GIT-ICS-80/03

SOFTWARE TOOLS SUBSYSTEM
USER'S GUIDE
2ND EDITION

APRIL, 1980

School of
Information and Computer Science

GEORGIA INSTITUTE
OF TECHNOLOGY

GIT-ICS-80/03

SOFTWARE TOOLS SUBSYSTEM
USER'S GUIDE
2ND EDITION

APRIL, 1980

T. Allen Akin
Perry B. Flinn
Daniel H. Forsyth, Jr.

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

INTRODUCTION TO THE
GEORGIA TECH SOFTWARE TOOLS SUBSYSTEM USER'S GUIDE

The documents following this Introduction comprise the most recent version of the User's Guide for the Georgia Tech Software Tools Subsystem for Prime 350 and larger computers. This Guide brings together in one place all the tutorial and reference information useful to novice and intermediate users of the Subsystem. It deals with several important aspects of Subsystem use: the user interface in general, unavoidable aspects of the underlying operating system, and the most-frequently used major commands. Each topic is covered in a separate document (available individually) and all documents are collected together with this Introduction to form the Guide itself. Experienced users, as well as beginning users who wish to expand their knowledge of the Subsystem, will find the Software Tools Subsystem Reference Manual valuable.

The development of the Georgia Tech Software Tools Subsystem was originally motivated by the text Software Tools by Brian W. Kernighan and P. J. Plauger, Addison-Wesley, 1976. That text is still the basic reference for the tools that it covers, particularly Ratfor, the text editor, the macro preprocessor, and the text formatter.

SOFTWARE TOOLS SUBSYSTEM TUTORIAL

USER'S GUIDE TO THE PRIMOS FILE SYSTEM

INTRODUCTION TO THE SOFTWARE TOOLS SUBSYSTEM TEXT EDITOR

USER'S GUIDE FOR THE SOFTWARE TOOLS SUBSYSTEM COMMAND INTERPRETER

USER'S GUIDE TO THE RATFOR PREPROCESSOR

SOFTWARE TOOLS TEXT FORMATTER USER'S GUIDE

Software Tools Subsystem Tutorial

T. Allen Akin
Perry B. Flinn
Daniel H. Forsyth, Jr.

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

April, 1980

TABLE OF CONTENTS

Foreword

The **Software Tools Subsystem** is powerful collection of program development and text processing tools developed at the Georgia Tech School of Information and Computer Science, for use on Prime 350 and larger computer systems. The tutorial that you are now reading is intended to serve as your first introduction to the Subsystem and its many capabilities. The information contained herein applies to Version 7 of the Subsystem as released in April 1980.

Introduction

The Software Tools Subsystem is a programming system based on the book Software Tools, by Brian W. Kernighan and P. J. Plauger, (Addison-Wesley Publishing Company, 1976), that runs under the Primos operating system on Prime 350 and larger computers. It allows much greater flexibility in command structure and input/output capabilities than Primos, at some small added expense in processing time.

This tutorial is intended to provide sufficient information for a beginning user to get started with the Subsystem, and to acquaint him with its basic features; it is by no means a comprehensive reference. Readers desiring a more detailed exposition of the Subsystem's capabilities are referred to the Software Tools Subsystem Reference Manual and to the remainder of the Software Tools Subsystem User's Guide, of which this Tutorial is a part.

Getting Started

Since the Subsystem is composed entirely of ordinary user-state programs, as opposed to being a part of the operating system, it must be called when needed.

The following example shows how a typical terminal session might begin. Items typed by the user are boldfaced.

```
OK, login login_name                              (1)
Password:                                         (2)
LOGIN_NAME (15) LOGGED IN AT 9:22 010578          (3)
OK, swt                                           (4)
Password:                                         (5)
]                                                 (6)
```

(1) A terminal session is initiated when you type the Primos 'login' command. "Login_name" here represents the login name that you were assigned when your account was established.

