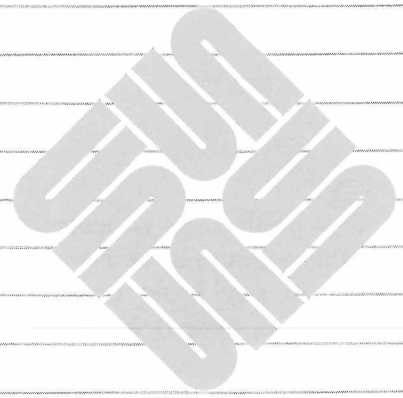
sun microsystems

FORTRAN Programmer's Guide *for the* Sun Workstation



Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1175-01
Revision D of 15 May, 1985

Credits and Acknowledgements

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.

Trademarks

Revision History

Version	Date	Comments
A	15 July 1983	First release of this Programmer's Guide.
B	1 November 1983	Incorporates corrections.
C	7 January 1984	Reorganized and some extra material added.
D α	19 November 1984	2.0 α release.
D β	5 February l985	2.0 β release.
D	15 May l985	2.0 release.

Contents

Contents

Tables

Figures

Preface

Purpose and Audience

This Programmer's Guide gives information you need to write FORTRAN programs on the Sun Workstation. It contains information useful to those who already know FORTRAN but have little familiarity with UNIX† and off, and enough basic commands to find your way around the UNIX file system. To refresh your memory of these basics, refer to the ***Beginner's Guide to the Sun Workstation*** or an introductory UNIX book. Also, refer to Appendix B for a summary of the differences between FORTRAN 66 and FORTRAN 77.

Conventions in Examples

Note the following conventions used in this manual to display information. After logging in, the Sun UNIX system prompt looks something like this:

```
hostname%
```

The hostname is different for every Sun Workstation. The basic UNIX prompt is merely the percent sign (%). However, most Sun Workstations have distinct hostnames and our examples are more easily distinguished if we use a symbol longer than a % sign. Hence, the examples in this manual use `hostname%` to denote the system prompt.

The system's 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`**.

```
hostname% echo hello
hello
hostname%
```

Organization

The manual is organized as follows:

Chapter 1 is an introduction to FORTRAN programming on the Sun Workstation. It describes how to gain access to the FORTRAN compiler, indicates tools available to the programmer, and lists helpful related documents.

Chapter 2 deals with maintaining FORTRAN source and object files. It contains more detailed information on using the compiler and its options. This chapter briefly discusses **make**, a tool for compiling large programs contained in multiple source files. It also describes how to maintain

† UNIX is a trademark of Bell Laboratories.

FORTRAN programs with **SCCS** (Source Code Control System) and how to build and use libraries.

Chapter 3 describes input and output (I/O) and other run-time environment issues. It covers gaining access to named and unnamed files and I/O devices.

Chapter 4 describes FORTRAN 77 representations of data in storage. This information is necessary for writing C, Pascal and FORTRAN 77 routines that can communicate with each other. In addition, useful run-time variables are discussed.

Chapter 5 describes debugging tools and their use.

Chapter 6 is a summary of deviations from the ANSI standard for FORTRAN 77. These deviations consist of extensions and violations.

Chapter 7 contains a brief description of the differences between FORTRAN 66 and FORTRAN 77.

Appendix A is a summary of the *Ratfor* language.

Appendix B is a table of the ASCII character set.

Appendix C is a list of I/O library error messages.

Appendix D is a bibliography.

Appendix E contains the manual pages for FORTRAN library routines from the *System Interface Manual for the Sun Workstation.*

Chapter 1

Introduction

The Sun Workstation provides a FORTRAN 77 compiler with several enhancements. For example, variable names can be up to 16 characters long, but they must still begin with a letter. Sun FORTRAN also supports recursion. These enhancements are described in Appendix A, "Deviations From the FORTRAN 77 Standard."

The Sun FORTRAN compiler is invoked with the command

```
hostname% f77
```

It implements the American National Standard (ANSI) of 1978 for FORTRAN. FORTRAN 77 includes most of the features of the 1966 standard (also called FORTRAN IV) plus new features such as the character data type, direct I/O, and internal I/O.

In addition to the *f77* compiler, other 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 and Text Processing*** manual section on ***vi***.

FORTRAN Tools

fpr is a FORTRAN 'output filter' for printing files that have FORTRAN carriage-control characters in column one. As noted in Appendix A, describing deviations from the ANSI standard, the UNIX implementation on the Sun system does not use carriage control since there are no explicit printer files. Thus, you use ***fpr*** when you want to transform files formatted with FORTRAN carriage control conventions into files formatted according to UNIX line printer conventions. For more information on ***fpr***, refer to the ***User's Manual for the Sun Workstation.***

Ratfor is 'Rational FORTRAN' — a preprocessor intended to add some control structures to FORTRAN that are similar to those in C. *Ratfor* was written in the days of FORTRAN 66 and is not as useful for FORTRAN 77, which has better control structures.

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

dbx is an interactive symbolic debugger that understands FORTRAN 77 programs.

dbxtool is a window- and mouse-based version of ***dbx***.

adb is an interactive, general purpose low-level debugger — it is not as easy to use as ***dbx***.

The online documentation consists of pages from the ***User's Manual*** that are called 'man pages'. The most commonly used pages for FORTRAN are:

- *f77*(1)
- *fpr*(1)
- *ratfor*(1)
- *fsplit*(1)
- *dbx*(1)
- *dbxtool*(1)

Also, see the manual pages in Section (3f) of the ***User's Manual for the Sun Workstation*** for other FORTRAN routines. ***f77*** invokes the FORTRAN compiler; ***fpr*** and ***ratfor*** are FORTRAN tools briefly explained above. ***fsplit*** splits a multi-routine FORTRAN file into individual files.

Other Sun manuals containing information on editing or using FORTRAN are

- ***Editing and Text Processing on the Sun Workstation***
- ***Programming Tools for the Sun Workstation***
- ***Commands Reference Manual for the Sun Workstation***
- ***System Interface Manual for the Sun Workstation***

Chapter 2

Developing and Maintaining FORTRAN programs

2.1. Using the FORTRAN 77 Compiler on the Sun Workstation

Creating, compiling, and running a FORTRAN 77 program on the Sun Workstation requires three steps:

1. Write a program in the FORTRAN 77 language using an editor. Give the file a *.f* suffix.
2. Compile the program using the *f77* command.
3. Run the program by typing the name of the executable output file.

The previous chapter contains information about tools you can use to create your FORTRAN program. Once you have created a FORTRAN 77 source file and named it ***filename.f***, invoke the compiler using the *f77* command. The specified files are then compiled, and object files are generated having the same names as the source files, but with the suffix `.o` appended in place of `.f`. For example, *f77* compiles `greetings.f` and puts the resulting object code result into a file named `greetings.o`. Finally, *f77* calls the UNIX linker to create an executable file with the name (by default) `a.out`. *f77* also understands other filename extensions (such as `.r` for Ratfor files — these topics are discussed later in this chapter).

For example, here is a very simple FORTRAN 77 program that displays a message on the workstation screen.

```
program  greetings
print *, 'Real programmers hack FORTRAN!'
end
```

Note: Remember to begin typing the source code in at least column seven by spacing over or using tabs.

2.2. Compiling and Running Your Program

Compile the program *greetings* using the *f77* command like this:

```
hostname% f77 greetings.f
greetings.f:
   MAIN greetings:
hostname%
```

Note that *f77* displays a message indicating the stage of the compilation. If you do not specify an output filename at compilation, the results end up in an executable file called *a.out*. You can then run that program by typing *a.out* on the command line:

```
hostname% a.out
  Real programmers hack FORTRAN!
hostname%
```

It is inconvenient to have the results of every FORTRAN 77 compilation end up in a file called `a.out`, since if such a file already exists, it is overwritten. To solve this problem, you can

- change the name of `a.out` after each compilation, using the *mv* command
- tell the *f77* compiler to place the executable file in a different file (such as one with the same name as the source minus the *.f* suffix). For example,

```
hostname% f77 -o greetings  greetings.f
greetings.f:
   MAIN greetings:
hostname%
```

places the executable file into `greetings`. Run the program by typing:

```
hostname% greetings
  Real programmers hack FORTRAN!
hostname%
```

The remainder of this chapter discusses the kinds of files that *f77* understands, the options that you may type on the *f77* command line, and other topics such as Makefiles and using the Source Code Control System (SCCS).

2.3. Source Files that *f77* Understands

f77 is a general-purpose 'driver' command for compiling and loading FORTRAN 77 and FORTRAN-related files. As mentioned above, FORTRAN 77 source code is contained in files having a *.f* suffix. Table 2-1 summarizes the filename extensions that *f77* understands.

Table 2-1: Filename Suffixes that *f77* Understands

Suffix	Language	Action
.f	FORTRAN 77	FORTRAN 77 source programs are compiled, and the object program is left in a file in the current directory whose name is that of the source with *.o* substituted for *.f*.
.F	FORTRAN 77	FORTRAN 77 source programs are processed by the C preprocessor before being compiled 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 77 programs need a few extra libraries and a different startup routine than do C programs.
.r	Ratfor	The Ratfor preprocessor processes source files and *f77* compiles the results.
.s	Assembler	The assembler processes assembly-language source files.
.o	Object Files	Object files are passed through to the linker.

Note: Files without the above filename extensions are simply passed to the loader.

2.4. Source Input Format

The *f77* compiler accepts two kinds of source lines: standard and UNIX–style.

2.4.1. Standard Source Lines

The standard source lines are in the format specified in the FORTRAN Standard:

- the first 72 characters of each line are scanned
- the first five columns must be blank or contain a numeric label
- column 6 is nonblank if the line is a continuation of the previous line or lines. *f77* pads such lines to 72 characters or truncates them as required.

Padding is significant in lines such as:

```
          1         2         3         4         5         6         7
C2345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012

      data sixtyh/60h
     1         /
```

A procedure can contain both kinds of lines, but each statement can contain only one kind.

2.4.2. UNIX Source Lines

A tab in columns 1-5 marks the beginning of a UNIX -style source line. The text following the tab is scanned as if it started in column 7. The line may be arbitrarily long. Continuation lines are identified by an ampersand (&) in column 1.

2.5. Source File Content

The FORTRAN language places no significance on whether compilation units, main programs, functions or subroutines reside in the same or different source files. An *f77* input file can contain any number of compilation units. However, there are two good reasons to keep each compilation unit in a separate source file. The first reason is to reduce the compilation overhead of changing one procedure. The second reason is to minimize loading of unreferenced functions.

f77 produces one *.o* file for each *.f* file it processes. If any routine in the *.o* file is referenced, the loader *ld* copies in the entire *.o* file. For example, if the file `subs.f` defines subroutines `a` and `b`, and the file `main.f` contains a main program that calls subroutine `a` but not `b`, then the `a.out` file produced by

```
hostname% f77 main.f sub.f
hostname%
```

contains the code for subroutine `b` even though the subroutine is not referenced. The *fsplit* command can be used to break up multiple-routine source files.

A final consideration in maintaining FORTRAN source files, is to maintain source in lower-case form. The *f77* compiler converts keywords and variables to lower case (unless the **-U** flag is set), but does not translate characters inside strings. Thus, tests of the following form fail:

```
      CHARACTER ANSWER*15
      INQUIRE (6, SEQUENTIAL=ANSWER)
      IF (ANSWER.NE.'YES') STOP 99
99    END
```

The `tr` command can be used to translate a source file from upper case to lower case or vice versa. For example,

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

2.6. Options to the *f77* Command

The list below contains the options that *f77* understands.

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

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

-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).

-F Apply the Ratfor preprocessor to relevant files and put the result in the file with the suffix changed to . *f*, but do not compile.

-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. Programs compiled without the **-fsky** option use the SKY board, but won't run as fast. If any part of a program is compiled using the **-fsky** option, you must also use this option when loading with the *f77* command, since a different set of startup routines is required.

-g Produce additional symbol table information for *dbx* (1). Also pass the **-lg** file to *ld* (1).

-I*dir*
Search first for '#include' files whose names do not begin with '/' in the directory of 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 preprocessor's.

-i2
Make the default integer and logical constants and variables **short**.

-m Apply the M4 preprocessor to each *.r* file before transforming it with the Ratfor preprocessor.

-N[**qxscn**]*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:

- **q** Maximum number of equivalenced variables. The default is 150.
- **x** Maximum number of external names (common block names, subroutine and function names). The default is 200.
- **s** Maximum number of statement numbers. The default is 401.
- **c** Maximum depth of nesting for control statements (for example, DO loops). The default is 20.
- **n** Maximum number of identifiers. The default is 1009.

-O Optimize the object code.

---

1 Sky is a trademark of SKY Computers, Inc.

-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. FORTRAN 77 DO loops are not performed at all if the upper limit is smaller than the lower limit.

-p Prepare object files for profiling, see *prof*(1).

-pg
Produce counting code in the manner of **-p**, but invoke a run-time 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).

