ARRAY`.

Arrays only — The following can be used only with an array:

□ The form *r*c* stores *r* copies of the constant *c* into an array, where *r* is a nonzero, unsigned integer constant, and *c* is any constant. For example, the program:

```
* nam3.f  Namelist "r*c" and  "r*  "
    REAL  PSI(10)
    NAMELIST  /GRID/ PSI
    WRITE ( *, * ) "Input?"
    READ ( *, GRID )
    WRITE ( *, GRID )
    STOP
    END
```

with the input:

```
 $GRID  PSI = 5*980    $
 ↑
```

column 2

loads 980.0 into the first 5 elements of the array `PSI`.

□ The form *r** skips *r* elements of an array (that is, does *not* change them) where *r* is an unsigned integer constant. For example, the same program of the above example, with this input:

```
 $GRID  PSI = 3*     5*980    $
 ↑
```

column 2

skips the first 3 elements and loads 980.0 into elements 4,5,6,7,8 of `PSI`.

Requesting names

If your program is doing namelist input from the terminal, you can request the group name and namelist names that it will accept. To do this, enter a question mark (?) in column two, and press RETURN. The group name and variable names for that group will be displayed, and then it will wait again for input.

For example:

```
demo% cat nam4.f
* nam4.f Namelist prompting for input
        CHARACTER*16   SAMPLE
        LOGICAL*4      NEW
        REAL*4         DELTA
        NAMELIST /CASE/ SAMPLE, NEW, DELTA
        WRITE ( *, * ) "Input?"
        READ ( *, CASE )
        STOP
        END
demo% f77 nam4.f
nam4.f:
 MAIN:
demo% a.out
  Input?
 ?
$case
 sample
 new
 delta
$end
 $case   sample="Test 2", delta=0.03 $
demo%
 ↑
```

column 2

7.4. Accessing Files from FORTRAN Programs

Data is transferred to or from devices or files by specifying a logical unit number in an I/O statement.

Logical unit numbers can be nonnegative integers or the character '*'. The '*' stands for the *standard input* if it appears in a read statement, or the *standard output* if it appears in a write or print statement. Standard input and standard output are explained in the section on preconnected units found later in this chapter.

Accessing Named Files

Before a program can access a file with a `READ`, `WRITE`, or `PRINT` statement, the file needs to be created and a connection established for communication between the program and the file. The file can already exist or be created at the time the program executes. The FORTRAN `OPEN` statement establishes a connection between the program and file to be accessed. `OPEN` can take a filename parameter (`FILE=`*filename*) to specify the file. Filenames can be:

- quoted character constants , such as:
  `FILE='myfile.out'`
- character variables, such as:
  `FILE=FILNAM`

- character expressions, such as:

```
     FILE=PREFIX(:LNBLNK(PREFIX)) // '/' //
  &     NAME(:LNBLNK (NAME)),...
```

Some ways a program can get filenames are:

- by reading from a file or terminal keyboard, such as:

```
      READ( 4, 401) FILNAM
```

- from the command line, such as:

```
      CALL GETARG( ARGNUMBER, FILNAM)
```

- from the environment, such as:

```
      CALL GETENV( STRING, FILNAM )
```

The example below shows one way to construct a filename:

```
      CHARACTER*1024 FUNCTION FULLNAME ( NAME )
      CHARACTER*(*)  NAME
      CHARACTER*1024 PREFIX
C
C In  path names starting with '~/':
C  replace the tilde with the home directory name;
C  prefix relative pathname by path to current directory;
C  leave absolute path names unchanged.
C
      IF ( NAME(1:1) .EQ. '/') THEN
          FULLNAME = NAME
      ELSE IF ( NAME(1:2) .EQ. '~/') THEN
          CALL GETENV( 'HOME', PREFIX )
          FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
     &                    NAME(2:LNBLNK(NAME))
      ELSE
          CALL GETCWD( PREFIX )
          FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
     &                    '/' // NAME(:LNBLNK(NAME))
      ENDIF
      END
```

Accessing Unnamed Files

When a program opens a FORTRAN file without a name, the run time system supplies a filename. There are several ways it can do this.

Opened as scratch

If you specify STATUS='SCRATCH' in the OPEN statement, then the system opens a file with a name of the form tmp.F*AAAxnnnnn*, where *nnnnn* is replaced by the current process ID, *AAA* is a string of three characters, and *x* is a letter; the *AAA* and *x* make the filename unique. This file is deleted upon termination of the program or execution of a CLOSE statement, unless STATUS='KEEP' is specified in the CLOSE statement.

Already open

If a FORTRAN program has a file already open, an OPEN statement that specifies only the file's logical unit number and the parameters to change can be used to change some of the file's parameters (specifically, BLANK and FORM). The system determines that it should not really OPEN a new file, but just change the parameter values. Thus, this looks like a case where the runtime system would make up a name, but it does not.

Other

In all other cases, the runtime system OPENs a file with a name of the form `fort.n`, where *n* is the logical unit number given in the OPEN statement.

Passing filenames to programs

The SunOS file system does not have any notion of temporary filename binding (or file equating) as some other systems do. Filename binding is the facility that is often used to associate a FORTRAN logical unit number with a physical file without changing the program. This mechanism evolved to communicate filenames more easily to the running program, because in FORTRAN 66 you could not open files by name. With SunOS or with UNIX there are several satisfactory ways to communicate filenames to a FORTRAN program including command-line arguments and environment-variable values. For example, see the file `ioinit.f` in `libI77`, which is discussed in Section 7.2, under Subsection "Logical Unit Preattachment." The program can then use those logical names to open the files. The next section describes two additional ways to change a program's input and output files without changing the program, called *redirection* and *piping*.

Preconnected units

When a FORTRAN program begins execution under SunOS, there are usually three units already open. These are called *preconnected units*. Their names are *standard input, standard output*, and *standard error*. In FORTRAN programs:

- standard input is logical unit 5
- standard output is logical unit 6
- standard error is logical unit 0

All three are connected to your workstation or window, unless file redirection or piping is done at the command level.

Other units

All other units are preconnected to files named `fort.`*n* where *n* is the corresponding unit number. These files need not exist, and are created only if the units are actually used, and if the first action to the unit is a WRITE or PRINT. That is, only if an OPEN statement does not override the preconnected name before any WRITE or PRINT is issued for that unit. For example, the program:

```
WRITE( 15 ) 2
END
```

writes a single unformatted record on the `fort.15` file.

7.5. Formats

Sun FORTRAN and FORTRAN 77 provide additional specifiers and enhancements to the FORTRAN 66 format specifications I, F, E, G, D, H, X, A, and L. For all specifiers, upper-case as well as lower-case characters are recognized in format statements and in all the alphabetic arguments to the I/O library routines.

If the external representation of a datum is too large for the field width specified, the specified field is filled with asterisks (*).

Alpha editing (A)

The A specifier is used with CHARACTER type data elements. In FORTRAN 66, this specifier could be used with any variable type. Sun FORTRAN supports the older usage, up to four characters to a word.

Blank control (B, BN, BZ)

For normal input, without any specific blank specifiers in the format, non-leading blanks in numeric input fields are interpreted as zeros or ignored, depending on the value of the BLANK= specifier currently in effect for the unit.

In FORTRAN 77, the following blank specifiers were added:

- BN — If BN precedes a specification, a non-leading blank in the input data is considered *null*, and is ignored.
- BZ — If BZ precedes a specification, a non-leading blank in the input data is considered *zero*.
- B — Sun FORTRAN has added the blank-control specifier B. It returns interpretation to the default mode of blank interpretation, consistent with S, which returns to default sign control.

For example, the following program reads the same data twice, once with BZ and once with BN, and prints the results both times:

```
host% cat bz1.f
        CHARACTER  CARD*16
*                     12341234
        DATA  CARD / " 82  82 " /

        READ ( CARD, 20 )  M, N
 20     FORMAT( I4, BZ, I4 )
        PRINT *, M, N

        READ ( CARD, 30 )  M, N
 30     FORMAT( I4, BN, I4 )
        PRINT *, M, N
        STOP
        END
host% f77 bz.f
bz.f:
 MAIN:
host% a.out
  82  820
  82  82
host% █
```

Rules and Restrictions for Blank Control

- Blank control specifiers apply to *input* only.
- A blank control specifier remains in effect until another blank control specifier is encountered, or format interpretation is complete.
- The B, BN, and BZ specifiers affect only I, F, E, D, and G editing.

Carriagecontrol ($)

The special edit descriptor $ suppresses the carriage return.* The action does *not* depend on the first character of the format. It is used typically for console prompts. For instance, you can use this to make a typed response follow the output prompt on the same line. This edit descriptor is constrained by the same rules as the colon (:).

As an example, the statements:

```
* dol1.f The $ edit descriptor with space
     WRITE ( *, 2 )
   2 FORMAT (' Enter the node number: ', $ )
     READ ( *, * ) NODENUM
     STOP
     END
```

produce a displayed prompt and user input response such as:

```
Enter the node number: 82
```

The first character of the format is printed out, in this case, a blank. For an *input* statement, the $ descriptor is ignored.

Commas in Formatted Input

If you are entering data that is controlled by a fixed-column format, then you can use commas to override the exacting column restrictions. Thus, the format:

```
(I10, F20.10, I4)
```

reads the record:

```
-345,.05e-3,12
```

correctly.

The I/O system is just being more lenient than described in the standard. In general, when doing a *formatted* read of *noncharacter* variables, commas override field lengths. More precisely: for I*w*, F*w*.*d*, E*w*.*d*[E*e*], and G*w*.*d* input fields, the field ends when *w* characters have been scanned or a comma has been

* If FORTRAN carriage control is enabled, then for formatted output, the first character of a record controls carriage movement.

scanned, whichever occurs first. If the latter, the field consists of the characters up to but not including the comma; the next field begins with the character following the comma.

Hollerith (*n*H)

FORTRAN 77 does not have the old Hollerith (*n* H) notation, although the ANSI standard recommends implementing the Hollerith feature in order to improve compatibility with old programs. Though this is not recommended, in Sun FORTRAN you can use Hollerith constants wherever a character constant can be used in FORMAT statements, assignment statements, and DATA starements. But such constants cannot be used as input data elements in list-directed or namelist input. For example, the two formats below are equivalent:

```
10 FORMAT( 8H Code = , A6 )
20 FORMAT(  " Code = ", A6 )
```

Sun FORTRAN also allows consecutive Hollerith constants without a separating comma. For example:

```
10 FORMAT( 5H flex   4Hible )
```

For compatibility with older programs, Sun FORTRAN also allows READ s into Hollerith edit descriptors, for example:

```
demo% cat holl.f
        PRINT 1
 1      FORMAT( 6Holder )
        READ 1
        PRINT 1
        STOP
        END
demo% f77 holl.f
holl.f:
 MAIN
demo% a.out
older
newer
newer
demo% █
```

Note that there is no list in this READ statement.

Octal and hexadecimal (O,Z)

The O and Z field descriptors for a FORMAT statement are for octal and hexadecimal notation, respectively. They can be used with any data type, taking the form:

O*w*[.*m*]
Z*w*[.*m*]

where *w* is the number of characters in the external field, and for output, *m*, if specified, determines the total number of digits in the external field (that is, if there are fewer than *m* nonzero digits, the field is zero-filled on the left to a total of *m* digits). The *m* has no effect on input.

Octal and hex input

A READ, with the O or Z field descriptors in the FORMAT, reads in *w* characters as octal or hexadecimal, respectively, and assigns the value to the corresponding member of the I/O list.

For example, if the external data field is:

```
654321
↑
```

column 1

then the input instructions:

```
   READ ( *, 2 ) M
2  FORMAT ( O6 )
```

will result in the octal value 654321 being loaded into the variable M. Further examples are included in the table below.

Table 7-3 *Sample Octal/Hex Input Values*

Format	*External Field*	*Internal (Octal or Hex) Value*
O4	1234^	1234
O4	16234	1623
O3	97^^^	Error: "9" not allowed
Z5	A23DE^	A23DE
Z5	A23DEF	A23DE
Z4	95.AF2	Error: "." not allowed

NOTE *The caret (^) indicates blanks.*

General rules for octal and hex input

- For octal values, the external field can contain only numerals 0 through 7.
- For hexadecimal values, the external field can contain only numerals 0 through 9 and the letters A through F or a through f.
- Signs, decimal points, and exponent fields are not allowed.
- All-blank fields are treated as having a value of zero.

- If an of item data is too big for the corresponding variable, an error message is displayed.

Octal and hex output

A `WRITE`, with the `O` or `Z` field descriptors in the `FORMAT`, writes out values as octal or hexadecimal integers, respectively. It writes to a field that is *w* characters wide, right-justified.

For example, the output instructions:

```
        M = 161
        WRITE ( *, 8 ) M
 8      FORMAT ( Z3 )
```

display as (161 decimal = A1 hex):

```
 A1
 ↑
```

column 2

Further examples are included in the table below.

Table 7-4 *Sample Octal/Hex Output Values*

Format	*Internal (Decimal) Value*	*External (Octal/Hex) Representation*
O6	32767	^77777
O2	14251	**
O4.3	27	^033
O4.4	27	0033
O6	-32767	100001
Z4	32767	7FFF
Z3.3	2708	A94
Z6.4	2708	^^0A94
Z5	-32767	^8001

NOTE *The caret (^) indicates blanks.*

General rules on octal and hex output

- Negative values are written as if unsigned; no negative sign is printed.
- The external field is filled with leading spaces, as needed, up to the width *w*.
- If the field is too narrow, it is filled with asterisks.

- If *m* is specified, the field is left-filled with leading zeros, to a width of *m*.

Radix control (R)

Radixes other than 10 can be specified for formatted integer I/O conversion. The specifier is patterned after P, the scale factor for floating-point conversion. It remains in effect until another radix is specified or format interpretation is complete. The specifier is R or *n*R, where $2 \leq n \leq 36$. If *n* is omitted, the default decimal radix is restored. The I/O item is treated as a 32-bit integer.

For an example, see "Sign Control" later in this section.

Remaining characters (Q)

You can get the length of an input record, or of the remaining portion of it that is unread, by the special edit descriptor Q in the FORMAT statement. This edit descriptor obtains the number of characters that remain to be read from the current record. For example:

```
%demo cat qed1.f
* qed1.f  Q edit descriptor (real & string)
        CHARACTER  CVECT(80)*1
        OPEN ( UNIT=4, FILE='qed1.dat' )
        READ ( 4, 1 ) R, L, ( CVECT(I), I=1,L )
 1      FORMAT ( F4.2, Q, 80 A1 )
        WRITE ( *, 2 ) R, L, '"', (CVECT(I),I=1,L), '"'
 2      FORMAT ( 1X, F7.2, 1X, I2, 1X, 80A1 )
        STOP
        END
demo% cat qed1.dat
8.10qwerty
demo% f77 qed1.f -o qed1
qed1.f:
 MAIN:
%demo qed1
   8.10  6 "qwerty"
%demo
```

The above example reads a field into the variable R, then reads the number of characters remaining after that field into L, then reads L characters into CVECT. Note that Q's position in the FORMAT statement corresponds to that of L in the READ statement. The next example puts the Q descriptor first:

```
demo% cat qed2.f
* qed2.f  The Q edit descriptor (string)
    CHARACTER  CVECT(80)*1
    OPEN ( UNIT=4, FILE='qed2.dat' )
    READ ( 4, 1 ) L, ( CVECT(I), I=1,L )
 1  FORMAT ( Q, 80A1 )
    WRITE ( *, 2 ) L, '"',  (CVECT(I),I=1,L), '"'
 2  FORMAT ( 1X,  I2, 1X, 80A1 )
    STOP
    END
demo% cat qed2.dat
qwerty
demo% f77 qed2.f -o qed2
qed2.f:
 MAIN:
demo% qed2
  6 "qwerty"
```

The above example gets the length of the input record. With the whole input string and its length, you can then parse it yourself.

Restrictions on the `Q` edit descriptor

- The list element it corresponds to must be of `INTEGER` or `LOGICAL` data type.
- This is strictly a *character* count: it gets the number of *characters* remaining in the input record; it does not get the number of integers or reals or anything else.
- This operates on files only — not with interactive (terminal) I/O.

Sign control (`SU, SP, SS, S`)

For normal output, without any specific sign specifiers, if a value is negative, a minus sign is printed in the first position to the left of the left-most digit; and if the value is positive, printing a plus sign depends on the implementation.

In FORTRAN 77, the following sign specifiers were added:

- `SP` — If `SP` precedes a specification, a sign is printed.
- `SS` — If `SS` precedes a specification, plus-sign printing is suppressed.
- `S` — If `S` precedes a specification, the system default is restored.

Sun FORTRAN has added the sign-control specifier `SU` to cause integer values to be interpreted as *unsigned.*

For example, the *unsigned* specifier can be used with the radix specifier to format a hexadecimal dump, as follows:

```
2000  FORMAT( SU, 16R, 8I10.8 )
```

Rules and Restrictions for Sign Control

- Sign-control specifiers apply to *output* only.
- A sign-control specifier remains in effect until another sign-control specifier is encountered, or format interpretation is complete.
- The `S`, `SP`, and `SS` specifiers affect only `I`, `F`, `E`, `D`, and `G` editing.
- The `SU` specifier affects only `I` editing.

Scale control (`P`)

`P` by itself is equivalent to `0P`. It resets the scale factor to the default value, `0`.

Tab control (`T`,`nT`, `TRn`,`TLn`)

An additional form of tab-control specification has been added. The ANSI standard forms `TRn`, `TLn`, and `Tn` are supported, where *n* is a positive nonzero number. If `T` or `nT` is specified, tabbing is to the next (or n-th) 8-column tab stop. Thus columns of alphanumerics can be lined up without counting.

Nondestructive tabbing is implemented for both internal and external formatted I/O: tabbing left or right on output does not affect previously written portions of a record. Tabbing right on output causes unwritten portions of a record to be filled with blanks. Tabbing right off the end of an input logical record is an error. Tabbing left beyond the beginning of an input logical record leaves the input pointer at the beginning of the record. The format specifier `T` may appear by itself, or be preceded or followed by a positive nonzero number. Tabbing left requires the ability to seek on the logical unit. Therefore, it is not allowed in I/O to a terminal or pipe. Likewise, nondestructive tabbing in either direction is possible only on a unit that can seek. Otherwise tabbing right or spacing with the X edit specifier writes blanks on the output.

Termination control (:)

The colon (:) is the conditional termination edit descriptor. If the I/O list is exhausted before the format, then the format terminates at the colon. For example:

```
* coll.f The colon (:) edit descriptor
      DATA  INIT / 3 /, LAST / 8 /
      WRITE ( *, 2 )  INIT
      WRITE ( *, 2 )  INIT, LAST
 2    FORMAT ( 1X "INIT = ", I2, :, 3X, "LAST = ", I2 )
      STOP
      END
```

The above code produces output such as the following:

```
INIT =  3
INIT =  3   LAST =  8
```

Without the colon, the output is more like this:

```
INIT =  3   LAST =
INIT =  3   LAST =  8
```

Vertical Format Control

Simple vertical format control is provided. The logical unit must be opened for sequential access with FORM='PRINT'. Control codes '0' and '1' in column one are replaced in the output file with '\n' and '\f', respectively. The control character '+' is not implemented and, like any other character in the first position of a record written to a 'print' file, is dropped. No vertical format control is recognized for direct formatted output or list-directed output. See fpr (1) for an alternative way of mapping FORTRAN carriage control to ASCII control characters.

Extensions to I*w*, E*w.d*, G*w.d*

FORTRAN 77 extends the formats I*w*, E*w.d*, and G*w.d* to include the forms:

I*w.m*   E*w.d*E*e*   G*w.d*E*e*

The *m* field

In I*w.m*, the *m* field specifies the *minimum* number of nonblank characters written out, with as many leading zeros as necessary to get a width of *m*. The *m* is ignored on input.

The *e* field

In `E`*w.d*`E`*e* or `G`*w.d*`E`*e*, the *e* field specifies the minimum number of digits or spaces in the exponent field on output. The default value for *e* is 2.

The form `E`*w.d.e*

With the form `E`*w.d.e*, if the value of the exponent is too large, the single character `E` or `D` in the exponent notation is dropped from the output to allow one more character position. If this is still not adequate, the *e* field is filled with asterisks (*). The form `E`*w.d.e* is allowed but is not standard.

Defaults for *w, d, e*

You can write field descriptors `A`, `D`, `E`, `F`, `G`, `I`, `L`, `O`, or `Z` without the *w, d,* or *e* field indicators. If these are left unspecified, the appropriate defaults will be used, based on the data type of the I/O list element. Typical format field descriptor forms that use *w, d, e* include:

`A`*w*, `I`*w*, `L`*w*, `O`*w*, `Z`*w*, `D`*w.d*, `E`*w.d*, `G`*w.d*, `E`*w.d*`E`*e*, `G`*w.d*`E`*e*

For example, with default *w*=7 for `INTEGER*2`, and 161 decimal = A1 hex:

```
      INTEGER*2  M
      M = 161
      WRITE ( *, 8 ) M
    8 FORMAT ( Z )
```

will display as follows:

```
     A1
     ↑
```

column 6

Table 7-5 *Default w, d, e Values in Format Field Descriptors*

Field Descriptor	*List Element*	*w*	*d*	*e*
I,O,Z	BYTE	7	-	-
I,O,Z	INTEGER*2, LOGICAL*2	7	-	-
I,O,Z	INTEGER*4, LOGICAL*4	12	-	-
O,Z	REAL*4	12	-	-
O,Z	REAL*8	23	-	-
O,Z	REAL*16	44	-	-
L	LOGICAL	2	-	-
F,E,D,G	REAL, COMPLEX*8	15	7	2
F,E,D,G	REAL*8, COMPLEX*16	25	16	2
F,E,D,G	REAL*16	42	33	3
A	LOGICAL*1	1	-	-
A	LOGICAL*2, INTEGER*2	2	-	-
A	LOGICAL*4, INTEGER*4	4	-	-
A	REAL*4, COMPLEX*8	4	-	-
A	REAL*8, COMPLEX*16	8	-	-
A	REAL*16	16	-	-
A	CHARACTER**n*	*n*	-	-

NOTE: Default for the A descriptor is the length of the corresponding I/O list element.

Summary of Formats

This table summarizes FORTRAN 66, FORTRAN 77, and Sun FORTRAN formats.

Table 7-6 *FORTRAN Format Specifiers*

Specifier	*FORTRAN 66*	*FORTRAN 77 Extensions*	*Sun FORTRAN Extensions*
Blank Control		BN, BZ	B
Carriage Control			$
Character Edit	wH, Aw	"xxxx" (string constant), A	
Floating-Point Edit	Fw.d, Ew.d, Gw.d, Dw.d	Ew.dEe, Dw.dEe, Gw.dEe	Ew.d.e, Dw.d.e, Gw.d.e
Hexadecimal Edit			Zw.m
Integer Edit	Iw	Iw.m	
Logical Edit	Lw		
Octal Edit			Ow.m
Position Control		Tn, TLn, TRn	nT, T
Position Edit	wX, /		
Radix Control			nR
Remaining Characters			Q
Scale Control		nP	P
Sign Control		S, SP, SS	SU
Terminate a Format		:	

7.6. Magnetic Tape I/O

Using tape files on a SunOS or UNIX systems is awkward because, historically, UNIX development was oriented toward small data sets residing on fast disks. Magnetic tape was used by early UNIX systems for archival storage and moving data between different machines. Unfortunately, many FORTRAN programs are intended to use large data sets from magnetic tape.

Using TOPEN

A FORTRAN tape I/O package (see TOPEN(3F)) offers a partial solution to the problem. FORTRAN programmers can transfer blocks between the tape drive and buffers declared as FORTRAN character variables. The programmer can then use internal I/O to fill and empty these buffers. This facility does not integrate with the rest of FORTRAN I/O (it even has its own set of tape logical units); thus, its use is discouraged.

Formatted

Sun FORTRAN provides facilities for transparent access to formatted, sequential files on magnetic tape. The tape block size may be optionally controlled by the OPEN statement's FILEOPT parameter. There is no bound on formatted record size and records may span tape blocks.

Unformatted

Connecting a magnetic tape for unformatted access is less satisfactory. Because of the implementation of unformatted records as a sequence of characters preceded and followed by character counts, the first word of the record must be backpatched after the length of the entire record is known. This is due to the sequential property of the medium, which makes it impossible to seek back and rewrite this word. Thus, the size of a record (+ 8 characters of overhead) cannot be bigger than the buffer size.

As long as this restriction is honored, the I/O system does not write records that span physical tape blocks, but writes short blocks when necessary. This representation of unformatted records is preserved (even though it is inappropriate for tape), so files can be freely copied between disk and tapes. Note that, since the block-spanning restriction does not apply to tape reads, files can be copied from disk to tape without any special considerations.

Tape File Representation

A FORTRAN file is represented on tape by a sequence of data records followed by an endfile record. The data is grouped into blocks, the maximum size determined when the file is opened. The records are represented the same as records in disk files: formatted records are followed by newlines, unformatted records are preceded and followed by character counts. In general, there is no relation between FORTRAN records and tape blocks; that is, records can span blocks, which can contain parts of several records. The only exception is that FORTRAN won't write an unformatted record that spans blocks; thus, the size of the largest unformatted record is eight characters less than the block size.

The dd conversion utility

An endfile record in FORTRAN maps directly into a tape mark. Thus, FORTRAN files are the same as tape system files. Because the representation of FORTRAN files on tape is the same as that used in the rest of UNIX, naive FORTRAN programs cannot read 80-column card images from tape. If you have an existing FORTRAN program and an existing data tape you wish to read with it, you should translate the tape using the dd(1) utility, which adds newlines and strips trailing blanks. For example:

```
demo% dd if=/dev/rmt0 ibs=20b cbs=80 conv=unblock | ftnprg
```

The GETC library routine

If you write or modify a program and don't want to use dd, you can use the GETC(3F) library routine to read characters from the tape. You can then assemble the characters into a character variable and use internal I/O to transfer formatted data. See also TOPEN(3F).

End-of-File

The end-of-file condition is reached when an endfile record is encountered during execution of a READ statement. The standard states that the file is positioned after the endfile record. In real life, this means that the tape read head is poised at the beginning of the next file on the tape. Thus, it would seem that you should be able to continue reading the next file on the tape; however, this doesn't work and is prohibited by the standard.

The standard also says that a BACKSPACE or REWIND statement may be used to reposition the file. This means that after reaching end-of-file, you can backspace over the endfile record and further manipulate the file (such as writing more records at the end), rewind the file, and reread or rewrite it.

Access on Multi-File Tapes

Each tape drive can be opened by many names. The name used determines certain characteristics of the connection, which are the recording density and whether the tape is automatically rewound when opened and closed. To access a file on a multiple-file tape, you should use the mt(1) utility to position the tape to the correct file, then open the file as a no-rewind magnetic tape such as /dev/nrmt0. Using the tape with this name also prevents it from being repositioned when it is closed. This means that if your program reads the file until end-of-file, then reopens it, it can access the next file on the tape. Any following programs can access the tape where you left it (preferably at the beginning of a file, or past the endfile record). If your program terminates prematurely it could leave the tape positioned in an unpredictable place.

8

Program Development

8

Program Development

This chapter introduces building programs with make, tracking changes with SCCS, and creating libraries with ar and ranlib.

8.1. Simple Program Builds

For a program that depends on only a single source file and some system libraries, you can compile all every time you change the program. Even in this simple case, the f77 command can involve much typing, and with options or libraries, a lot to remember. A simple script or alias can help.