(2) Primos asks you to enter your login password (if you have one) and turns off the terminal's printer. You then type your password (which is not echoed) followed by a newline. (Note: password checking on login is a feature of Georgia Tech Primos; at other installations, this line will be omitted.)

(3) Primos acknowledges a successful login by typing your login name, your process number (in parentheses), and the current time and date.

(4) Primos indicates it is ready to accept commands by typing "OK,". (Whenever you see this prompt, Primos is waiting for you to type a command.) Type 'swt' (for "Software Tools") to start up the Subsystem.

(5) 'Swt' prompts you for your Subsystem password. This password will have been assigned to you by your Subsystem Manager at the time he created your Subsystem account. (Note: Under Georgia Tech Primos, Subsystem passwords are never issued, nor prompted for by 'swt'.) After you receive the prompt, enter your Subsystem password. It will not be printed on the terminal.

(6) The Subsystem's command interpreter prompts with "]", indicating that it is ready to accept commands.

When the Subsystem command interpreter has told you it is waiting for something to do (by typing the "]"), you may proceed to enter commands. Each command consists of a 'command-name', followed by zero or more 'arguments', all separated from each other by blanks:

```
command-name  argument  argument  ...
```

The command name is necessary so that the command interpreter knows what it is you want it to do. On the other hand, the arguments, with a few exceptions, are completely ignored by the command interpreter. They consist of arbitrary sequences of characters which are made available to the command when it is invoked. For this reason, the things that you can type as arguments depend on what command you are invoking.

When you have finished typing a command, you inform the command interpreter of this by hitting the "newline" key. (On some terminals, this key is labeled "return", or "cr". If both the "newline" and "return" keys are present, you should use "return".)

Incidentally, if you get some strange results from including any of the characters

" ' # | , ; ( ) { } [ ] >

within a command name or argument, don't fret. These are called "meta-characters" and each has a special meaning to the command interpreter. We will explain some of them later on. For a more complete description of their meaning, see the User's Guide for the Software Tools Subsystem Command Interpreter.

Correcting Typographical Errors

If you are a perfect typist, you can probably skip this part. But, if you are like most of us, you will make at least a few typos in the course of a session and will need to know how to correct them.

There are three special characters used in making corrections. The "erase" character causes the last character typed on the line to be deleted. If you want to delete the last three characters you have typed so far, you should type the erase character three times. If you have messed up a line so badly that it is beyond repair, you can throw away everything you have typed on that line in one fell swoop by typing the "kill" character. The result will be that two backslashes ( ) are printed, and the cursor or carriage is repositioned to the beginning of the line. Finally, the "retype" character retypes the present line, so you can see exactly what erasures and changes have been made. You may then continue to edit the line, or enter it by striking the return key.

When you log into the Subsystem for the very first time, your erase, kill and retype characters are control-h (backspace), DEL (RUBOUT on some terminals), and control-r, respectively. You can, however, change their values to anything you wish, and the new settings will be remembered from session to session. The 'ek' command is used to set erase and kill characters:

```
ek erase kill
```

"Erase" should be replaced by any single character or by an ASCII mnemonic (like "BS" or "SUB"). The indicated character will be used as the new erase character. Similarly, "kill" should be replaced by a character or mnemonic to be used as the new kill character. For instance, if you want to change your erase and kill characters back to the default values of "BS" and "DEL", you can use the following command:

```
ek BS DEL
```

(By the way, we recommend that you do not use "e" or "k" for your erase or kill character. If you do, you will be hard pressed to change them ever again!)

Adjusting to Terminal Characteristics

Unfortunately, not all terminals have full upper/lower case capability. In particular, most of the older Teletype models can handle only the upper case letters. In the belief that the use of "good" terminals should not be restricted by the limitations of the "bad" ones, the Subsystem preserves the distinction between upper and lower case letters.

To allow users of upper-case-only terminals to cope with programs that expect lower case input, the 'term' command provides case conversion facilities:

```
term -nolcase
```

This command instructs the Subsystem to force all subsequent upper case letters you type to be converted to lower case, and all lower case letters sent to your terminal by the computer to be converted to upper case. In short, it means "I have no lower-case."