-R*x*
Use the string *x* as a Ratfor option in processing *.r* files.

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

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

-u Make the default type of a variable 'undefined' rather than using FORTRAN implicit typing.

-v Print the version number of the compiler and the name of each pass as the compiler executes.

-w Suppress all warning messages.

-w66
Suppress 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 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* .

Other flags, all library names (arguments beginning with **-lib**), and any names not ending with one of the understood suffixes are passed to the linker.

2.6.1. Language Preprocessors

cpp is the C language preprocessor, which is invoked during the first pass of a FORTRAN compilation if the source filename has the extension **.F**. The main uses of this preprocessor for FORTRAN programs are for constant definitions and conditional compilation. The details on *cpp* syntax and options are found in *cpp*(1). (Also see the **D***name* option in "Options to the *f77* Command.")

M4 is UNIX macro processor that is primarily used on Ratfor programs before transforming them with the Ratfor preprocessor. Files must have an *.r* extension. For details about its usage see the section called "M4 — a Macro Processor" in the ***Programming Tools for the Sun Workstation.***

2.7. Managing Program Builds With make

make is a program that manages the process of building big programs or libraries.

When you develop programs that depend on only a single source file and possibly a few system-supplied libraries, you simply need to run the compiler every time you change the program. Even with the simplest compilation, the *f77* command line can involve a lot of typing. If it contains long lists of option flags or libraries, the command line can also be hard to remember.

To save some time, you can create a simple shell script or *csh* alias to compile the source for you every time. For instance, to compile a small program contained in the file **example.f**, that uses the **core** library, you could write a shell script called **fexample** that contains just one line:

```
f77 example.f -libcore77 -libcore -o example
```

Whenever you want to recompile **example.f**, you only have to type:

```
hostname% fexample
hostname%
```

But when you are developing programs made from multiple source files, such simple methods are insufficient. You need to remember which files have been edited since the last time they were compiled, and compile only those files. Then link together the resulting relocatable files along with any libraries you use into a program file.

If you forget to recompile even one of the files that has been edited, the object will be inconsistent with the source. But if you recompile your whole program after every editing session, you waste time, since not every source file needs recompiling. To help you recompile only what needs compiling, use the program **make**.

The features of **make** are fully discussed in the chapter "Make — a Program for Maintaining Computer Programs" in the *Programming Tools* manual, and are summarized in the *Sun User's Manual* on page **make**(1). This section shows you how **make** is normally used to maintain large FORTRAN programs, and provides a simple example.

2.7.1. Use

In order for **make** to help you maintain consistent programs, you must tell it what files depend on other files, and what to do in order to transform one object into another. You encode this information into a file called the **Makefile** in the directory where you are developing the program.

When **make** is invoked with no arguments, it looks for a file named **makefile** or **Makefile** in the current directory, and causes the first program or file for which it finds a dependency list to be "made," or in other words, created. (Most people prefer to use the name **Makefile**, because it is easier to find in the alphabetized output of **ls**.)

Suppose that you have a simple program of four files: **pattern.f**, **computepts.f**, **startupcore.f**, and **commonblock**. Assume that **commonblock** is included by **pattern.f** and **computepts.f**, and that you wish to compile them into a program called *pattern*. The **make** paradigm for such simple programs is that programs are made from, and thus depend on, relocatable (.o) files. And, relocatable files are made from, and thus depend on, the corresponding source files and any included files. The dependencies for this example are

shown in this table:

Table 2-2: Example Makefile Targets and Dependencies

Target	Depends on
pattern	pattern.o, computepts.o, startupcore.o
pattern.o	pattern.f, commonblock
computepts.o	computepts.f, commonblock
startupcore.o	startupcore.f

Furthermore, the program ***pattern*** is made by linking together the three relocatable files (plus a series of libraries). Each FORTRAN source file compilation produces corrresponding relocatable files. The Makefile to express this looks like:

```
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
```

The model for a `Makefile` entry is as follows:

- The first line of an entry begins with a list of target files, separated by blank spaces.
- The targets are followed by a colon (:) and a list of the files the targets depend on.
- The second and subsequent lines are shell command lines, each indented by a tab character.
- The execution of these lines causes the target file to be brought up-to-date with the files it depends on.

Since the command lines executed in order to create the target file are arbitrary shell commands, they can do much more than simple compilation. To continue our example, 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 to your program that looks like:

```
if (argstring .eq. "-v") then
    print *,COMPILETIME
    call exit(0)
endif
```

and then use the C preprocessor to define the word COMPILETIME as a quoted string that can be printed. The output of the preprocessor, for example, might be

```
print *, "jan15..."
```

To do this, you must also change the name of the source file containing this code to `pattern.F`, so the C preprocessor runs over it. We also change the compilation line for

`pattern.F` in the Makefile to look like this:

```
f77 "-DCOMPILETIME=\"`date`\"" -c -u pattern.F
```

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 set 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*.

Note that this example is for illustrative purposes only, since you are unlikely to care when the program was last compiled. You may, though, be interested in when the program was last edited or changed. That information can be obtained using SCCS, as described later in this chapter.

2.7.2. Macros and Rules

`Make` has several other bells and whistles that can make your job easier. Two discussed here are *macros* and *rules*.

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 modifying the Makefile error-prone, since the same changes must be made in two places in the file. To help with this problem, **make** does some simple parameterless macro substitution in interpreting a Makefile. In this case, you can add the following to the beginning of your Makefile:

```
OBJ = pattern.o computepts.o startupcore.o
```

and change the description of the program pattern into:

```
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. To indicate an actual dollar sign (as when your shell command contains shell variables), type two dollar signs: $$.

A useful property of **make** macros is that their initial values 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
```

When you issue the **make** command, everything compiles as before. But if you give the command

```
make "FFLAGS=-u -O"
```

then the `-O` flag, as well as the `-u` flag, is passed to *f77*.

Another form of shorthand **make** offers you is its set of ***rules***. A rule is a pattern for creating a command that **make** issues to create one sort of file from another. For instance, the **make** rule for making a relocatable file out of the corresponding FORTRAN source file is to use the *f77* compiler, passing as arguments any flags specified by the FFLAGS macro, the **-c** flag, and the name of the source file to be compiled. Since there are three compilations in our example, two of them the same, we can make use of this rule. You should still explicitly state the dependencies, and must explicitly state the nonstandard command for compiling `pattern.F`. The Makefile now looks like this:

```
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=-u
pattern: $(OBJ)
^I      f77 $(OBJ) -lcore77 -lcore -lsunwindow -lpixrect -o pattern
pattern.o: pattern.F commonblock
^I      f77 $(FFLAGS) "-DCOMPILETIME=\"`date`\"" -c  pattern.F
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

2.8. Tracking and Controlling Changes to Programs with SCCS

SCCS stands for 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 SCCS system provided by Sun is explained in several papers and manual sections, the most approachable of which is "Source Code Control System" in ***Programming Tools for the Sun Workstation.*** Although addressed mainly to the C language programmer, that manual provides a thorough introduction to the mechanics of using SCCS. This section uses the previous program to show how to maintain a FORTRAN program under SCCS.

2.8.1. Using SCCS

To begin, you must create the SCCS subdirectory beneath the directory in which your program is being developed. Do this with the command:

```
hostname% mkdir SCCS
hostname%
```

Now put your source files under SCCS control. Before doing this, though, you should put in each file one or more SCCS "ID keywords," which are filled in with a version number each time the file is the object of 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 out 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 be identified by initialized data of the form:

```
character*50 sccsid
data sccsid/"%Z%%M%    %I%    %E%\n"/
```

You can also replace the word COMPILETIME by a parameter that is automatically updated whenever the file is accessed with `get`:

```
character*(*) COMPILETIME
parameter(COMPILETIME="%E%")
```

and correspondingly remove the −DCOMPILETIME from our Makefile. Finally, the included file "commonblock" is annotated with a FORTRAN comment:

```
C       %Z%%M%    %I%    %E%
```

Now you can put these files under control of SCCS with the command

```
hostname% sccs create Makefile commonblock startupcore.f computepts.f pattern.F
hostname%
```

Your files now look like this after SCCS keyword expansion:

Makefile:

```
#       @(#)Makefile    1.1     84/03/01
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=-u
pattern: $(OBJ)
^I      f77 $(OBJ) -lcore77 -lcore -lsunwindow -lpixrect -o pattern
pattern.o: pattern.F commonblock
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

commonblock:

```
C       @(#)commonblock 1.1     84/03/01
        integer nmax, npoints
        real x, y
        parameter( nmax = 200 )
        common npoints
        common x(nmax), y(nmax)
```

computepts.f:

```
        subroutine computepts
        double precision t, dt, pi
        parameter (pi=3.1415927)
        include 'commonblock'
        integer i
        character*50 sccsid
        data sccsid/"@(#)computepts.f   1.1     84/03/05\n"/

C       compute x/y coordinates of npoints points on a unit circle
C       as index i moves from 1 to npoints, parameter t sweeps from
C       0 to pi(2 + npoints/2) in increments of (pi/2)*(1 + 4/npoints)
        t = 0.0
        dt = (pi/2.0)*(1.0 + 4.0/dble(npoints))
        do 10 i = 1, npoints+1
            x(i) = cos(t)
            y(i) = sin(t)
            t = t+dt
10      continue
        return
        end
```

startupcore.f:

```
        subroutine startupcore
        include '/usr/include/f77/usercore77.h'
C       make initializing calls to core library

        integer pixwindd, InitializeCore, InitializeVwsurf, SelectVwsurf
        external pixwindd
        character*4 envreturn
        character*50 sccsid
        data sccsid/"@(#)startupcore.f  1.1     84/03/05\n"/

        if (InitializeCore(BASIC, NOINPUT, TWOD) .ne.0) call exit(1)
        call getenv( "WINDOW_ME", envreturn )
        if (envreturn .eq. "     ") then
```

```
        write(0,*)"must run in a window"
        call exit(2)
    endif
    if (InitializeVwsurf( pixwindd, FALSE) .ne. 0) call exit(2)
    if (SelectVwsurf( pixwindd) .ne. 0) call exit(3)
    call SetWindow( -1.5, 1.5, -2.0, 2.0 )
    call CreateTempSeg()
    return
    end


    subroutine closecore
    include '/usr/include/f77/usercore77.h'
C   make terminating calls to core library
    integer pixwindd
    external pixwindd

    call CloseTempSeg()
    call DeselectVwsurf( pixwindd )
    call TerminateCore()
    return
    end
```

pattern.F:

```
    program star
C   draw a star of n points, argument n
    include 'commonblock'
    character*10 argument
    integer i, iargc, lnblnk
    character*(*) COMPILETIME
    parameter(COMPILETIME="84/03/05")
    character*50 sccsid
    data sccsid/"@(#)pattern.F 1.1      84/03/05\n"/

    if (iargc() .lt. 1 ) then
        call getarg( 0, argument)
        i = lnblnk(argument)
        write (0,*) "usage: ",argument(:i)," -v or ",argument(:i)," nnn"
        call exit (0)
    endif
    call getarg( 1, argument )
    if (argument .eq. "-v") then
        print *, COMPILETIME
        call exit(0)
    endif
    read( argument, '(i3)') npoints
    npoints = npoints*4
    if (npoints .le. 0 .or. npoints .gt. nmax-1) then
        write( 0,*) npoints/4, "out of range [1..", (nmax-1)/4,"]"
        call exit(12)
    endif
    call computepts
    call startupcore
    call moveabs2( x(1),y(1) )
```

```
call polylineabs2( x(2), y(2), npoints)
pause
call closecore
end
```

Of course, in doing this, you have an even more ridiculous example, since you don't need the preprocessor any longer to drop in the compilation date and the **-v** argument is without purpose, since you can use the **what** command, which gives you much more detail.

2.8.1.1. Editing Files Under SCCS Control

Once your source code is under SCCS control, there are two main tasks you'll be using SCCS for: (1) to ***check out*** a file so that you can edit it, and (2) to ***check in*** a file you are done editing. A file is checked out using the **sccs edit** command. The command

```
hostname% sccs edit computepts.f
hostname%
```

makes a writable copy of **computepts.f** in the current directory, and records your login name. Other users are prevented from checking out the same file while you have it checked out, but they can find out what files are checked out and by whom.

When you have completed your current editing task, check in the file using the **sccs delget** command.

```
hostname% sccs delget computepts.f
hostname%
```

This causes the SCCS system to do the following:

- make sure that you have the same login name as the user who checked the file out
- make a record of what was changed in this editing session
- delete the writeable copy of **computepts.f** from the current directory
- replace it by a read-only copy with the SCCS keywords expanded

This is actually a composite of two simpler SCCS commands called **delta** and **get**. **delta** does the first three items in the above list, and **get** does the fourth.