Making a Script

For example, to compile a small program contained in the file example.f, that uses the SunCore graphics library, you can save a one-line shell script onto a file called fex, that looks like this:

```
f77 example.f -lcore77 -lcore -o example
```

You may need to put execution permissions on fex:

```
demo% chmod +x fex
```

Making an Alias

Or you can set up an alias to do the same command:

```
demo% alias fex "f77 example.f -lcore77 \
        -lcore -o example"
```

Using a Script or Alias

Either way, to recompile example.f, you type only **fex**:

```
demo% fex
```

Limitations

With multiple source files, forgetting one compile makes the objects inconsistent with the source. Recompiling all files after every editing session wastes time, since not every source file needs recompiling. Forgetting an option or a library produces questionable executables. The make program can help.

8.2. Program Builds with the make Program

The make program recompiles only what needs recompiling, and it uses only the options and libraries you want. This section shows you how to use normal, basic make, and it provides a simple example. For a full discussion of make, see the chapter "Make — a Program for Maintaining Computer Programs" in the *Programming Tools* manual. For a summary, see the *Sun User's Manual* page make(1).

The makefile

The way you tell make what files depend on other files, and what processes to apply to which files, is to put this information into a file called the *makefile*, in the directory where you are developing the program.

For example, suppose that you have a simple program of four source files and a makefile, as listed below:

```
demo% ls
Makefile
commonblock
computepts.f
pattern.f
startupcore.f
demo% ■
```

For this example, assume that both pattern.f and computepts.f do an include of commonblock, and you wish to compile each .f file and link the three relocatable files (plus a series of libraries) into a program called pattern.

The makefile for this example is listed below:

```
demo% cat Makefile

pattern : pattern.o computepts.o startupcore.o
    f77 pattern.o computepts.o startupcore.o -lcore77 \
        -lcore -lsunwindow -lpixrect -o pattern

pattern.o : pattern.f commonblock
        f77 -c -u pattern.f

computepts.o : computepts.f commonblock
        f77 -c -u computepts.f

startupcore.o : startupcore.f
        f77 -c -u startupcore.f
demo% ■
```

The first line of this makefile says "make pattern", and "pattern depends on pattern.o, computepts.o, and startupcore.o". The second line is the command for making pattern, and the third line is a continuation of the second.

There are four such paragraphs or entries in this makefile. The structure of these entries is:

- **Dependencies** — Each entry starts with a line that names the file to make, and names all the files it depends on.
- **Commands** — Each entry has one or more subsequent lines that contain shell commands, and that tell how to build the target file for this entry. These subsequent lines must each be indented by a tab,

Using make

When make is invoked with no arguments:

```
demo% make
```

it looks for a file named makefile or Makefile in the current directory, and takes its instructions from there.

Its general actions are:

- From the makefile, it gets all the target files it must make, and what files they depend on. It also gets the commands used to make the target files.
- It gets the date and time changed for each file.
- If any target file is not up to date with the files it depends on, then that target is rebuilt, using the commands from the makefile for that target.

The C Preprocessor

You can use the C preprocessor for such things as passing strings to f77. The example is just an extension of the example above.

Let's say that you want your program to print the time it was compiled when it is given a command-line argument of **–v**. You need to add code that looks like:

```
IF (ARGSTRING .EQ. "-v") THEN
    PRINT *, CTIME
    CALL EXIT(0)
ENDIF
```

and use the C preprocessor to define CTIME as a quoted string that can be printed. The next two examples show how to do this.

The .F Suffix

The C preprocessor is applied if the filenames have the suffix .F, so we change the filename:

```
demo% mv pattern.f pattern.F
```

The –D Option

The **–D** option defines a name to have a specified value for the C preprocessor, as if by a "#define" line. So we change the compilation line for pattern.F in the makefile to look like this:

```
demo% f77 "-DCTIME=\" `date` \" " -c -u pattern.F
```

Essentially, the part up to the -c option gets the output of the date command, puts quotes around it, stuffs that into CTIME, and passes that on to the C preprocessor. (If you want to follow the gory details, see the note below.[†])

The preprocessor now converts CTIME to "jan15...", so that:

```
PRINT *, CTIME
```

becomes

```
PRINT *, "jan15..."
```

The purpose here is to show *how* such strings are passed to the C preprocessor; the particular string passed is not useful, but the *method* is the same.

Macros with make

The make program does simple paramaterless *macro* substitutions. In the example above, the list of relocatable files that go into the target program pattern appears twice: once in the dependencies and once in the f77 command that follows. This makes changing the makefile error-prone, since the same changes must be made in two places in the file. In this case, you can add the following to the beginning of your makefile:

Sample macro definition

```
OBJ = pattern.o computepts.o startupcore.o
```

and change the description of the program pattern into:

Sample macro use

```
pattern: $(OBJ)
        f77 $(OBJ) -lcore77 -lcore -lsunwindow \
           -lpixrect -o pattern
```

Note the peculiar syntax in the above example: a *use* of a macro is indicated by a dollar sign immediately followed by the name of the macro in parentheses. For macros with single-letter names, the parentheses may be omitted.

[†] The innermost single quotes are back-quotes or grave accents. They indicate that the output of the command contained in them (in this case the date command) is to be substituted in place of the backquoted word(s). The next level of quote marks is what makes this define a FORTRAN quoted string, so it can be used in the print statement. These marks must be escaped (or "quoted") by preceding backslashes because they are nested inside another pair of quote marks. The outermost marks indicate to the interpreting shell that the enclosed characters are to be interpreted as a single argument to the f77 command. They are necessary because the output of the date command contains blanks, so that without the outermost quoting it would be interpreted as several arguments, which would not be acceptable to f77.

Overriding Macro Values

The initial values of make macros can be overridden with command-line options to make. For instance, if you add the line:

```
FFLAGS=-u
```

to the top of your makefile, and change each command for making FORTRAN source files into relocatable files by deleting that flag, the compilation of computepts.f looks like this:

```
        f77 $(FFLAGS) -c  computepts.f
```

and the final link looks like this:

```
f77 $(FFLAGS) $(OBJ) -lcore77 -lcore -lsunwindow \
      -lpixrect -o pattern
```

If you issue the bare make command, everything compiles as before. But if you give the command:

```
demo% make "FFLAGS=-u -O"
```

then the **-O** flag, as well as the **-u** flag, is passed to f77.

Suffix Rules in make

If you don't tell make how to make a relocatable file, it uses one of its default rules, in this case:

> Use the f77 compiler, pass as arguments any flags specified by the FFLAGS macro, the **-c** flag, and the name of the source file to be compiled.

We can take advantage of this rule twice in our example, but still must explicitly state the dependencies, and the nonstandard command for compiling the pattern.F file. The makefile is:

```
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=-u
pattern: $(OBJ)
      f77 $(OBJ) -lcore77 -lcore -lsunwindow \
            -lpixrect -o pattern
pattern.o: pattern.F commonblock
      f77 $(FFLAGS) "-DCTIME=\" 'date' \" " -c  pattern.F
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

8.3. Tracking and Controlling Changes with SCCS

SCCS is Source Code Control System. It provides a way to:

- keep track of a source file's evolution (change history)
- prevent different programmers from changing the same source file at the same time
- keep track of the version number by providing version stamps

The basic three operations of SCCS are: putting files under SCCS control, checking out a file for editing, and checking in a file. This section shows you how to use SCCS to do these things and provides a simple example, using the previous program. It describes normal, basic SCCS, and introduces only three SCCS commands: `create`, `edit`, and `delget`.[†]

Putting Files under SCCS

Putting files under SCCS control involves making the SCCS directory, inserting SCCS ID keywords into the files (optional), and creating the SCCS files.

Making the SCCS directory

To begin, you must create the SCCS subdirectory in the directory in which your program is being developed:

```
demo% mkdir SCCS
demo% █
```

The '`SCCS`' must be upper case.

Inserting SCCS ID keywords

Put one or more SCCS "ID keywords" into each file. These will later be filled in with a version number each time the file is checked in with a `get` or `delget` SCCS command. There are three likely places to put such strings:

- in comment lines,
- in parameter statements, or
- in initialized data.

The advantage of the last is that the version information appears in the compiled object program, and can be printed using the `what` command. Included header files containing only parameter and data definition statements should not generate any initialized data, so the keywords for those files usually are put in comments or in parameter statements. Finally, in the case of some files, like ASCII data files or makefiles, the source is all there is, so the SCCS information can go in comments, if anywhere.

Let's identify the makefile with a `make` comment containing the keywords:

```
#        %Z%%M%    %I%    %E%
```

The source files `startupcore.f` and `computepts.f` and `pattern.f` can

[†] For more on SCCS, see "Source Code Control System" in *Programming Utilities and Libraries* Although addressed mainly to the C language programmer, that manual provides a thorough introduction to the mechanics of using SCCS.

9

Debugging and Profiling

9.1. Introduction

This chapter describes tools for debugging and measuring the resource use of FORTRAN programs. The most versatile and powerful tool for debugging on the Sun workstation is the symbolic debugger `dbx` or `dbxtool`. With `dbx` you can display and modify variables, set breakpoints, trace variables, and invoke procedures in the program being debugged without having to recompile. `dbxtool` is a Sun workstation debugger that lets you make more effective use of `dbx` by replacing the original, terminal-oriented interface with a window- and mouse-based interface.

The `adb` debugger is an older binary-oriented debugger, which is occasionally useful as a supplement to `dbx`.

The **-C**, **-u**, and **-v** flags are useful for debugging, see Section 9.4 .

The simplest way to measure resource consumption is with the `time`(1) command. The `gprof`(1) command provides a detailed procedure-by-procedure analysis of execution time, including how many times a procedure was called, who called it and who it called, and how much time was spent in the procedure and by the routines that it called.

The following program is used in several examples:

a1.f:

```
      program silly
      real twobytwo(2,2)
      data twobytwo /4 *-1 /
      n = 2
      call mkidentity( twobytwo, n )
      print *, determinant(twobytwo)
      end
```

a2.f:

```
      subroutine mkidentity(matrix,dim)
      real matrix(dim,dim)
      integer dim
      do 10 m = 1, dim
      do 20 n = 1, dim
      if(m.eq.n) then
          matrix(m,n) = 1.
      else
          matrix(m,n) = 0.
      endif
20    continue
10    continue
      return
      end
```

a3.f:

```
      real function determinant(m)
      real m(2,2)
      determinant = m(1,1)*m(2,2) - m(1,2)/m(2,1)
      return
      end
```

9.2. Using dbx

This section summarizes the use of dbx and describes some of its FORTRAN–specific aspects. Complete documentation for dbx and dbxtool can be found in the dbx(1) and dbxtool(1) man pages and in *Debugging Tools*.

dbx commands

compile

To use dbx or dbxtool, you must compile and load your program with the **-g** flag.[*] For example:

```
demo% f77 -o silly -g a1.f a2.f a3.f
```

or:

```
demo% f77 -c -g a1.f a2.f a3.f
demo% f77 -g -o silly a1.o a2.o a3.o
```

dbx

[1] The -g and -O options are incompatible. If used together, the -g option cancels the -O option.

To run the program under the control of dbx, change to the directory where the sources and programs reside, then type the dbx command and the name of the executables file:

```
demo% dbx silly
Reading symbolic information...
Read 635 symbols
(dbx)
```

breakpoint

To set a breakpoint before the first executable statement, wait for the (dbx) prompt, then type **stop in MAIN**, as follows:

```
(dbx) stop in MAIN
(1) stop in MAIN
(dbx)
```

run

After the (dbx) prompt appears, type **run** to begin execution. When the breakpoint is reached, dbx displays a message showing where it stopped, in this case at line 4 of file a1.f:

```
(dbx) run
Running: silly
stopped in MAIN at line 4 in file "a1.f"
    4           n = 2
(dbx)
```

print

The command **print n** at this point displays 0, since the statement n=2 has not been executed yet:

```
(dbx) print n
'a1'MAIN'n = 0
(dbx)
```

The command **next** advances execution to line 5, and if the **print n** command is now repeated it displays a 2:

```
(dbx) next
stopped in MAIN at line 5 in file "a1.f"
    5           call mkidentity( twobytwo, n )
(dbx) print n
'a1'MAIN'n = 2
(dbx)
```

The command **print twobytwo** displays the entire matrix, one element per line. Note that square brackets (not parentheses) are used to reference array elements:

```
(dbx) print twobytwo
twobytwo = [1,1]        -1.0
        [2,1]   -1.0
        [1,2]   -1.0
        [2,2]   -1.0

(dbx)
```

The command **print matrix** fails because subroutine `mkidentity` is not active at this point and the bounds of the adjustable array `matrix` are not know:

```
(dbx) print matrix
"matrix" is not active
(dbx)
```

Execution can be continued in three ways:

continue The **continue** command resumes execution without setting further breakpoints.

next The `next` command sets a one-time breakpoint, in this case at line 5 of file `a1.f`, and continues execution until that point is reached.

step The **step** command sets a breakpoint at the next source line to be executed, in this case line 4 of file `a2.f`. If the next statement is a subroutine or function call, then **step** sets a breakpoint at the first source line of the *subprogram*, but **next** sets the breakpoint at the first source line after the call but still in the *calling program.*

Throughout a debugging session, `dbx` defines a procedure and a source file (the file that contains the source for the current procedure) as "current." Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, **stop at 5** sets one of three different breakpoints depending on whether the current file is `a1.f`, `a2.f`, or `a3.f`.

Likewise, **print n** displays a different storage location when the current function is `main` than when it is `mkidentity`.

where The **where** command shows where in the program execution stopped and how execution reached this point.

which The **which** command shows exactly which variable `n` is being referenced.

func/file The **func** and **file** commands can be used to alter `dbx`'s definition of the current procedure.

status The **status** command lists the breakpoints in effect and the **delete** command removes breakpoints.

It is possible to call a subroutine or function in the program at any point when execution has stopped. The effect is exactly as if the source had contained a call at that point. For example if, after the initial breakpoint described above, you typed **print determinant (twobytwo)** the value 0 would display, since mkidentity would not yet have modified twobytwo.

This facility is often useful for special-case printing. For example, in a program it might be meaningful to trace the row and column sums of different matrices. A subroutine called matsum that does this could be compiled into a program and invoked by the user at appropriate breakpoints.

postmortem

Assume that file a3.f was modified as follows:

```
real function determinant(m,dim)
real m(dim,dim)
integer dim
determinant = m(1,1)*m(2,2) - m(1,2)*m(2,1)
return
end
```

Execution results in a "segmentation violation" as soon as determinant is invoked and a core file (a copy of the program's image in memory) is produced. The command **dbx silly core** correlates this program image with the program, as follows:

```
demo% dbx silly core
```

where

Then **where** commands can determine which routines were active at the time of the exception:

```
(dbx) where
determinant(m =  ARRAY , dim = 16776938),
    line 5 in "a3.f"
MAIN, line 6 in "a1.f"
MAIN(0x1, 0xfffeb0, 0xfffeb8) at 0x82fa
(dbx)
```

Structures and pointers

The dbx debugger recognizes the Sun FORTRAN data types of *structure*, *record*, *union*, and *pointer*. The following examples show using dbx with these data types.

Compile for dbx using the **-g** option, load it in dbx, and list it:

```
demo% f77 -o debstr -g debl.f
debl.f:
 MAIN:
demo% dbx debstr
Reading symbolic information...
Read 604 symbols
(dbx) list 1,30
     1   * dbxl.f: Show dbx with structures and pointers
     2        STRUCTURE  /PRODUCT/
     3                INTEGER*4      ID
     4                CHARACTER*16   NAME
     5                CHARACTER*8    MODEL
     6                REAL*4         COST
     7                REAL*4         PRICE
     8        END STRUCTURE
     9
    10        RECORD /PRODUCT/ PROD1, PROD2
    11        POINTER (PRIOR, PROD1), (CURR, PROD2)
    12
    13        PRIOR = MALLOC( 36 )
    14        PROD1.ID = 82
    15        PROD1.NAME = "Schlepper"
    16        PROD1.MODEL = "XL"
    17        PROD1.COST = 24.0
    18        PROD1.PRICE = 104.0
    19        CURR = MALLOC( 36 )
    20        PROD2 = PROD1
    21        WRITE ( *, * ) PROD2.NAME
    22        STOP
    23        END
(dbx)
```

Set a breakpoint at a specific line number, and run it under dbx:

```
(dbx) stop at "debl.f":21
(1) stop at "debl.f":21
(dbx) run
Running: debstr
stopped in main at line 21 in file "debl.f"
   21          WRITE ( *, * ) PROD2.NAME
(dbx)
```

Print a record:

```
(dbx) print prod1
*prod1 = (
        id     =        82
        name   =        "Schlepper        "
        model  =        "XL        "
        cost   = 24.0
        price  = 104.0
)
(dbx)
```

If you tell dbx to print a record, it displays all fields of the record, including field names.

Inquire about a record:

```
(dbx) whatis prod1
 (based variable) record /product/ prod1
(dbx) whatis product
structure /product/
        integer*4 id
        character*16 name
        character*8 model
        real cost
        real price
end structure product
(dbx)
```

Print a pointer, then quit dbx:

```
(dbx) print prior
prior = 150244
(dbx) quit
demo%
```

If you tell it to print a pointer, it displays the contents of that pointer, which is the address of the variable pointed to.

Parameters

The dbx debugger recognizes parameters — the compiler generates pseudo variables for parameters when programs are compiled for dbx with the **-g** option. The following examples show using dbx with parameters.

Compile for dbx using the **-g** option, load it in dbx and list it. Print some parameters.

```
demo% f77 -o silly -g deb2.f a2.f a3.f
deb2.f:
deb2.f:
 MAIN silly:
a2.f:
a2.f:
        mkidentity:
a3.f:
a3.f:
        determinant:
Linking:
demo% dbx silly
Reading symbolic information...
Read 269 symbols
(dbx) list 1,30
    1           program silly
    2           parameter ( n=2, nn=n*n )
    3           real twobytwo(n,n)
    4           data twobytwo /nn *-1 /
    5           call mkidentity( twobytwo, n )
    6           print *, determinant(twobytwo)
    7           end
(dbx) print n
`deb2`MAIN`n = 2
(dbx) print nn
nn = 4
(dbx) quit
demo%
```

9.3. Using adb

The adb debugger can also be used to provide a stack traceback but at a lower level. The adb program does NOT display a prompt; it just waits.

start

For example, adb silly core starts up adb and the command $c displays something like the following:

```
_abort[d590]() + 4
_sigdie[0](b,0,fffe30) + 152
__sigtramp[11ab0]() + 20
_determinant_[81dc](1801c) + 36
_MAIN_[8074]() + 36
_main[82a0](1,fffeb0,fffeb8) + 54
```

This is interpreted as follows. The startup routine main, called the FORTRAN MAIN routine, which in turn called the function determinant (note the underscores appended to FORTRAN external names). Somewhere around 36 (hex) bytes from the beginning of determinant an exception occurred. The exception is recorded as a call to the signal dispatcher sigtramp. sigtramp noted that the particular signal was handled by sigdie, a signal handling routine in the FORTRAN library, and then called it. sigdie printed a message and then called abort to halt execution. The command determinant_,10?ia displays

10(hex) machine instructions and their addresses starting from the entry point `determinant`.

quit To quit `adb`, type **$q** or **$Q** or **^D**.

`adb` can be used on any program regardless of whether or not it was compiled with the **-g** debugging flag. Variables can be displayed in a variety of formats, but their addresses must be known. The addresses of some external variables are easy to determine. For example, the command `__BLNK__/D` prints the first four bytes after label `__BLNK__` in a decimal format, which is equivalent to the `dbx` `print n` command if `n` is the first variable in blank common. The addresses of local variables are usually difficult to determine.

As another example, consider the program:

```
    write(4) 4
    end
```

When executed, this program creates a file named `fort.4` which contains a single unformatted record. An unformatted record includes two count words containing the record length at the beginning and end of the record. You can examine this data file with `adb` as follows:

```
demo% adb fort.4 -
```

Then display the first three words of the data file in decimal (location 0 with a repeat count of three):

```
0,3?D
0:              4               4               4
$q
demo%
```

9.4. Compiler flags

The compiler provides three flags that are useful for debugging FORTRAN programs: **-C** **-u** and **-v**

-C The **-C** flag causes the compiler to generate subscript checking code that catches certain kinds of out-of-bounds array subscripts.
For example, if line 7 of file `a2.f` were changed to:

```
    matrix(2*m,2*n) = 1.
```

Execution would produce the message

```
Subscript out of range on file line 7,
procedure mkidenti.
Attempt to access the 10-th element of
variable matrix.
```

The -c does NOT catch all subscript range errors.

The current implementation does not catch all out of range subscripts. For example, if `dim` is greater than 2, then a reference of the form `matrix(2*dim,1)`, though illegal, does not produce an error. An error is flagged only if a subscript expression causes a reference outside the linearized internal representation of the array.

-u The **-u** flag causes all variables to be initially declared "UNDEFINED", so that an error is flagged for variables that are not explicitly declared. The **-u** flag is useful for discovering mistyped variables. When **-u** is set, all variables are treated as undefined until explicitly declared. Use of an undefined variable is accompanied by an error message.

-v The **-v** flag produces a log of the various phases of the compiler along with information about the resources used by each phase. This can be useful in tracking the origin of ambiguous error messages and in reporting compiler failures.

9.5. Profiling Tools

The simplest way to gather data about the resources consumed by a program is to use the `time` command or, in the C shell to issue the `set time` command. After the program terminates, the shell prints a line like this:

```
6.5u 17.1s 1:16 31% 11+21k 354+210io 135pf+0w
```

This indicates that the program spent six seconds executing user code, 17 seconds executing kernel code on behalf of the user, and took one minute and 16 seconds to complete, so that approximately 31 per cent of the machine's resources were dedicated to this program. Memory usage during execution averaged 11 kilobytes of shared (program) memory and 21 kilobytes of private (data) memory. Input and output operations done by the program resulted in 564 disk accesses of which 354 were reads and 210 were writes. The program caused 135 page faults and was never swapped out.

To obtain a more detailed account of how the program spent its time we can compile and link it with the **-pg** flag, for example:

```
demo% f77 -o silly -pg a1.f a2.f a3.f
```

After execution completes, a file named `gmon.out` is written in the working directory. This file contains profiling data that can be interpreted with `gprof`(1). To generate meaningful timing information, execution must complete normally. The command **`gprof silly`** invokes `gprof` and asks it to correlate the `gmon.out` file with the program in file `silly`. `gprof` produces two summaries of how the total time (user time plus system time) the program uses is distributed across the program's procedures. Both user routines and library routines are accounted for.

The "flat" profile lists the procedures along with the number of times each procedure was called and the number of seconds spent in the routine. This information can be useful, but does not allow you to determine the calling structure of the program and how time is distributed across it. For example, if you discover that a vector cross-product function that is called from many points in a program is taking up most of the execution time, you can't tell who it calls most often and causes it to do the most work. The second summary produced by gprof, the "graph" profile, can help answer these questions.

For example, if you modify MAIN to call mkidentity 1000 times, then compile your source files with the **-pg** flag and call gprof to produce timing profiles, an entry in the graph profile might look like this:

```
                      0.18        0.24    1000/1000         _MAIN_ [4]
[3]         95.5      0.18        0.24    1000              _mkidentity_ [3]
                      0.24        0.00    4000/4000         lmult [5]
```

In the graph profile above, the line that begins with "[3]" is called the function line, the lines above it the "parent lines", and the lines below it the "descendant" lines. The function line in the example above reveals that mkidenity was called 1000 times, a total of 0.18 seconds were spent in mkidentity itself and 0.24 seconds were spent in routines called by mkidentity. 95.5 per cent of the program's execution time is attributable to mkidentity and its descendants.

The single parent line reveals that MAIN was the only procedure to call mkidentity, that is, all 1000 invocations of mkidentity came from MAIN. Thus, all of the 0.18 seconds spent in mkidentity were spent on behalf of MAIN and all 0.24 seconds of mkidentity s descendants were spent on behalf of MAIN. If mkidentity had also been called from another procedure there would be two parent lines and the 0.18 seconds of "self" time and 0.24 seconds of "descendant time" would be divided between MAIN and the other caller.

The descendant lines are interpreted similarly. In this example, mkidentity has only called one function, lmult, the 32-bit integer multiply routine. lmult is called 4000 times in this program and all of these calls come from mkidentity. lmult has a descendant time of zero, which suggests that it calls no other routines (this could be confirmed by examining the lmult entry).

When you enable profiling, the running time of a program is significantly increased. The fact that mcount the utility routine used to gather the raw profiling data, is usually at the top of the flat profile shows this. To eliminate this overhead in the completed version of the program, recompile all source files without the **-pg** flag. The overhead incurred by mcount should be ignored when interpreting the flat profile. The graph profile automatically subtracts time attributed to mcount when computing percentages of total runtime.

The FORTRAN library includes three routines that return the total time used by the calling process — see dtime (3F), etime (3F), and tcov (1).

10

The VMS Extensions

10

The VMS Extensions

10.1. Overview

This chapter provides the following information about the VMS extensions:

- □ A summary of the VMS extensions *recognized* by `f77`
- □ A complete description of the VMS source code converter `f77cvt` and the conversions it makes
- □ A summary of the VMS extensions *not* supported by `f77` or `f77cvt`.
- □ A table of the VMS Intrinsic Functions supported by `f77` or `f77cvt`.

10.2. Background

Sun FORTRAN has added the VMS extensions to make it as easy as possible to port FORTRAN programs from VMS environments to Sun workstations.

These extensions involve three systems:

- □ The Sun FORTRAN compiler, which supports many of the VMS features,
- □ The source code conversion program, which converts most of the remaining extensions into statements that the Sun FORTRAN compiler will accept.
- □ The debugger, which supports many of the VMS features

Used together, the compiler and converter provide almost complete compatibility with VMS FORTRAN.

Sun FORTRAN includes:

- □ Relics of previous DEC FORTRAN compilers or VMS-specific features that provide little in terms of the language, such as the `TYPE` and `ACCEPT`.
- □ Features that improve readability but not capability; examples: unlabeled `DO/END DO`, `DO WHILE`, and end-of-line comments.
- □ Significant language enhancements, such as namelist I/O, structures, and unions.

Supporting features of the first category lends itself to separate pre-processing. For this reason, most features in the first category are converted into standard FORTRAN by the source code converter. The converter prints diagnostic messages for those features that remain unconverted.

The features in the second category are either implemented as compiler extensions or converted by the source code converter, depending on ease of conversion and general usefulness.

The features in the last category are implemented as extensions to the Sun FORTRAN compiler.

10.3. The VMS Extensions Recognized by f77

This is a summary of the VMS extensions are recognized by f77. They are described elsewhere in this manual.

- Namelist I/O
- Unlabeled DO ... END DO
- Indefinite DO WHILE ... END DO
- The BYTE data type
- Logical operations on integers, and arithmetic operations on logicals
- Additional field and edit descriptors for FORMAT s:

  Remaining characters (Q)
  Carriage Control ($)
  Octal (O)
  Hexadecimal (X)
  Hexadecimal (Z)

- Default field indicators for *w*, *d*, and *e* field descriptors in FORMAT s
- READ s into Hollerith edit descriptors
- The additional option APPEND for the OPEN statement:

```
        OPEN( ... , ACCESS='APPEND', ... )
```

- Long names (32 characters) and acceptance of "_" and "$" in names
- Long (132-character) source lines
- Records, structures, unions, and maps
- Getting addresses via the LOC function
- Passing arguments via the %VAL function

10.4. The Source Code Converter

In general, the source-code conversion program accepts files that contain valid VMS FORTRAN† source code and produces source files acceptable to both the Sun and VMS FORTRAN compilers. The converter accepts options that correspond to the VMS compiler options, and if it finds one it doesn't know, it generates a diagnostic message.

† Designed for VMS FORTRAN, level 4.0, as documented.

Usage

The syntax of the source code converter command is as follows:

f77cvt [*options*] *filename*.[vf | for] [*filename*.[vf | for] ...]

For example:

```
demo% f77cvt growth.for  fft.vf
```

Input

Each filename must have one of the forms:

name.vf   *name*.for

The suffix .vf or .for can be any mixture of upper and lower case. Each file must contain valid VMS FORTRAN source code.

Output

For each *input* file there is a corresponding *output* file, with the same filename but a different suffix. Each output *filename* is in one of the following forms:

name.f   *name*.F

The filename gets a .f or a .F suffix, depending on the -D option. (Options are explained in the following section.)

Each output file is a FORTRAN source file whose code is acceptable to both the Sun and VMS compilers.*

The output has an optional detailed summary of the conversions performed, as well as an identification of the lines that could not be converted. The unconvertible are written to standard error.

Using f77cvt with f77

Generally, you should be able to run f77 immediately after a f77cvt run. For example:

```
demo% f77cvt myprog.for
          ... various messages from   f77cvt   ...
demo% f77 myprog.f
           ... various messages from f77 ...
demo%
```

In general, with existing VMS FORTRAN programs, first use f77cvt, then use f77.

[1] Except that there may be VMS FORTRAN constructs that the program can't convert, which generate a message on standard error. See the Section *Unsupported VMS FORTRAN* .

Converter Options

The converter accepts the following options:

-b Prevents the converter from creating `BLOCK DATA` subprograms for initialized `COMMON` variables.

-d Enables VMS FORTRAN debugging statements. If specified, lines with a `D` in column 1 are converted into FORTRAN statements. If this option is not used, these debugging statements are converted into comments.

-D Also enables VMS FORTRAN debugging statements, but the debugging statements are enclosed by a pair of preprocessor statements, as shown here:

```
#ifdef DEBUG
     .
     .
#endif
```

The `-D` and `-d` are mutually exclusive; do *not* specify both.

-e Indicates that the input file(s) contains extended source lines (up to 132 characters). If your file contains extended lines and you omit this option, the source lines are truncated to 72 characters.

-E Allows the output file(s) to contain extended source lines. If this option is not specified, the output lines are broken into 72-column lines.

-i Inserts the text of the `INCLUDE` files into the converted program.

-Nc*x* Sets *x* as the maximum number of levels that control structures can be nested. The default is 20.

-Nd*x* Sets *x* as the maximum number of levels that data structures and unions can be nested. The default is 20.

-Nl*x* Sets *x* as the maximum number of continuation lines per statement. (If you use `-Nl50`, then you can have a total of 51 lines: one first line and 50 after it.) The default is 19, that is, one initial line and 19 continuation lines.

-P Suppresses the generation of preprocessor line numbers for `dbx` and `f77`.

-s Produces warning messages if Sun-specific FORTRAN extensions are generated; examples include character/non-character equivalencing, 32-character variable names, `IMPLICIT UNDEFINED` statements, and hexadecimal constants.

-v Enables *verbose* mode; that is, the converter places a FORTRAN comment before each converted line, describing the conversion of the source code.

Conversion Description

The major items converted are described in some detail in the following pages.

Embedded Comments

End-of-line comments are converted into comment lines preceding the statement.

Debug Statements

Debugging lines are converted into comment lines or FORTRAN statements, depending on whether the `-D` or `-d` options are set.
(See Section 10.4, under Subsection *Usage*)

VMS Tab-format

Tab-format source lines are converted into standard FORTRAN source lines.

Initializing `BLOCK DATA`

Initialization of variables in common blocks is moved to a `BLOCK DATA` subprogram. (But see also the **-b** option.)

Radix-50 Constants

Radix-50 constants are relics from older DEC FORTRAN compilers, implemented to save memory on procedure names, and so forth. These constants are converted to Sun bit-string constants—that is, no type is assumed.

The `IMPLICIT NONE` Statement

The VMS statement:

```
   IMPLICIT NONE
```

is converted to the equivalent:

```
   IMPLICIT UNDEFINED (A-Z)
```

The `VIRTUAL` statement

The `f77cvt` program converts a `VIRTUAL` statement to a `DIMENSION` statement (a synonym).

Nonstandard `PARAMETER` Statements

The alternate `PARAMETER` statement syntax is converted to standard syntax, including a type declaration if necessary. For example, the following statement:

```
    PARAMETER FLAG1 = .TRUE.
```

is converted to:

```
    LOGICAL   FLAG1
    PARAMETER (FLAG1 = .TRUE.)
```

Note that an ambiguous statement that could be interpreted as either a `PARAMETER` statement or an assignment statement is always taken to be the former. For example:

```
    PARAMETER S = .TRUE.
```

is interpreted as the PARAMETER statement about the variable S:

```
PARAMETER S = .TRUE.
```

rather than the assignment statement about the variable PARAMETERS:

```
PARAMETERS = .TRUE.
```

Initializers in Declarations

Initialization of variables within declaration statements is changed to the standard declaration statement followed by a DATA statement; for example:

```
CHARACTER*10 NAME /'Nell'/
```

is converted to:

```
CHARACTER*10 NAME
DATA NAME /'Nell'/
```

If such initialization statements involve variables in COMMON, then they are moved to a BLOCK DATA subprogram.

Non-CHARACTER Format Specifiers

If a runtime format specifier is not of type CHARACTER, the converter creates a CHARACTER variable and EQUIVALENCEs that variable to the one specified. The result is used for the format. The ability to equivalence the character and non-character variables is a Sun extension to standard FORTRAN.

Omitted Actual Arguments

Whenever the converter finds an omitted argument in a subroutine call (that is, two consecutive commas), it inserts a zero.

Variables and Literals of Type REAL*16

The converter changes variables of type REAL*16 to DOUBLE PRECISION. Likewise, it converts literals such as 1.0Q1 to the 1.0D1 form.

Using a CHARACTER Alias for Non-CHARACTER Variables

Standard FORTRAN requires the FILE= specifier for OPEN and INQUIRE to be an expression of type CHARACTER. VMS FORTRAN, however, also accepts a numeric variable or array element reference. If the converter finds a numeric variable or array element, it creates a CHARACTER variable of the correct size and EQUIVALENCEs it to the one specified. The result is used as the FILE= specifier in the new source statement.

Consecutive Operators

VMS FORTRAN allows two consecutive arithmetic operators when the second operator is a unary + or – . The converter inserts parentheses into such expressions so as to preserve the meaning and precedence. For example:

```
X = A ** -B
```

The VMS compiler would evaluate the `-` operator and *then* exponentiate. To get the same result in Sun FORTRAN, the expression above is converted to:

```
    X = A ** (-B)
```

Illegal `REAL` Expressions

When the converter finds a `REAL` expression where it expects an integer expression, it inserts an explicit type conversion to `INTEGER`. Examples are:

- □ Alternate `RETURN`
- □ Dimension declarators and array subscripts
- □ Substring selectors
- □ Computed `GO TO`
- □ Logical unit number, record number, and record length.

Hex and Octal Constants

Hex and octal constants are converted to Sun FORTRAN syntax. These include typeless hex, typeless octal, and integer octal constants.

Typeless Numeric Constants

Typeless numeric constants are so named because their expressions assume data types based on how they are used. However, in Sun FORTRAN such constants must be distinguished from character strings. Thus, in `DATA` statements such constants are converted to typeless hexadecimal or octal constants; elsewhere, they are converted to the type required by the context. For example, in the statements:

```
    JCOUNT = ICOUNT + '777'O
    TEMP = 'FFF99A'X
```

are converted to:

```
    JCOUNT = ICOUNT + 451
    TEMP = 2.35076E-38
```

The context defines `'777'O` as `INTEGER*2` and `'FFF99A'X` as `REAL*8`.

Octal Integer Constants

Similarly, a VMS FORTRAN octal *integer* constant is converted to its decimal form. For example:

```
    JCOUNT = ICOUNT + "703
```

is converted to:

```
   JCOUNT = ICOUNT + 511
```

The VMS FORTRAN notation "703, signals `f77cvt` to convert from the *integer* octal constant to its *integer* decimal equivalent, 451 in this case. Note that 703 cannot be the start of a character constant, because VMS FORTRAN character constants are delimited by apostrophes, not quotes.

Nonstandard Length Specifiers

The `f77cvt` program converts to standard syntax any nonstandard length specifiers in function declarations. For example, the VMS-specific syntax for the `FUNCTION` statement is:

```
[type] FUNCTION name[*m]([p[,p] ...])
```

where *m* is an unsigned, nonzero integer constant specifying the length of the data type. The syntax in Sun FORTRAN is:

```
[type] [*m] FUNCTION name([p[,p]...])
```

For example:

```
  INTEGER FUNCTION FCN*2 ( A, B, C )
```

is converted to:

```
  INTEGER*2 FUNCTION FCN ( A, B, C )
```

Old TYPE and ACCEPT Statements

The `TYPE` statement is converted into a `PRINT` statement; likewise, `ACCEPT` is converted into `READ`.

Alternate Return Arguments

The nonstandard "&" syntax for alternate-return actual arguments is converted to the standard FORTRAN "*" syntax. For example:

```
   CALL SUB(..., &label, ...)
```

is converted to:

```
   CALL SUB(..., *label, ...)
```

The ENCODE and DECODE statements

The `ENCODE` and `DECODE` statements are converted into standard internal `READ` and `WRITE` statements, respectively.

These statements require a buffer of `CHARACTER` type. If their buffer is a `CHARACTER` variable, it is used, otherwise the converter creates a `CHARACTER` variable and equivalences it to the buffer. If the buffer is a dummy argument, then it cannot be in an `EQUIVALENCE` statement, so the `ENCODE/DECODE` statement is flagged as an untranslatable feature.

Record Specifier 'N in Direct-access I/O

The VMS nonstandard record specifier 'N for direct-access I/O statements is converted to the standard REC=N. For example:

```
READ ( K ' N ) LIST
```

is converted to:

```
READ ( UNIT=K, REC=N ) LIST
```

where the logical unit number is K and the number of the record is N.

Old OPEN and INQUIRE Options

The TYPE and NAME options for both the OPEN and INQUIRE statements are converted into the standard STATUS and FILE options, respectively.

The RECORDSIZE option for OPEN is changed to RECL.

The OPEN for unformatted files Options

In Sun FORTRAN unformatted files are opened with the logical record size in *bytes*. If the VMS OPEN statement states or implies an *unformatted* file, then the new OPEN statement has its logical record size multiplied by four.

If the converter cannot determine formatted/unformatted, then it issues a warning message that the record size may need to be adjusted. This could happen if the information is passed in variable character strings.

Warning: The record size returned by an INQUIRE statement is *not* adjusted by the converter.

The DISPOSE=*p* in the CLOSE Statement

The VMS DISPOSE=*p* clause in the CLOSE statement is converted to STATUS=*p* .

Line Numbers for dbx

To make conversion debugging easier, the converter passes through line numbers from the original VMS FORTRAN program, for dbx. For example, the converter generates:

```
# line "filename"
```

The compiler and debugger both print the offending line number of the .vf source file instead of the .f or .for file into which it has been converted. That way, you can more easily find the source of the diagnostic message in the original VMS file. This conversion can be suppressed by using the **-P** converter option.

Special Intrinsic Functions

The converter processes certain intrinsic functions:

`%VAL` is converted to `%VAL`
`%LOC` is converted to `LOC`

`%REF` (*expr*) is converted to *expr*
(with a warning if *expr* is of type `CHARACTER`)

`%DESCR` is reported as an untranslatable feature.

Backslash in character string

A backslash in a character string is doubled on output, since Sun FORTRAN *does* treat backslash as an escape character.

Logical filenames in the `INCLUDE` statement

If the **`-i`** converter option is set, then the converter will translate VMS logical filenames on the `INCLUDE` statement if it finds the environment variable `LOGICALNAMEMAPPING` to define the mapping between the logical names and the UNIX pathname.

The environment variable should be set to a string with the following syntax:

> "*lname1=path1*; *lname2=path2*; ... "

where each *lname* is a logical name and each path is the pathname of a UNIX directory (without a trailing '/'). All blanks are ignored when parsing this string.

Logical names in a file name are delimited by the first ":" in the VMS filename. The converter will convert filenames of the form:

```
lname1:file to path1/file
```

For logical names, upper/lower case is significant. If a logical name is encountered on the `INCLUDE` statement which is not specified in the `LOGICALNAMEMAPPING`, the filename is used unchanged.

Conversion Samples

This section lists samples of each kind of conversion made by `f77cvt`. Some of the examples include a line just above the coding that shows column numbers; such a line is *not* part of the coding, but is there only as a guide.

VMS FORTRAN

Conversion by `f77cvt`

Embedded comment

```
DO 10 I = 1, 80 ! Find EoL
```

Preceding comment line

```
1      7
c Find EoL
       do 10 i = 1, 80
```

Debug statement

```
1
D      PRINT *, I
```

If neither **-d** nor **-D** is set:

```
1      7
c      write(unit=*, fmt=*) i
```

Debug statement

```
1
D      PRINT *, I
```

If **-d** is set:

```
1      7
       write(unit=*, fmt=*) i
```

Debug statement

```
1

D      PRINT *, I
```

If **-D** is set:

```
1      7
#ifdef DEBUG
       write(unit=*, fmt=*) i
#endif
```

`Tab-format line`

```
1       3
9 {tab} X = 0.0
```

`Standard FORTRAN line`

```
1      7
    9  x = 0.0
```

`continuation line`

```
1       3
{tab}   if ( i .GT. 0 )
{tab} 1 X = 0.0
```

`Standard FORTRAN line`

```
1      7
  if (i .gt. 0) x - 0.0
```

Initialize in declaration

```
CHARACTER*10 NAME /'Nell'/
```

Initialize in `DATA` statement

```
character*10 name
data name /'Nell'/
```

VMS FORTRAN	Conversion by f77cvt
Initialize in common block COMMON / ID / NAME, ADDR CHARACTER*10 NAME /'Nell'/ ...	Move to BLOCK DATA subprogram common / id / name, addr character name*10 ... block data v00000...00000001 common / id / name, addr character *10 name real addr data name /'Nell'/ end
Radix-50 constants DATA IOTA / 7RA SKOSH /	Sun bit-string data iota / x'472b0653'/
IMPLICIT NONE	implicit undefined (a-z)
VIRTUAL A(4,6)	dimension a(4,6)
PARAMETER FLAG1 = .TRUE.	logical flag1 parameter (flag1 = .true.)
Non-character format INTEGER X WRITE(6, X) list	Equivalenced character format character a integer x equivalence (x, a) write(6, a) list
Omitted actual argument CALL SUBX (A,,Z)	Insert zero call subx (a, 0, z)
Type REAL*16 REAL*16 X 1.0Q1	double precision x 1.0d1
File specifier, non-character INTEGER X OPEN(8, FILE=X)	Equivalenced file specifier character a integer x equivalence (x, a) open (8, file=a)
Consecutive operators X = A ** -B	Insert parentheses x = a ** (-b)

VMS FORTRAN	Conversion by f77cvt
Illegal REAL expression	Explicit type conversion

REAL R
...
RETURN R
DIMENSION A(R)
STRZ(1:R)
GO TO (30, 50, 90), R
READ(UNIT=R, ...)
READ(... REC=R, ...)
OPEN(..., RECL=R, ...)
	real r
	...
	return int(r)
	dimension a(int(r))
	strz(1:int(r))
	go to (30, 50, 90), int(r)
	read(unit=int(r), ...)
	read(..., rec=int(r), ...)
	open(..., recl=int(r), ...)

Typeless numerics in DATA	Sun syntax
DATA N1 / '777'O / DATA N2 / '1F'X /	data n1 / o'777' / data n2 / x'1f' /
Typeless numerics not in DATA	To a type, depending on context

JCOUNT = ICOUNT + '777'O
TEMP = 'FFF99A'X
	jcount = icount + 451
	temp = 2.35076e-38

Octal integer constants	Decimal
JCOUNT = ICOUNT + "703	jcount = icount + 511
Nonstandard function length	Standard function length
INTEGER FUNCTION FCN*2 (A,B)	integer*2 function fcn (a,b)
Old TYPE/ACCEPT statements	Print/Read
TYPE *, LIST ACCEPT *, LIST	PRINT *, LIST READ *, LIST
Alternate-return "&"	Alternate-return "*"
CALL SUB(A, &90)	call sub(a, *90)
ENCODE/DECODE char buffer	
CHARACTER A*32 ENCODE (32, A, 1) LIST DECODE (32, A, 1) LIST	character a*32 read (a, 1) list write(a, 1) list
ENCODE/DECODE non char buffer	
INTEGER I*4 ENCODE (16, 1, I) LIST DECODE (16, 1, I) LIST	integer i*4 character z*16 equivalence (i, z) read (z, 1) list write(z, 1) list
K'N (direct access read):	
READ (K ' N) LIST	read (unit=k, rec=n) list

VMS FORTRAN	**Conversion by f77cvt**

OPEN options
TYPE='OLD'
NAME='FFT.DATA'
RECORDSIZE=10
	
	status='OLD'
	file='FFT.DATA'
	recl=10

OPEN unformatted files OPEN(1, ACCESS='DIRECT', & RECL=20)	New RECL ← old RECL * 4 OPEN(1, ACCESS='DIRECT', & RECL=80)

INQUIRE options
TYPE=ANSWER
NAME=ANSWER
RECORDSIZE=NCPR
	
	status=answer
	file=answer
	recl=ncpr

CLOSE option DISPOSE='KEEP'	 status='KEEP'
%VAL: CALL SUBX (%VAL(A), B)	 call subx (%val(a), b)
%LOC: I = %LOC(X)	 i = loc (x)
%REF: CALL SUBX (%REF(*expr*))	 call subx (*expr*) {warns if *expr* is character}
%DESCR: CALL SUBX (%DESCR(A))	Reported as untranslatable
Backslash in string STRZ = 'ABCD\abcd'	 strz = 'ABCD\\abcd'
Logical filename in INCLUDE statement[*] INCLUDE 'vers4:common.h'	 INCLUDE projA/version/4/common.h

[1] This conversion is done if f77cvt is run with the -i option, and if the environment variable LOGICALNAMEMAPPING is set to "vers4=progA/version/4/" .

Unsupported VMS FORTRAN

Most VMS FORTRAN extensions are either incorporated into the Sun FORTRAN compiler or converted to equivalent Sun FORTRAN statements by the source code converter. This section lists the few VMS statements that remain. The converter writes diagnostic messages to standard error for any unconverted statements in the `.vf` source file and does not pass the statements on to either the `.f` or `.F` file.

The following VMS FORTRAN features are not supported in Sun FORTRAN:

- `DEFINE FILE` statement
- `DELETE` statement
- `UNLOCK` statement
- `FIND` statement
- `REWRITE` statement
- Variable `FORMAT` expressions
- A first character option of +, `0`, or `1` with the `$` edit descriptor (supports only the space option)
- `KEYID` and key specifiers in `READ` statements
- Nonstandard `INQUIRE` specifiers:

  ```
  CARRIAGECONTROL
  DEFAULTFILE
  KEYED
  ORGANIZATION
  RECORDTYPE
  ```

- Nonstandard `OPEN` specifiers:

  ```
  ASSOCIATEVARIABLE
  BLOCKSIZE
  BUFFERCOUNT
  CARRIAGECONTROL
  DEFAULTFILE
  DISP[OSE]
  EXTENDSIZE
  INITIALSIZE
  KEY
  MAXREC
  NOSPANBLOCKS
  ORGANIZATION
  READONLY
  RECORDTYPE
  SHARED
  USEROPEN
  ```

- Quadruple-precision `REAL`
- The intrinsic function `%DESCR`.

□ The INCLUDE statement

Some aspects of the INCLUDE statement are converted: see "Logical filenames in the INCLUDE statement," above. The INCLUDE statement is operating system–dependent, so it cannot be completely converted automatically. The VMS version allows a module-name and a LIST control directive that are indistinguishable from a continuation of a UNIX filename. Also, VMS ignores alphabetic case, so if the programmer was inconsistent about capitalization, distinctions would be made where none were intended.

□ Getting a long integer — expecting a short

In VMS FORTRAN you can pass a long integer argument to a subroutine that expects a short integer. This will work if the long integer fits in 16 bits, because the VAX addresses an integer by its low-order byte. This does *not* work on Sun-2, Sun-3 or Sun-4 systems.

□ An ENCODE/DECODE buffer that is a dummy argument

These statements require a buffer of CHARACTER type. If their buffer is a CHARACTER variable, it is used, otherwise the converter creates a CHARACTER variable and equivalences it to the buffer. If the buffer is a dummy argument, then it cannot be in an EQUIVALENCE statement, so the ENCODE/DECODE statement is flagged as an error.

□ The VMS system calls

10.5. The VMS Intrinsics

This section lists the VMS FORTRAN intrinsic functions recognized by f77 or converted by f77cvt, and the one function not covered by either. There are roughly 150 intrinsic functions beyond those mandated by the ANSI standard. Some functions are in more than one category. All functions take one argument, unless otherwise indicated.

Table 10-1 *Double-Precision Complex Functions Recognized by Sun FORTRAN*

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
CDABS	specific	absolute value	COMPLEX*16	REAL*8
CDEXP	specific	exponential, e^a	COMPLEX*16	COMPLEX*16
CDLOG	specific	natural log	COMPLEX*16	COMPLEX*16
CDSQRT	specific	square root	COMPLEX*16	COMPLEX*16
CDSIN	specific	sine	COMPLEX*16	COMPLEX*16
CDCOS	specific	cosine	COMPLEX*16	COMPLEX*16
DCMPLX	specific	convert to Dcomplex	any numeric	COMPLEX*16
DCONJG	specific	complex conjugate	COMPLEX*16	COMPLEX*16
DIMAG	specific	imaginary part of complex	COMPLEX*16	REAL*8
DREAL	specific	real part of complex	COMPLEX*16	REAL*8

Table 10-2 *Degree-Based Trigonometric Functions Recognized by Sun FORTRAN*

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
SIND	generic	sine	–	–
SIND	specific	sine	REAL*4	REAL*4
DSIND	specific	sine	REAL*8	REAL*8
COSD	generic	cosine	–	–
COSD	specific	cosine	REAL*4	REAL*4
DCOSD	specific	cosine	REAL*8	REAL*8
TAND	generic	tangent	–	–
TAND	specific	tangent	REAL*4	REAL*4
DTAND	specific	tangent	REAL*8	REAL*8
ASIND	generic	arc sine	–	–
ASIND	specific	arc sine	REAL*4	REAL*4
DASIND	specific	arc sine	REAL*8	REAL*8
ACOSD	generic	arc cosine	–	–
ACOSD	specific	arc cosine	REAL*4	REAL*4
DACOSD	specific	arc cosine	REAL*8	REAL*8

Table 10-2 *Degree-Based Trigonometric Functions Recognized by Sun FORTRAN— Continued*

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
ATAND	generic	arc tangent	–	–
ATAND	specific	arc tangent	REAL*4	REAL*4
DATAND	specific	arc tangent	REAL*8	REAL*8
ATAN2D	generic	arc tangent of a1/a2	–	–
ATAN2D	specific	arc tangent of a1/a2	REAL*4	REAL*4
DATAN2D	specific	arc tangent of a1/a2	REAL*8	REAL*8

Table 10-3 *Bit-Manipulation Functions Recognized by Sun FORTRAN*

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
IBITS	generic	from a1, initial bit a2, extract a3 bits	–	–
IIBITS	specific	from a1, initial bit a2, extract a3 bits	INTEGER*2	INTEGER*2
JIBITS	specific	from a1, initial bit a2, extract a3 bits	INTEGER*4	INTEGER*4
ISHFT	generic	shift a1 logically by a2 bits[*]	–	–
ISHFTC	generic	in a1, circular shift by a2 places, of right a3 bits	–	–
IISHFTC	specific	in a1, circular shift by a2 places, of right a3 bits	INTEGER*2	INTEGER*2
JISHFTC	specific	in a1, circular shift by a2 places, of right a3 bits	INTEGER*4	INTEGER*4

The following are translated by f77cvt. See comments about multiple integer sizes for Table 7.

Table 10-4 *Bit-Manipulation Functions Converted by* f77cvt

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*	*Translation*
IAND	generic	bitwise AND of a1, a2	–	–	AND
IIAND	specific	bitwise AND of a1, a2	INTEGER*2	INTEGER*2	AND
JIAND	specific	bitwise AND of a1, a2	INTEGER*4	INTEGER*4	AND
IOR	generic	bitwise OR of a1, a2	–	–	OR
IIOR	specific	bitwise OR of a1, a2	INTEGER*2	INTEGER*2	OR
JIOR	specific	bitwise OR of a1, a2	INTEGER*4	INTEGER*4	OR
IEOR	generic	bitwise exclusive OR of a1, a2	–	–	XOR
IIEOR	specific	bitwise exclusive OR of a1, a2	INTEGER*2	INTEGER*2	XOR
JIEOR	specific	bitwise exclusive OR of a1, a2	INTEGER*4	INTEGER*4	XOR

[*] ISHFT — If *a2* is positive, then logical shift left, is negative, then logical shift right.

Table 10-4 *Bit-Manipulation Functions Converted by* f77cvt*— Continued*

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*	*Translation*
NOT **INOT** **JNOT**	generic specific specific	bitwise complement bitwise complement bitwise complement	– INTEGER*2 INTEGER*4	– INTEGER*2 INTEGER*4	NOT NOT NOT
IISHFT **JISHFT**	specific specific	shift a1 logically left by a2 bits shift a1 logically left by a2 bits	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4	ISHIFT ISHIFT
IBSET **IIBSET** **JIBSET**	generic specific specific	in a1, set bit a2 to 1 in a1, set bit a2 to 1; return new a1 in a1, set bit a2 to 1; return new a1	– INTEGER*2 INTEGER*4	– INTEGER*2 INTEGER*4	BIS BIS BIS
BTEST **BITEST** **BJTEST**	generic specific specific	if bit a2 of a1 is 1, return .TRUE. if bit a2 of a1 is 1, return .TRUE. if bit a2 of a1 is 1, return .TRUE.	– INTEGER*2 INTEGER*4	– LOGICAL*2 LOGICAL*4	BIT BIT BIT
IBCLR **IIBCLR** **JIBCLR**	generic specific specific	in a1, set bit a2 to 0; return new a1 in a1, set bit a2 to 0; return new a1 in a1, set bit a2 to 0; return new a1	– INTEGER*2 INTEGER*4	– INTEGER*2 INTEGER*4	BIC BIC BIC

Table 10-5 *Quad-Precision Real Functions Converted by* f77cvt

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*	*Translation*
QSQRT **QLOG** **QLOG10** **QEXP**	specific specific specific specific	square root natural log common log exponential, e^a	REAL*16 REAL*16 REAL*16 REAL*16	REAL*16 REAL*16 REAL*16 REAL*16	DSQRT DLOG DLOG10 DEXP
QSIN **QSIND** **QCOS** **QCOSD** **QTAN** **QTAND**	specific specific specific specific specific specific	sine sine (degrees) cosine cosine (degrees) tangent tangent (degrees)	REAL*16 REAL*16 REAL*16 REAL*16 REAL*16 REAL*16	REAL*16 REAL*16 REAL*16 REAL*16 REAL*16 REAL*16	DSIN DSIND DCOS DCOSD DTAN DTAND
QASIN **QASIND** **QACOS** **QACOSD**	specific specific specific specific	arc sine arc sine arc cosine arc cosine	REAL*16 REAL*16 REAL*16 REAL*16	REAL*16 REAL*16 REAL*16 REAL*16	DASIN DASIND DACOS DACOSD

Table 10-5 *Quad-Precision Real Functions Converted by* f77cvt*— Continued*

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*	*Translation*
QATAN **QATAND**	specific specific	arc tangent arc tangent	REAL*16 REAL*16	REAL*16 REAL*16	DATAN DATAND
QATAN2 **QATAN2D**	specific specific	arc tangent of a1/a2 arc tangent of a1/a2	REAL*16 REAL*16	REAL*16 REAL*16	DATAN2 DATAN2D
QSINH **QCOSH** **QTANH**	specific specific specific	hyperbolic sine hyperbolic cosine hyperbolic tangent	REAL*16 REAL*16 REAL*16	REAL*16 REAL*16 REAL*16	DSINH DCOSH DTANH
QABS **QINT** **QNINT**	specific specific specific	absolute value truncation toward zero nearest integer, INT(a+.5*sign(a))	REAL*16 REAL*16 REAL*16	REAL*16 REAL*16 REAL*16	DABS DINT DNINT
SNGLQ **DBLEQ** **QEXT** **QEXT** **QEXTD** **QFLOAT**	specific specific generic specific specific specific	convert to REAL*4 convert to REAL*8 convert to REAL*16 convert to REAL*16 convert to REAL*16 convert to REAL*16	REAL*16 REAL*16 any numeric REAL*4 REAL*8 any integer	REAL*4 REAL*8 REAL*16 REAL*16 REAL*16 REAL*16	SNGL DBLE DBLE DBLE DBLE DBLE
QMAX1 **QMIN1** **QDIM** **QMOD** **QSIGN**	specific specific specific specific specific	maximum* minimum* positive difference† remainder of a1/a2 transfer sign, |a1|*sign(a2)	REAL*16 REAL*16 REAL*16 REAL*16 REAL*16	REAL*16 REAL*16 REAL*16 REAL*16 REAL*16	DMAX1 DMIN1 DDIM DMOD DSIGN

* QMAX1, QMIN1: at least two arguments.

† positive difference: a1-min(a1,a2))

The possibility of multiple integer types is not addressed by the standard. Sun FORTRAN copes with their existence by treating a specific INTEGER → INTEGER function name (IABS, etc.) as a special sort of generic: the argument type is used to select the appropriate runtime routine name, which is not accessible to the programmer. VMS FORTRAN takes a similar approach but makes the specific names available. The f77cvt program converts the VMS specific name back into the corresponding pseudo-generic name.

Table 10-6 *Integer Functions Converted by* f77cvt

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*	*Translation*
IIABS	specific	absolute value	INTEGER*2	INTEGER*2	IABS
JIABS	specific	absolute value	INTEGER*4	INTEGER*4	IABS
IMAX0	specific	maximum*	INTEGER*2	INTEGER*2	MAX0
JMAX0	specific	maximum*	INTEGER*4	INTEGER*4	MAX0
IMIN0	specific	minimum*	INTEGER*2	INTEGER*2	MIN0
JMIN0	specific	minimum*	INTEGER*4	INTEGER*4	MIN0
IIDIM	specific	positive difference†	INTEGER*2	INTEGER*2	IDIM
JIDIM	specific	positive difference†	INTEGER*4	INTEGER*4	IDIM
IMOD	specific	remainder of a1/a2	INTEGER*2	INTEGER*2	MOD
JMOD	specific	remainder of a1/a2	INTEGER*4	INTEGER*4	MOD
IISIGN	specific	transfer sign, |a1|*sign(a2)	INTEGER*2	INTEGER*2	ISIGN
JISIGN	specific	transfer sign, |a1|*sign(a2)	INTEGER*4	INTEGER*4	ISIGN

Some VMS FORTRAN functions coerce to a particular INTEGER type. Sun FORTRAN always coerces to the prevailing INTEGER type. The converter recognizes such functions, converts each to its generic name, and issues a warning.

Table 10-7 *Converted Functions that VMS Coerces to a Particular Type*

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*	*Translation*
IINT	specific	truncation toward zero	REAL*4	INTEGER*2	INT
JINT	specific	truncation toward zero	REAL*4	INTEGER*4	INT
IIDINT	specific	truncation toward zero	REAL*8	INTEGER*2	IDINT
JIDINT	specific	truncation toward zero	REAL*8	INTEGER*4	IDINT
IQINT	generic	truncation toward zero	REAL*16	INTEGER	IDINT
IIQINT	specific	truncation toward zero	REAL*16	INTEGER*2	IDINT
JIQINT	specific	truncation toward zero	REAL*16	INTEGER*4	IDINT

* At least two arguments

† positive difference: a1-min(a1,a2))

Table 10-7 *Converted Functions that VMS Coerces to a Particular Type— Continued*

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*	*Translation*
ININT **JNINT**	specific specific	nearest integer, INT(a+.5*sign(a)) nearest integer, INT(a+.5*sign(a))	REAL*4 REAL*4	INTEGER*2 INTEGER*4	NINT NINT
IIDNNT **JIDNNT**	specific specific	nearest integer, INT(a+.5*sign(a)) nearest integer, INT(a+.5*sign(a))	REAL*8 REAL*8	INTEGER*2 INTEGER*4	IDNINT IDNINT
IQNINT **IIQNNT** **JIQNNT**	generic specific specific	nearest integer, INT(a+.5*sign(a)) nearest integer, INT(a+.5*sign(a)) nearest integer, INT(a+.5*sign(a))	REAL*16 REAL*16 REAL*16	INTEGER INTEGER*2 INTEGER*4	IDNINT IDNINT IDNINT
IIFIX **JIFIX**	specific specific	fix fix	REAL*4 REAL*4	INTEGER*2 INTEGER*4	IFIX IFIX
IMAX1 **JMAX1**	specific specific	maximum maximum	REAL*4 REAL*4	INTEGER*2 INTEGER*4	MAX1 MAX1
IMIN1 **JMIN1**	specific specific	minimum minimum	READ*4 READ*4	INTEGER*2 INTEGER*4	MIN1 MIN1

In other cases, each VMS specific name is converted into a Sun generic name.

Table 10-8 *Other Conversions by* f77cvt

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*	*Translation*
FLOATI **FLOATJ** **DFLOAT** **DFLOTI** **DFLOTJ**	specific specific generic specific specific	convert to REAL*4 convert to REAL*4 convert to REAL*8 convert to REAL*8 convert to REAL*8	INTEGER*2 INTEGER*4 INTEGER INTEGER*2 INTEGER*4	REAL*4 REAL*4 REAL*8 REAL*8 REAL*8	REAL REAL DBLE DBLE DBLE
AIMAX0 **AJMAX0**	specific specific	maximum maximum	INTEGER*2 INTEGER*4	REAL*4 REAL*4	AMAX0 AMAX0
AIMIN0 **AJMIN0**	specific specific	minimum minimum	INTEGER*2 INTEGER*4	REAL*4 REAL*4	AMIN0 AMIN0

The zero-extend functions are *not* recognized by Sun FORTRAN and are *not* converted by `f77cvt`.

Table 10-9 *Zero-Extend Functions*

Name	*Gen/Spec*	*Function*	*Arg Type*	*Action*
`ZEXT`	generic	zero-extend	–	error message
`IZEXT`	specific	zero-extend	short log or int	error message
`JZEXT`	specific	zero-extend	log or int	error message

11

The C—FORTRAN Interface

11

The C—FORTRAN Interface

This chapter describes conventions for interfacing C and FORTRAN. It is intended as a guide to programmers who want to use FORTRAN with modules written in C. It assumes more sophisticated knowledge of FORTRAN and C than do most of the other parts of this manual. The Pascal—FORTRAN interface is covered in comparable chapters of the *Pascal Programmer's Guide*. The Modula-2—FORTRAN interface is similar to the Pascal—FORTRAN interface.

11.1. Command Line Arguments

The function IARGC (3F) returns the number of command-line arguments, and GETARG (3F) copies an argument into a variable in the program. For example:

```
      CHARACTER ARG*70
c         How many command-line arguments ?
      NARGS = IARGC()
c         Get an argument and write it out.
      DO 10 I = 1, NARGS
          CALL GETARG( I, ARG )
          PRINT '(A)', ARG
10    CONTINUE
      END
```

This loops through the parameter list copying a parameter into ARG and then writing it to standard output. Since ARG is only 70 characters long, any longer parameter is truncated. If the above program is compiled as myecho, you can test it as follows:

```
demo% myecho this is a sample
this
is
a
sample
demo% myecho *
calc.f
mycat.f
myecho
myecho.f
demo% █
```

11.2. Exiting with status

The subroutine EXIT() can set the shell status variable to indicate whether the program was successful or not. The default is that status is set to zero. The following statement:

```
   CALL EXIT( 8 )
```

sets status to 8, then terminates execution of the program. The current value of status can be displayed as follows:

```
demo% echo $status
1
demo% █
```

The **echo** command sets status back to zero after showing its value. The value of status can be tested in shell scripts.

The ABORT routine can terminate a program, set status to 138 to dump memory to the file core, and print a message on standard error:

```
   CALL ABORT(" sample error message")
```

and causes a program to terminate after writing out:

```
   abort: sample error message
   bus error (core dumped)
```

11.3. Interprocedure Interface

To write C procedures that call or are called by FORTRAN procedures, you must know the conventions for procedure names, data representation, return values, and argument lists that both languages use.

Procedure Names

The FORTRAN compiler appends an underscore to the name of a common block or procedure to distinguish it from C procedures or external variables with the same user-assigned name. If the name has exactly 32 characters, the underscore is not appended. All FORTRAN library procedure names have embedded underscores to reduce clashes with user-assigned subroutine names.

Data Representations

This table summarizes corresponding FORTRAN and C declarations:

Table 11-1 *FORTRAN and C Declarations*

In standard FORTRAN, variables of type INTEGER, LOGICAL, and REAL occupy the same amount of memory. Since LOGICAL*1 or BYTE violate such rules, they are not standard FORTRAN and can result in nonportable programs.

FORTRAN	C
`INTEGER*2 X`	`short int x;`
`INTEGER X`	`long int x;`
`LOGICAL X`	`long int x;`
`BYTE X or LOGICAL*1 X`	`char x;`
`REAL X`	`float x;`
`DOUBLE PRECISION X`	`double x;`
`COMPLEX X`	`struct { float r, i; } x;`
`DOUBLE COMPLEX X`	`struct { double dr, di; } x;`
`CHARACTER*6 X`	`char x[6];`

Return Values

A FORTRAN function of type INTEGER, LOGICAL, REAL, or DOUBLE PRECISION is equivalent to (as far as returning values is concerned) a C function that returns the corresponding type.

Functions

The FORTRAN function:

```
COMPLEX FUNCTION F( ... )
```

is equivalent to the C function:

A COMPLEX or DOUBLE COMPLEX function is equivalent to a C routine having an additional initial argument that points to the return value storage location.

```
f_( temp, ... )
struct { float r, i; } *temp;
  ...
```

And FORTRAN:

```
CHARACTER*15 FUNCTION G( ... )
```

is equivalent to C:

A character-valued FORTRAN function is equivalent to a C function with two extra initial arguments: data address and length.

```
g_( result, length, ... )
char result[ ];
long int length;
  ...
```

and could be invoked in C with:

Since every character argument in the list is associated with an additional argument giving the string's length, such FORTRAN strings need not terminate with a null character, as required by C.

Return a `float`

If `MAIN` declares `REPEAT` as an `INTEGER`, `LOGICAL`, `REAL`, or `DOUBLE PRECISION` function, then the two initial arguments would not be present, so the return value could be passed back to the FORTRAN program with a `return` statement. In the current implementation of the C compiler it is impossible to return a `float`, since the language requires it be promoted to a `double` whenever it is used in an expression and the value in a `return` statement is an expression. To construct a C function that returns a FORTRAN `REAL` it is necessary to use a trick as is illustrated below. The function `incr_` is FORTRAN callable and returns a `REAL` value one greater than its `REAL` argument.

`incr.c`:

```
int
/* returns a single-precision floating-point value */
incr_( float_ptr )
float *float_ptr;
{
    float f;

    f = *float_ptr;
    f ++;
    return *((int*)&f);
}
```

Thus, the FORTRAN program below:

`testincr.f`:

```
REAL INCR
PRINT *, INCR( 1. )
END
```

prints `2.000000` on standard output:

```
demo% f77 testincr.f incr.c
testincr.f:
testincr.f:
 MAIN:
incr.c:
Linking:
demo% a.out
   2.000000
demo% █
```

Calling FORTRAN from C

The following example illustrates a C program that calls a FORTRAN function.

main.c:

```
#include <stdio.h>

main()
{
    char string[100], repeat_val[50];
    int repeat_(), repeat_len, i, count;

    repeat_len = sizeof(repeat_val);
    count = 10;
    repeat_(repeat_val, repeat_len, "*",
        &count, sizeof("*")-1);

    strncpy(string,repeat_val,repeat_len);
    for(i=repeat_len;i<100;i++) {
        repeat_val[i] = ' ';
    }
    printf( "%s\n", repeat_val );
}
```

repeat.f:

```
    FUNCTION REPEAT( C, N )
    CHARACTER REPEAT*(*), C*(*)
    IF ( N.GT. LEN(REPEAT)) THEN
        WRITE( 0, '(A)') 'Repeat count is too large'
        N = LEN(REPEAT)
    ENDIF
    REPEAT = ''
    DO 10 I = 1, N
10  REPEAT(I:I) = C(1:1)
    RETURN
    END
```

This program can be compiled with the **cc** command as indicated:

```
demo% cc main.c repeat.f -lF77 -lI77 -lU77 -lc -lm
```

The observations made above now apply in reverse. The caller must set up more actual arguments than are apparent as formal parameters to the FORTRAN function. Arguments that are not lengths of character strings must be passed by address. The two statements following the call to `repeat` are equivalent to the work done by the character assignment statement in `repeat.f`.

Note that the FORTRAN function attempts to reference the `stderr` stream (unit 0). Before a FORTRAN program starts, the FORTRAN I/O library is initialized to connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively. In this example, the initialization does not occur since execution begins with the C `main`. Thus output is written to a file named `fort.0` instead of to the `stderr` stream.

The C program should initialize I/O by inserting the following line at the start of the program:

```
call f_init()
```

This establishes the preconnection of units 0, 5, and 6. At the end of the program, you can insert:

```
call f_exit()
```

although it may not be strictly necessary.

With a C main program, some of the routines in the FORTRAN library may not work correctly. In particular, the `signal`() routine, the `getarg`() rroutine, and the `iargc`() function.

Sharing Input/Output Streams

A C function called from a FORTRAN program must take the FORTRAN I/O environment into consideration to perform I/O on open file descriptors. The FORTRAN I/O library is implemented largely on top of the C standard I/O library. Every open unit in a FORTRAN program has an associated standard I/O file structure. For the `stdin`, `stdout`, and `stderr` streams, the file structure need not be explicitly referenced, so it is easy to share these streams between a FORTRAN program and a C function (as illustrated in the first example).

It is more difficult to share a stream that a FORTRAN program explicitly opens, since there is no way to obtain and pass the file structure. One possible solution that allows shared writing is to call `FLUSH` (3F) to empty the stream associated with a unit, and then to call `GETFD` (3F) to obtain the UNIX file descriptor associated with that unit number. This file descriptor can then be passed to the C function, which can use it as an argument to `write`(2) calls.

File Descriptors and `stdio`

In almost every discussion of input and output in FORTRAN programs, I/O channels are in terms of FORTRAN unit numbers. The I/O system does not actually deal with these units, but with SunOS *file descriptors.** The FORTRAN runtime system always translates from one to the other, so most FORTRAN programs don't have to know about file descriptors.

In addition to FORTRAN units and SunOS file descriptors, many C programs use a set of subroutines called *standard I/O* (or `stdio`). Many of the functions of the FORTRAN I/O system are implemented using standard I/O, which in turn is implemented using the SunOS I/O system calls. Some of the characteristics of these systems are listed in Table 11-2 .

[1] The information in this section is of interest mostly to users writing C routines that interface to FORTRAN programs. More about this is covered in Chapter 11, *The C—FORTRAN Interface.*

Table 11-2 *Characteristics of Three I/O Systems*

	FORTRAN Units	Standard I/O File Pointers	SunOS File Descriptors
Files Open	opened for reading and writing	opened for reading; or opened for writing; or opened for both; or opened for appending see `OPEN`(3S)	opened for reading; or opened for writing; or opened for both
Attributes	formatted or unformatted	always unformatted, but can be read or written with format-interpreting routines	always unformatted
Access	direct or sequential	direct access if the physical file representation is direct access, but can always be read sequentially	direct access if the physical file representation is direct access, but can always be read sequentially
Structure	record	character stream	character stream
Form	arbitrary, nonnegative integers	pointers to structures in the user's address space	integers from 0-63

File permissions

In C, programmers traditionally open input files for reading and output files for writing. Sometimes files are opened for both operations since SunOS lets you open files with reading and/or writing permissions assigned to the owner or others. In FORTRAN it's not possible for the system to foresee what use you will make of the file since there's no parameter to the `OPEN` statement that gives that information. Thus, FORTRAN always attempts to `OPEN` a file with the maximum permissions possible: first for both reading and writing, then for each separately. This occurs transparently and should only be of concern if you try to perform a `READ`, `WRITE`, or `ENDFILE` that you don't have permission for. Magnetic tape operations are an exception to this general freedom, since you could have write permission on a file while not having a write ring on the tape.

12

Ratfor — A FORTRAN Preprocessor

12

Ratfor — A FORTRAN Preprocessor

12.1. Overview

Since Ratfor was designed, the new FORTRAN 77 language has appeared. FORTRAN 77 provides some of the control structures that were the major reasons for Ratfor's existence and so Ratfor might not be as appropriate in the Sun system (which supports FORTRAN 77) but is still useful for porting programs written in it to Sun workstations.

FORTRAN has the advantages of universality and relative efficiency. The Ratfor language attempts to conceal the main deficiencies of FORTRAN 66 while retaining its desirable qualities by providing decent control flow statements. Ratfor features include:

statement grouping
using { and } in the style of C

decision making
via `if-else` and `switch` statements

looping constructs
using `while`, `for`, `do`, and `repeat-until` statements

controlled exits from loops
using `break` and `next` statements

free-form input
multiple statements per line and automatic continuation

unobtrusive comments
signaled by a # sign anywhere on the line

translation
of >, >=, etc., into .GT., .GE., etc.

return (*expression*)
statement for functions

symbolic parameters
via the `define` statement

source file inclusion
via the `include` statement

Ratfor is implemented as a preprocessor that translates this language into FORTRAN 66.

Once the control flow and cosmetic deficiencies of FORTRAN are hidden, the resulting language is remarkably pleasant to use. *Ratfor* programs are markedly easier to read, write, debug, maintain, and modify than their FORTRAN 66 equivalents.

You can easily write *Ratfor* programs that are portable to other environments. *Ratfor* itself is written in this way, making it portable; versions of *Ratfor* are now available on at least two dozen different types of computers at over 500 locations.

This chapter discusses design criteria for a FORTRAN preprocessor, the Ratfor language and its implementation, and user experience.

12.2. Introduction

FORTRAN is often chosen, since it is frequently the only language supported on a local computer. It is the closest thing to a universal programming language currently available — with care you can write large, truly portable FORTRAN 66 programs. Finally, FORTRAN 66 is often the most 'efficient' language available, particularly for programs requiring much computation.

But FORTRAN can be unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops, which express the logic of the program. The conditional statements in FORTRAN are primitive. The arithmetic `if` forces the user into at least two statement numbers and two (implied) `goto`'s; it leads to unintelligible code. The logical `if` is better in that the test part can be stated clearly, but is hopelessly restrictive because only one FORTRAN statement can follow the `if` statement. And of course there can be no ELSE part to a FORTRAN `if` — you can't specify an alternative action if the `if` is not satisfied.

The FORTRAN `do` restricts the user to going forward in an arithmetic progression. It is fine for '1 to N in steps of 1 (or 2 or ...)', but there is no direct way to go backwards, or even (in ANSI FORTRAN) to go from 1 to N–1. The `do` is also useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that FORTRAN programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

Ratfor defines a new language that overcomes these deficiencies, and translates it into the unpleasant one with a preprocessor. The preprocessor idea is not new. A recent listing shows more than 50 preprocessors, at least half a dozen of which are widely available.

[1] This chapter is a revised and expanded version of a paper published in *Software — Practice and Experience*, October 1975.

Using the `ratfor` translator

`ratfor` is the basic translator; it takes either a list of file names or the standard input and writes FORTRAN on the standard output. Options include **-6***x*, which causes the character given for *x* to be used as a continuation character in column 6 (UNIX uses **&** in column 1), and **-C**, which copies *Ratfor* comments into the generated FORTRAN. For example:

```
demo% ratfor [ options ] file ...
```

compiles the specified files. Files with names ending in **.r** are *Ratfor* source; other files are assumed to be for the loader. The flags **-C** and **-6***x* described above are recognized, as are

–c compile without loading

–f save intermediate FORTRAN **.f** files

–r Ratfor only; implies **–c** and **-f**

–U flag undeclared variables (not universally available). Other flags are passed on to the loader.

Using the `f77` compiler

The **f77** command accepts **.r** files and passes them to the **ratfor** preprocessor before it compiles them. The **-m4** option to **f77** applies the M4 macro processor to each **.r** file before transforming it with the Ratfor preprocessor. For example:

```
demo% f77 mymain.f sub1.r sub2.r sub3.f
```

12.3. Language Description

Design

The language is the same as standard FORTRAN 66 except for two aspects. First, since control flow is central to any program regardless of the specific application, the primary task of *Ratfor* is to conceal this part of FORTRAN from the user by providing decent control flow structures. These structures are sufficient and comfortable for structured programming without `goto`'s. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the 'cosmetic' deficiencies of FORTRAN, to provide a language that is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — *Ratfor* does nothing about the host of other weaknesses of FORTRAN 66. Although it would be straightforward to extend it to provide character strings, they are not needed by everyone, and the preprocessor would be harder to implement. Throughout, the design principle used has been that *Ratfor doesn't know any FORTRAN*. Any language feature requiring that *Ratfor* really understand FORTRAN has been omitted.

The rest of this chapter contains an informal description of the *Ratfor* language. The control flow aspects and cosmetic changes will look familiar if you are used to languages such as Algol, PL/I, and Pascal.

Statement grouping

FORTRAN 66 provides no way to group statements together, short of making them into a subroutine. The standard construction 'if a condition is true, do this group of things,' for example,

```
if (x > 100)
    { call error("x>100"); err = 1; return }
```

can't be written directly in FORTRAN. Instead a programmer is forced to translate this relatively clear thought into murky FORTRAN, by stating the negative condition and branching around the group of statements:

```
      if (x .le. 100) goto 10
          call error(5hx>100)
          err = 1
          return
10    ...
```

When the program doesn't work or must be modified, it must be translated back into a clearer form before you can be sure what it's doing.

Ratfor eliminates this error-prone and confusing back and forth translation; the first form is the way the computation is written in *Ratfor*. A group of statements can be treated as a unit by enclosing them in braces { and }. This is true throughout the language — wherever a single *Ratfor* statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than `begin` and `end`, `do` and `end`.

Cosmetics contribute to the readability of code. The character '>' is clearer than '.GT.', so *Ratfor* translates it appropriately. Although many FORTRAN compilers permit character strings in quotes (such as `"x>100"`), they are not allowed in ANSI FORTRAN, so *Ratfor* converts quoted strings into the right number of *n*H's: computers count better than people do.

Ratfor is a free-form language — statements can appear anywhere on a line, and several can appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
    call error("x>100")
    err = 1
    return
}
```

In this case, no semicolon is needed at the end of each line, since Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the `if` is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
    write(6, 20) y, z
```

No continuation is needed here because the statement on the first line is clearly continued on the second. In general *Ratfor* continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language allows freedom in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital to make the logical structure of the program clear.

The `else` clause

Ratfor provides an `else` statement to handle the construction 'if a condition is true, do this, *otherwise* do that.'

```
if (a <= b)
    { sw = 0; write(6, 1) a, b }
else
    { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of `a` and `b`, then the larger, and sets `sw` appropriately.

The FORTRAN equivalent of this code is circuitous indeed:

```
    if (a .gt. b) goto 10
        sw = 0
        write(6, 1) a, b
        goto 20
10  sw = 1
    write(6, 1) b, a
20  ...
```

This is a mechanical translation, so shorter forms exist but all translations suffer from the same problem: they are less clear and understandable than untranslated code. To understand the FORTRAN version, you must scan the entire program to make sure that no other statement branches to statements 10 or 20 before you know that this is an `if-else` construction. With the *Ratfor* version, there is no question about how you get to the parts of the statement, since the `if-else` is a single unit that can be read, understood, or ignored as required.

As mentioned before, if the statement following an `if` or an `else` is a single statement, then no braces are needed:

```
if (a <= b)
    sw = 0
else
    sw = 1
```

The syntax of the `if` statement is

```
if (legal FORTRAN condition)
     Ratfor statement
else
     Ratfor statement
```

where the `else` part is optional. The *legal* FORTRAN *condition* is anything that can legally go into a FORTRAN Logical `if`. *Ratfor* does not check this clause, since it does not know enough FORTRAN to know what is permitted. The *Ratfor statement* is any *Ratfor* FORTRAN statement, or a collection of them surrounded by braces.

Nested `if´s`

Since the statement that follows an `if` or an `else` can be any *Ratfor* statement, it is possible for another `if` or `else` to follow it. As a useful example, consider this problem: the variable `f` is to be set to –1 if `x` is less than zero, to +1 if `x` is greater than 100, and to 0 otherwise. In *Ratfor*, you would write

```
if (x < 0)
     f = -1
else if (x > 100)
     f = +1
else
     f = 0
```

Here the statement after the first `else` is another `if-else`. Logically it is just a single statement, although it is rather complicated.

Any version written in straight FORTRAN is necessarily indirect because FORTRAN does not let you say what you mean.

Following an `else` with an `if` is one way to write a multi-way branch in *Ratfor*. In general, the structure

```
if (...)
     - - -
else if (...)
     - - -
else if (...)
     - - -
   ...
else
     - - -
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a `switch` statement that does the same job in certain special cases; in more general situations, you must make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is satisfied. The code associated with this condition is executed, and then the entire structure is exited. The trailing `else` part handles the 'default' case, where none of the other conditions apply. If there is no default action, this final `else` part is omitted:

```
if (x < 0)
    x = 0
else if (x > 100)
    x = 100
```

Ambiguity in `if-else`

There is one thing to notice about complicated structures involving nested `if`'s and `else's`. Consider

```
if (x > 0) if (y > 0)
        write(6, 1) x, y
       else
        write(6, 2) y
```

There are two `if`'s and only one `else`, so you don't know which `if` goes with the `else`.

This is a genuine ambiguity in *Ratfor*. The ambiguity is resolved by saying that in such cases the `else` goes with the closest previous `else'ed` un- `if`. In this case, the `else` goes with the inner `if`, as is indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces. In the case above, you would write

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y
}
```

which does not change the meaning but leaves no doubt in the reader's mind. If you want the other association, you *must write*

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
}
else
    write(6, 2) y
```

The switch statement

The switch statement provides a clean way to express multi-way branches that branch on the value of some integer-valued expression. The syntax is

```
switch (expression) {

    case expr1 :
        statements
    case expr2, expr3 :
        statements
    ...
    default:
        statements
}
```

Each case is followed by a list of comma-separated integer expressions. The *expression* following switch is compared against the case expressions *expr1*, *expr2*, and so on in turn until one matches, at which time the statements following that case are executed. If no case matches *expression*, and there is a default section, the statements in it are executed; if there is no default, nothing is done. In all situations, as soon as some block of statements is executed, the entire switch is exited immediately. (Readers familiar with C should beware that this behavior is not the same as the C switch.)

The do statement

The do statement in *Ratfor* is quite similar to the do statement in FORTRAN, except that it uses no statement number. The statement number, serves only to mark the end of the do, and this can be done just as easily with braces. Thus

```
    do i = 1, n {
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
    }
```

is the same as

```
    do 10 i = 1, n
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
10  continue
```

The syntax is:

```
do legal-FORTRAN-DO-text
    Ratfor statement
```

The part that follows the keyword do has to be something that can legally go into a FORTRAN do statement. Thus, if a local version of FORTRAN allows do limits to be expressions (which is not permitted in ANSI FORTRAN 66), they can be used in a Ratfor do .

The *Ratfor statement* part is often enclosed in braces, but like the if, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
    x(i) = 0.0
```

A slightly more complicated routine,

```
do i = 1, n
    do j = 1, n
        m(i, j) = 0
```

sets the entire array m to zero.

```
do i = 1, n
    do j = 1, n
        if (i < j)
            m(i, j) = -1
        else if (i == j)
            m(i, j) = 0
        else
            m(i, j) = +1
```

sets the upper triangle of m to −1, the diagonal to zero, and the lower triangle to +1. (The operator == is 'equals', that is, '.EQ.'.) In each case, the statement that follows the do is logically a *single* statement, even though complicated, and thus needs no braces.