To provide further assistance to users whose terminals have a limited character set, this command also activates a set of "escape" conventions to allow them to enter other special characters not on their keyboard, and to provide for their printing. When the "escape" character (@) precedes another, the two characters together are recognized by the Subsystem as a single character according to the following list:

```
@A   ->   A      (note that  A -> a in "nolcase" mode)
     ...
@Z   ->   Z
@(   ->   {
@)   ->   }
@    ->   ~
@`   ->   `
```

```
@!   ->   |
```

All other characters are mapped to themselves when escaped; thus, "@-" is recognized as "@-". If you must enter a literal escape character, you must enter two: "@@".

If you have given a "term -nolcase" command you must use escapes to enter upper case letters, since everything would otherwise be forced to lower case. For example,

```
@A
```

is used to transmit an upper case 'A', while

```
A
```

is used to transmit a lower case 'A'.

All output generated when the "-nolcase" option is in effect is forced to upper case for compatibility with upper-case-only terminals. However, the distinction between upper and lower case is preserved by prefixing each letter that was originally upper case with an escape character. The same is true for the special characters in the above list. Thus,

```
Software Tools Subsystem
```

would be printed as

```
@SOFTWARE @TOOLS @SUBSYSTEM
```

under the "-nolcase" option.

Finishing Up

When you're finished using the Subsystem, you have several options for getting out. The first three simply terminate the Subsystem, leaving you face to face with bare Primos. We cover them here only for the sake of completeness, and on the off chance that you will actually want to use Primos by itself.

First, you may type

```
] stop
OK,
```

which effects an orderly exit from the Subsystem's command interpreter and gives control to Primos' command interpreter. You will be immediately greeted with "OK,", indicating that Primos is ready to heed your call.

Second, you may enter a control-c (hold the "control" key down, then type the letter "c") immediately after the "]" prompt from the command interpreter. TAKE HEED that this is the standard method of generating an end-of-file signal to a program that is trying to read from the terminal and is widely used throughout the Subsystem. Upon seeing this end-of-file signal, the command interpreter assumes you are finished and automatically invokes the 'stop' command.

The third and least desirable method is to type a control-p or BREAK. This interrupts any running program, including the command interpreter, and (with few exceptions) returns to Primos. This last method is definitely not the method of choice, since it is likely to interrupt the Subsystem at an inconvenient point, causing things to be left in a state of chaos. You are strongly encouraged to choose either of the first two methods over this one.

Finally, we come to the method you will probably want to use most often. The 'bye' command simply ends your terminal session and disconnects you from the computer. The following example illustrates its use. (Once again, user input is boldfaced.)

```
] bye                                                    (1)
LOGIN NAME (15) LOGGED OUT AT 10:16 010578               (2)
TIME USED=  0:25  10:39  1:33                            (3)
OK,                                                      (4)
```

(1) You type the 'bye' command to end your terminal session.

(2) Primos acknowledges, printing the time of logout.

(3) Primos prints a summary of times used.

- The first time is the number of hours and minutes of connect time.
- The second time is the number of minutes and seconds of CPU time.
- The third time is the number of minutes and seconds spent doing disk i/o.

(4) Primos signals it is ready for a new login.

Online Documentation

Users, old and new alike, often find that their memories need jogging on the use of a particular command. It is convenient, rather than having to look something up in a book or a manual, to have the computer tell you what you want to know. Two Subsystem commands, 'help' and 'usage,' attempt to address this need.

The 'Help' Command

The 'help' command is designed to give a comprehensive description of the command in question. The information provided includes the following: a brief, one-line description of what the command does; the date of the last modification to the documentation; the usage syntax for the command (what you must type to make it do what you want it to do); a detailed description of the command's features; a few examples; a list of files referenced by the command; a list of the possible messages issued by the command; a list of the command's known bugs or shortcommings; and a cross reference of related commands or documentation.