2.9. Building Libraries

A ***library*** is a collection of subprograms. Each member of this collection is called a library ***element*** or ***module***. There are many examples of libraries on the Sun system. The libraries used implicitly or explicitly in the above example were the

- Core graphics libraries: /usr/lib/libcore.a and /usr/lib/libcore77.a
- FORTRAN libraries: /usr/lib/libF77.a, /usr/lib/libI77.a, and /usr/lib/libU77.a
- math library: /lib/libm.a
- C library: /lib/libc.a

A ***relocatable*** library is one whose elements are relocatable (.o) files. Relocatable libraries provide an easy way for commonly used subroutines to be shared among several programs that use them. The programmer need only name the library when linking the program and those library modules that resolve references in the program are loaded. The advantages of doing this are:

- only the needed modules are loaded
- the programmer need not change the link command line as subroutine calls are added and removed during program development.

2.9.1. Using Libraries

When the linker ***searches*** a library, it extracts from it those elements whose entry points (i.e., subprogram or entry names, or names of COMMON blocks initialized in BLOCKDATA subprograms) are referenced in other parts of the program it is linking.

When the linker extracts a library element, it takes the whole thing; since an element corresponds to the result of a compilation, this means that routines that are compiled together are always linked together. This is a difference between UNIX and some other systems and may affect the way you divide up your libraries.

Another important difference between UNIX and other systems is that when you link programs, the order really matters. The linker processes its input files in the order that they appear on the command line, (i.e., left-to-right). When the linker is to decide whether or not a library element is to be loaded, its decision is based only on the relocatable modules it has already processed. For example, if our FORTRAN program is in two files, `main.f` and `graf.f`, and only the latter accesses the Core library, it would be an error to reference that library before the reference of `graf.f` or `graf.o`:

```
(Wrong!) hostname%f77 main.f -lcore77 -lcore graf.f -o myprog

(Right)  hostname%f77 main.f graf.f -lcore77 -lcore -o myprog
```

Order can matter within libraries as well. If you build a ***sequential*** library, then elements at the end of the library should not reference entry points defined in elements that precede them. This is because these libraries are searched in the order in which they are presented (i.e., front-to-back). There are two ways to get around this problem: make sure the library is constructed in the right order, or build a ***random*** library. The programs ***lorder*** and ***tsort*** are usually sufficient for ordering interdependent library elements for one-pass linking: see the manual page ***lorder***(1) for instructions.

2.9.1.1. `ranlib`

Random libraries are built from sequential libraries using the program `ranlib`. `ranlib` builds a table of contents for the library, indicating to the linker which entry points are defined in library elements. Elements in random libraries can refer to one another indiscriminately. Random libraries are preferred on the Sun system, and the linker issues a warning message if it encounters any sequential libraries. Random libraries have the unfortunate property that `ranlib` must be rerun on them whenever the library is changed or copied. Extremely careful individuals use `lorder` and `tsort` to sort their libraries, and then apply `ranlib` to them.

The `-M` flag, which *f77* passes to the loader, is useful for determining what routines are obtained from libraries.

Using the program example from the previous section, suppose you want to put the module `startupcore.o` into a library. Also suppose that you take out the calls to the Core library from the main program, and encapsulate them in a routine `drawpoly`, which you place in the file `drawpoly.f`:

```
subroutine drawpoly( x, y, n)
integer n
real x(n), y(n)
character*50 sccsid
data sccsid/"@(#)drawpoly.f      1.1      84/03/05\n"/
call moveabs2( x(1),y(1) )
call polylineabs2( x(2), y(2), n)
end
```

The following statement can call this routine from the main program:

```
call drawpoly( x, y, npoints )
```

The library named `polylib.a` is created using the `ar` and `ranlib` commands:

```
hostname% ar crv polylib.a startupcore.o drawpoly.o
hostname% ranlib polylib.a
```

and can be referenced in an *f77* command line:

```
hostname% f77 pattern.o computepts.o polylib.a -lcore -lsunwindow
-lpixrect -o pattern
hostname%
```

If a library element is recompiled and must be replaced in its library, use `ar` and `ranlib` again:

```
hostname% ar rv polylib.a drawpoly.o
hostname% ranlib polylib.a
hostname%
```

This time `ar` is given the `rv` flags; `c` is used only for creating. A library need not be specially flagged for the linker; the linker recognizes a library when it encounters one.

2.10. Transporting FORTRAN Programs

If you have developed FORTRAN code on another system, parts of it may need to be changed so it can run on Sun Workstations. This section describes some implementation details you need to know when you transport FORTRAN programs.

2.10.1. General Hints

Keep these Sun FORTRAN conventions in mind when transporting your program from another machine:

- Your source code must have a `.f` filename extension to be recognized by the FORTRAN compiler (*f77*).

- If you are entering in code manually (instead of downloading), you must make sure to start typing in your code in column six or greater (this can be done conveniently by tabbing over).

2.10.2. Time Functions

These are the time functions supported in the Sun Extension to standard FORTRAN

idate() — Returns date in integer array
itime() — Returns time in integer array
ctime(), ***gmtime***(), ***time***() — Returns system time
dtime(), ***etime***() — Returns elapsed system time

You should check your code to make sure that these functions are used in place of other time functions. For example, Sun does not support the following features which are found on other machines:

(CAL)
mclock(h) -- time-of-day in 10h format
mdate(h) -- date in 10h format
milsec() -- milliseconds of job CP time
second() -- seconds of job CP time (floating)

(CRAY)
clock(h) -- like mclock
date(h) -- line mdate
date(h) -- julian date in ASCII
second() -- as above
timef() -- wallclock time since last call to timef, fp mill.

2.10.3. Formats

In most cases, formats in other FORTRAN programs are transportable to Sun Workstations. Sun FORTRAN format features that may be different from other versions of FORTRAN are

- **a** behaves as usual on non-string data, but be careful to use only four characters to a word.
- **r** sets arbitrary radix for following i formats
- **$** suppresses newline
- **:** conditional termination
- **h** FORTRAN 66 feature
- **su** select unsigned output for following i formats:

```
format(z4) => format( su, 16r, i4)
```

For example, these format specifiers are not available with FORTRAN

- **d** same as e (HP, CRAY)

`g` used on integers or logicals (IBM)

`k,@,o`
octal (HP, CRAY)

`q` extended precision (real*16)(IBM)

`r` right-justified characters (on old FORTRAN versions, usually left-justified inside a word) (CRAY, HP)

`z` hexadecimal (IBM,CRAY)

2.10.4. Carriage Control

UNIX doesn't really have carriage control (see "Carriage Control" in Chapter 3). There are two ways that FORTRAN carriage control conventions can be accomodated:

- For simple jobs, use `open( N, form='print')`. You then get single or double spacing, formfeed, and stripping off of column 1 (remember, it's legal to reopen unit 6 if you're just changing the form parameter to 'print', for example open(6,form='print')).
- Use the ***fpr*** filter to transform FORTRAN carriage control conventions into the UNIX carriage control format (see ***fpr***(1)). You can then print files using ***lpr***.

2.10.5. File Equates

See the I/O section on "Gaining Access to Files", about piping and redirection. You can also use hard or soft links.

2.10.6. Data representation

See the section in Chapter 4 concerning data representation for exact representation of different kinds of data. This section points out information necessary for transporting FORTRAN programs. You should remember the following:

- The first four bytes of a `real*8` are not the same as a `real*4`
- The default sizes for reals, integers, and logicals are the same (as they should be, according to the standard) except when the `-i2` flag is used, which shrinks integers and logicals to two bytes but leaves reals as four bytes.
- There is no `logical*1`. Use `character*1` instead.
- Character variables can be freely mixed and equivalenced with variables of other types, but you should be careful of potential alignment problems.
- Integer, logical, real, double precisions, complex, double complex types must always be aligned on even-byte boundaries. The compiler does this for you, but may create holes in common blocks if character variables are mixed with any of the other types. Programs having knowledge of holes created by more restrictive alignment on other machines are not portable.
- Our floating-point arithmetic does not cause exceptions on overflow or divide-by-zero. It does deliver IEEE indeterminate forms in cases where exceptions would otherwise be signaled. See the data representation section in Chapter 4 for more details. The extreme finite,

normalized values are delivered by the flmin(), flmax(), etc. functions (see ***range***(3f)). The indeterminate forms can be written and read using formatted and list-directed I/O statements.

2.10.7. Hollerith

The information in this section is useful for transporting older programs — not for writing or heavily modifying a program. It is recommended that you use character variables for the purpose covered in this section.

You can initialize things as with standard FORTRAN , but remember that Sun Workstations are 32-bit machines. Thus, the maximum number of characters per data type is as follows:

DATA TYPE	MAX CHARACTERS PER DATUM
integer*2	2
integer*4	4
logical	4 (or 2 if **-i2** flag given)
integer	4 (or 2 if **-i2** flag given)
real	4
real*4	4
double precision	8
real*8	8
complex	8
double complex	16
complex*16	16

For example:

```
double complex x(2)
data x /16hHello there, sai, 16hlor, new in town/
write (6,'(4a8, "?")' )x
end
```

You cannot pass Holleriths as parameters or used them in expressions, or even comparisons. They are interpreted as character-type expressions in these contexts. If you must, you can initialize a data item of a compatible type with a Hollerith, and then pass it around. For example,

```
      integer function DoYouLoveMe()
      double precision fortran, beloved
      integer yes, no
      data yes,no/ 3hyes, 2hno /
      data fortran/ 7hfortran/
10    format( "Whom do you love? ",$)
      write (6,10)
      read  (5,20) beloved
20    format( a8)
      DoYouLoveMe = no
      if ( beloved .eq. fortran ) DoYouLoveMe = yes
      return
      end
```

```
program trouble
integer yes, no
integer DoYouLoveMe
data yes,no/ 3hyes, 2hno /

if ( DoYouLoveMe() .eq. yes ) then
    print *, 'You are sick'
else
    print *, 'See if I ever speak to you again'
endif
end
```

All these things produce warning messages from the compiler. Use the **-w66** flag to suppress these messages.

Chapter 3

Input and Output

The first half of this chapter describes the UNIX file system and how it relates to the FORTRAN I/O system. The second half discusses FORTRAN I/O as implemented on the Sun Workstation. Topics covered include:

- Accessing files
- Logical units and preconnected units
- UNIX file descriptors
- FORTRAN I/O, file access modes, and file types
- FORTRAN 77 implementation
- Extensions to FORTRAN 77 I/O
- Running older programs
- Magnetic Tape I/O

For a more detailed discussion of the UNIX file system structure, refer to the *Beginner's Guide to the Sun Workstation.*

3.1. The UNIX File System

The UNIX system file structure 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 make a distinction between directories and subdirectories.

A sequence of branching directory and filenames in the file system tree describe a ***path***. Files are at the ends of paths, and can not have anything branching from them. When moving around in the file system, ***down*** means away from the root and ***up*** means toward the root. Refer to Figure 3-1 for a diagram showing the UNIX file system tree structure.

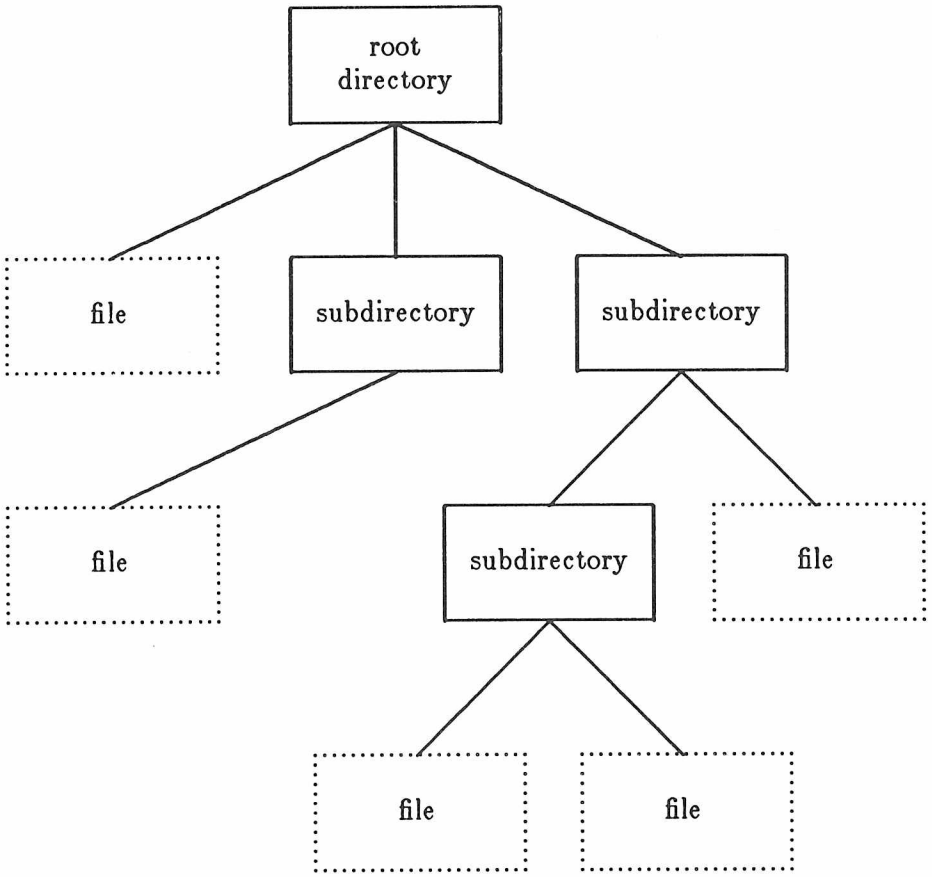


Figure 3-1: Diagram showing UNIX file system structure

All UNIX files have names and all files branch from directories. 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 (/).