Using break and next

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. break causes an immediate exit from the do; in effect it is a branch to the statement *after* the do. next is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
    if (x(i) < 0.0)
        next
    process positive element
}
```

break and next also work in the other *Ratfor* looping constructions which are discussed in the next few sections.

break and next can be followed by an integer that indicates the level to break or iterate the enclosing loop; thus,

```
break 2
```

exits from two levels of enclosing loops, and `break 1` is equivalent to `break`. `next 2` iterates the second enclosing loop. (Realistically, multi-level `break`'s and `next`'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The `while` statement

One of the problems with the FORTRAN 66 `do` statement is that it generally must be done at least once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

it is typically done once with `I` set to 2, even though common sense suggests that perhaps it shouldn't be. Of course a *Ratfor* `do` can easily be preceded by a test such as

```
if (j <= k)
    do i = j, k  {
        ---
    }
```

but is often overlooked by programmers.

A more serious problem with the `do` statement is that it encourages a program to be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be adjusted to fit the requirements imposed by the FORTRAN `do`, it is that much harder to write and understand.

To overcome these difficulties, *Ratfor* provides a `while` statement, which is simply a loop: 'while some condition is true, repeat this group of statements.' It has no preconceptions about why looping is happening. For example, the routine to compute sin(x) using the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
    # returns sin(x) to accuracy e, by
    # sin(x) = x - x**3/3! + x**5/5! - ...

    sin = x
    term = x

    i = 3
    while (abs(term)>e & i<100) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
        i = i + 2
    }

    return
    end
```

Notice that if the routine is entered with `term` already smaller than `e`, the loop is done *zero times*, that is, no attempt is made to compute `x**3`; thus, a potential underflow is avoided. Since the test is made at the top of a `while` loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test `i<100` is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character '#' in a line marks the beginning of a comment. Comments and code can coexist on the same line, which is not possible with FORTRAN's 'C in column 1' convention. Blank lines are also permitted anywhere (they are not in FORTRAN 66) to emphasize the natural divisions of a program.

The syntax of the `while` statement is

```
while (legal FORTRAN condition)
    Ratfor statement
```

As with `if`, *legal* FORTRAN *condition* is something that can go into a FORTRAN logical `if`, and *Ratfor statement* is a single statement or multiple statements in braces.

The `while` encourages a style of coding not normally practiced by FORTRAN programmers. For example, suppose `nextch` is a function that returns the next input character both as a function value and in its argument. Then a loop to find the first nonblank character is

```
while (nextch(ich) == iblank)
    ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the `while`; if it were not present, the `while` would control the next statement. When the loop is exited, `ich` contains the first nonblank. Of course the same code can be written in FORTRAN as

```
100 if (nextch(ich) .eq. iblank) goto 100
```

but many FORTRAN programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The `for` statement

The `for` statement is another *Ratfor* loop, which attempts to carry the separation of loop body from reason-for-looping a step further than the `while`. A `for` statement allows explicit initialization and increment steps as part of the statement. For example, a `do` loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
     ...
     i = i + 1
}
```

Initializing and incrementing `i` has been moved into the `for` statement, making it easier to see at a glance what controls the loop.

The `for` and `while` versions have the advantage that they are done zero times if `n` is less than 1; this is not true of the `do`.

The loop of the sine routine in the previous section can be rewritten with a `for` as

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
     term = -term * x**2 / float(i*(i-1))
     sin = sin + term
}
```

The syntax of the `for` statement is

```
for ( init ; condition ; increment )
     Ratfor statement
```

init is any single FORTRAN statement, which gets done once before the loop begins. *increment* is any single FORTRAN statement that gets done at the end of each pass through the loop before the test. *condition* is anything that is legal in a logical `if`. Any of *init*, *condition*, and *increment* can be omitted, although the semicolons *must* always be present. A nonexistent *condition* is treated as always true, so "`for(;;)`" is an infinite repeat. (But see the `repeat-until` in the next section.)

The for statement is particularly useful for such things as backward loops, chaining along lists, and loops that might be done zero times, which are hard to express with a do statement as well as obscure to write out with if's and goto's. For example, here is a backwards do loop that finds the last nonblank character on a card:

```
for (i = 80; i > 0; i = i - 1)
    if (card(i) != blank)
        break
```

('!=' is the same as '.NE.'). The code scans the columns from 80 down to 1. If a nonblank is found, the loop is immediately exited. break and next work in for's and while's just as in do's. If i reaches zero, the card is all blank.

This code is rather nasty to write with a regular FORTRAN do, since the loop must go forward, and you must explicitly set up proper conditions when you fall out of the loop. Forgetting this is a common error. Thus,

```
    DO 10 J = 1, 80
        I = 81 - J
        IF (CARD(I) .NE. BLANK) GO TO 11
10  CONTINUE
    I = 0
11  ...
```

The version that uses the for handles the termination condition properly for free; i is zero when you fall out of the for loop.

The increment in a for need not be an arithmetic progression; the following program walks along a list (stored in an integer array ptr) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
    sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The repeat-until statement

In spite of warnings, there are times when you really need a loop that tests at the bottom after one pass through. This service is provided by the repeat-until:

```
repeat
    Ratfor statement
until (legal FORTRAN condition)
```

The *Ratfor statement* part is done once, then the condition is evaluated. If it is .true., the loop is exited; if it is .false., another pass is made.

The `until` part is optional, so a bare `repeat` is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as `stop`, `return`, or `break`, or an implicit stop such as running out of input with a READ statement.

As a matter of observed fact, the `repeat-until` statement is much less used than the other looping constructions; in particular, it is typically outnumbered ten to one by `for` and `while`. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on `break` and `next`

`break` exits immediately from `do`, `while`, `for`, and `repeat-until`. `next` goes to the test part of `do`, `while` and `repeat-until`, and to the increment step of a `for`.

The `return` statement

The standard FORTRAN mechanism for returning a value from a function uses the name of the function as a variable that can be assigned to. The last value stored in it is the function value upon return. For example, here is a routine `equal` that returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value –1.

```
# equal - compare str1 to str2;
#    return 1 if equal, 0 if not
    integer function equal(str1, str2)
    integer str1(100), str2(100)
    integer i

    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == -1) {
            equal = 1
            return
        }
    equal = 0
    return
    end
```

In many languages (e.g., PL/I) one instead says

```
return (expression)
```

to return a value from a function. Since this is often clearer, *Ratfor* provides such a `return` statement — in a function `F`, `return (expression)` is equivalent to

```
{ F = expression; return }
```

For example, here is `equal` again:

```
# equal - compare str1 to str2;
#   return 1 if equal, 0 if not
    integer function equal(str1, str2)
    integer str1(100), str2(100)
    integer i

    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == -1)
            return(1)
    return(0)
    end
```

If there is no parenthesized expression after `return`, a normal RETURN is made. (Another version of `equal` is presented shortly.)

Cosmetics

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand. Accordingly, *Ratfor* provides a number of cosmetic facilities that can be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line. Long statements are continued automatically, as are long conditions in `if`, `while`, `for`, and `until`. Blank lines are ignored. Multiple statements can appear on one line if they are separated by semicolons. No semicolon is needed at the end of a line, if *Ratfor* can make some reasonable guess about whether the statement ends there. Lines ending with any of these characters

```
= + - * , | & ( _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a FORTRAN label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

Translation services

Text enclosed in matching single or double quotes is converted to *n*H... but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '\' serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
  " \\\' "
```

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing FORTRAN program). Use '%' only for ordinary statements, not for the condition parts of `if`, `while`, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '%':

character	*translation*	*character*	*translation*
==	.eq.	!=	.ne.
>	.gt.	>=	.ge.
<	.lt.	<=	.le.
&	.and.	|	.or.
!	.not.	^	.not.

In addition, the following translations are provided for input devices with restricted character sets.

character	*translation*	*character*	*translation*
[	{	]	}
($	{	$)	}

The `define` statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by nonalphanumerics) it is replaced by the rest of the definition line. (Comments and trailing whitespace are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

`define` is typically used to create symbolic parameters:

```
define  ROWS    100
define  COLS    50
dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS  |  j > COLS) ...
```

Alternately, definitions can be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis, which allows for multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, since they help clarify the function of what would otherwise be mysterious numbers. As an example, here is the routine `equal` again, this time with symbolic constants.

```
define  YES     1
define  NO      0
define  EOS     -1
define  ARB     100

# equal - compare str1 to str2;
#   return YES if equal, NO if not
    integer function equal(str1, str2)
    integer str1(ARB), str2(ARB)
    integer i

    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == EOS)
            return(YES)
    return(NO)
    end
```

The `include` statement

The statement

```
    include file
```

inserts the file found on input stream *file* into the *Ratfor* input in place of the `include` statement. The standard usage is to place COMMON blocks on a file, and `include` that file whenever a copy is needed:

```
subroutine x
    include commonblocks
    ...
    end

subroutine y
    include commonblocks
    ...
    end
```

This ensures that all copies of the COMMON blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, `else` clauses without an `if`, and most errors involving missing parentheses in statements. Beyond that, since *Ratfor* knows no FORTRAN, the FORTRAN compiler reports any errors, so you will need to occasionally have to relate a FORTRAN diagnostic back to the *Ratfor* source.

Keywords are reserved — using `if`, `else`, etc., as variable names typically wreak havoc. Don't leave spaces in keywords or use the Arithmetic `if`.

The FORTRAN *n*H convention is not recognized anywhere by *Ratfor*; use quotes instead.

12.4. Implementation

Ratfor was originally written in C on the UNIX operating system. The language is specified by a context-free grammar, and the compiler constructed using the YACC compiler-compiler.

The *Ratfor* grammar is simple and straightforward, being essentially

```
prog     stat
    | prog   stat
stat     if (...) stat
    | if (...) stat else stat
    | while (...) stat
    | for (...; ...; ...) stat
    | do ... stat
    | repeat stat
    | repeat stat until (...)
    | switch (...) { case ...: prog ...
          default: prog }
    | return
    | break
    | next
    | digits   stat
    | { prog }
    | anything unrecognizable
```

The observation that *Ratfor* knows no FORTRAN follows directly from the rule that says a statement is 'anything unrecognizable.' In fact, most of FORTRAN falls into this category, since any statement that does not begin with one of the keywords is by definition 'unrecognizable.'

Code generation is also simple. If the first thing on a source line is not a keyword (like `if`, `else`, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when `if` is recognized, two consecutive labels L and L+1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the `if` is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s), the code

```
L   continue
```

is generated, unless there is an `else` clause, in which case the code is

```
    goto L+1
L   continue
```

In this latter case, the code

```
L+1 continue
```

is produced after the *statement* part of the `else`. Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing `else`,

```
    if (i > 0) x = a
```

should be left alone and not converted into

```
    if (.not. (i .gt. 0)) goto 100
    x = a
100 continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program where this kind of 'inefficiency' makes even a measurable difference. In the few cases where it is important, the offending lines can be protected by

'%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no compiler-compiler is available.

The C version of *Ratfor* is used on UNIX. C compilers are not as widely available as FORTRAN, however, so there is also a *Ratfor* written in itself and originally bootstrapped with the C version. The *Ratfor* version was written so it could be translated into the portable subset of FORTRAN described in [2]. Thus it is portable, having been run essentially without change on at least twelve distinct machines. The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v \pm c$; avoiding expressions in places like `do` loops; consistency in subroutine argument usage and in COMMON declarations. *Ratfor* itself does not generate nonstandard FORTRAN.

The *Ratfor* version is about 1500 lines of *Ratfor* (compared to about 1000 lines of C); this compiles into 2500 lines of FORTRAN. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the *Ratfor* version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine-coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