'Help' is called in the following manner:

```
help command-1 command-2 ...
```

If help is available for the specified commands, it is printed; otherwise, 'help' tells you that no information is available.

'Help' will only print out about as many lines as will fit on most CRT screens, and then prompt you with the message "more?". This allows you to read the information before it rolls off the screen, and also lets you stop getting the information for a command if you find you're not really interested. To stop the output, just type an "n" or a "q" followed by a NEWLINE. To continue, you may type anything else, including just a NEWLINE.

Several special cases are of interest. One, the command "help" with no arguments is the same as "help general", which gives general information on the Subsystem and explains how to use the help command. Two, the command "help -i" produces an index of all commands supported under the Subsystem, along with a short description of each. Finally, "help bnf" gives an explanation of the conventions used in the documentation to describe command syntax.

Examples of the use of 'help':

```
] help                          (1)
] help -i                       (2)
] help rp ed term               (3)
] help bnf                      (4)
] help guide                    (5)
```

(1) General information pertaining to the Subsystem, along with an explanation of the 'help' command, is listed on the terminal.

(2) A list of currently supported commands and subprograms, each with a short description, is listed on the terminal.

(3) Information on the Ratfor preprocessor, the Software Tools text editor, and the terminal configuration program is printed on the terminal.

(4) A description of the notational conventions used to describe command syntax is printed.

(5) Information on how to obtain the Subsystem User's Guides is listed on the terminal.

Since beginning users frequently find printed documentation helpful, you may find the following procedure useful. Unfortunately, it involves many concepts not yet discussed, so it will be rather cryptic; nevertheless, it will allow you to produce a neatly-formatted copy of output from 'help'.

```
] help -p | os >/dev/lps/f              (1)
] help -p rp se term | os >/dev/lps/f    (2)
] help -p -i | os >/dev/lps/f           (3)
```

(1) The general information entry is printed on the line printer.

(2) Information on the Ratfor preprocessor, the screen editor, and the terminal configuration program is printed on the line printer.

(3) The index of available commands and subprograms is printed on the line printer.

The 'Usage' Command

Whereas 'help' produces a fairly comprehensive description of the command in question, the 'usage' command gives only a brief summary of the syntax of the command. The syntax is expressed in a notation known as Backus-Naur Form (BNF for short) which is itself explained by typing "help bnf".

The 'usage' command is used in the same way as the 'help' command, as the following examples illustrate.

```
] usage usage                                    (1)
] usage fmt help                                 (2)
```

(1) The syntax of the 'usage' command is printed.

(2) Usage information on the Software Tools text formatter and the 'help' command is printed.

Utility Commands

The Subsystem supports several utility commands of general interest. These commands have proven to be important in daily use, and so have been given a separate section in this tutorial. Each will be discussed in turn.

The 'Lf' Command

The 'lf' (for "list files") command is the preferred method for obtaining information about files. Used by itself without any arguments, 'lf' prints the names of all the files in your current directory in a multi-column format. This, however, is by no means all that 'lf' can do. In fact, used in its general form, an 'lf' command looks something like this:

```
lf  options  files
```

The "files" part is simply a list of files and/or directories that you want information about. If the "files" part is omitted, 'lf' assumes you mean the current directory. For each file in the list, information about that file is printed; for each directory listed, information about each file within that directory is printed.

The "options" part of the command controls what information is to be printed. It is composed of a dash ("-") followed by a string of single character option specifiers. Some of the more useful options are the following:

- c — print information in a single column format.
- d — for each directory in the list, print information about the directory itself instead of about its contents.
- l — print all known information about the named files.
- w — print the size (in 16-bit words) of each named file.

(As always, if you would like complete information on 'lf', just use 'help'.) As we said above, if no options are given, then only the names of the files are printed.

Here are some examples of 'lf' commands:

```
] lf                                    (1)
] lf -l                                 (2)
] lf //lkj                              (3)
] lf -cw //lkj =extra=/news             (4)
```

(1) List the names of all files in the current directory, in a multi-column format.