While you are logged on to a UNIX system, you are said to be in a directory. When you first log on, you are in your ***home*** directory. At any time, wherever you are, that directory is called your ***current working directory***. It is often useful to list your current working directory. The ***pwd*** command and the ***getcwd*** library call print 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*** library call change to a different current working directory. Additional explanations of the file system organization and relevant shell commands are located in the ***Beginner's Guide to the Sun Workstation.***

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.

To describe a file anywhere in the directory structure, you can list the sequence of directory, subdirectory and filenames, separated by slash characters, between the root and the file you want to describe. This is called an ***absolute path name*** because it begins at the root of the directory tree (indicated by the first /). It is also the complete filename for this file. An example of an absolute path name is shown in **Figure 3-2**.

/usr/you/mail/record

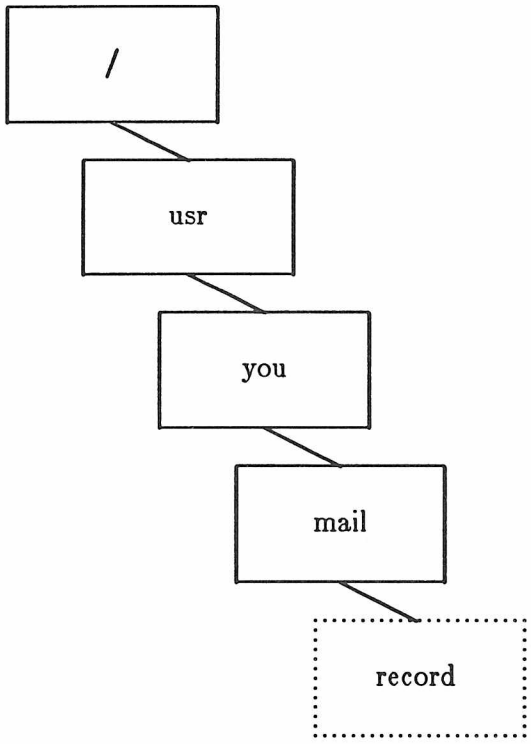


Figure 3-2: Absolute Path Name

Alternatively, from anywhere in the directory structure, you can describe a ***relative path name*** of a file. Relative path names begin in the directory you are in (the current directory) instead of the root. Refer to **Figure 3-3** for an illustration of a relative path name.

A complete UNIX file specification has the general form:

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

A typical example of a complete UNIX file specification, or absolute path name is:

```
/usr/src/sun/doc/fortran.manuals/programmers.guide
```

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 path name is less than or equal to 1024.

mail/record (from /usr/you)

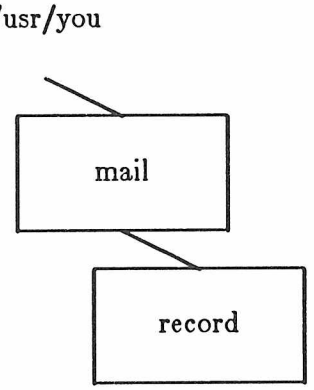


Figure 3-3: Relative Path Name

3.2. Gaining Access to Files From FORTRAN Programs

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

- `open`
- `close`
- `read`
- `write`
- `print`
- `backspace`
- `endfile`
- `rewind`
- `inquire`

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.

3.2.1. Accessing Files With Names

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 77 `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
  [*ex*: `file='myfile.out',...`]
- character variables
  [*ex*: `file=filnam,...`]
- character expressions
  [*ex*: `file=prefix(:lnblnk(prefix)) // '/' // name(:lnblnk (name)),...`]

Some ways a program can get filenames are

- by reading from a file or terminal with a FORTRAN read statement
  [*ex*: `read(4,401) filnam`]
- from the command line by way of the `getarg` function
  [*ex*: `call getarg(argumentnumber, filnam)`]
- from the environment with `getenv`
  [*ex*: `call getenv(string, filnam)`]

The program fragment in **Figure 3-4** shows one way filenames may be constructed.

```
      character*1024 function fullname ( name )
      character*(*) name
      character*1024 prefix
C
C     in  path names starting with '~/', replace
C         the tilde with the home directory name;
C     prefix relative path names with path to current working 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
```

Figure 3-4: Program example showing one way to construct filenames

3.2.2. Accessing Files Without Names

When a program opens a FORTRAN file without a name, the runtime system supplies a filename. There are several ways it can do this. If `status='scratch'` is specified in the `open` statement, then the run-time system opens a file with a name of the form `tmp.F`*nnnnnn*, where *nnnnnn* is replaced by the current process ID. This file is deleted upon termination of the program or execution of a `close` statement, unless `status='keep'` is specified in the `close` statement.

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 runtime 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 run-time system would make up a name, but is not.

In all other cases, the run-time system opens a file with a name of the form fort.*n*, where *n* is the logical unit number given in the `open statement`.

The `inquire` statement can also be used to determine the name of an `open` file by giving its logical unit number. More information on the `open` and `inquire` statements is found later in this chapter.

The UNIX 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 UNIX, there are several satisfactory ways to communicate filenames to a FORTRAN 77 program including command line arguments and environment variable values. For example, see the routine `ioinit.f` in libU77, which is discussed in "Preattachment of Logical Units" later in this chapter. The program can then use those logical names to `open` the files. The next section recommends two additional ways to change a program's input and output files without changing the program, called ***piping*** and ***redirection***.

3.2.3. Preconnected Units

When a UNIX FORTRAN or C language program begins execution, 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 unit 6
- standard error is unit 0

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

All other units are preconnected to files named `fort.`*n* where *n* is the ccorresponding unit number. These files need not exist, and are only created if their units are used and an `open` statement does not override the preconnected name. For example, the program

```
write (15) 2
end
```

writes a single unformatted record in file `fort.15`.

3.2.4. 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. As shown in the example below, the shell command line

```
hostname% 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 file `mydata`. Similarly, the shell command line

```
hostname% 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 terminal. (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.

3.2.5. Piping

It is also possible, in UNIX to 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:

```
hostname% 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:

```
hostname% 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***.

3.2.6. UNIX File Descriptors

In almost every discussion of input and output in FORTRAN 77 programs, I/O channels are in terms of FORTRAN unit numbers. The UNIX I/O system does not actually deal with these units, but with UNIX ***file descriptors***. The FORTRAN runtime system always translates from one to the other, so most FORTRAN programs don't have to have to know about file descriptors.

The information in this section is of interest mostly to users writing C routines that interface to FORTRAN 77 programs. More about this is covered in Chapter 4, "Data Representations." In addition to FORTRAN units and UNIX file descriptors, many C programs use a set of subroutines called ***standard I/O*** (or `stdio`). Many of the functions of the FORTRAN 77 I/O system are implemented using standard I/O, which in turn is implemented using the UNIX I/O system calls. Some of the characteristics of these systems are listed in Table 3-1.

Table 3-1: Characteristics of Three I/O Systems

	FORTRAN-77 Units	Standard I/O File Pointers	UNIX 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	byte stream	byte stream
Form	arbitrary, nonnegative integers	pointers to structures in the user's address space	integers from 0-19

3.3. FORTRAN I/O

UNIX is not as format-oriented as FORTRAN. UNIX treats all files as sequences of bytes instead of collections of record structures. The FORTRAN run-time system keeps track of file formats and access modes. It provides the FORTRAN file facilities using the FORTRAN I/O

system, which includes the FORTRAN libraries and the standard I/O library.

Table 3-2: Summary of FORTRAN Input and Output

Type of file		Access mode	
		Sequential	Direct
Formatted	internal	file must be character variable, character array element, character array, or substring	file must be a character array; each "record" is a single element of the array
	external	contains only formatted records of same or variable length	contains only formatted records; all must be the same length
unformatted	internal	(not allowed)	(not allowed)
	external	contains only unformatted records	reads: one logical record at a time; writes: leaves unfilled part of record undefined
List-directed	internal	reads: bytes are read until the I/O list is satisfied, or until 'end-of-file' is reached; implemented to make command line decoding easier; writes: records are filled until I/O list is satisfied; WRITES SHOULD BE AVOIDED	(not allowed)
	external	no associated format statement; values input or output depend on types in I/O list	(not allowed)

Note: Under List-directed internal files, writes should be avoided because the number of items written on a line of output and the lengths of the items vary with the values of the items (see "List-Directed Output" later in this chapter).

3.4. Implementation Details

Some details of the current implementation may be useful in understanding constraints on FORTRAN 77 I/O.

3.4.1. Logical Units

The maximum number of logical units that a program can have **open** at one time is the same as the UNIX system limit, currently 20.

The standard logical units, 0, 5, and 6, are named internally **stderr**, **stdin**, and **stdout** respectively. These are not actual filenames and can not be used for **open**ing these units. **Inquire** does not return these names and indicates that the above units are not named unless they have been **open**ed 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).

3.4.2. Vertical Format Control

Simple vertical format control is implemented. The logical unit must be **open**ed for sequential access with **form = 'print.'** Control codes '0' and '1' 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.

3.4.3. open

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

open (***parameter list***)

where

parameters is a list of optional keywordd specifiers, separated by commas. Valid specifiers 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**. If not specified, a default filename can be created. An **open** statement need not specify a filename.

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'), 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.F*nnnn*' 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 'sequential' or 'direct'. If not specified, 'sequential' is assumed.

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 accesible files are allowed; thus, tty, pipes and magnetic tape are not allowed. If form='unformatted' the size of each transfer depends upon the data transferred. If form is not specified, unformatted transfer is assumed.

If access='sequential' recl is prohibited since records are of varying size. No padding of records is done and files don't have to be randomly accessible; thus, tty, pipes and tapes can be used. If not specified form='formatted' is assumed. If form='formatted' each record is terminated with a newline (0 character. This means that each record actually has one extra byte per record. 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 = formfeed, and blank = single space). If form='unformatted' each record is preceded and terminated with an integer * 4 count making each record 8 bytes longer than normal. This is inconsistent with UNIX, thus is only useful for communicating between FORTRAN programs. In addition, each write defines one record and each read reads one record (unread bytes are flushed). With magnetic tape records cannot span tape blocks, so fileopt='buffer=...' suboption must be at least 8 bytes greater than the largest record you write.

form An optional character expression. The options are 'formatted', 'UNformatted' or 'print.' If not specified, formatted is assumed. Interacts with access.

recl recl=*length* specifies the record length in bytes. Required if access='direct', prohibited if access='sequential.'

err An optional statement label to jump to if an error occurs during the open.

iostat An optional variable name that receives the error status from an open.

Note: Either err=*label* or iostat=*name* must be coded to avoid a disaster when an error occurs on an open.

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. The possible values are

- 'old' — the file already exists (nonexistence is an error);

- 'new'— the file doesn't exist (existence is an error) and `file=`*name* is required;
- 'unknown' — existence is unknown (the default); and
- 'scratch' — `file=`*name* is prohibited and the file is removed upon close (exception: if you specify `status='keep'` in an explicit `closefP of the unit`) .

`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);
- 'buffer=*nnnn*' — the size of the I/O buffer to use (magnetic tape only). `buffer` is only necessary when writing, since the I/O system defaults to 65 character buffers for tape, allowing reads to anything smaller than that;
- 'eof' — opens a file at end-of-file rather than at the beginning (useful for appending data to the file). For example:

```
open(7,file='junkfile',form='formatted',fileoptg='eof,buffer=2048')
```

By default, files are positioned at their beginning upon opening, but see *ioinit*(3f) for alternatives. Existing files are never truncated on opening. Sequentially-accessed, external files are truncated to the current file position on `close`, `backspace`, or `rewind` only if the last access to the file was a write. An `endfile` always causes such files to be truncated to the current file position.

3.4.4. FORTRAN and UNIX file permissions

In C, programmers traditionally open up input files for reading, output files for writing or some files for both since UNIX allows for reading and/or writing permissions by the owner, owner's group or anyone. In FORTRAN it's not possible for the system to forsee what use you make of the file since there's no parameter to the open statement that gives us 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 to do. The only exception is with magnetic tape, where if you could have write permission but without a write ring, an error displays on the screen.

3.4.5. `inquire`

The `inquire` statement gives information about a unit (inquire-by-unit) or a file (inquire-by-file). It sets values of integer, logical, and character variables by specifing keywords that correspond to the values of unit, connection, or file properties. These properties can be grouped as follows:

- Unit properties: A unit alone only has the properties of existence and of being connected or not. Only units that exist can be opened but you can inquire about a unit even if it doesn't exist.