12.5. Experience

Good Things

'It's so much better than FORTRAN' is the most common response of users when asked how well *Ratfor* meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics convert FORTRAN 66 from a bad language into quite a reasonable one, assuming that FORTRAN data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in *Ratfor* is at least twice as fast as in FORTRAN. More important, debugging and subsequent revision are much faster than in FORTRAN. Partly this is because the code can be *read*. The looping statements that test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in *Ratfor*; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in *Ratfor* tend to be as readable as programs written in languages like Pascal. Once you are freed from the shackles of FORTRAN's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a *Ratfor* implementation of the linear table search discussed by Knuth in [14]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
    ;
if (i > m) {
    m = i
    B(i) = 1
}
else
    B(i) = B(i) + 1
```

A large corpus (5400 lines) of *Ratfor*, including a subset of the *Ratfor* preprocessor itself, can be found in [13].

Bad Things

The biggest single problem is that the FORTRAN compiler detects many syntax errors — not *Ratfor*. The compiler then prints a message in terms of the generated FORTRAN, which in a few cases may be difficult to relate back to the offending *Ratfor* line, especially if the implementation conceals the generated FORTRAN. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not as difficult as you might think. Since *Ratfor* generates no variables (only a simple pattern of `if`'s and `goto`'s), data-related errors like missing `dimension` statements are easy to find in FORTRAN. Furthermore, *Ratfor*'s ability to catch trivial syntactic errors like unbalanced parentheses and quotes has steadily improved.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic `if`. A few standard FORTRAN constructions are not accepted by *Ratfor*, which could be a problem to users with many existing FORTRAN programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stopgap. The best long-term solution is provided by the program Struct [4], which converts arbitrary FORTRAN programs into *Ratfor*.

Users who export programs often complain that the generated FORTRAN is 'unreadable' because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (*Ratfor* now has an option to copy *Ratfor* comments into the generated FORTRAN), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since *Ratfor* is relatively easy to modify, there are now several dialects of *Ratfor*. Fortunately, most of the differences so far are in character set, or in invisible aspects like code generation.

12.6. Conclusions

Ratfor demonstrates that with modest effort it is possible to convert FORTRAN from a bad language into a good one. A preprocessor is clearly a useful way to improve the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in 'features' — things that the user can trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for *Ratfor* to prepare a neatly formatted listing of its input or output. You are presumably capable of the self-discipline required to prepare neat input that reflects your thoughts. It is much more important that the language provide free-form input so you *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

A

ASCII Character Set

A

ASCII Character Set

dec	*oct*	*hex*	*name*	*dec*	*oct*	*hex*	*name*	*dec*	*oct*	*hex*	*name*	*dec*	*oct*	*hex*	*name*
0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	‘
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	’	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	–	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

B

FORTRAN Statements

B

FORTRAN Statements

The following table provides selected examples of all Sun FORTRAN statements. The purpose of the table is to jog the memory on syntax details for the more common variations of each statement type.

In this table, the following conventions are used:

C is a character variable
CA is a character array
I is an integer variable
U is an external unit

R is an real variable
N is an numeric variable
L is a logical variable
S is a switch variable

Table B-1 *FORTRAN Statements*

Name	*Examples*	*Comments*
Assign	`ASSIGN 9 TO I`	

Assignment	`C = "abc"`
	`C = S // "abc"`
	`C = S(I:M)`
		Character

	`L = L1 .OR. L2`
	`L = I .LE. 80`
		Logical

	`N = N+1`	Arithmetic

	`CURR = NEXT`
	`NEXT.ID = 82`
		See Record.

Automatic	`AUTOMATIC A, B, C`
	`AUTOMATIC REAL  P, D, Q`
	`IMPLICIT AUTOMATIC REAL (X-Z)`	

Backspace	`BACKSPACE I`
	`BACKSPACE( UNIT=U )`
	`BACKSPACE( UNIT=U, IOSTAT=I, ERR=9 )`	

Block Data	`BLOCK DATA`
	`BLOCK DATA COEFFS`	

Byte	`BYTE A, B, C`
	`BYTE A, B, C(10)`	

Table B-1 *FORTRAN Statements— Continued*

Name	*Examples*	*Comments*
Call	CALL P(A, B) CALL P(A, B, *9) CALL P	Alternate Return
Character	CHARACTER C*80, D*1(4) CHARACTER*16 A, B, C	

Close	CLOSE(UNIT=I)
	CLOSE(UNIT=U, ERR=90, IOSTAT=I)	

Common	COMMON / DELTAS / H, P, T COMMON X, Y, Z COMMON P, D, Q(10,100)	
Complex	COMPLEX U, V COMPLEX U(3,6) COMPLEX U*16	Double Complex
Continue	100 CONTINUE	
Data	DATA A, C / 4.01, "z" / DATA (V(I),I=1,3) /.7, .8, .9/ DATA ARRAY(4,4) / 1.0 / DATA B,O,X,Y /B'0011111', O'37', X'1f', Z'1f'/	
Dimension	DIMENSION ARRAY(4,4) DIMENSION V(1000), W(3)	
Do	DO 100 I = INIT, LAST, INCR ... 100 CONTINUE	
	DO I = INIT, LAST ... END DO	Unlabeled do
	DO WHILE (DIFF .LE. DELTA) ... END DO	Do while
	DO 100 WHILE (DIFF .LE. DELTA) ... 100 CONTINUE	
Double Complex	DOUBLE COMPLEX U, V DOUBLE COMPLEX U(3,6)	Complex*16
Double Precision	DOUBLE PRECISION A, D, Y(2)	Real*8
Else	ELSE	See Block If.
Else If	ELSE IF	See Block If.

Table B-1 *FORTRAN Statements— Continued*

Name	*Examples*	*Comments*
End	END	
End Do	END DO	See Do.

Endfile	ENDFILE(UNIT=I)
	ENDFILE I
	ENDFILE(UNIT=U, IOSTAT=I, ERR=9)	

End If	END IF	See Block If
End Map	END MAP	See Map.
End Structure	END STRUCTURE	See Structure.
End Union	END UNION	See Union.
Entry	ENTRY SCHLEP(X, Y) ENTRY SCHLEP(A1, A2, *4) ENTRY SCHLEP	
Equivalence	EQUIVALENCE (V(1), A(1,1)) EQUIVALENCE (V, A) EQUIVALENCE (X,V(10)), (P,D,Q)	
External	EXTERNAL RNGKTA, FIT	
Format	10 FORMAT(// X, I3, F6.1, E12.2, A, L2) 10 FORMAT(2X, 2I3, 3F6.1, 4E12.2, 2A6, 3L2) 10 FORMAT(// D6.1, G12.2) 10 FORMAT(2D6.1, 3G12.2) 10 FORMAT(2I3.3, 3G6.1E3, 4E12.2E3) 10 FORMAT('a quoted string', " another", I2) 10 FORMAT(18Ha hollerith string, I2) 10 FORMAT(1X, T10, A1, T20, A1) 10 FORMAT(5X, TR10, A1, TR10, A1, TL5, A1) 10 FORMAT(" Init=", I2, :, 3X, "Last=", I2) 10 FORMAT(1X, "Enter pathname ", $) 10 FORMAT(F4.2, Q, 80 A1) 10 FORMAT('Octal ', O6, ', Hex ' Z6)	
Function	FUNCTION Z(A, B) FUNCTION W(P,D, *9) CHARACTER FUNCTION R*4(P,D,*9) INTEGER*2 FUNCTION M(I, J)	
Go To	GO TO 99	Unconditional
	GO TO I, (10, 50, 99) GO TO I	Assigned
	GO TO (10, 50, 99), I GO TO I	Computed

Table B-1 *FORTRAN Statements— Continued*

Name	*Examples*	*Comments*
If	IF (I-K) 10, 50, 90	Arithmetic if
	IF (L) RETURN	Logical if

	IF (L) THEN
	  N=N+1
	  CALL CALC
	ELSE
	  K=K+1
	  CALL DISP
	END IF
		Block if

	IF (C .EQ. 'a') THEN
	  NA=NA+1
	  CALL APPEND
	ELSE IF (C .EQ. 'b') THEN
	  NB=NB+1
	  CALL BEFORE
	ELSE IF (C .EQ. 'c') THEN
	  NC=NC+1
	  CALL CENTER
	END IF
		Block if
		with else-if

Implicit	IMPLICIT COMPLEX (U-W,Z) IMPLICIT UNDEFINED (A-Z)	
Include	INCLUDE 'project02/header'	

Inquire	INQUIRE(UNIT=3, OPENED=OK)
	INQUIRE(FILE='mydata', EXIST=OK)
	INQUIRE(UNIT=3, OPENED=OK, IOSTAT=ERRNO)	

Integer	INTEGER C INTEGER C*2, D(4) INTEGER*4 A, B, C	
Intrinsic	INTRINSIC SQRT, EXP	
Logical	LOGICAL C LOGICAL B*1, C*1 LOGICAL*1 B, C LOGICAL*4 A, B, C	

Map	MAP
	   CHARACTER *16 MAJOR
	END MAP
	MAP
	   INTEGER*2 CREDITS
	   CHARACTER*8 GRAD_DATE
	END MAP
		See Structure and Union.

Table B-2 *More FORTRAN Statements*

Name	*Examples*	*Comments*
Namelist	NAMELIST /CASE/ S, N, D	

Open	OPEN(UNIT=3, FILE="data.test")
	OPEN(UNIT=3, IOSTAT=ERRNO)	

Parameter	PARAMETER (A="xyz"), (PI=3.14) PARAMETER (A="z", PI=3.14) PARAMETER (X=11, Y=X/3)	
Pointer	POINTER (P, V), (I, X)	
Program	PROGRAM FIDDLE	
Print	PRINT *, A, I	List-directed
	PRINT 10, A, I	Formatted
	PRINT 10, M	Array M
	PRINT 10, (M(I),I=J,K)	Implied-DO
	PRINT 10, C(I:K)	Substring
	PRINT '(A6,I3)', A, I PRINT FMT='(A6,I3)', A, I	Character constant format Character variable format
	PRINT S, I PRINT FMT=S, I	Switch variable has format number
	PRINT G	Namelist
Read	READ *, A, I	List-directed
	READ 1, A, I	Formated
	READ 10, M	Array M
	READ 10, (M(I),I=J,K)	Implied-DO
	READ 10, C(I:K)	Substring
	READ '(A6,I3)', A, I	Character constant format
	READ(1, 2) X, Y READ(UNIT=1, FMT=2) X,Y READ(1, 2, ERR=8,END=9) X,Y READ(UNIT=1, FMT=2, ERR=8,END=9) X,Y	Formatted read from a file
	READ(*, 2) X, Y	Formatted read from standard input
	READ(*, 10) M	Array M
	READ(*, 10) (M(I),I=J,K)	Implied-DO
	READ(*, 10) C(I:K)	Substring
	READ(1, *) X, Y READ(*, *) X, Y	List-directed read from a file — from standard input
	READ(1, '(A6,I3)') X, Y READ(1, FMT='(A6,I3)') X, Y	Character constant format
	READ(1, C) X, Y READ(1, FMT=C) X, Y	Character variable format

Table B-2 *More FORTRAN Statements— Continued*

Name	*Examples*	*Comments*

	`READ( 1, S )  X, Y`
	`READ( 1, FMT=S )  X, Y`
		Switch variable has format number

	`READ( *, G )`
	`READ( 1, G )`
		Namelist read
		Namelist read from a file

	`READ( 1, END=8, ERR=9 ) X, Y`	Unformatted sequential access
	`READ( 1, REC=3 ) V`	Unformatted direct access
	`READ( 1, 2, REC=3 ) V`	Formatted direct access
	`READ( CA, 1, END=8, ERR=9 ) X, Y`	Internal formatted sequential access
	`READ( CA, *, END=8, ERR=9 ) X, Y`	Internal list-directed sequential access
	`READ( CA, REC=4, END=8, ERR=9 ) X, Y`	Internal direct access

Real	`REAL R`
	`REAL R*4, M(4)`
	`REAL*8 A, B, C`
		
		Double Precision

Record	`RECORD /PROD/ CURR,PRIOR,NEXT`	

Return	`RETURN`
	`RETURN 2`
		Standard return
		Alternate return

Rewind	`REWIND 1`
	`REWIND I`
	`REWIND( UNIT=U, IOSTAT=I, ERR=9 )`	

Save	`SAVE  A, /B/, C`
	`SAVE`	

Static	`STATIC A, B, C`
	`STATIC REAL  P, D, Q`
	`IMPLICIT STATIC REAL (X-Z)`	

Stop	`STOP`
	`STOP "all gone"`	

Structure	`STRUCTURE /PROD/`
	`   INTEGER*4    ID`
	`   CHARACTER*16 NAME`
	`   CHARACTER*8  MODEL`
	`   REAL*4       COST`
	`   REAL*4       PRICE`
	`END STRUCTURE`	

Subroutine	`SUBROUTINE SHR( A, B, *9 )`
	`SUBROUTINE SHR( A, B )`
	`SUBROUTINE SHR`
		Alternate Return

Table B-2 *More FORTRAN Statements— Continued*

Name	*Examples*	*Comments*

Union	UNION
	  MAP
	    CHARACTER*16 MAJOR
	  END MAP
	
	  MAP
	    INTEGER*2 CREDITS
	    CHARACTER*8 GRAD_DATE
	  END MAP
	END UNION
		See Structure.

Write	WRITE(1, 2) X, Y WRITE(UNIT=1, FMT=2) X, Y WRITE(1, 2, ERR=8, END=9) X, Y WRITE(UNIT=1, FMT=2, ERR=8, END=9) X, Y	Formatted write to a file
	WRITE(*, 2) X, Y WRITE(*, 10) M	Formatted write to standard output Array M
	WRITE(*, 10) (M(I),I=J,K)	Implied-DO
	WRITE(*, 10) C(I:K)	Substring
	WRITE(1, *) X, Y WRITE(*, *) X, Y	List-directed write to a file — to standard output
	WRITE(1, ’(A6,I3)’) X, Y WRITE(1, FMT=’(A6,I3)’) X, Y	Character constant format
	WRITE(1, C) X, Y WRITE(1, FMT=C) X, Y	Character variable format
	WRITE(1, S) X, Y WRITE(1, FMT=S) X, Y	Switch variable has format number

	WRITE(*, CASE)
	WRITE(1, CASE)
		Namelist write
		Namelist write to a file

	WRITE(1, END=8, ERR=9) X, Y	Unformatted sequential access
	WRITE(1, REC=3) V	Unformatted direct access
	WRITE(1, 2, REC=3) V	Formatted direct access
	WRITE(CA, 1, END=8, ERR=9) X, Y	Internal formatted sequential access
	WRITE(CA, *, END=8, ERR=9) X, Y	Internal list-directed sequential access
	WRITE(CA, REC=4, END=8, ERR=9) X, Y	Internal direct access

C

Intrinsic Functions

Intrinsic Functions

Table C-1 *Sun FORTRAN Intrinsic Functions*

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Truncation	int(a)	1	AINT	AINT	Real	Real
	See Note 1			DINT	Double	Double
Nearest Whole Number	int(a+.5) if a≥0	1	ANINT	ANINT	Real	Real
	int(a-.5) if a < 0			DNINT	Double	Double
Nearest Integer	int(a+.5) if a≥0	1	NINT	NINT	Real	Integer
	int(a-.5) if a < 0			IDNINT	Double	Integer
Absolute Value	|a|	1	ABS	IABS	Integer	Integer
				ABS	Real	Real
				DABS	Double	Double
	See Note 6			CABS	Complex	Real
				ZABS ♦	Dcomplex	Double
	$(ar^2 + ai^2)$**(1/2)			CDABS ♦	Dcomplex	Double
Remainder	a1-int(a1/a2)*a2	2	MOD	MOD	Integer	Integer
	See Note 1			AMOD	Real	Real
				DMOD	Double	Double
Transfer of Sign	|a1| if a2 ≥ 0	2	SIGN	ISIGN	Integer	Integer
	-|a1| if a2 < 0			SIGN	Real	Real
				DSIGN	Double	Double
Positive Difference	a1-a2 if a1 > a2	2	DIM	IDIM	Integer	Integer
	0 if a1 ≤ a2			DIM	Real	Real
				DDIM	Double	Double
Double Precision Product	a1 * a2	2		DPROD	Real	Double
Choosing Largest Value	max(a1, a2, ...)	≥ 2	MAX	MAX0	Integer	Integer
				AMAX1	Real	Real
				DMAX1	Double	Double
				AMAX0	Integer	Real
				MAX1	Real	Integer
Choosing Smallest Value	min(a1, a2, ...)	≥ 2	MIN	MIN0	Integer	Integer
				AMIN1	Real	Real
				DMIN1	Double	Double
				AMIN0	Integer	Real
				MIN1	Real	Integer

Table C-1 *Sun FORTRAN Intrinsic Functions— Continued*

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Type Conversion	Conversion to Integer int(a) See Note 1	1	INT	– INT IFIX IDINT – –	Integer Real Real Double Complex Dcomplex	Integer Integer Integer Integer Integer Integer
	Conversion to Real See Note 2	1	REAL	REAL FLOAT – SNGL – –	Integer Integer Real Double Complex Dcomplex	Real Real Real Real Real Real
	Conversion to Double See Note 3	1	DBLE DREAL	– – – – –	Integer Real Double Complex Dcomplex	Double Double Double Double Double
	Conversion to Complex See Note 4	1 or 2	CMPLX	– – – – –	Integer Real Double Complex Dcomplex	Complex Complex Complex Complex Complex
	Conversion to Dcomplex	1 or 2	DCMPLX ♦	– – – – –	Integer Real Double Complex Dcomplex	Dcomplex Dcomplex Dcomplex Dcomplex Dcomplex
	Conversion to Integer See Note 5	1		ICHAR IACHAR ♦	Character	Integer
	Conversion to Character See Note 5	1		CHAR ACHAR ♦	Integer	Character

Table C-2 *More Sun FORTRAN Intrinsic Functions*

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Imaginary Part of a Complex	ai See Note 6	1	*IMAG*	AIMAG DIMAG ♦	Complex Dcomplex	Real Double
Conjugate of a Complex	(ar, -ai) See Note 6	1	*CONJG*	CONJG DCONJG ♦	Complex Dcomplex	Complex Dcomplex
Square Root	a**(1/2)	1	*SQRT*	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex

Table C-2 *More Sun FORTRAN Intrinsic Functions— Continued*

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
				ZSQRT ♦ CDSQRT ♦	Dcomplex Dcomplex	Dcomplex Dcomplex
Exponential	e**a	1	*EXP*	EXP DEXP CEXP ZEXP ♦ CDEXP ♦	Real Double Complex Dcomplex Dcomplex	Real Double Complex Dcomplex Dcomplex
Natural Logarithm	log(a)	1	*LOG*	ALOG DLOG CLOG ZLOG ♦ CDLOG ♦	Real Double Complex Dcomplex Dcomplex	Real Double Complex Dcomplex Dcomplex
Common Logarithm	log10(a)	1	*LOG10*	ALOG10 DLOG10	Real Double	Real Double

Table C-3 *Sun FORTRAN Bitwise Functions*

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Bitwise Operations See Note 13	Complement	1		NOT ♦	Integer	Integer
	And	2		AND ♦	Integer	Integer
	Inclusive Or	2		OR ♦	Integer	Integer
	Exclusive Or	2		XOR ♦	Integer	Integer
	Shift See Note 14	2		ISHFT ♦	Integer	Integer
	Left Shift See Note 14	2		LSHIFT ♦	Integer	Integer
	Right Shift See Note 14	2		RSHIFT ♦	Integer	Integer
	Bit Extraction	3		IBITS ♦	Integer	Integer
	Bit Set	2		IBSET ♦	Integer	Integer
	Bit Test	2		BTEST ♦	Integer	Logical
	Bit Clear	2		IBCLR ♦	Integer	Integer
	Circular Shift	3		ISHFTC ♦	Integer	Integer

Table C-4 *Sun FORTRAN Trigonometric Functions*

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Sine	sin(a)	1	*SIN*	SIN DSIN CSIN ZSIN ♦ CDSIN ♦	Real Double Complex Dcomplex Dcomplex	Real Double Complex Dcomplex Dcomplex
Sine (degrees)	sin(a)	1	*SIND* ♦	SIND ♦ DSIND ♦	Real Double	Real Double
Cosine	cos(a)	1	*COS*	COS DCOS CCOS ZCOS ♦ CDCOS ♦	Real Double Complex Dcomplex Dcomplex	Real Double Complex Dcomplex Dcomplex
Cosine (degrees)	cos(a)	1	*COSD* ♦	COSD ♦ DCOSD ♦	Real Double	Real Double
Tangent	tan(a)	1	*TAN*	TAN DTAN	Real Double	Real Double
Tangent (degrees)	tan(a)	1	*TAND* ♦	TAND ♦ DTAND ♦	Real Double	Real Double
Arcsine	arcsin(a)	1	*ASIN*	ASIN DASIN	Real Double	Real Double
Arcsine (degrees)	arcsin(a)	1	*ASIND* ♦	ASIND ♦ DASIND ♦	Real Double	Real Double
Arccosine	arccos(a)	1	*ACOS*	ACOS DACOS	Real Double	Real Double
Arccosine (degrees)	arccos(a)	1	*ACOSD* ♦	ACOSD ♦ DACOSD ♦	Real Double	Real Double
Arctangent	arctan(a)	1	*ATAN*	ATAN DATAN	Real Double	Real Double
	arctan(a1/a2)	2	*ATAN2*	ATAN2 DATAN2	Real Double	Real Double
Arctangent (degrees)	arctan(a)	1	*ATAND* ♦	ATAND ♦ DATAND ♦	Real Double	Real Double
	arctan(a1/a2)	2	*ATAN2D* ♦	ATAN2D ♦ DATAN2D ♦	Real Double	Real Double
Hyperbolic Sine	sinh(a)	1	*SINH*	SINH DSINH	Real Double	Real Double
Hyperbolic Cosine	cosh(a)	1	*COSH*	COSH DCOSH	Real Double	Real Double
Hyperbolic Tangent	tanh(a)	1	*TANH*	TANH DTANH	Real Double	Real Double

Table C-5 *Sun FORTRAN Character Functions*

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Conversion See Note 5	Conversion to Character	1		CHAR ACHAR ♦	Integer	Character
	Conversion to Integer	1		ICHAR IACHAR ♦	Character	Integer
Index of a Substring	Location of Substring a2 in String a1 See Note 10	2		INDEX	Character	Integer
Length	Length of Character Entity See Note 11	1		LEN	Character	Integer
Lexically Greater Than or Equal	a1 ≥ a2 See Note 12	2		LGE	Character	Logical
Lexically Greater Than	a1 > a2 See Note 12	2		LGT	Character	Logical
Lexically Less Than or Equal	a1 ≤ a2 See Note 12	2		LLE	Character	Logical
Lexically LessThan	a1 < a2 See Note 12	2		LLT	Character	Logical

Table C-6 *Sun FORTRAN Miscellaneous Functions*

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Environ-mental Inquiries *See Note 15*	Base of Number System	1	*EPBASE* ♦	-	Integer	Integer
				-	Real	Integer
				-	Double	Integer
	Number of Significant Bits	1	*EPPREC* ♦	-	Integer	Integer
				-	Real	Integer
				-	Double	Integer
	Minimum Exponent	1	*EPEMIN* ♦	-	Real	Integer
				-	Double	Integer
	Maximum Exponent	1	*EPEMAX* ♦	-	Real	Integer
				-	Double	Integer
	Least Nonzero Number	1	*EPTINY* ♦	-	Real	Real
				-	Double	Double
	Largest Number Representable	1	*EPHUGE* ♦	-	Integer	Integer
				-	Real	Real
				-	Double	Double
	Epsilon See Note 16	1	*EPMRSP* ♦	-	Real	Real
				-	Double	Double

Table C-6 *Sun FORTRAN Miscellaneous Functions— Continued*

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Location	Address-of See Note 17	1		`LOC` ♦	Any	Integer
Allocate	Allocate memory and return the address See Note 17	1		`MALLOC` ♦	Integer	Integer
Deallocate	Deallocate memory allocated by MALLOC	1		`FREE` ♦	Any	None. This is a subroutine.

Notes:

These tables and notes 1 through 12 are based on the "*Table of Intrinsic Functions*," from ANSI X3.9-1978 Programming Language FORTRAN, with the Sun FORTRAN extensions added. For other subroutines and functions, see also the FORTRAN man pages, online or at the end of this manual.

General:

- Functions marked with a ♦ are extensions to the standard.
- An intrinsic that takes an `INTEGER` argument accepts either `INTEGER*2` or `INTEGER*4`.
  An intrinsic that returns an `INTEGER` value returns the prevailing `INTEGER` type: `INTEGER*4` unless the **`-i2`** option is selected.
  The exceptions are `LOC` and `MALLOC`, which always return an `INTEGER*4`.
- The abbreviation "Double" stands for *Double Precision.*
- The abbreviation "Dcomplex" stands for *Double Precision Complex.*
- A function with a *generic* name returns a value with the same type as the argument — except for type conversion functions, the nearest integer function, and absolute value of a complex argument. If there is more than one argument, they must all be of the same type.
- If a function name is used as an *actual* argument, then it must be a *specific* name.
- If a function name is used as a *dummy* argument, then it does not identify an intrinsic function in the subprogram, and it has a data type according to the same rules as for variables and arrays.

(1) `INT`

If `A` is type integer, then `INT(A)` is `A`.
If `A` is type real or double precision then:
    if `|A| < 1`, then `INT(A)` is 0
    if `|A| ≥ 1`, then
        `INT(A)` is the greatest integer that
        does not exceed the magnitude of `A`,
        and whose sign is the same as the sign of `A`.
        (Such a mathematical integer value may be too large
        to fit in the computer integer type.)
If `A` is type complex or double complex then
        apply the above rule to the real part of `A`.

If `A` is type real then `IFIX(A)` is the same as `INT(A)`.

(2) `REAL`

If `A` is type real, then `REAL(A)` is `A`.
If `A` is type integer or double precision, then
    `REAL(A)` is as much precision of the significant part of `A`
    as a real datum can contain.
If `A` is type complex, then `REAL(A)` is the real part of `A`.
If `A` is type double complex, then
    `REAL(A)` is as much precision of the significant part of
    the real part of `A` as a real datum can contain.

(3) `DBLE`

If `A` is type double precision, then `DBLE(A)` is `A`.
If `A` is type integer or real, then `DBLE(A)` is
    as much precision of the significant part of `A`
    as a double precision datum can contain.
If `A` is type complex, then `DBLE(A)` is
    as much precision of the significant part of the real part of `A`
    as a double precision datum can contain.
If `A` is type Dcomplex, then `DBLE(A)` is the real part of `A`.

(4) CMPLX

If A is type complex, then CMPLX(A) is A .
If A is type integer, real, or double precision, then
    CMPLX(A) is REAL(A) + 0i .
If A is type integer, real, or double precision, then
    CMPLX(A1,A2) is REAL(A1) + REAL(A2)*i
If A is type double complex, then
    CMPLX(A) is REAL(DBLE(A)) + i*REAL(DIMAG(A)) .

If CMPLX has two arguments, then
    they must be of the same type, and
    they may be one of integer, real, or double precision.

If CMPLX has one argument, then
    it may be one of integer, real, double precision, complex,
    or Dcomplex.

(4') DCMPLX

If A is type Dcomplex, then DCMPLX(A) is A .
If A is type integer, real, or double precision, then
    DCMPLX(A) is DBLE(A) + 0i .
If A1 and A2 are type integer, real, or double precision, then
    DCMPLX(A1,A2) is DBLE(A1) + DBLE(A2)*i .

If DCMPLX has two arguments, then
    they must be of the same type, and
    they may be one of integer, real, or double precision.

If DCMPLX has one argument, then
    it may be one of integer, real, double precision, complex,
    or Dcomplex.

(5) ICHAR

ICHAR(A) is the position of A in the collating sequence.
The first position is 0, the last is N-1 , 0 ≤ ICHAR(A) ≤ N-1,
where N is the number of characters in the collating sequence,
and A is of type character of length one. CHAR and ICHAR
are inverses in the following sense:
  ICHAR(CHAR(I)) = I, for 0 ≤ I ≤ N-1 .
  CHAR(ICHAR(C)) = C, for any character C capable of
  representation in the processor.

(6) Complex

A Complex value is expressed as an ordered pair of reals, (ar, ai),
where ar is the real part and ai is the imaginary part.

(7) Radians

All angles are expressed in radians, unless the "*Intrinsic Function*" column includes the remark "*(degrees)*".

(8) Complex Function

The result of a function of type complex is the principal value.

(9) Argument types

All arguments in an intrinsic function reference must be of the same type.

(10) INDEX

INDEX(X, Y) is the place in X where Y starts. That is, it is the starting position within character string X of the first occurrence of character string Y .
If Y does not occur in X, then INDEX(X, Y) is 0.
If LEN(X) < LEN(Y), then INDEX(X, Y) is 0.

(11) Argument to LEN

The value of the argument of the LEN function need not be defined at the time the function reference is executed.

(12) Lexical Compare

LGE(X, Y) is true if X=Y or if X follows Y in the collating sequence; otherwise it is false.

LGT(X, Y) is true if X follows Y in the collating sequence; otherwise it is false.

LLE(X, Y) is true if X=Y or if X precedes Y in the collating sequence; otherwise it is false.

LLT(X, Y) is true if X precedes Y in the collating sequence; otherwise it is false.

If the operands for LGE, LGT, LLE, and LLT are of unequal length, the shorter operand is considered as if it were extended on the right with blanks.

(13) Bit Functions

Bitwise operations are described in Chapter 10 — *The VMS Extensions* .

(14) SHIFT

`LSHIFT` shifts *a1* logically left by *a2* bits.
`RSHIFT` shifts *a1* logically right by *a2* bits.
`ISHIFT` shifts *a1* logically right if *a2* is positive and left if *a2* is negative.
The `LSHIFT` and `RSHIFT` functions are the FORTRAN analogs of C's "`and`" operators. As in C, the semantics depend on the hardware.

(15) Environmental inquiries

Only the type of the argument is significant.

(16) Epsilon

Epsilon is the least `e` such that `1.0 + e ≠ 1.0` .

(17) `LOC` and `MALLOC`

The `LOC` function returns the 32-bit address of a variable or of an external procedure. The function call `MALLOC(` *n* `)` allocates a block of at least *n* bytes, and returns the 32-bit address of that block.

D

FORTRAN Runtime Error Messages

D

FORTRAN Runtime Error Messages

D.1. Overview

The FORTRAN I/O library, the FORTRAN signal handler, and parts of the SunOS operating system (when called by FORTRAN library routines) can all generate FORTRAN error messages.

D.2. SunOS Error Messages

SunOS error messages include system call failures, C library errors, and shell diagnostics. The system call error messages are found in *intro* (2) in the *SunOS Reference Manual.* System calls made via the FORTRAN library do not produce error messages directly. The following system routine in the FORTRAN library calls C library routines which produce an error message:

```
CALL SYSTEM("rm /")
END
```

The following message is printed:

```
rm: / directory
```

D.3. Signal Handler Error Messages

Before beginning execution of a program, the FORTRAN library sets up a signal handler (`sigdie`) for signals that could cause termination of the program. `sigdie` prints a message that describes the signal, flushes any pending output, and generates a core image.

Presently the only arithmetic exception caught is the `INTEGER*2` division with a denominator of zero. All other arithmetic exceptions are silently ignored.

A signal handler error example follows when the subroutine `SUB` tries to access parameters that are not passed to it:

```
CALL SUB()
END
SUBROUTINE SUB(I,J,K)
I=J+K
RETURN
END
```

The following error message is printed:

```
*** Segmentation violation
Illegal instruction (core dumped)
```

D.4. I/O Error Messages

The following error messages are generated by the FORTRAN I/O library. The error numbers are returned in the `IOSTAT` variable if the `ERR` return is taken.

As an example, the following program tries to do an unformatted write to a file opened for formatted output:

```
      WRITE( 6 ) 1
      END
```

and gets an error messages like the following:

```
sue: [103] unformatted io not allowed
logical unit 6, named 'stdout'
lately: writing sequential unformatted external IO
Illegal instruction (core dumped)
```

100 *error in format*
See the error message output for the location of the error in the format. It can be caused by more than 10 levels of nested parentheses or an extremely long format statement.

101 *illegal unit number*
It is illegal to close logical unit 0. Negative unit numbers are not allowed. The upper limit is $2^{31}-1$.

102 *formatted io not allowed*
The logical unit was opened for unformatted I/O.

103 *unformatted io not allowed*
The logical unit was opened for formatted I/O.

104 *direct io not allowed*
The logical unit was opened for sequential access, or the logical record length was specified as 0.

105 *sequential io not allowed*
The logical unit was opened for direct access I/O.

106 *can't backspace file*
The file associated with the logical unit can't seek. It may be a device or a pipe.

107 *off beginning of record*
The format specified a left tab beyond the beginning of an internal input record.

108 *can't stat file*
The system can't return status information about the file. Perhaps the directory is unreadable.

109 *no * after repeat count*
Repeat counts in list-directed I/O must be followed by an * with no blank spaces.

110 *off end of record*
A formatted write tried to go beyond the logical end-of-record. An unformatted read or write will also cause this.

111 *truncation failed*
The truncation of an external sequential file on `close`, `backspace`, or `rewind` could not be done.

112 *incomprehensible list input*
List input has to be as specified in the declaration.

113 *out of free space*
The library dynamically creates buffers for internal use. You ran out of memory for this (i.e., your program is too big).

114 *unit not connected*
The logical unit was not open.

115 *read unexpected character*
Certain format conversions can't tolerate nonnumeric data.

116 *blank logical input field*
`logical` data must be T or F

117 *'new' file exists*
You tried to open an existing file with `status='new'`.

118 *can't find 'old' file*
You tried to open a nonexistent file with `status='old'`.

119 *unknown system error*
Shouldn't happen, but .....

120 *requires seek ability*
Direct access requires seek ability. Sequential unformatted I/O requires seek ability on the file due to the special data structure required. Tabbing left also requires seek ability.

121 *illegal argument*
Certain arguments to `open`, etc. will be checked for legitimacy. Often only non-default forms are looked for.

122 *negative repeat count*
The repeat count for list-directed input must be a positive integer.

123 *illegal operation for unit*
An operation was requested, which was not possible for a device associated with the logical unit. This error is returned by the tape I/O routines if attempting to read past end-of-tape, etc.

124 *too many files open - no free descriptors*
You tried to open one more file than the maximum number of files allowed open. The current limit is 64.

125 *attempted operation on unit that's not open*
The logical unit was not open.

126 *illegal input for namelist*
A namelist read encountered an invalid data item.

E

Bibliography

E

Bibliography

The following books or documents describe aspects of FORTRAN 66, FORTRAN 77, and related subjects. This list is not necessarily complete. No particular endorsement is implied.

1. American National Standards Institute. 1978. *American National Standard Programming Language FORTRAN, ANSI X3.9-1978.* New York.

2. —. 1966. *American National Standard FORTRAN.* New York.

3. Ageloff, Roy, and Richard Mojena. 1981. *Applied FORTRAN 77 featuring Structured Programming.* Wadsworth.

4. Brainerd, Walter S., et al. 1978. *FORTRAN 77 Programming.* Harper and Row.

5. Cray Research Inc. 1982. Cray-1 Computer Systems FORTRAN (CFT) Reference Manual. SR-0009.

6. Day, A. C. 1979. *Compatible Fortran.* Cambridge University Press.

7. Digital Equipment Corporation. 1984. Programming in VAX FORTRAN. [AA-D034D-TE].

8. Dock, V. Thomas. 1979. *Structured FORTRAN 77 Programming.* West.

9. Etter, D. M. 1984. *Problem Solving with FORTRAN 77.* Benjamin/Cummings Publishing Company, Inc.

10. Hume, J. N., and R. C. Holt. 1979. *Programming FORTRAN 77.* Reston.

11. Katzan, Harry, Jr. 1978. *FORTRAN-77.* Van Nostrand-Reinhold.

12. Kernighan, B. W. January 1977. RATFOR — A Preprocessor for a Rational Fortran. *Bell Laboratories Computing Science Technical Report #55*,

13. Kernighan, B. W., and P. J. Plauger. 1976. *Software Tools*. Addison-Wesley.

14. Knuth, D. E. December 1974. Structured Programming with goto Statements. *Computing Surveys.*

15. Meissner, Loren P., and Elliott I. Organick. 1980. *FORTRAN-77 Featuring Structured Programming.* Addison-Wesley.

16. Merchant, Michael J. 1979. *ABC's of FORTRAN 77 Programming*. Wadsworth.

17. Page, Rex, and Richard Didday. 1980. *FORTRAN 77 for Humans.* West.

18. Wagener, Jerrold L. 1980. *Principles of FORTRAN 77 Programming.* Wiley.

19. *IEEE Standard For Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, IEEE, NY, 1985.

F

Manual Pages for FORTRAN

F

Manual Pages for FORTRAN

NAME

intro – introduction to FORTRAN Manual Pages

DESCRIPTION

This section includes the man pages for **f77, f77cvt, fpr,** and **fsplit**.

NAME

f77 – Sun FORTRAN compiler

SYNOPSIS

f77 [**–66**] [**–a**] [**–align** *_block*] [**–ansi**] [**–c**] [**–C**] [**–dryrun**] [**–D***name*[=*def*]] [**–e**]
[*float_option*] [**–fstore**] [**–f**] [**–F**] [**–g**] [**–help**] [**–i2**] [**–i4**]
[**–I***pathname*] [**–l***lib*] [**–L***dir*] [**–misalign**] [**–N**[cdlnqsx]*nnn*]
[**–o** *outfile*] [**–onetrip**] [**–O**[123]] [**–p**] [**–pg**] [**–pipe**] [**–Qoption** *prog opt*]
[**–Qpath** *pathname*] [**–Qproduce** *sourcetype*] [**–S**] [**–temp**=*dir*] [**–time**]
[**–u**] [**–U**] [**–v**] [**–w**[66]] *sourcefile* ...

DESCRIPTION

f77 is the Sun FORTRAN compiler, which translates programs written in the Sun FORTRAN programming language into executable load modules or into relocatable binary programs for subsequent linking with **ld**(1). Sun FORTRAN is a superset of FORTRAN 77, with many extensions, including those to provide compatibility with VMS FORTRAN (in conjunction with **f77cvt**(1)). In addition to the many flag arguments (options), **f77** accepts several types of files.

Files with names ending in **.f** are taken to be Sun FORTRAN source files; they are compiled, and each object program is put in the current directory in a file with the same name as the source, with **.o** substituted for **.f**.

Files with names ending in **.F** are also taken to be Sun FORTRAN source files, but they are preprocessed by the C preprocessor (equivalent to a **cc** –E command) before they are compiled by the **f77** compiler.

Files with names ending in **.c** or **.s** are taken to be C or assembly source files and are compiled or assembled, producing **.o** files.

Files with names ending in **.il** are taken to be in-line expansion code template files; these are used to expand calls to selected routines in-line when the –O option is in effect.

Files with names ending in **.vf** or **.for** are assumed by the **f77cvt**(1) source code converter (not by the **f77** compiler) to be valid VMS FORTRAN source files and are converted to source files acceptable to both Sun FORTRAN and VMS FORTRAN compilers, except for possible VMS FORTRAN features which it can't convert, which are reported by error messages.

OPTIONS

See **ld**(1) for link-time options.

–66 Report non-FORTRAN 66 constructs as errors.

–a Insert code to count how many times each basic block is executed. Invokes a runtime recording mechanism that creates a **.d** file for every **.f** file (at normal termination). The **.d** file accumulates execution data for the corresponding source file. The **tcov**(1) utility can then be run on the source file to generate statistics about the program.

–align *_block_* Cause the common block whose FORTRAN name is *block* to be page-aligned: its size is increased to a whole number of pages, and its first byte is placed at the beginning of a page. This option is passed on to the linker; it's a linker option. For example, the command "**f77 -align _BUFFO_ GROWTH.F**" causes BUFFO to be page-aligned.

–ansi Identify all non-ANSI extensions. Note that **f77cvt** provides an option to flag any Sun FORTRAN extensions that it uses during the conversion of a VMS FORTRAN source file.

–c Suppress linking with **ld**(1) and produce a **.o** file for each source file. A single object file can be named explicitly using the **–o** option.

–C Compile code to check that subscripts are within the declared array bounds.

–dryrun Show but do not execute the commands constructed by the compilation driver.

–D*name*[=*def*] Define a symbol *name* to the C preprocessor, **cpp**(1). Equivalent to a **#define** directive in the source. If no *def* is given, *name* is defined as '1' (**.F** suffix files only).

−e Accept extended source lines, up to 132 characters long.

−f Align local data and common blocks on 8-byte boundaries. Resulting code may not be standard and may not be portable.

float_option Floating-point code generation option. This option does not apply to the Sun-4, which generates SPARC floating-point instructions. For the Sun-2 and Sun-3, *float_option* can be one of:

> **−f68881**
> Generate in-line code for the Motorola MC68881 floating-point coprocessor (Sun-3 only).
>
> **−ffpa** Generate in-line code for the Sun-3 Floating-Point Accelerator board (Sun-3 only).
>
> **−fsky** Generate in-line code for the Sky Floating-Point Processor (Sun-2 only).
>
> **−fsoft** Generate software floating-point calls (Sun-2 and Sun-3 systems, for which this is the default).
>
> **−fstore** Insure that expressions allocated to extended precision registers are rounded to storage precision whenever an assignment occurs in the source code. Only has effect when **−f68881** is specified. (Sun-3 only)
>
> **−fswitch**
> Generate runtime-switched floating-point calls. The compiled object code is linked at runtime to routines that support one of the above types of floating-point code. This was the default in previous releases. Only for use with programs that are floating-point intensive and which must be portable to machines with various floating-point options (Sun-2 or Sun-3).

−F Apply the C preprocessor to **.F** files. Put the result in corresponding **.f** files, but do not compile them. No linking is done.

−g Produce additional symbol table information for **dbx**(1) and pass the **−lg** flag to **ld**(1).

−help Display an equivalent of this list of options.

−i2 Make the default size of integer and logical constants and variables two bytes.

−i4 Make the default size of integer and logical constants and variables four bytes (this is the default).

−I*pathname* Add *pathname* to the list of directories in which to search for **#include** files with relative filenames (not beginning with /). The preprocessor first searches for **#include** files in the directory containing *sourcefile,* then in directories named with **−I** options (if any), and finally in **/usr/include/f77** (applies to processing of **.F** suffix files only).

−l*lib* Link with object library *lib* (for **ld**(1)).

−L*dir* Add *dir* to the list of directories containing object-library routines (for linking using **ld**(1)).

−misalign Allow for misaligned data in memory. Use this option **only** if you get a warning that **COMMON** or **EQUIVALENCE** statements cause data to be misaligned.
WARNING: With this option, the compiler will generate very much *slower* code for references to dummy arguments. If you can, you should recode the indicated section instead of recompiling with this option.

−N[cdlnqsx]*nnn* Make static tables in the compiler bigger. **f77** complains if tables overflow and suggests you apply one or more of these flags. These flags have the following meanings:

- **c** Maximum depth of nesting for control statements (for example, **DO** loops). Default is 20.
- **d** Maximum depth of nesting for data structures and unions. Default is 20.
- **l** Maximum number of continuation lines for a continued statement. The default is 19 (1 initial and 19 continuation).
- **n** Maximum number of identifiers. Default is 1009.
- **q** Maximum number of equivalenced variables. Default is 150.
- **s** Maximum number of statement numbers. Default is 401.
- **x** Maximum number of external names (common block, subroutine, and function names). Default is 200.

Multiple −**N** options increase sizes of multiple tables.

−o *outfile* Name the output file *outfile. outfile* must have the appropriate suffix for the type of file to be produced by the compilation (see FILES, below). *outfile* cannot be the same as *sourcefile* (the compiler will not overwrite the source file).

−onetrip Compile **DO** loops so that they are performed at least once if reached. Otherwise, Sun FORTRAN **DO** loops are not performed at all if the upper limit is smaller than the lower limit.

−O[123] Optimize the object code. This invokes both the global intermediate code optimizer and the object code optimizer.

- **−O1** Peephole Optimization only. Do not use **−O1** unless **−O2** and **−O3** result in excessive compilation time, or running out of swap space.
- **−O2** Partial optimization. Does a restricted set of global optimizations. Do not use **−O2** unless **−O3** results in excessive compilation time, or running out of swap space. (Same as **−P**)
- **−O3** Global Optimization. (same as **−O**)

  If the optimizer runs out of swap space, try any of the following possibly corrective measures (listed in increasing order of difficulty):

  Change from **-O3** to **-O2**
  Divide large, complicated routines into smaller, simpler ones.
  Increase the limit for the stacksize: insert the line
  "limit stacksize 8 megabytes"
  into your **.cshrc** file.
  Repartition you disk with two to four times as much swap space. Backup everything first. You may well need help from your system administrator to do this.

−p Prepare the object code to collect data for profiling with **prof**(1). Invokes a runtime recording mechanism that produces a **mon.out** file (at normal termination).

−pg Prepare the object code to collect data for profiling with **gprof**(1). Invokes a runtime recording mechanism that produces a **gmon.out** file (at normal termination).

−pipe Use pipes, rather than intermediate files between compilation stages. Very cpu-intensive.

−P See the **−O2** option.

−Qoption *prog opt*

Pass the option *opt* to the program *prog*. The option must be appropriate to that program and may begin with a minus sign. *prog* can be one of: **as, c2, cg, cpp, f77pass1, iropt, inline** or **ld**.

–Qpath *pathname*
Insert directory *pathname* into the compilation search path (to use alternate versions of programs invoked during compilation). This path will also be searched first for certain relocatable object files that are implicitly referenced by the compiler driver (such files as ***crt*.o** and **bb_link.o**).

–Qproduce *sourcetype*
Produce source code of the type *sourcetype*, where *sourcetype* can be one of:

.o	Object file from **as**(1).
.s	Assembler source (from **f77pass1, inline, c2** or **cg**).

–S Compile the named programs, and leave the assembly language output on corresponding files suffixed **.s** (no **.o** file is created).

–temp=*dir* Set directory for temporary files to be *dir*.

–time Report execution times for the various compilation passes.

–u Make the default type of a variable 'undefined', rather than using the FORTRAN default rules.

–U Do not convert upper case letters to lower case. The default is to convert upper case letters to lower case, except within character string constants.

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

–w[66] Suppress all warning messages. **–w66** suppresses only FORTRAN 66 compatibility warnings.

Other arguments are taken to be either linker option arguments, or **f77**-compatible object programs, typically produced by an earlier run, or libraries of **f77**-compatible routines. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program in the file specified by the **–o** option, or in a file named **a.out** if the **–o** option is not specified.

ENVIRONMENT

FLOAT_OPTION When no floating-point option is specified, the compiler uses the value of this environment variable (if set). Recognized values are: **f68881, ffpa, fsky, fswitch** and **fsoft.**

FILES

a.out	executable output file
file.**a**	library of object files
file.**d**	**tcov**(1) test coverage input file
file.**f**	Sun FORTRAN source file
file.**F**	Sun FORTRAN source file for **cpp**(1)
file.**for**	VMS FORTRAN source file for **f77cvt**(1)
file.**vf**	VMS FORTRAN source file for **f77cvt**(1)
file.**il**	**inline** expansion file
file.**o**	object file
file.**s**	assembler source file
file.**S**	assembler source for **cpp**(1)
file.**tcov**	output from **tcov**(1)
/lib/c2	optional optimizer
/lib/cg	Sun FORTRAN code generator
/lib/compile	compiler command-line processing driver
/lib/cpp	macro preprocessor
/lib/crt0.o	runtime startup
/lib/Fcrt1.o	startup code for **–fsoft** option
/lib/gcrt0.o	startup for gprof-profiling
/lib/libc.a	standard library, see **intro**(3)
/lib/mcrt0.o	startup for profiling
/lib/Mcrt1.o	startup code for **–f68881** option

/lib/Scrt1.o	startup code for **−fsky** option
/lib/Wcrt1.o	startup code for **−ffpa** option
/usr/include/f77	directory searched by the Sun FORTRAN **INCLUDE** statement
/usr/bin/f77	compiler command-line processing driver
/usr/bin/f77cvt	VMS FORTRAN source code converter
/usr/lib/f77pass1	Sun FORTRAN parser
/usr/lib/libc_p.a	profiling library, see **intro**(3)
/usr/lib/libF77.a	Sun FORTRAN library: General - other than I/O or UNIX interface
/usr/lib/inline	inline expander of library calls
/usr/lib/libI77.a	Sun FORTRAN library: I/O routines
/usr/lib/libm.a	math library
/usr/lib/libpfc.a	startup code for combined Sun Pascal and Sun FORTRAN programs
/usr/lib/libU77.a	Sun FORTRAN library: interface to UNIX system calls
/tmp/*	compiler temporary files
mon.out	file produced for analysis by **prof**(1)
gmon.out	file produced for analysis by **gprof**(1)

SEE ALSO

cc(1), **f77cvt**(1), **fpr**(1), **fsplit**(1), **gprof**(1), **ld**(1), **prof**(1)

Sun FORTRAN Programmer's Guide

Floating-Point Programmer's Guide for the Sun Workstation

DIAGNOSTICS

The diagnostics produced by **f77** itself are intended to be self-explanatory. Occasional messages may be produced by the linker.

NAME

f77cvt – VMS FORTRAN to Sun FORTRAN source code converter

SYNOPSIS

f77cvt [**–b**] [**–d**] [**–D**] [**–e**] [**–E**] [**–i**] [**–Ncx**] [**–Ndx**] [**–Nlx**] [**–P**] [**–s**] [**–v**]
filename.**vf** | *filename*.**for** ...

DESCRIPTION

f77cvt is the VMS FORTRAN source code converter. It converts source files written in valid VMS FORTRAN into FORTRAN source files acceptable to both the Sun FORTRAN and VMS FORTRAN compilers, except for possible VMS FORTRAN features which it can't convert, which are reported by error messages. It optionally produces warning messages if Sun-specific FORTRAN extensions are generated.

The converter accepts input files with filenames ending in **.vf** or **.for**, in upper, lower or mixed case.

It produces output files with filenames ending in **.f** or **.F** depending on the **–D** option.

The converter accepts options that roughly correspond to the VMS FORTRAN compiler options.

A successful completion of **f77cvt** *filename*.**vf** can be followed immediately by **f77** *filename*.**f** or **f77** *filename*.**F**, depending on the **–D** option.

OPTIONS

–b Prevents the converter from creating **BLOCK DATA** subprograms for initialized **COMMON** variables.

–d Enables VMS FORTRAN debugging statements. If specified, lines with a **D** or **d** in column 1 are converted into FORTRAN statements. If this option is not used, these debugging statements are converted into comments.

–D Also enables VMS FORTRAN debugging statements, but with this option the debugging statements are enclosed by a pair of preprocessor statements, as shown here:

```
#ifdef DEBUG
      ...
#endif
```

The preceding two options are mutually exclusive; a diagnostic will result if you specify both of them.

–e Indicates that the input file(s) contains extended source lines (up to 132 characters long). If your file contains extended lines and you omit this option, the source lines are truncated to 72 characters.

–E Allows the output file(s) to contain extended source lines. If this option is not specified, the output lines are broken into 72-column lines.

–i Inserts the text of **INCLUDE** files into the converted programs.

–Nc*x* Sets *x* as the maximum number of levels that control structures can be nested.

–Nd*x* Sets *x* as the maximum number of levels that data structures and unions can be nested.

–Nl*x* Sets *x* as the maximum number of continuation lines for a continued statement. (If you use **-Nl50,** then you can have a total of 51 lines: one first line and 50 after it.) The default is 19, 1 first line and 19 after it.

–P Suppresses the generation of preprocessor line numbers for **dbx** and **f77**.

–s Produces warning messages if Sun-specific FORTRAN extensions are generated.

–v Enables *verbose* mode; that is, the converter places a FORTRAN comment before each converted line, describing the conversion of the source code.

FILES

filename.**f**	FORTRAN source file (output from **f77cvt** without the **–D** option)
filename.**F**	FORTRAN source file (output from **f77cvt** with the **–D** option)
filename.**for**	valid VMS FORTRAN source file (input to **f77cvt**)
filename.**vf**	valid VMS FORTRAN source file (input to **f77cvt**)

SEE ALSO

f77(1), **fpr**(1), **fsplit**(1)
Sun FORTRAN Programmer's Guide

DIAGNOSTICS

The diagnostics produced by **f77cvt** itself are intended to be self-explanatory.

NAME

fpr – print FORTRAN file

SYNOPSIS

fpr

DESCRIPTION

fpr transforms files formatted according to FORTRAN's carriage control conventions into files formatted according to UNIX line printer conventions.

fpr copies its input onto its output, replacing the carriage control characters with characters that will produce the intended effects when printed using **lpr**(1). The first character of each line determines the vertical spacing as follows:

(blank)	one line
0	two lines
1	to first line of next page
+	no advance

A blank line (that is, an empty line) is treated as if its first character is a blank. A blank that appears as a carriage control character is deleted. A zero is changed to a newline. A one is changed to a form feed. The effects of a "+" are simulated using backspaces.

Note that **fpr** is known as **asa** in UNIX System V.

EXAMPLES

```
a.out | fpr | lpr
```

```
fpr < f77.output | lpr
```

BUGS

Results are undefined for input lines longer than 170 characters.

NAME

fsplit – split a multi-routine FORTRAN file into individual files

SYNOPSIS

fsplit [**–e** *efile*] . . . [*file*]

DESCRIPTION

fsplit takes as input either a file or standard input containing FORTRAN source code. It attempts to split the input into separate routine files of the form *name*.**f**, where *name* is the name of the program unit (function, subroutine, block data or program). The name for unnamed block data subprograms has the form **blkdta***NNN*.**f** where *NNN* is three digits and a file of this name does not already exist. For unnamed main programs the name has the form **main***NNN*.**f** . If there is an error in classifying a program unit, or if *name*.**f** already exists, the program unit will be put in a file of the form **zzz***NNN*.**f** where **zzz***NNN*.**f** does not already exist.

Normally each subprogram unit is split into a separate file. When the **–e** option is used, only the specified subprogram units are split into separate files. For example:

```
fsplit –e readit –e doit prog.f
```

will split **readit** and **doit** into separate files.

DIAGNOSTICS

If names specified via the **-e** option are not found, a diagnostic is written to *standard error*.

BUGS

fsplit assumes the subprogram name is on the first noncomment line of the subprogram unit. Nonstandard source formats may confuse **fsplit.**

It is hard to use –**e** for unnamed main programs and block data subprograms since you must predict the created file name.

NAME

intro – introduction to FORTRAN library functions

DESCRIPTION

This section describes those functions that are in the FORTRAN runtime library. The functions listed here provide an interface from **f77** programs to the system in the same manner as the C library does for C programs. They are automatically loaded as needed by the FORTRAN compiler **f77 (1).**

Most of these functions are in **libU77.a** . Some are in **libF77.a** or **libI77.a** . A few intrinsic functions are described for the sake of completeness.

For efficiency, the SCCS ID strings are not normally included in the **a.out** file. To include them, simply declare

```
EXTERNAL f77lid
```

in any **f77** module.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
abort	abort(3F)	terminate abruptly with memory image
access	access(3F)	determine accessibility of a file
alarm	alarm(3F)	execute a subroutine after a specified time
bit	bit(3F)	and, or, xor, not, rshift, lshift, bic, bis, bit, setbit functions
chdir	chdir(3F)	change default directory
chmod	chmod(3F)	change mode of a file
ctime	time(3F)	return system time
drand	rand(3F)	return random values
dtime	etime(3F)	return elapsed execution time
etime	etime(3F)	return elapsed execution time
exit	exit(3F)	terminate process with status
f77_ieee_environment	f77_ieee_environment(3F)	IEEE floating-point mode, status, and signals
f77_floatingpoint	f77_floatingpoint(3F)	IEEE floating-point definitions
fdate	fdate(3F)	return date and time in an ASCII string
fgetc	getc(3F)	get a character from a logical unit
flmax	range(3F)	return extreme values
flmin	range(3F)	return extreme values
flush	flush(3F)	flush output to a logical unit
fork	fork(3F)	create a copy of this process
fpecnt	trpfpe(3F)	trap and repair floating-point faults
fputc	putc(3F)	write a character to a FORTRAN logical unit
free	free(3F)	memory deallocator
fseek	fseek(3F)	reposition a file on a logical unit
fstat	stat(3F)	get file status
ftell	fseek(3F)	reposition a file on a logical unit
gerror	perror(3F)	get system error messages
getarg	getarg(3F)	return command line arguments
getc	getc(3F)	get a character from a logical unit
getcwd	getcwd(3F)	get pathname of current working directory
getenv	getenv(3F)	get value of environment variables
getfd	getfd(3F)	get the file descriptor of an external unit number
getgid	getuid(3F)	get user or group ID of the caller
getlog	getlog(3F)	get user's login name

getpid	getpid(3F)	get process id
getuid	getuid(3F)	get user or group ID of the caller
gmtime	time(3F)	return system time
hostnm	hostnm(3F)	get name of current host
iargc	getarg(3F)	return command line arguments
idate	idate(3F)	return date or time in numerical form
ierrno	perror(3F)	get system error messages
index	index(3F)	return index of the first occurrence of character string a2 in character string a1
inmax	range(3F)	return the maximum positive integer value
ioinit	ioinit(3F)	change f77 I/O initialization
irand	rand(3F)	return random values
isatty	ttynam(3F)	find name of a terminal port
itime	idate(3F)	return date or time in numerical form
kill	kill(3F)	send a signal to a process
len	index(3F)	return declared length of character string
libm_single	libm_single(3F)	single-precision Fortran entry points for all libm functions
libm_double	libm_double(3F)	double-precision Fortran entry points for all libm functions
link	link(3F)	make a link to an existing file
lnblnk	index(3F)	return position of last non blank in character string
loc	loc(3F)	return the address of an object
long	long(3F)	integer object conversion
lstat	stat(3F)	get file status
ltime	time(3F)	return system time
malloc	malloc(3F)	memory allocator
perror	perror(3F)	get system error messages
putc	putc(3F)	write a character to a FORTRAN logical unit
qsort	qsort(3F)	quick sort
rand	rand(3F)	return random values
rename	rename(3F)	rename a file
rindex	index(3F)	return index of the last occurrence of character string a2 in character string a1
short	long(3F)	integer object conversion
signal	signal(3F)	change the action for a signal
sleep	sleep(3F)	suspend execution for an interval
stat	stat(3F)	get file status
symlnk	link(3F)	make a link to an existing file
system	system(3F)	execute a UNIX command
tclose	topen(3F)	f77 tape I/O
time	time(3F)	return system time
topen	topen(3F)	f77 tape I/O
tread	topen(3F)	f77 tape I/O
trewin	topen(3F)	f77 tape I/O
tskipf	topen(3F)	f77 tape I/O
tstate	topen(3F)	f77 tape I/O
ttynam	ttynam(3F)	find name of a terminal port
twrite	topen(3F)	f77 tape I/O
unlink	unlink(3F)	remove a directory entry
wait	wait(3F)	wait for a process to terminate

NAME

abort – terminate abruptly with memory image

SYNOPSIS

subroutine abort (string)
character*(*) string

DESCRIPTION

Abort cleans up the I/O buffers and then aborts producing a **core** file in the current directory. If *string* is given, it is written to logical unit 0 preceded by ''**abort:**''.

FILES

/usr/lib/libF77.a

SEE ALSO

abort(3)

NAME

access – determine accessibility of a file

SYNOPSIS

integer function access (name, mode)
character*(*) name, mode

DESCRIPTION

Access checks the given file, *name,* for accessability with respect to the caller according to *mode. Mode* may include in any order and in any combination one or more of:

r	test for read permission
w	test for write permission
x	test for execute permission
(blank)	test for existence

An error code is returned if either argument is illegal, or if the file can not be accessed in all of the specified modes. 0 is returned if the specified access would be successful.

FILES

/usr/lib/libU77.a

SEE ALSO

access(2), perror(3F)

NAME

alarm – execute a subroutine after a specified time

SYNOPSIS

integer function alarm (time, proc)
integer time
external proc

DESCRIPTION

This routine arranges for subroutine *proc* to be called after *time* seconds. If *time* is ''0'', the alarm is turned off and no routine will be called. The returned value will be the time remaining on the last alarm.

FILES

/usr/lib/libU77.a

SEE ALSO

alarm(3C), sleep(3F), signal(3F)

BUGS

A subroutine cannot pass its own name to *alarm* because of restrictions in the standard.

NAME

bit – and, or, xor, not, rshift, lshift, bic, bis, bit, setbit functions

SYNOPSIS

(generic) function and (word1, word2)

(generic) function or (word1, word2)

(generic) function xor (word1, word2)

(generic) function not (word)

(generic) function rshift (word, nbits)

(generic) function lshift (word, nbits)

subroutine bic (bitnum, word)
integer*4 bitnum, word

subroutine bis (bitnum, word)
integer*4 bitnum, word

subroutine setbit (bitnum, word, state)
integer*4 bitnum, word, state

logical function bit (bitnum, word)
integer*4 bitnum, word

DESCRIPTION

The *and*, *or*, *xor*, *not*, *rshift*, and *lshift* functions are generic functions expanded inline by the compiler. Their arguments must be **integer** or **logical** values (short or long). The returned value has the data type of the first argument.

Bits are numbered such that bit 0 is the *least* significant bit, and bit 31 is the *most* significant.

and computes the bitwise 'and' of its arguments.

or computes the bitwise 'or' of its arguments.

xor computes the bitwise 'exclusive or' of its arguments.

not returns the bitwise complement of its argument.

lshift is a logical left shift with no end around carry.

rshift is an arithmatic right shift with sign extension. No test is made for a reasonable value of *nbits*.

Bic, *bis*, and *setbit* are external subroutines which operate on integer*4 arguments.

bis sets *bitnum* in *word*.

bic clears *bitnum* in *word*.

setbit sets *bitnum* in *word* to 1 if *state* is nonzero and clears it otherwise.

bit is an external function which tests *bitnum* in *word* and returns .true. if *bitnum* is a 1 (one), and returns .false. if *bitnum* is a 0 (zero).

FILES

/usr/lib/libF77.a

NAME

chdir – change default directory

SYNOPSIS

integer function chdir (dirname)
character*(*) dirname

DESCRIPTION

The default directory for creating and locating files will be changed to *dirname*. Zero is returned if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

chdir(2), cd(1), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

Use of this function may cause **inquire** by unit to fail.

Certain FORTRAN file operations reopen files by name. Using *chdir* while doing I/O may cause the runtime system to lose track of files created with relative pathnames (including files created by OPEN statements without file names).

NAME

chmod – change mode of a file

SYNOPSIS

integer function chmod (name, mode)
character*(*) name, mode

DESCRIPTION

This function changes the filesystem *mode* of file *name*. *Mode* can be any specification recognized by *chmod*(1). *Name* must be a single pathname.

The normal returned value is 0. Any other value will be a system error number.

FILES

/usr/lib/libU77.a	
/bin/chmod	exec'ed to change the mode.

SEE ALSO

chmod(1)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

etime, dtime – return elapsed execution time

SYNOPSIS

real function etime (tarray)
real tarray(2)

real function dtime (tarray)
real tarray(2)

DESCRIPTION

These two routines return elapsed runtime in seconds for the calling process. *Dtime* returns the elapsed time since the last call to *dtime*, or the start of execution on the first call.

The argument array returns user time in the first element and system time in the second element. Elapsed time, the returned value, is the sum of user and system time.

The resolution is determined by the system clock frequency.

FILES

/usr/lib/libU77.a

SEE ALSO

getrusage(2)

NAME

exit – terminate process with status

SYNOPSIS

subroutine exit (status)
integer status

DESCRIPTION

Exit flushes and closes all the process's files, and notifies the parent process if it is executing a *wait*. The low-order 8 bits of *status* are available to the parent process. (Therefore *status* should be in the range 0 – 255)

This call will never return.

The C function *exit* may cause cleanup actions before the final 'sys exit'.

FILES

/usr/lib/libF77.a

SEE ALSO

exit(2), fork(2), fork(3f), wait(2), wait(3f)

NAME

f77_floatingpoint – Fortran IEEE floating-point definitions

SYNOPSIS

#include <f77/f77_floatingpoint.h>

DESCRIPTION

This file defines constants and types used to implement standard floating-point according to ANSI/IEEE Std 754-1985. Use these constants and types to write more easily understood .F source files that will undergo automatic preprocessing prior to Fortran compilation.

IEEE Rounding Modes:

fp_direction_type The type of the IEEE rounding direction mode. Note that the order of enumeration varies according to hardware.

fp_precision_type The type of the IEEE rounding precision mode, which only applies on systems that support extended precision such as Sun-3's with 68881's.

SIGFPE handling:

sigfpe_code_type The type of a SIGFPE code.

sigfpe_handler_type The type of a user-definable SIGFPE exception handler called to handle a particular SIGFPE code.

SIGFPE_DEFAULT A macro indicating the default SIGFPE exception handling, namely for IEEE exceptions to continue with a default result, and to abort for other SIGFPE codes.

SIGFPE_IGNORE A macro indicating an alternate SIGFPE exception handling, namely to ignore and continue execution.

SIGFPE_ABORT A macro indicating an alternate SIGFPE exception handling, namely to abort with a core dump.

IEEE Exception Handling:

N_IEEE_EXCEPTION The number of distinct IEEE floating-point exceptions.

fp_exception_type The type of the N_IEEE_EXCEPTION exceptions. Each exception is given a bit number.

fp_exception_field_type The type intended to hold at least N_IEEE_EXCEPTION bits corresponding to the IEEE exceptions numbered by *fp_exception_type*. Thus *fp_inexact* corresponds to the least significant bit and *fp_invalid* to the fifth least significant bit. Some operations may set more than one exception.

IEEE Classification:

fp_class_type An enumeration of the various classes of IEEE floating-point values and symbols.

FILES

/usr/include/f77/f77_floatingpoint.h

SEE ALSO

ieee_environment(3M), f77_ieee_environment(3F)

NAME

IEEE environment - mode, status, and signal handling subprograms for IEEE arithmetic

SYNOPSIS

#include <f77/f77_floatingpoint.h>

integer function ieee_flags(action,mode,in,out)
character*(*) action, mode, in, out

integer function ieee_handler(action,exception,hdl)
character*(*) action, exception
sigfpe_handler_type hdl

sigfpe_handler_type function sigfpe(code, hdl)
sigfpe_code_type code
sigfpe_handler_type hdl

DESCRIPTION

These subprograms provide modes and status required to fully exploit ANSI/IEEE Std 754-1985 arithmetic in a Fortran program. They correspond closely to the functions *ieee_flags(3M), ieee_handler(3M),* and *sigfpe(3).*

EXAMPLES

The following examples illustrate syntax.

```
integer ieeer
character*1 mode, out, in
ieeer = ieee_flags('clearall',mode, in, out)
```

sets ieeer to 0, rounding direction to 'nearest', rounding precision to 'extended', and all accrued exception-occurred status to zero.

```
character*1 out, in
ieeer = ieee_flags('clear','direction', in, out)
```

sets ieeer to 0, and rounding direction to 'nearest'.

```
character*1 out
ieeer = ieee_flags('set','direction','tozero',out)
```

sets ieeer to 0 and the rounding direction to 'tozero' unless the hardware does not support directed rounding modes; then ieeer is set to 1.