`exist` (if inquire-by-unit)

`number` (if inquire-by-file)

- Connection properties: The association between a FORTRAN unit and a file. It can have associated with it properties associated with the `open` statement: sequential or direct, formatted or unformatted, and a record length. Its properties interact with file properties. For example, some types of connections (e.g., direct) may not be allowed with some files (e.g., magnetic tape).

  `opened`

  `access`

  `form`

  `recl`

  `nextrec`

  `blank`

- File properties: File properties are its name, existence and how it can be connected (`formatted`, `unformatted`, `sequential` and `direct`).

  `exist` (if inquire-by-file)

  `named` (if inquire-by-unit)

  `name` (if inquire-by-unit)

  `sequential`

  `direct`

  `formatted`

  `unformatted`

Simple examples are:

```
inquire(unit=3, namexx)
inquire(file='junk', exist=l, opened=isopen, number=n)
```

The options to `inquire` are as follows:

`file=` a character variable specifies the file the `inquire` is about. Trailing blanks in the filename are ignored. Files have the properties of a name, existence (or nonexistence), and the ability to be connected to in certain ways (`formatted`, UN`formatted`, `sequential`, or `direct`). It can be connected to a unit in the current program or not.

`unit=` a positive integer variable that refers to files after they are `opened`. Exactly one of `file=` or `unit=` must be used.

`iostat=`, `err=`
are as before.

`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.

`named=` a logical variable that is assigned `.true.` if the file has a name, or `.false.` otherwise.

`name=` a character variable that is assigned the name of the file (inquire- by-file) or the name of the file connected to the unit (inquire-by-unit). The name is the full name of the file. When performing an inquire-by-unit, the name parameter is undefined unless both the `opened` and named parameters indicate .true..

`access=` a character variable that is assigned the value `'sequential'` if the connection is for sequential I/O, `'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, or `'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, `'zero'` if blanks are being converted to zeros and the file is connected for formatted I/O.

Remember that the people who wrote the ANSI standard probably weren't thinking of your needs. Here is an example, in which declarations are omitted.

```
open(1, file="/dev/console")
```

On a UNIX system this statement `opens` the console for formatted sequential I/O. An `inquire` statement 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, and is ignoring blanks in numeric fields.

In the UNIX 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.

3.4.6. Close

`close` severs the connection between a unit and a file. The unit number must be given. The optional parameters are `iostat=` and `err=` (see `open` for meanings), and `status=` 'keep' or 'delete.' `keep` is the default (except for scratch files). `delete` means that the file will be removed. A simple example is

```
close(3, err=17)
```

3.4.7. Format Interpretation

In the Sun implementation, most formats are compiled; only those that are unknown until runtime require parsing at runtime. Upper- as well as lower-case characters are recognized in format statements and 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 (*).

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** must be followed by a positive nonzero number. If it is not, it has a different meaning. 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 **X** writes blanks on the output.

3.4.8. List-Directed Output

In formatting list-directed output, the I/O system tries to prevent output lines longer than 80 characters. Each pair of external data is separated by two spaces. List-directed output of `complex` values includes an appropriate comma. List-directed output distinguishes between `real` and `double precision` values and formats them differently. The output system reasonably interprets the output of a character string that includes '\n.'

3.4.9. 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 UNIX errors; these are described in ***intro***(2) in the Sun ***System Interface Manual***. Error numbers ≥ 100 come from the I/O library, and are described further in Appendix C 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.

3.5. Non-'ANSI Standard' Extensions

Several extensions have been added to the I/O system to provide for functions omitted or poorly defined in the standard. Programmers should be aware that these are not portable. Refer to Chapter 6 for a complete description of deviations from the FORTRAN 77 standard.

3.5.1. Format Specifiers

FORTRAN 77 provides additional specifiers to the FORTRAN 66 format specifications I, F, E, G, D, H, X, A, and L. A brief description of some of the expanded features follows here. Table 3-3 summarizes FORTRAN 66, FORTRAN 77 and extended ***f77*** format specifiers.

Table 3-3: FORTRAN Format Specifiers

Format Specifier	FORTRAN 66	FORTRAN 77	*f77* Extensions
Integer Editing	Iw	Iw.m	
Floating-Point Editing	Fw.d, Ew.d, Gw.d, Dw.d	Ew.dEe, Dw.dEe, Gw.dEe	Ew.d.e, Dw.d.e, Gw.d.e
Character Editing	wH, Aw	"*xxxx*" (string constant), A	
Logical Editing	Lw		
Position Editing	wX, /	Tn, TLn, TRn	
Position Control			nT, T
Sign Control		S, SP, SS, SP	
Blank Control		BN, BZ	B
Scale Control		nP	P
Conditional Newline			$
Conditional Termination of Format Editing		:	
Signed/Unsigned Integer Control			SU
Radix Control			nR

The FORTRAN 66 formats I*w*, E*w.d*, and G*w.d* have been extended in FORTRAN 77 to include the forms

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

The *e* field specifies the minimum number of digits or spaces in the exponent field on output. The form **E**w.d.e is allowed but is not standard. If the value of the exponent is too large, the exponent notation **e** or **d** 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 default value for *e* is 2.

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

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

B is an acceptable edit control specifier. It causes return to the default mode of blank interpretation. This is consistent with **S**, which returns to default sign control.

A format control specifier has been added to suppress the newline at the end of the last record of a formatted sequential write. The specifier is a dollar sign (**$**) and is constrained by the same rules as the colon (**:**). It is used typically for console prompts. For example:

```
write (*, "('enter value for x: ',$)")
read (*,*) x
```

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.

In conjunction with the above, a sign-control specifier has been added to cause integer values to be interpreted as unsigned during output conversion. The specifier is **SU** and remains in effect until another sign control specifier is encountered, or format interpretation is complete. Radix and 'unsigned' specifiers could be used to format a hexadecimal dump, as follows:

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

Note: Unsigned integer values greater than (2**30 - 1), cannot be read by FORTRAN 77 input routines. All internal values are output correctly.

3.5.2. Print Files

The ANSI standard is ambiguous regarding the definition of a 'print' file. Since UNIX 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" earlier in the chapter).

The `inquire` statement returns `print` in the `form=` string 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')
```

3.5.3. Scratch Files

A `close` statement with `status='keep'` must be specified for temporary files. 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.

3.5.4. List-Directed I/O

List-directed input has been modified 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, list-directed I/O has been implemented. During internal, list-directed reads, bytes 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 input list is satisfied. The length of an internal array element should be at least 20 bytes 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.

3.6. Running Older Programs

Traditional FORTRAN 77 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 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 **open**ing.

3.6.1. Preattachment of Logical Units

The *ioinit* routine can also be used to attach logical units to specific files at runtime. It looks for names of a user-specified form in the environment and **open**s the corresponding logical unit for sequential formatted I/O. Names must be of the form PREFIX*nn*, where *PREFIX* is specified in the call to *ioinit* and *nn* is the logical unit to be **open**ed. Unit numbers < 10 must include the leading '0.'

ioinit should prove adequate for most programs as written. However, it is written in FORTRAN 77 specifically so that it may serve as an example for similar user-supplied routines. A copy may be retrieved by issuing the command

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

3.7. Magnetic Tape I/O

Previously, magnetic tape was not seriously usable from UNIX FORTRAN programs. This is because of the simple implementation of the FORTRAN I/O system in terms of UNIX's Standard I/O package, which is set up to deal only with disks and sequential devices such as terminals and pipes.

The *f77* tape I/O package implemented at Berkeley (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.

Sun Microsystems has reimplemented parts of the FORTRAN I/O system so that FORTRAN programmers are allowed to use magnetic tape transparently. We provide facilities that make it easy to use tape as sequentially-accessed formatted files. Since the standard I/O package buffering scheme provided is still used, there is no bound on formatted record size, and records may span physical tape blocks.

Connecting a magnetic tape for unformatted access is less satisfactory. Because of the implementation of unformatted records as a sequence of bytes preceded and followed by byte 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 bytes 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 bewteen 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.)

3.7.1. 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 byte 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 largest unformatted record is eight less than the size of the block.

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,

```
dd if=/dev/rmt0 ibs=20b cbs=80 conv=unblock | fort_prog
```

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 bytes into a character variable and use internal I/O to transfer formatted data. See also *topen(3F)*.

3.7.2. 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, it 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.

3.7.3. Endfile

When writing to a UNIX disk file, endfile causes the file to be truncated 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 causes two endfile records to be written, 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. This is consistent with the standard.

3.7.4. Backspace

Backspace does one of two things, depending on whether or not end-of-file has been reached. If it has, then backspace 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, backspace 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, search backwards looking for the record separator (newline or ^J); for UN`formatted` records, use the byte-count trailer that is part of the record.

3.7.5. Rewind

REWIND positions you at the beginning of the file you were just reading or writing. When writing a sequential file (such as tape), it does an implicit END`file` action first. If you are reading the endfile record, rewind backspaces over that and all the data records preceeding.

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 using the *system(3f)* library call.

3.7.6. open

`open` 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 a tape or if the `fileoptg='buffer=..'` subparameter is specified. The default buffer size for tape is 64K bytes.

3.7.7. Accesing Files on Multiple-File Tapes

Each tape drive can be **open**ed by many names. The name used determines certain characteristics of the connection, which are the recording density and whether the tape is automatically rewound upon **open** and close. 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 re**open**s 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). Thus, if your program terminates prematurely it could leave the tape positioned in an unpredictable place.

Chapter 4

The Runtime Environment

This chapter describes useful runtime parameters, *f77* data representations, and the conventions you must be aware of to interface C and FORTRAN 77 procedures. It is intended as a guide to write modules in languages other than FORTRAN 77 and have those modules interface to FORTRAN 77 code. The Pascal—FORTRAN interface is covered in Appendix C of the *Pascal Programmer's Guide.*

4.1. Command Line Arguments

It is often useful to pass a program parameters on the command line. The function *iargc*(1) returns the number of command line parameters. The subroutine *getarg*(1) copies a parameter into a variable in the program (similar to the C shell's *echo* command). For example,

```
      character arg*70
c
c       find out how many command line arguments there were
      nargs=iarc()
c       one at a time, get an argument and write it out
      do 10 i = 1, nargs
      call getarg(i, arg)
      print '(a)', arg
10     continue
      end
```

The program 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 it is compiled in `myecho` you can test it as follows:

```
hostname% myecho this is a sample
          this
          is
          a
          sample

hostname% myecho *
          calc.f
          mycat.f
          myecho
          myecho.f
          myecho.o
```

4.2. Exiting with status

Using the subroutine *exit*(), a FORTRAN program 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 listed by typing

```
hostname% echo $status
```

Note that the `echo` command sets the variable back to zero after listing its value. The value of `status` can be used in shell script conditional statements or in batch jobs.

`abort` can be used to terminate a program setting `status` to 138, dumping memory to the file `core`, and printing a message on standard error as in

```
call abort(" sample error message ")
```

and causes a program to terminate after writing out

```
abort: sample error message
Bus error (core dumped)
```

4.3. 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. Values that can fit into a single byte are aligned on a byte boundary.

integer*2
: occupies 16 bits (two bytes or one word), aligned on a word boundary.

integer or **integer***4
: occupies 32 bits (four bytes or two words), aligned on a word boundary.

real or **real***4
: occupies 32 bits (four bytes 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 77 `real` elements conform to the proposed IEEE standard[1]. The layout of a `real` element is shown in Table 4-1.

double precision or **real***8
: elements occupies 64 bits (eight bytes 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 77 `double precision` elements conform to the IEEE standard for double precision floating-point data as defined in [25]. The layout of a `double precision` element is shown in Table 4-1.

[1] See p.754 [25].

complex
elements are represented by two `real` elements. The first element represents the real part of the number, and the second represents the imaginary part.

double complex
elements are represented by two `double precision` elements. The first element represents the real part of the number, and the second represents the imaginary part.

logical*2
occupies two bytes (16 bits) of storage, aligned on a word boundary. The value 0 represents `.false.` and 1 represents `.true.`. Any other value is an 'undefined' logical value.

logical or **logical*4**
occupies four bytes (32 bits) of storage, aligned on a word boundary. The value 0 represents the value `.false.` and 1 represents `.true.`. Any other value is an 'undefined' logical value.

4.4. 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 byte of the byte sequence required to represent that object.

4.4.1. Representation of real and double precision

`real` and `double precision` data elements are represented according to the proposed IEEE standard:

Table 4-1: Representation of Real and Double Precision Numbers

	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

`real` and `double precision` numbers are composed of the following parts:

- a one-bit sign. The sign bit is a `1` if the number is negative.
- a biased exponent. The exponent is eight bits for a `real` number, and is eleven bits for a `double precision` number. The values of all zeroes and ones are reserved values.
- a normalized significand, with the high-order `1` bit 'implicit.' The fraction is 23 bits for a **real** number and 52 bits for a `double precision` number. A `real` or `double`

`precision` number is represented by the form:

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

where *f* is the bits in the mantissa.

4.4.2. Representation of Extremal Numbers

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

subnormal numbers
are nonzero numbers with an exponent of zero. The form of a subnormal number is

$$2^{exponent-bias+1} * 0.f$$

where *f* is the bits in the fraction.

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.

The largest finite `double precision` number is approximately 1.797693e+308; the smallest positive normalized `double precision` number is approximately 2.225074e-308. The largest finite `real` number is approximately 3.402823e+38; the smallest positive, normalized `real` number is approximately 1.175494e-38.

4.4.3. Hexadecimal Representation of Selected Numbers

Table 4-2: Hexadecimal Representation of Selected Numbers

Value	Real	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	7FFxxxxxxxxxxxx

4.4.4. Deviations from the Proposed IEEE Standard

Deviations from the proposed IEEE standard in this implementation are as follows:

- Remainder is not provided
- Ordered comparisons involving NaNs do not conform
- User-defined rounding modes are not supported. Only round-to-nearest mode is provided for most operations, except that conversion from a floating-point number to an integer value in either integer format (INT) or floating format (AINT) is provided only in round-toward-zero mode
- Exceptions are neither recorded nor reported
- Signaling NaNs are not provided

4.4.5. Arithmetic Operations on Extreme Values

This section describes the results from the basic arithmetic operations using 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 4-3 summarizes the abbreviations used in the following tables:

Table 4-3: Abbreviations for Numbers

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

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

Note: Inf + Inf = Inf; Inf - Inf = NaN

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

Note: NS means either Num or Sub result possible.

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

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

Notes:

- If either x or y is NaN, then x.EQ.y is FALSE and x.NE.y is TRUE, while x.LT.y, x.LE.y, x.GT.y and x.GE.y are undefined.
- +0 compares equal to -0.

Max				
Left Operand	Right Operand			
	0	Sub	Num	Inf
0	0	Sub	Num	Inf
Sub	Sub	Sub	Num	Inf
Num	Num	Num	Num	Inf
Inf	Inf	Inf	Inf	Inf

Min				
Left Operand	Right Operand			
	0	Den	Num	Inf
0	0	0	0	0
Sub	0	Sub	Sub	Sub
Num	0	Sub	Num	Num
Inf	0	Sub	Num	Inf

Note: Results of Max and Min are undefined if any argument is NaN.

4.5. Inter-Procedure Interface

To write C procedures that call or are called by FORTRAN 77 procedures, you must know the conventions for procedure names, data representation, return values, and argument lists that both languages use.

4.5.1. Procedure Names

f77 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. FORTRAN 77 library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

4.5.2. Data Representations

Table 4-4 summarizes corresponding FORTRAN 77 and C declarations:

Table 4-4: FORTRAN and C Declarations

FORTRAN	C
integer*2 x	short int x;
integer x	long int x;
logical x	long int 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];

According to FORTRAN 77 rules, `integer`, `logical`, and `real` data occupy the same amount of memory.

4.5.3. 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. 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. Thus,

```
complex function f( . . . )
```

is equivalent to

```
f_(temp, . . .)
struct { float r, i; } *temp;
 . . .
```

A character-valued FORTRAN function is equivalent to a C routine with two extra initial arguments: data address and length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
g_(result, length, . . .)
char result[ ];
long int length;
 . . .
```

and could be invoked in C with

```
char chars[15];
 . . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were integer-valued functions whose values specify which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. If the subroutine has no entry points with alternate return arguments, the returned value is undefined. The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed `goto`

```
goto (1, 2, 3), nret( )
```

4.5.4. Argument Lists

All FORTRAN 77 arguments are passed by reference. In addition, for every argument that is of type `character` or that is a dummy procedure, an argument is passed giving the length of the value. The string lengths are `long int` quantities passed by value. The order of arguments is then:

- Extra arguments for the return values of complex and character functions
- Address for each datum or function
- A `long int` for each character or procedure argument

Thus, the FORTRAN call in

```
external f
character*7 s
integer b(3)
 . . .
call sam(f, b(2), s)
```

is equivalent to the C call in

```
int f_();
char s[7];
long int b[3];
 . . .
sam_(f_, &b[1], s, 0L, 7L);
```

Note that the first element of a C array always has subscript zero, but FORTRAN 77 arrays begin at 1 by default. FORTRAN 77 arrays are stored in column-major order, C arrays are stored in row-major order.

4.5.5. Examples

This section presents two examples that illustrate interlanguage conventions. The first example shows how a C function can be called from a FORTRAN program and the second shows how a FORTRAN function can be called from a C program. The called function has the task of building a character string by repeating a character *n* times, where the character and *n* are arguments.

4.5.5.1. Calling C from FORTRAN

```
file "main.f"

        CHARACTER STRING*100, REPEAT*50
        STRING=REPEAT('*',10)
        PRINT *,STRING
        END

file "repeat.c"
#include <stdio.h>

repeat_(retval_ptr, retval_len, char_ptr, n_ptr, char_len)
char *retval_ptr, *char_ptr;
int  retval_len, *n_ptr, char_len;
{
        int count, i;
        char *cp;

        count = *n_ptr;
        if(count > retval_len) {
                fprintf(stderr,"repeat count too large0);
                count = retval_len;
        }

        cp = retval_ptr;
        for(i=0;i<count;i++) {
                *cp++ = *char_ptr;
        }

        for(i=count;i<retval_len;i++) {
                *cp++ = ' ';
        }
}
```

This program can be compiled with the command

```
% f77 main.f repeat.c
```

Since the *f77* compiler appends a trailing underscore to all external names in FORTRAN programs, you need to add an underscore to the name of the C function called. `repeat`'s list of formal arguments is more complicated than the list of actual arguments in MAIN. The additional complication is due to housekeeping details related to the management of character strings. If `repeat` were a FORTRAN function, the compiler would hide these details; however, since repeat is written in "C" the housekeeping must be explicit.

MAIN declares `repeat` as a function that returns a character string of length 50. The mechanism used to return character strings is to prepend two additional arguments to the beginning of the argument list. The first of these (`retval_ptr`) points to the start of the string and the second (`retval_len`) gives the string's length. MAIN passes two actual arguments: a character string and an integer. Both `char_ptr` and `n_ptr` are passed by address. Finally, for every character argument in the list of actuals, an additional argument giving the character's length is passed. In this example, `char_len` gives the length of the string pointed to by `char_ptr`. Note that FORTRAN strings are always accompanied by a

length and need not terminate with a null character.

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 1) it is used in an expression and 2) the left hand side of 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. `Incr` is a FORTRAN callable function that returns a `real` value one greater than its its `real` argument.

```
int /* actually returns a single precision floating point value */
incr_(float_ptr)
float *float_ptr;
{
        float f;

        f = *float_ptr;
        f ++;
        return *((int*)&f);
}
```

Thus, the program

```
real incr
print *,incr(1.)
end
```

prints `2.`.

4.5.5.2. Calling FORTRAN from C

The second example illustrates a C program that calls a FORTRAN function.

```
file "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("%s0,repeat_val);
}
file "repeat.f"

        function repeat(c,n)
        character repeat*(*),c*(*)
        if(n.gt.len(repeat)) then
                write(0,'(a)')'repeat count too large'
                n = len(repeat)
        endif
        repeat = ''
        do10i=1,n
10      repeat(i:i)=c(1:1)
        return
        end
```

This program can be compiled with the command

% **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.

4.5.6. 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.

Chapter 5

Debugging and Profiling FORTRAN Programs

5.1. Introduction

This chapter describes tools for debugging and measuring the resource usage of FORTRAN programs. The most versatile and powerful tool for debugging FORTRAN programs on the Sun workstation is the symbolic debugger ***dbx***, or its window- and mouse-based version ***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. ***adb*** is an older binary-oriented, debugger, which is occasionally useful as a supplement to ***dbx***.

The *f77* compiler provides two flags that are useful for debugging:

- The **-C** flag causes the compiler to generate subscript checking code that catches certain kinds of out-of-bounds array subscripts.
- 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 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.

To provide examples of how these tools work, the following program is used throughout this chapter:

file `a1.f`:

```
program silly
real twobytwo(2,2)
data twobytwo/4*-1/
n=2
call mkidentity(twobytwo,n)
print *,determinant(twobytwo)
end
```

file `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
```

file `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
```

5.2. Using *dbx*

This section briefly 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.

To use ***dbx***, you must compile and load your program with the **-g** flag. For example,

```
hostname% f77 -o silly -g a1.f a2.f a3.f
```

or

```
hostname% f77 -c -g a1.f a2.f a3.f ; f77 -g -o silly a1.o a2.o a3.o
```

To run the program under the control of ***dbx***, type the following command in the directory where the sources and programs reside:

```
hostname% dbx silly
```

To set a breakpoint before the first executable statement, type

```
(dbx) stop in MAIN
```

after the `(dbx)` prompt appears, then type "`run`" to begin execution. When the breakpoint is reached, ***dbx*** displays a message showing that it is stopped at line 4 of file `a1.f`.

The `where` command shows where in the program execution stopped and how execution reached this point.

The command `print n` at this point displays 0, since the statement `n=2` has not been executed yet. The command `next` advances execution to line 5, and if the `print n`

command is now repeated it displays a 2.

The command `print twobytwo` displays the entire matrix, one element per line. Note that square brackets (not parentheses) are used to reference array elements. The command `print matrix` fails because subroutine `mkidentity` is not active at this point and the bounds of the adjustable array `matrix` are not known.

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`. The `which` command shows exactly which variable *n* is being referenced. The `function` and `file` commands can be used to alter *dbx*'s definition of the current procedure. The `status` command lists the breakpoints in effect and the `delete` command removes breakpoints.

Execution can be continued in three ways: `continue` resumes execution without setting further breakpoints, `next` sets a one-time breakpoint at line 5 of file `a1.f` and continues execution until that point is reached; and `step` sets a breakpoint at the next source line to be executed—in this case, line 4 of file `a2.f`.

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` had not yet 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 your program and invoked by the user at appropriate breakpoints.

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, which then allows `where` commands to determine which routines were active at the time of the exception:

```
determinant(m =  ARRAY , dim = 16776938), line 5 in "a3.f"
MAIN, line 6 in "a1.f"
main(0x1, 0xfffeb0, 0xfffeb8) at 0x82fa
```

5.3. Using adb

The *adb* debugger can also be used to provide a stack traceback but at a lower level. For example, `adb silly core` starts up *adb* and the command `$c` displays something like

```
_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 to be 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`.

adb can be used on any program regardless of whether it was compiled with the 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. To examine this file you could type

```
% adb fort.4 -
```

to invoke *adb*, and the command `0,3?D` to display the first three words of the file in decimal (location 0 with a repeat count of three).

5.4. Compiler flags

The compiler provides three optional flags that are useful for debugging a FORTRAN program: **-C**, **-u**, and **-v**. The **-C** flag causes the compiler to generate code that tests whether subscript expressions are in bounds. 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.
```

Note that 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.

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. 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.

5.5. 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,

```
hostname% 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`. 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 considered.

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 calls it most often and causes it to do the most work. The second summary produced by `gprof`, the "graph" profile, can help us 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 a graph profile, the line that ends with "[3]" is called 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 "descendant time" descendants were spent on behalf of MAIN. If `mkidentity` had been called from two procedures 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.

For programs that wish to keep track of their own timing, the FORTRAN library includes two routines that return the total time used by the calling process — see *dtime*(3F) and *etime*(3F).

Chapter 6

Deviations from the Fortran 77 Standard

FORTRAN 77 includes almost all of FORTRAN 66 as a subset. Chapter 7 contains a brief description of the differences between FORTRAN 66 and FORTRAN 77.

The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

This chapter is in two major parts. The first part describes extensions to the ANSI standard that the Sun FORTRAN compiler (*f77*) and run-time system implement. The second part describes areas where this compiler and runtime system violate the ANSI standard, usually because the compiler or runtime system cannot correctly implement the ANSI standard.

6.1. Extensions to the FORTRAN 77 Standard

In addition to implementing the language specified in the ANSI standard, the Sun *f77* compiler implements some extensions. Some of them are useful additions to the language. The remaining ones make it easier to communicate with C procedures or to permit compilation of old FORTRAN 66 programs.

6.1.1. Double Complex Data Type

The new type `double complex` is defined. Each datum is represented by a pair of double-precision real variables. A double complex version of each `complex` built-in function is provided. The specific function names begin with `z` instead of `c`.

6.1.2. Internal Files

The FORTRAN 77 standard introduces 'internal files' (memory arrays) but restricts their use to formatted sequential I/O statements. The Sun *f77* I/O system also permits internal files to be used in direct formatted reads and writes.

6.1.3. Implicit Undefined statement

FORTRAN 66 has a fixed rule that the type of a variable that does not appear in a type statement is `integer` if its first letter is `i`, `j`, `k`, `l`, `m`, or `n`, and `real` otherwise. FORTRAN 77 has an `implicit` statement for overriding this rule. As an aid to good programming practice, the Sun *f77* compiler has an additional data type named `undefined`. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and *f77* issues a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **-u** compiler flag on the command line is equivalent to beginning each procedure with this statement.