```
character*16 out
ieeer = ieee_flags('clear','exception','all',out)
```

sets ieeer to 0 and clears all accrued exception-occurred bits. If subsequently overflow, invalid, and inexact exceptions are generated then

```
character*16 out
ieeer = ieee_flags('get','exception','overflow',out)
```

sets ieeer to 25 and out to 'overflow'.

A user-specified signal handler might look like this:

```
        integer function sample_handler ( sig, code, sigcontext )
        integer sig
        integer code
        integer sigcontext(5)
c       Sample user-written sigfpe code handler.
c       Prints a message and terminates.
c       sig .eq. SIGFPE always.
c       The structure of sigcontext is defined in <signal.h>.
        print *,’ ieee exception code ’,code,’ occurred at pc ’,sigcontext(4)
        call abort(’ ieee exception occurred ’)
        end
```

and it might be set up like this:

```
        extern sample_handler
        integer ieeer
        ieeer = ieee_handler ( ’set’, ’overflow’, sample_handler )
        if (ieeer .ne. 0) print *,’ ieee_handler can not set overflow ’
```

FILES

/usr/include/f77/f77_floatingpoint.h
/usr/lib/libm.a

SEE ALSO

floatingpoint(3), signal(3), sigfpe(3), f77_floatingpoint(3F), ieee_flags(3M), ieee_handler(3M)

NAME

fdate – return date and time in an ASCII string

SYNOPSIS

subroutine fdate (string)
character*24 string

character*24 function fdate()

DESCRIPTION

Fdate returns the current date and time as a 24 character string in the format described under *ctime*(3). Neither 'newline' nor NULL will be included.

Fdate can be called either as a function or as a subroutine. If called as a function, the calling routine must define its type and length. For example:

```
character*24  fdate
write(*,*) fdate()
```

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3), time(3F), idate(3F)

NAME

flush – flush output to a logical unit

SYNOPSIS

subroutine flush (lunit)

DESCRIPTION

Flush causes the contents of the buffer for logical unit *lunit* to be flushed to the associated file. This is most useful for logical units 0 and 6 when they are both associated with the control terminal.

FILES

/usr/lib/libI77.a

SEE ALSO

fclose(3S)

NAME

fork – create a copy of this process

SYNOPSIS

integer function fork()

DESCRIPTION

Fork creates a copy of the calling process. The only distinction between the 2 processes is that the value returned to one of them (referred to as the 'parent' process) will be the process id of the copy. The copy is usually referred to as the ‘child’ process. The value returned to the ‘child’ process will be zero.

All logical units open for writing are flushed before the fork to avoid duplication of the contents of I/O buffers in the external file(s).

If the returned value is negative, it indicates an error and will be the negation of the system error code. See perror(3F).

A corresponding *exec* routine has not been provided because there is no satisfactory way to retain open logical units across the exec. However, the usual function of *fork/exec* can be performed using *system*(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

fork(2), wait(3F), kill(3F), system(3F), perror(3F)

NAME

fseek, ftell – reposition a file on a logical unit

SYNOPSIS

integer function fseek (lunit, offset, from)
integer offset, from

integer function ftell (lunit)

DESCRIPTION

lunit must refer to an open logical unit. *offset* is an offset in bytes relative to the position specified by *from.* Valid values for *from* are:

0 meaning 'beginning of the file'
1 meaning 'the current position'
2 meaning 'the end of the file'

The value returned by *fseek* will be 0 if successful, a system error code otherwise. (See perror(3F))

Ftell returns the current position of the file associated with the specified logical unit. The value is an offset, in bytes, from the beginning of the file. If the value returned is negative, it indicates an error and will be the negation of the system error code. (See perror(3F))

FILES

/usr/lib/libU77.a

SEE ALSO

fseek(3S), perror(3F)

NAME

getarg, iargc – return command line arguments

SYNOPSIS

subroutine getarg (k, arg)
character*(*) arg

function iargc ()

DESCRIPTION

A call to *getarg* will return the **k***th* command line argument in character string *arg*. The 0*th* argument is the command name.

Iargc returns the index of the last command line argument.

FILES

/usr/lib/libU77.a

SEE ALSO

execve(2), getenv(3F)

NAME

getc, fgetc – get a character from a logical unit

SYNOPSIS

integer function getc (char)
character char

integer function fgetc (lunit, char)
character char

DESCRIPTION

These routines return the next character from a file associated with a fortran logical unit, bypassing normal fortran I/O. *Getc* reads from logical unit 5, normally connected to the control terminal input.

The value of each function is a system status code. Zero indicates no error occured on the read; −1 indicates end of file was detected. A positive value will be either a UNIX system error code or an f77 I/O error code. See perror(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

getc(3S), intro(2), perror(3F)

NAME

getcwd – get pathname of current working directory

SYNOPSIS

integer function getcwd (dirname)
character*(*) dirname

DESCRIPTION

The pathname of the default directory for creating and locating files will be returned in *dirname*. The value of the function will be zero if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

chdir(3F), perror(3F), getwd(3)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

getenv – get value of environment variables

SYNOPSIS

subroutine getenv (ename, evalue)
character*(*) ename, evalue

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *ename=value* and returns *value* in *evalue* if such a string is present, otherwise fills *evalue* with blanks.

FILES

/usr/lib/libU77.a

SEE ALSO

execve(2), environ(5)

NAME

getfd – get the file descriptor of an external unit number

SYNOPSIS

```
integer function getfd(unitn)
integer unitn
```

DESCRIPTION

Getfd returns the 'file descriptor' of an external unit number if the unit is connected and −1 otherwise.

FILES

/usr/lib/libI77.a

SEE ALSO

open(2)

NAME

getlog – get user's login name

SYNOPSIS

subroutine getlog (name)
character*(*) name

character*(*) function getlog()

DESCRIPTION

Getlog will return the user's login name or all blanks if the process is running detached from a terminal.

FILES

/usr/lib/libU77.a

SEE ALSO

getlogin(3)

NAME

getpid – get process id

SYNOPSIS

integer function getpid()

DESCRIPTION

Getpid returns the process ID number of the current process.

FILES

/usr/lib/libU77.a

SEE ALSO

getpid(2)

NAME

getuid, getgid – get user or group ID of the caller

SYNOPSIS

integer function getuid()

integer function getgid()

DESCRIPTION

These functions return the real user or group ID of the user of the process.

FILES

/usr/lib/libU77.a

SEE ALSO

getuid(2)

NAME

hostnm – get name of current host

SYNOPSIS

integer function hostnm (name)
character*(*) name

DESCRIPTION

This function puts the name of the current host into character string *name*. The return value should be 0; any other value indicates an error.

FILES

/usr/lib/libU77.a

SEE ALSO

gethostname(2)

NAME

idate, itime – return date or time in numerical form

SYNOPSIS

subroutine idate (iarray)
integer iarray(3)

subroutine itime (iarray)
integer iarray(3)

DESCRIPTION

Idate returns the current date in *iarray*. The order is: day, mon, year. Month will be in the range 1-12. Year will be ≥ 1969.

Itime returns the current time in *iarray*. The order is: hour, minute, second.

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3F), fdate(3F)

NAME

index, rindex, lnblnk, len – tell about character strings

SYNOPSIS

(intrinsic) function index (string, substr)
character*(*) string, substr

integer function rindex (string, substr)
character*(*) string, substr

function lnblnk (string)
character*(*) string

(intrinsic) function len (string)
character*(*) string

DESCRIPTION

Index (rindex) returns the index of the first (last) occurrence of the substring *substr* in *string*, or zero if it does not occur. *Index* is an f77 intrinsic function; *rindex* is a library routine.

Lnblnk returns the index of the last non-blank character in *string*. This is useful since all f77 character objects are fixed length, blank padded. Intrinsic function *len* returns the declared size of the character string argument.

FILES

/usr/lib/libF77.a

NAME

ioinit – change f77 I/O initialization

SYNOPSIS

logical function ioinit (cctl, bzro, apnd, prefix, vrbose)
logical cctl, bzro, apnd, vrbose
character*(*) prefix

DESCRIPTION

This routine will initialize several global parameters in the f77 I/O system, and attach externally defined files to logical units at run time. This connection exists only until broken; if you close the unit, then the connection no longers holds. The effect of the flag arguments applies to logical units opened after *ioinit* is called. The exception is the preassigned units, 5 and 6, to which *cctl* and *bzro* will apply at any time. *ioinit* is written in Fortran-77.

By default, carriage control is not recognized on any logical unit. If *cctl* is **.true.** then carriage control will be recognized on formatted output to all logical units except unit 0, the diagnostic channel. Otherwise the default will be restored.

By default, trailing and embedded blanks in input data fields are ignored. If *bzro* is **.true.** then such blanks will be treated as zero's. Otherwise the default will be restored.

By default, all files opened for sequential access are positioned at their beginning. It is sometimes necessary or convenient to open at the END-OF-FILE so that a write will append to the existing data. If *apnd* is **.true.** then files opened subsequently on any logical unit will be positioned at their end upon opening. A value of **.false.** will restore the default behavior.

Many systems provide an automatic association of global names with fortran logical units when a program is run. There is no such automatic association in f77. However, if the argument *prefix* is a non-blank string, then names of the form **prefix**NN will be sought in the program environment. The value associated with each such name found will be used to open logical unit NN for formatted sequential access.

For example, if the program *myprogram* has the call:

```
call ioinit ( .true., .false., .false., 'FORT', .false.)
```

then when the following sequence

```
% setenv FORT01 mydata
% setenv FORT12 myresults
% myprogram
```

would result in logical unit 1 opened to file *mydata* and logical unit 12 opened to file *myresults*. Both files would be positioned at their beginning. Any formatted output would have column 1 removed and interpreted as carriage control. Embedded and trailing blanks would be ignored on input.

If the argument *vrbose* is **.true.** then *ioinit* will report on its activity.

The effect of

```
call ioinit (.true., .true., .false., '', .false.)
```

can be achieved without the actual call by including ''–lI66'' on the *f77* command line. This gives carriage control on all logical units except 0, causes files to be opened at their beginning, and causes blanks to be interpreted as zero's.

The internal flags are stored in a labeled common block with the following definition:

```
integer*2 ieof, ictl, ibzr
```

```
common /ioiflg/ ieof, ictl, ibzr
```

FILES

/usr/lib/libI77.a	f77 I/O library
/usr/lib/libI66.a	sets older fortran I/O modes

SEE ALSO

getarg(3F), getenv(3F), ''Introduction to the f77 I/O Library''

BUGS

Prefix can be no longer than 30 characters. A pathname associated with an environment name can be no longer than 255 characters.

The ''+'' carriage control does not work.

NAME

kill – send a signal to a process

SYNOPSIS

function kill (pid, signum)
integer pid, signum

DESCRIPTION

Pid must be the process id of one of the user's processes. *Signum* must be a valid signal number (see signal(3)). The returned value will be 0 if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

kill(2), signal(3), signal(3F), fork(3F), perror(3F)

NAME

libm_double - Double-precision Fortran access to libm functions

SYNOPSIS

```
#include <f77/f77_floatingpoint.h>

doubleprecision function d_acosh (x)
doubleprecision function d_asinh (x)
doubleprecision function d_atanh (x)
doubleprecision function d_cbrt (x)
doubleprecision function d_ceil (x)
fp_class_type function id_fp_class (x)
doubleprecision function d_copysign (x,y)
doubleprecision function d_erf (x)
doubleprecision function d_erfc (x)
doubleprecision function d_expm1 (x)
integer function id_finite (x)
doubleprecision function d_floor (x)
doubleprecision function d_hypot (x,y)
integer function id_ilogb (x)
integer function id_irint (x)
integer function id_isinf (x)
integer function id_isnan (x)
integer function id_isnormal (x)
integer function id_issubnormal (x)
integer function id_iszero (x)
doubleprecision function d_infinity ()
doubleprecision function d_j0 (x)
doubleprecision function d_j1 (x)
doubleprecision function d_jn (n,x)
doubleprecision function d_lgamma (x)
doubleprecision function d_logb (x)
doubleprecision function d_log1p (x)
doubleprecision function d_log2 (x)
doubleprecision function d_max_normal ()
doubleprecision function d_max_subnormal ()
doubleprecision function d_min_normal ()
doubleprecision function d_min_subnormal ()
doubleprecision function d_nextafter (x,y)
doubleprecision function d_quiet_nan (n)
doubleprecision function d_remainder (x,y)
doubleprecision function d_rint (x)
doubleprecision function d_scalb (x,y)
doubleprecision function d_scalbn (x,n)
doubleprecision function d_signaling_nan (n)
integer function id_signbit (x)
doubleprecision function d_significand (x)
subroutine d_sincos(x, s, c)
doubleprecision function d_y0 (x)
doubleprecision function d_y1 (x)
doubleprecision function d_yn (n,x)

doubleprecision x, y, s, c
integer n
```

DESCRIPTION

These functions provide access to double-precision *libm* functions that do not correspond to standard Fortran generic intrinsic functions.

FILES

/usr/lib/libm.a

SEE ALSO

intro(3M)

NAME

libm_single - Single-precision Fortran access to libm functions

SYNOPSIS

```
#include <f77/f77_floatingpoint.h>

real function r_acosh (x)
real function r_asinh (x)
real function r_atanh (x)
real function r_cbrt (x)
real function r_ceil (x)
fp_class_type function ir_fp_class (x)
real function r_copysign (x,y)
real function r_erf (x)
real function r_erfc (x)
real function r_expm1 (x)
integer function ir_finite (x)
real function r_floor (x)
real function r_hypot (x,y)
integer function ir_ilogb (x)
integer function ir_irint (x)
integer function ir_isinf (x)
integer function ir_isnan (x)
integer function ir_isnormal (x)
integer function ir_issubnormal (x)
integer function ir_iszero (x)
real function r_infinity ()
real function r_j0 (x)
real function r_j1 (x)
real function r_jn (n,x)
real function r_lgamma (x)
real function r_logb (x)
real function r_log1p (x)
real function r_log2 (x)
real function r_max_normal ()
real function r_max_subnormal ()
real function r_min_normal ()
real function r_min_subnormal ()
real function r_nextafter (x,y)
real function r_quiet_nan (n)
real function r_remainder (x,y)
real function r_rint (x)
real function r_scalb (x,y)
real function r_scalbn (x,n)
real function r_signaling_nan (n)
integer function ir_signbit (x)
real function r_significand (x)
subroutine r_sincos(x, s, c)
real function r_y0 (x)
real function r_y1 (x)
real function r_yn (n,x)

real x, y, s, c
integer n
```

DESCRIPTION

These functions provide access to single-precision *libm* functions that do not correspond to standard Fortran generic intrinsic functions.

FILES

/usr/lib/libm.a

SEE ALSO

intro(3M), single_precision(3M)

NAME

link, symlnk – make a link to an existing file

SYNOPSIS

function link (name1, name2)
character*(*) name1, name2

integer function symlnk (name1, name2)
character*(*) name1, name2

DESCRIPTION

Name1 must be the pathname of an existing file. *Name2* is a pathname to be linked to file *name1*. *Name2* must not already exist. The returned value will be 0 if successful; a system error code otherwise.

Symlnk creates a symbolic link to *name1*.

FILES

/usr/lib/libU77.a

SEE ALSO

link(2), symlink(2), perror(3F), unlink(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

loc – return the address of an object

SYNOPSIS

function loc (arg)

DESCRIPTION

The returned value will be the address of *arg*.

FILES

/usr/lib/libU77.a

NAME

long, short – integer object conversion

SYNOPSIS

integer*4 function long (int2)
integer*2 int2

integer*2 function short (int4)
integer*4 int4

DESCRIPTION

These functions provide conversion between short and long integer objects. *Long* is useful when constants are used in calls to library routines and the code is to be compiled with '–i2'. *Short* is useful in similar context when an otherwise long object must be passed as a short integer.

FILES

/usr/lib/libF77.a

NAME

perror, gerror, ierrno – get system error messages

SYNOPSIS

subroutine perror (string)
character*(*) string

subroutine gerror (string)
character*(*) string

character*(*) function gerror()

function ierrno()

DESCRIPTION

Perror will write a message to fortran logical unit 0 appropriate to the last detected system error. *String* will be written preceding the standard error message.

Gerror returns the system error message in character variable *string*. *Gerror* may be called either as a subroutine or as a function.

Ierrno will return the error number of the last detected system error. This number is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call; that value is a more reliable indicator of what caused the error condition.

FILES

/usr/lib/libU77.a

SEE ALSO

intro(2), perror(3), ‘‘Introduction to the f77 I/O Library’’

BUGS

String in the call to *perror* can be no longer than 127 characters.

The length of the string returned by *gerror* is determined by the calling program.

NOTES

UNIX system error codes are described in *intro*(2). The f77 I/O error codes and their meanings are:

100	‘‘error in format’’
101	‘‘illegal unit number’’
102	‘‘formatted io not allowed’’
103	‘‘unformatted io not allowed’’
104	‘‘direct io not allowed’’
105	‘‘sequential io not allowed’’
106	‘‘can’t backspace file’’
107	‘‘off beginning of record’’
108	‘‘can’t stat file’’
109	‘‘no * after repeat count’’
110	‘‘off end of record’’
111	‘‘truncation failed’’
112	‘‘incomprehensible list input’’
113	‘‘out of free space’’
114	‘‘unit not connected’’
115	‘‘read unexpected character’’
116	‘‘blank logical input field’’
117	‘‘’new’ file exists’’
118	‘‘can’t find ’old’ file’’
119	‘‘unknown system error’’

120	‘‘requires seek ability’’
121	‘‘illegal argument’’
122	‘‘negative repeat count’’
123	‘‘illegal operation for unit”
124	‘‘too many files open - no free descriptors’’
125	‘‘attempted operation on unit that's not open’’
126	‘‘illegal input for namelist”

NAME

putc, fputc – write a character to a FORTRAN logical unit

SYNOPSIS

integer function putc (char)
character char

integer function fputc (lunit, char)
character char

DESCRIPTION

These funtions write a character to the file associated with a FORTRAN logical unit bypassing normal FORTRAN I/O. *Putc* writes to logical unit 6, normally connected to the control terminal output.

The value of each function will be zero unless some error occurred; a system error code otherwise. See perror(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

putc(3S), intro(2), perror(3F)

NAME

qsort – quick sort

SYNOPSIS

subroutine qsort (array, len, isize, compar)
external compar
integer*2 compar

DESCRIPTION

One dimensional *array* contains the elements to be sorted. *len* is the number of elements in the array. *isize* is the size of an element, typically -

4 for **integer** and **real**
8 for **double precision** or **complex**
16 for **double complex**
(length of character object) for **character** arrays

Compar is the name of a user supplied integer*2 function that will determine the sorting order. This function will be called with 2 arguments that will be elements of *array*. The function must return -

negative if arg 1 is considered to precede arg 2
zero if arg 1 is equivalent to arg 2
positive if arg 1 is considered to follow arg 2

On return, the elements of *array* will be sorted.

FILES

/usr/lib/libU77.a

SEE ALSO

qsort(3)

NAME

rand, drand, irand – return random values

SYNOPSIS

function irand (iflag)

function rand (iflag)

double precision function drand (iflag)

integer*4 iflag

DESCRIPTION

These functions use *random*(3) to generate sequences of random numbers.
If *iflag* is '0', the generator returns the next random number in the sequence.
If *iflag* is '1', the generator is restarted and the first random value is returned.
If *iflag* is otherwise non-zero, it is used as a new seed for the random number generator, and the first new random value is returned. The three functions share the same 256 byte state array.

Irand returns positive integers in the range 0 through 2147483647. *Rand* and *drand* return values in the range 0.0 through 1.0 .

FILES

/usr/lib/libF77.a

SEE ALSO

random(3)

NAME

inmax – return maximum positive integer

SYNOPSIS

function inmax()

DESCRIPTION

Function *inmax* returns the maximum positive integer value, 2147483647 .

FILES

/usr/lib/libF77.a

SEE ALSO

libm_single(3f), libm_double(3f).

NAME

rename – rename a file

SYNOPSIS

integer function rename (from, to)
character*(*) from, to

DESCRIPTION

From must be the pathname of an existing file. *To* will become the new pathname for the file. If *to* exists, then both *from* and *to* must be the same type of file, and must reside on the same filesystem. If *to* exists, it will be removed first.

The returned value will be 0 if successful; a system error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

rename(2), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

signal – change the action for a signal

SYNOPSIS

integer function signal(signum, proc, flag)
integer signum, flag
external proc

DESCRIPTION

When a process incurs a signal (see *signal*(3)) the default action is usually to clean up and abort. The user may choose to write an alternative signal handling routine. A call to *signal* is the way this alternate action is specified to the system.

Signum is the signal number (see *signal*(3)). If *flag* is negative, then *proc* must be the name of the user signal handling routine. If *flag* is zero or positive, then *proc* is ignored and the value of *flag* is passed to the system as the signal action definition. In particular, this is how previously saved signal actions can be restored. Two possible values for *flag* have specific meanings: 0 means "use the default action" (See NOTES below), 1 means "ignore this signal".

A positive returned value is the previous action definition. A value greater than 1 is the address of a routine that was to have been called on occurrence of the given signal. The returned value can be used in subsequent calls to *signal* in order to restore a previous action definition. A negative returned value is the negation of a system error code. (See *perror*(3F))

FILES

/usr/lib/libU77.a

SEE ALSO

kill(1), signal(3), kill(3F)

NOTES

f77 arranges to trap certain signals when a process is started. The only way to restore the default **f77** action is to save the returned value from the first call to *signal.*

If the user signal handler is called, it will be passed the signal number as an integer argument.

NAME

sleep – suspend execution for an interval

SYNOPSIS

subroutine sleep (itime)

DESCRIPTION

Sleep causes the calling process to be suspended for *itime* seconds. The actual time can be up to 1 second less than *itime* due to granularity in system timekeeping.

FILES

/usr/lib/libU77.a

SEE ALSO

sleep(3)

NAME

stat, lstat, fstat – get file status

SYNOPSIS

integer function stat (name, statb)
character*(*) name
integer statb(13)

integer function lstat (name, statb)
character*(*) name
integer statb(13)

integer function fstat (lunit, statb)
integer statb(13)

DESCRIPTION

These routines return detailed information about a file.
Stat and *lstat* do the inquiry by *filename*.
fstat does the inquiry by FORTRAN logical *lunit*.
The value of each function is zero if successful, and an error code otherwise.
The variable 'statb' receives the stat structure for the file.

Calling Sequences:

stat :
integer stat, statb(13)
character name*(*)
ierr = stat (name, statb)

fstat :
integer fstat, logunit, statb(13)
ierr = fstat (logunit, statb)

lstat :
integer lstat, statb(13)
character name*(*)
ierr = lstat (name, statb)

The meaning of the information returned in array *statb* is as described for the structure *stat* under *stat*(2). Spare values are not included. The order is shown below:

statb(1)	device inode resides on
statb(2)	this inode's number
statb(3)	protection
statb(4)	number of hard links to the file
statb(5)	user-id of owner
statb(6)	group-id of owner
statb(7)	the device type, for inode that is device
statb(8)	total size of file
statb(9)	file last access time
statb(10)	file last modify time
statb(11)	file last status change time
statb(12)	optimal blocksize for file system i/o ops
statb(13)	actual number of blocks allocated

FILES

/usr/lib/libU77.a

SEE ALSO

stat(2), access(3F), perror(3F), time(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

system – execute a UNIX command

SYNOPSIS

integer function system (string)
character*(*) string

DESCRIPTION

System causes *string* to be given to your shell as input as if the string had been typed as a command. If environment variable **SHELL** is found, its value will be used as the command interpreter (shell); otherwise *sh*(1) is used.

The current process waits until the command terminates. The returned value will be the exit status of the shell. See *wait*(2) for an explanation of this value.

FILES

/usr/lib/libU77.a

SEE ALSO

execve(2), wait(2), system(3)

BUGS

String can not be longer than NCARGS–50 characters, as defined in <sys/param.h>.

NAME

time, ctime, ltime, gmtime – return system time

SYNOPSIS

integer function time()

character*24 function ctime (stime)
integer*4 stime

subroutine ltime (stime, tarray)
integer*4 stime, tarray(9)

subroutine gmtime (stime, tarray)
integer*4 stime, tarray(9)

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. This is the value of the UNIX system clock.

Ctime converts a system time to a 24 character ASCII string. The format is described under *ctime*(3). No 'newline' or NULL will be included.

Ltime and *gmtime* dissect a UNIX time into month, day, etc., either for the local time zone or as GMT. The order and meaning of the 9 elements returned in *tarray* is described under *ctime*(3).

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3), idate(3F), fdate(3F)

NAME

topen, tclose, tread, twrite, trewin, tskipf, tstate – f77 tape I/O

SYNOPSIS

integer function topen (tlu, devnam, label)
integer tlu
character*(*) devnam
logical label

integer function tclose (tlu)
integer tlu

integer function tread (tlu, buffer)
integer tlu
character*(*) buffer

integer function twrite (tlu, buffer)
integer tlu
character*(*) buffer

integer function trewin (tlu)
integer tlu

integer function tskipf (tlu, nfiles, nrecs)
integer tlu, nfiles, nrecs

integer function tstate (tlu, fileno, recno, errf, eoff, eotf, tcsr)
integer tlu, fileno, recno, tcsr
logical errf, eoff, eotf

DESCRIPTION

These functions provide a simple interface between f77 and magnetic tape devices. A ''tape logical unit'', *tlu*, is ''topen''ed in much the same way as a normal f77 logical unit is ''open''ed. All other operations are performed via the *tlu*. The *tlu* has no relationship at all to any normal f77 logical unit.

Topen associates a device name with a *tlu*. *Tlu* must be in the range 0 to 3. The logical argument *label* should indicate whether the tape includes a tape label. This is used by *trewin* below. *Topen* does not move the tape. The normal returned value is 0. If the value of the function is negative, an error has occured. See *perror*(3f) for details.

Tclose closes the tape device channel and removes its association with *tlu*. The normal returned value is 0. A negative value indicates an error.

Tread reads the next physical record from tape to *buffer*. *Buffer* **must** be of type **character**. The size of *buffer* should be large enough to hold the largest physical record to be read. The actual number of bytes read will be returned as the value of the function. If the value is 0, the end-of-file has been detected. A negative value indicates an error.

Twrite writes a physical record to tape from *buffer*. The physical record length will be the size of *buffer*. *Buffer* **must** be of type **character**. The number of bytes written will be returned. A value of 0 or negative indicates an error.

Trewin rewinds the tape associated with *tlu* to the beginning of the first data file. If the tape is a labelled tape (see *topen* above) then the label is skipped over after rewinding. The normal returned value is 0. A negative value indicates an error.

Tskipf allows the user to skip over files and/or records. First, *nfiles* end-of-file marks are skipped. If the current file is at EOF, this counts as 1 file to skip. (Note: This is the way to reset the EOF status for a *tlu*.) Next, *nrecs* physical records are skipped over. The normal returned value is 0. A negative value indicates an error.

Finally, *tstate* allows the user to determine the logical state of the tape I/O channel and to see the tape drive control status register. The values of *fileno* and *recno* will be returned and indicate the current file and record number. The logical values *errf*, *eoff*, and *eotf* indicate an error has occurred, the current file is at EOF, or the tape has reached logical end-of-tape. End-of-tape (EOT) is indicated by an empty file, often referred to as a double EOF mark. It is not allowed to read past EOT although it is allowed to write. The value of *tcsr* will reflect the tape drive control status register. See *tm*(4S) for details.

FILES

/usr/lib/libU77.a

SEE ALSO

tm(4S), perror(3f)

NAME

ttynam, isatty – find name of a terminal port

SYNOPSIS

character*(*) function ttynam (lunit)

logical function isatty (lunit)

DESCRIPTION

Ttynam returns a blank padded path name of the terminal device associated with logical unit *lunit*.

Isatty returns **.true.** if *lunit* is associated with a terminal device, **.false.** otherwise.

FILES

/dev/*
/usr/lib/libU77.a

DIAGNOSTICS

Ttynam returns an empty string (all blanks) if *lunit* is not associated with a terminal device in directory '/dev'.

NAME

unlink – remove a directory entry

SYNOPSIS

integer function unlink (name)
character*(*) name

DESCRIPTION

Unlink causes the directory entry specified by pathname *name* to be removed. If this was the last link to the file, the contents of the file are lost. The returned value will be zero if successful; a system error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

unlink(2), link(3F), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

wait – wait for a process to terminate

SYNOPSIS

integer function wait (status)
integer status

DESCRIPTION

Wait causes its caller to be suspended until a signal is received or one of its child processes terminates. If any child has terminated since the last *wait,* return is immediate; if there are no children, return is immediate with an error code.

If the returned value is positive, it is the process ID of the child and *status* is its termination status (see *wait*(2)). If the returned value is negative, it is the negation of a system error code.

FILES

/usr/lib/libU77.a

SEE ALSO

wait(2), signal(3F), kill(3F), perror(3F)

Index

Revision History

Version	Date	Comments
1	12 November 1986	α release preceding Release 4.0 FCS.
50	15 May 1987	Beta Release Notes Only.
A	4 Sept 1987	FCS Release Notes Only.
A	11 March 1988	1.1 4.0 alpha - full manual
A	21 March 1988	1.1 4.0 beta - full manual
A	6 May 1988	1.1 4.0 FCS full manual

Corporate Headquarters
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 37-29639

For U.S. Sales Office locations, call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Sun House
31-41 Pembroke Broadway
Camberley
Surrey GU15 3XD
England
0276 62111
TLX 859017

Australia: (02) 436 4699
Canada: 416 477-6745
France: (1) 46 30 23 24
Germany: (089) 95094-0
Japan: (03) 221-7021
Nordic Countries: (08) 764 78 10
Switzerland: (1) 82 89 555
The Netherlands: 02155 24888
UK: 0276 62111

Europe, Middle East, and Africa, call European Headquarters:
0276 62111

Elsewhere in the world, call Corporate Headquarters:
415 960-1300
Intercontinental Sales