6.1.4. Recursion

Procedures can call themselves, directly or through a chain of other procedures. But note that a subroutine or function can not 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 makes FORTRAN programs nonportable.

6.1.5. Automatic Storage

Two new keywords are recognized, `static` and `automatic`. These keywords can appear as 'types' in type statements and in `IMPLICIT` statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared `automatic` for each invocation of the procedure. Automatic variables cannot appear in `equivalence`, `data`, or `save` statements.

6.1.6. Source Input Format

The standard expects programs to to be in 72-column format. Except in comment lines, the first five characters are the statement number, the sixth is the continuation character, and the next 66 are the body of the line. If a line of this format contains fewer than 72 characters, *f77* pads it with blanks. Characters after the 72nd are ignored.

In order to make it easier to type FORTRAN 77 programs, this compiler also accepts input in variable-length lines. An ampersand ('&') in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by *f77*. Lines containing a tab among the first six characters or lines beginning with an ampersand are not padded with blanks, nor does *f77* ignore characters past the 72nd character in lines of this format.

In the standard, there are only 26 letters — FORTRAN 77 is a one-case language. Consistent with ordinary UNIX system usage, this compiler expects lower-case input. By default, the compiler converts all upper case characters to lower-case except those inside character constants. However, if the **-U** compiler flag is specified, upper-case letters are not transformed. In this mode, it is possible to specify external names with upper-case letters in them, and to have distinct variables differing only in case. However, when **-U** is specified, FORTRAN 77 reserved words are only recognized in lower case.

6.1.7. Include Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file `stuff`. `includes` can be nested to a reasonable depth, currently ten.

6.1.8. Binary Initialization Constants

A `logical`, `real`, or `integer` variable can be initialized in a `data` statement by a binary constant denoted by a letter and followed by a quoted string. If the letter is `b`, the string is binary, and only zeroes and ones are permitted. If the letter is `o`, the string is octal, with digits `0–7`. If the letter is `z` or `x`, the string is hexadecimal, with digits `0–9`, `a–f`. Thus, the statements

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of `a` to ten.

6.1.9. Character Strings

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

Table 6-1: Backslash Escape Sequences

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

Standard FORTRAN 77 has only one quoting character — the apostrophe. This 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.

Every unequivalenced scalar local character variable and every character string constant is aligned on a word boundary. Each character string constant appearing outside a `data` statement is followed by a null character to ease communication with C routines.

6.1.10. Hollerith

FORTRAN 77 does not have the old Hollerith (*n*h) notation, although the FORTRAN 77 standard recommends implementing the Hollerith feature in order to improve compatibility with old programs. In this compiler, Hollerith data can be used in place of character string constants, and can also be used to initialize noncharacter variables in `data` statements.

6.1.11. Equivalence Statements

As a very 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. The Sun *f77* 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.

6.1.12. One-Trip DO Loops

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, as in

```
do 10 i = 2, 1
```

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. In order to accommodate old programs, though they violate the FORTRAN 66 standard, the **-onetrip** compiler flag makes *f77* generate nonstandard loops.

6.1.13. Commas in Formatted Input

The I/O system attempts to be more lenient than described in the standard when it seems worthwhile. When doing a formatted read of noncharacter variables, commas can be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

reads the record

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

correctly.

6.1.14. Short Integers

f77 accepts declarations of type `integer*2`. 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 halfword integers are of C type `short int`. An expression involving only objects of type `integer*2` is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled

using the `-i2` flag, all integer constants that fit 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, all quantities of type `logical` are short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

6.1.15. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the FORTRAN 77 standard. In addition, there are functions for performing bitwise Boolean operations (`or`, `and`, `xor`, and `not`) and for accessing the UNIX command arguments (`getarg` and `iargc`) and environment (`getenv`).

6.2. Violations of the Standard

There are only a few ways in which this implementation of FORTRAN 77 system violates the ANSI FORTRAN 77 standard.

6.2.1. Dummy Procedure Arguments

If any argument of a procedure is of type `character`, all dummy procedure arguments of that procedure must be declared in an `external` statement. This requirement arises as a subtle corollary of the way character string arguments are represented and of the one-pass nature of the compiler. A warning is printed if a dummy procedure argument is not declared `external`. Code is correct without any external declarations if there are no `character` arguments.

6.2.2. T and TL Formats

The implementation of the `t` (absolute tab) and `tl` (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record that has already been processed. The I/O library uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to predeclare any record lengths except where specifically required by FORTRAN 77 or the operating system.

6.2.3. Carriage Control

The ANSI standard leaves the logical unit(s) that are treated as 'printer' files as implementation-dependent. In this implementation, there are no printer files and thus carriage control specifiers such as '+' are not implemented. It would be difficult to implement these carriage-control characters correctly and still provide UNIX-like file I/O.

Furthermore, the carriage control implementation is asymmetrical. A file written with carriage control interpretation cannot be read again with the same characters in column 1.

An alternative to interpreting carriage control internally is to run the output file through a FORTRAN 'output filter' before printing (see the *fpr*(1) command in the *User's*Manual).

6.2.4. Assigned Goto

The optional list associated with an assigned `goto` statement is not checked against the actual assigned value during execution.

6.2.5. Default files

Files created by default uses of `rewind` or `endfile` statements are opened for `sequential formatted` access. There is no way to redefine such a file to allow `direct` or `unformatted` access.

6.2.6. Lower case strings

It is not clear if the ANSI standard requires internally generated strings to be upper case or not. As currently written, the `inquire` statement returns lower-case strings for any alphanumeric data.

6.2.7. Exponent representation on Ew.dEe output

If the field width for the exponent is too small, the ANSI standard allows dropping the exponent character, but only if the exponent is > 99. This system does not enforce that restriction.

6.2.8. Repeat counts for null values

Repeat counts for null values on list-directed input are not recognized correctly.

Chapter 7

Differences Between FORTRAN 77 and FORTRAN 66

The following is a very brief description of the differences between the 1966 [2] and 1977 [1] standard languages. We assume that you are familiar with FORTRAN 66.

7.1. Deleted FORTRAN 66 Features

7.1.1. Hollerith

The notion of 'Hollerith' (***n*h**) as data has officially been removed from the standard, although this compiler, like almost all in the foreseeable future, still supports this anachronism.

7.1.2. Extended Range

In FORTRAN 66, under a set of very 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.

7.2. Program Form

7.2.1. Blank Lines

Completely blank lines are now legal comment lines.

7.2.2. Program and Block Data Statements

A main program can now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures can also have a name:

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure can appear in a program. This system does not enforce that rule. The standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

7.2.3. ENTRY Statement

Multiple entry points are now legal. Subroutine and function subprograms can have additional entry points, declared by an `entry` statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the `entry` line. All variable declarations 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 entry point, with the type determined by declared entry name type. If any entry is a character-valued function, then all entries must be. 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.

7.2.4. DO Loops

`do` variables and range parameters may now be of `integer`, `real`, or `double precision` types. The use of floating-point `do` variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against it. The action of the `do` statement is now defined for all values of the `do` parameters. The statement

```
do 10 i = 1, u, d
```

performs $\max(0, \lfloor (u-l)/d \rfloor)$ iterations. The `do` variable has a predictable value when exiting a loop — the value at the time a `goto` or `return` terminates the loop; otherwise, it is the value that failed the limit test.

7.2.5. Alternate Returns

In a `subroutine` or subroutine `entry` statement, some of the arguments can be denoted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the 'alternate returns' is described in the section named "Alternate Returns" found later in this chapter.

7.2.6. CHARACTER Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted, it is assumed equal to 1. A character string argument can have a constant length, or the length can be declared to be the same as that of the corresponding actual argument at runtime by a statement like

```
character*(*) a
```

There is an intrinsic function `len` that returns the actual length of a character string. Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

7.2.7. IMPLICIT Statement

The traditional implicit declaration rules still hold — a variable whose name begins with `i, j, k, l, m,` or `n` is of type `integer`, other variables are of type `real`, unless otherwise declared. This general rule may be overridden with an `implicit` statement:

```
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 assumed `complex`, and so on. It is still poor practice to depend on implicit typing, but this statement is part of the standard.

7.2.8. PARAMETER Statement

It is now possible to give a constant a symbolic name, as in

```
parameter (x=17, y=x/3, pi=3.14159d0, s='hello')
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

7.2.9. Array Declarations

Arrays can now have as many as seven dimensions — only three were permitted in FORTRAN 66. The lower bound of each dimension can be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound can be an integer expression involving constants, arguments, and variables in common:

```
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:

```
integer a(5, *), b(*), c(0:1, -2:*)
```

7.2.10. SAVE Statement

A FORTRAN 66 rule that is not widely known is that 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. FORTRAN 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
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.

7.2.11. INTRINSIC Statement

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. If an intrinsic function is to be passed to another procedure, it must be declared `intrinsic`. Declaring it `external` (as in FORTRAN 66) passes a function other than the built-in one.

7.3. Expressions

7.3.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

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

There are no null (zero-length) character strings in FORTRAN 77. The Sun compiler has two different quotation marks, " ' " and " " ".

7.3.2. Concatenation

Character string concatenation has been added and is marked by a double slash ('//'). The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The values of

```
'ab' // 'cd'
```

and

```
'abcd'
```

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared adjustable with a '*(*)' modifier, or a substring denotation with nonconstant position values can appear are on the right sides of assignments).

7.3.3. 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.

7.3.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

```
a(i,j) (m:n)
```

is the string of *(n-m+1)* characters beginning at the m^{th} character of the character array element *a(i,j)*. The result is undefined unless $m<=n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

7.3.5. Exponentiation

It is now permissible to raise 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 now defined:

```
a**b**c = a ** (b**c)
```

7.3.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. For instance, it is permissible to combine integer and complex quantities in an expression.

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 may now be general integer expressions; the old *cv±c'* rules have been removed. `do` loop bounds may be general integer, real, or double-precision expressions. Computed `goto` expressions and I/O unit numbers can be general integer expressions.

7.4. Executable Statements

7.4.1. IF-THEN-ELSE

At last, the if-then-else branching structure has been added to FORTRAN. It is 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. If the logical expression in the Block If statement is `.true.`, the statements following it up to the next `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 are true, control passes to the statements following the `else` statement, if any. The `else` must follow all `else if`s in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures. A case construct can be set up:

```
if (s .eq. 'ab') then
 . . .
else if (s .eq. 'cd') then
 . . .
else
 . . .
end if
```

7.4.2. Alternate Returns

Some of the arguments of a subroutine call can be statement labels preceded by an asterisk, as in

```
call joe(j, *10, m, *2)
```

A `return` statement may have an integer expression, such as

```
return k
```

If the entry point has n alternate return (asterisk) arguments and if $1 \leq k \leq n$, the return is followed by a branch to the k^{th} statement label; otherwise the usual return to the statement following the `call` is executed.

7.5. Input/Output

7.5.1. Format Variables

A format can be the value of a character expression (constant or otherwise), or be stored in a character array, as in

```
write(6, '(i5)') x
```

7.5.2. END=, ERR=, and IOSTAT= Clauses

A `read` or `write` statement can contain `end=`, `err=`, and `iostat=` clauses, as in

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the units on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and `a` and `x` are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the `iostat=` clause is given a value when the I/O statement finishes (the value is assigned to the name on the right side of the equal sign). This value is zero if all went well, negative for end of file, and positive for an error.

7.5.3. Formatted I/O

7.5.3.1. Character Constants

Character constants in formats are copied literally to the output. Character constants cannot be read into.

```
write(6,'(i2," isn""t ",i1)') 7, 4
```

produces

```
 7 isn't 4
```

Here the format is the character constant

```
(i2,' isn"t ',i1)
```

and the character constant

```
isn't
```

is copied into the output.

7.5.3.2. Positional Editing Codes

`t`, `tl`, `tr`, and `x` codes control where the next character is in the record. `tr`*n* or *n*`x` specifies that the next character is *n* to the right of the current position. `tl`*n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. `t`*n* says that the next character is to be character number *n* in the record.

7.5.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list, otherwise it has no effect. In the fragment

```
x='("hello", :, " there", i4)'
write(6, x) 12
write(6, x)
```

the first `write` statement prints:

```
hello there 12
```

while the second only prints

```
hello
```

7.5.3.4. Optional Plus Signs

According to the standard, each implementation has the option of putting plus signs in front of nonnegative numeric output. The `sp` format code can be used to make the optional plus signs actually appear for all subsequent items while the format is active. The `ss` format code guarantees that the I/O system does not insert the optional plus signs, and the `s` format code restores the default behavior of the I/O system. Since *f77* doesn't normally put out optional plus signs, the `ss` and `s` codes have the same effect.

7.5.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks are ignored following a `bn` code in a format statement, and are treated as zeros following a `bz` code in a format statement. The default for a unit can be changed by using the `open` statement. Blanks are ignored by default.

7.5.3.6. Unrepresentable Values

The ANSI standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks.

7.5.3.7. iw.m

A new integer output code **iw.m.** is the same as **iw**, except that there are at least *m* digits in the output field, including, if necessary, leading zeros. The case **iw.0** is special, since if the value being printed is 0, the output field is entirely blank. `iw.1` is the same as `iw`.

7.5.3.8. Floating Point

On input, exponents can start with the letter `E`, `D`, `e`, or `d`. All have the same meaning. On output, always use `e`. The `e` and `d` format codes also have identical meanings. A leading zero before the decimal point in `e` output without a scale factor is optional with the implementation. *f77* does not print it. There is a `g`*w.d* format code which is the same as `e`*w.d* and `f`*w.d* on input, but which chooses `f` or `e` formats for output depending on the size of the number and of *d*.

7.5.3.9. 'A' Format Code

A codes are used for character values. `a`*w* use a field width of *w*, while a plain `a` uses the length of the character item.

7.5.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a,b
```

Similarly, the standard output units is specified by a `print` statement or an asterisk unit in a `write`:

```
print 10
write(*, 10)
```

7.5.5. List-Directed Formatting

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

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and (possibly) a comma. Values, except for character strings, cannot contain blanks. 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. Values can be preceded by repetition counts, as in

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

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

For output, a suitable format is chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

7.5.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records can be written or read in any order, using direct access I/O statements.

Direct access `read` and `write` statements have an extra argument, `rec=`, which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array `a`.

The size of the records must be given by an `open` statement (see below). Direct access files can be connected for either formatted or unformatted I/O.

7.5.7. 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 very precise term to use here, but internal files are dealt with using `read` and `write`.) Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,'(a)') x
read(x,'(i3,i4)') n1,n2
```

which 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.

f77 also supports a compatible extension, direct I/O on 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.

7.5.8. open

The `open` statement connects a file with a unit, or alters some property of the connection.

It has the following format:

open(***parameter list***)

where

parameters is a list of optional specifiers, separated by commas. For valid specifiers see the section called "open" in Chapter 3.

7.5.8.1. close

`close` severs the connection between a unit and a file. The unit number must be given. The optional parameters are `iostat=` and `err=` with their usual meanings, and `status=` either '`keep`' or '`delete`.' Scratch files cannot be kept; otherwise `keep` is the default. `delete` means the file will be removed. A simple example is

```
close(3, err=17)
```

7.5.8.2. inquire

The `inquire` statement gives information about a unit (inquire by unit) or a file (inquire by file). It sets values of integer, logical, and character variables by specifing keywords that correspond to the values of unit, connection, or file properties. For the semantics of this command see "inquire" in Chapter 3.

Appendix A

Ratfor — A Preprocessor for a Rational FORTRAN

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 fron loops
using `break` and `next` statements

free-form input
multiple statements per line and automatic continuation

unobtrusive comments
signalled 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.

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 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 appendix discusses design criteria for a FORTRAN preprocessor, the Ratfor language and its implementation, and user experience.

Note that since the original Ratfor was designed, the new FORTRAN 77 language has appeared on the scene. 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.

A.1. 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.

A.1.1. 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 ***-6x***, 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.

Rc provides an interface to the ***Ratfor*** command, which is much the same as ***cc***. Thus

```
hostname% rc [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

[1] This chapter is a revised and expanded version of a paper published in *Software — Practice and Experience*, October 1975.

-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.

A.2. Language Description

A.2.1. 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 appendix contains an informal description of the *Ratfor* language. The control flow aspects and cosmetic changes will look familiar if you are used to languages like Algol, PL/I, and Pascal.

A.2.2. 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 (like `"""x>100""" )`, they are not allowed in ANSI FORTRAN, so ***Ratfor*** converts quoted strings into the right number of **L '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.

A.2.3. 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.

A.2.4. 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
```

A.2.5. if-else ambiguity

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
```

A.2.6. 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`.)

A.2.7. 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.

A.2.8. '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.)

A.2.9. 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 commonsense 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.

A.2.10. 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.

A.2.11. 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.

A.2.12. 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`.

A.2.13. '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.)

A.2.14. 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.

A.2.15. 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)
```

A.2.16. 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
[	{	]	}
(\$	{	\$)	}

A.2.17. '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
```

A.2.18. '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

A.2.19. 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.

A.3. 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 [22]. 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.

A.4. Experience

A.4.1. 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 [17]:

```
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 [15].

A.4.2. 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 [3], 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.

A.5. 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.

Appendix B

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

Appendix C

Runtime Error Messages

The FORTRAN I/O library, the FORTRAN signal handler or parts of the UNIX operating system (when called by FORTRAN library routines) can all generate FORTRAN error messages. UNIX error messages include system call failures, C library errors, and shell diagnostics.

C.1. UNIX error messages

UNIX error messages include system call failures, C library errors, and shell diagnostics. The system call error messages are found in *intro*(2) in the Sun *System Interface Manual*. 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
```

This example shows that system calls made via the FORTRAN library do not produce error messages directly. For example, the program

```
integer unlink
        i=unlink("/")
        if(i.ne.0) call perror("unlink")
        end
```

produces the message

```
unlink: Invalid argument
```

whereas the program

```
integer unlink
        i=unlink("/")
        end
```

produces no output.

C.2. 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.

As an example of a signal handler error, the subroutine **sub** tries to access parameters that are not passed:

```
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)
```

C.3. FORTRAN 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. For example, this program tries to do an unformatted write to a file opened for formatted output:

```
write(6) 1
end
```

```
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 error message output for the location of the error in the format. 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. 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 just right.

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. Logical data must be T or F.

116 *blank logical input field*

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.

Appendix D

Bibliography

The following books or documents describe aspects of FORTRAN 66, FORTRAN 77, *Ratfor 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. Baker, B. S. December 1975. *Struct — A Program which Structures FORTRAN.* Bell Laboratories internal memorandum.

4. Brainerd, Walter S., et al. 1978. *FORTRAN 77 Programming.* Harper and Row.

5. Day, A. C. 1979. *Compatible Fortran.* Cambridge University Press.

6. Dock, V. Thomas. 1979. *Structured FORTRAN 77 Programming.* West.

7. Feldman, S. I. June 1979. The Programming Language EFL. *Bell Laboratories Technical Report.*

8. *For-word: FORTRAN Development Newsletter*, August 1975.

9. Hall, A. D. August 1971. The Altran System for Rational Function Manipulation — A Survey. *CACM.*

10. Hume, J. N., and R. C. Holt. 1979. *Programming FORTRAN 77.* Reston.

11. Johnson, S. C. January 1978. A Portable Compiler: Theory and Practice. *Proc. 5th ACM Symp. on Principles of Programming Languages*

12. Johnson, S. C. 1978. YACC — Yet Another Compiler-Compiler. *Bell Laboratories Computing Science Technical Report #32.*

13. Katzan, Harry, Jr. 1978. *FORTRAN-77.* Van Nostrand-Reinhold.

14. Kernighan, B. W., and D. M. Ritchie. 1978. *The C Programming Language*, Prentice-Hall.

15. Kernighan, B. W. January 1977. RATFOR — A Preprocessor for a Rational Fortran. *Bell Laboratories Computing Science Technical Report #55,*

16. Kernighan, B. W., and P. J. Plauger. 1976. *Software Tools*. Addison-Wesley.

17. Knuth, D. E. December 1974. Structured Programming with goto Statements. *Computing Surveys.*

18. Meissner, Loren P., and Elliott I. Organick. 1979. *FORTRAN-77 Featuring Structured Programming.* Addison-Wesley.

19. Merchant, Michael J. 1979. *ABC's of FORTRAN 77 Programming.* Wadsworth.

20. Page, Rex, and Richard Didday. 1980. *FORTRAN 77 for Humans.* West.

21. Ritchie, D. M., and K. L. Thompson. July 1974. *The UNIX Time-sharing System.* CACM.

22. Ryder, B. G. October 1974. The PFORT Verifier. *Software—Practice & Experience.*

23. United States of America Standards Institute. March 7, 1966. *USA Standard FORTRAN, USAS X3.9-1966.* New York. Clarified in *Comm. ACM* **12,** *289 (1969) and Comm. ACM* **14,** *628 (1971).*

24. Wagener, Jerrold L. 1980. *Principles of FORTRAN 77 Programming.* Wiley.

25. *A Proposed Standard For Binary Floating-Point Arithmetic*, Draft 10.0 of IEEE Task, p754. December l982.

Appendix E

FORTRAN Library Routines

NAME

intro – introduction to FORTRAN library functions

DESCRIPTION

This section describes those functions that are in the FORTRAN run-time 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 77 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
bessel functions		bessel.3fof two kinds for integer orders
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
dflmax	range.3f	return extreme values
dflmin	range.3f	return extreme values
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
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
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	tell about character objects
inmax	range.3f	return extreme values
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	tell about character objects
link	link.3f	make a link to an existing file
lnblnk	index.3f	tell about character objects
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
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	tell about character objects
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
trpfpe	trpfpe.3f	trap and repair floating point faults
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 preceeded 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

bessel functions — of two kinds for integer orders

SYNOPSIS

function besj0 (x)

function besj1 (x)

function besjn (n, x)
integer*4 n

function besy0 (x)

function besy1 (x)

function besyn (n, x)
integer*4 n

double precision function dbesj0 (x)
double precision x

double precision function dbesj1 (x)
double precision x

double precision function dbesjn (n, x)
integer*4 n
double precision x

double precision function dbesy0 (x)
double precision x

double precision function dbesy1 (x)
double precision x

double precision function dbesyn (n, x)
integer*4 n
double precision x

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

DIAGNOSTICS

Negative arguments cause *besy0*, *besy1*, and *besyn* to return a huge negative value. The system error code will be set to EDOM (33).

FILES

/usr/lib/libF77.a

SEE ALSO

j0(3m), perror(3F)

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.

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 result in the run-time 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

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 if 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 $\geq$ 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 objects

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 size of the character object 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. 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 f77 program *myprogram* included 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

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"

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)

DESCRIPTION

These functions use *random*(3) to generate sequences of random numbers. 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

flmin, flmax, dflmin, dflmax, inmax – return extreme values

SYNOPSIS

function flmin()

function flmax()

double precision function dflmin()

double precision function dflmax()

function inmax()

DESCRIPTION

Functions *flmin* and *flmax* return the minimum and maximum positive floating point values respectively. Functions *dflmin* and *dflmax* return the minimum and maximum positive double precision floating point values. Function *inmax* returns the maximum positive integer value.

These functions can be used by programs that must scale algorithms to the numerical range of the processor.

The values returned by *flmin* and *dflmin* are the smallest normalized IEEE format floating point values. The values returned by *flmax* and *dflmax* are the largest finite IEEE format floating point values.

The approximate values of these functions for the Sun Workstation are:

flmin 1.175494e−38

flmax 3.402823e+38

dflmin 2.2250738590e−308

dflmax
1.7976931349e+308

inmax 2147483647

FILES

/usr/lib/libF77.a

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(12)

integer function lstat (name, statb)
character*(*) name
integer statb(12)

integer function fstat (lunit, statb)
integer statb(12)

DESCRIPTION

These routines return detailed information about a file. *Stat* and *lstat* return information about file *name*; *fstat* returns information about the file associated with fortran logical unit *lunit*. 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.

The value of either function will be zero if successful; an error code otherwise.

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* disect 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

trpfpe, fpecnt – trap and repair floating point faults

SYNOPSIS

subroutine trpfpe (numesg, rtnval)
double precision rtnval

integer function fpecnt ()

common /fpeflt/ fperr
logical fperr

DESCRIPTION

NOTE: This routine applies only to Vax computers. It is a null routine on the PDP11.

Trpfpe sets up a signal handler to trap arithmetic exceptions. If the exception is due to a floating point arithmetic fault, the result of the operation is replaced with the *rtnval* specified. *Rtnval* must be a double precision value. For example, "0d0" or "dflmax()".

The first *numesg* occurrences of a floating point arithmetic error will cause a message to be written to the standard error file. Any exception that can't be repaired will result in the default action, typically an abort with core image.

Fpecnt returns the number of faults since the last call to *trpfpe*.

The logical value in the common block labelled *fpeflt* will be set to **.true.** each time a fault occurs.

FILES

/usr/lib/libF77.a

SEE ALSO

signal(3f), range(3f)

BUGS

This routine works only for *faults*, not *traps*. This is primarily due to the Vax architecture.

If the operation involves changing the stack pointer, it can't be repaired. This seldom should be a problem with the f77 compiler, but such an operation might be produced by the optomizer.

The POLY and EMOD opcodes are not dealt with.

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)

READER COMMENT SHEET

Dear Customer,
We who work here at Sun Microsystems wish to provide the best possible documentation for our products. To this end, we solicit your comments on this manual. We would appreciate your telling us about errors in the content of the manual, and about any material which you feel should be there but isn't.

Typographical Errors:
Please list typographical errors by page number and actual text of the error.

Technical Errors:
Please list errors of fact by page number and actual text of the error.

Content:
Please list errors of fact by page number and actual text of the error.