(2) List the names of all files in the current directory, including all information that is known about each file.

(3) List the names of all files in the directory named "lkj".

(4) List the names and sizes of lkj's files in a single-column format, followed by the names and sizes of all files in directory "=extra=/news".

Before discussing filename specifications, a short overview of the Primos file system is needed.

Primos files are stored on several disk packs, each with a unique name. Each pack contains a master file directory (mfd), which contains a pointer to each primary directory on that disk. Each of these primary directories (one for each user, and several special ones for the system) may contain sub-directories, which may themselves contain further sub-directories, ad infinitum. Any directory may also contain ordinary files of text, data, or program code.

The Software Tools Subsystem allows the user to specify the location of any file with a construct known as a "pathname." Pathnames have several elements.

- The first characters of a pathname may be a slash, followed by a disk packname or octal logical disk number, followed by another slash (e.g., "/user_disk/" or "/3/"), in which case the named disk will be searched for the remainder of the pathname. The disk name may be omitted, implying that all disks are to be sear-

ched. When a pathname begins with a slash, the file search operation begins in the master file directory of the disk or disks selected; when it does not begin with a slash, the file search operation begins in the current directory.

- The remainder of the pathname consists of "nodes", separated by slashes. Each node contains the name of a sub-directory or a file, possibly followed by a colon (":") and a password. For example

```
bin
extra
mfd:xxxxxx
```

are nodes.

When nodes are strung together, they describe a path to a file, from anywhere in the file system. Hence the term "pathname." For example,

```
/sys/bin
```

describes the primary directory named "bin", located on the disk whose packname is "sys".

```
//extra/users
```

describes the file named "users" in the directory named "extra" on some unknown disk (all disks will be searched);

```
myfile
```

describes the file or directory named "myfile" in the current directory;

```
mydir:pword/file2
```

describes the file named "file2" in the directory named "mydir" (with password "pword"), which must be a sub-directory of the current directory, since the lack of a leading slash means that the search for the directory is to begin in the current directory.

Certain important Subsystem directories have been given alternative names, called "templates," in order to allow the Subsystem manager to change their location on disk without disturbing existing programs (or users). A template consists of a name surrounded by equals signs ("="). For example, the Subsystem command directory is named "bin". On a standard system, you could list its contents with the command

```
] lf //bin
```

If the Subsystem Manager at your installation had changed the location of the command directory, the command above would not work. To avoid this problem, you could use the template for "bin":

```
] lf =bin=
```

Now your command works regardless of the location of the command directory. There exist templates for all of the most important Subsystem directories; for more information on them, and on pathnames in general, see the User's Guide to the Primos File System.

Now go back and look at the pathnames in the examples for lf. They should be somewhat less mysterious.

The 'Cat' Command

'Cat' is an alias for Kernighan and Plauger's program 'concat', which appears on page 78 of Software Tools. It has a simple function: to concatenate the files named in its argument list, and print them on standard output. If no files are named, it takes input from standard input. (More on standard input and output in the next section, which has examples using 'cat.' For now, just assume that standard input comes from the terminal and standard output goes to the terminal.)

Here are some samples of how to use 'cat'. For more important and useful ones, see the following section.

```
] cat myfile                          (1)
] cat part1 part2 part3               (2)
] cat                                 (3)
```

(1) Prints the file named "myfile" on the user's terminal; i.e., "myfile" is concatenated with nothing and printed on standard output.

(2) Prints the concatenation of the files named "part1", "part2", and "part3" on the terminal.

(3) Copies standard input to standard output. On a terminal, this would cause anything you typed to 'cat' to be echoed back to you. (If you try this, the way to stop is to type a control-c as the first character on the line. As we said before, lots of programs use this end-of-file convention.)

Deleting Files

Sooner or later, you will find it necessary to get rid of some files. The 'del' command serves this need very nicely. It is used like this:

```
del  file1  file2  file3  ...
```

to remove as many files as you wish. Remember that each file can be specified by a pathname, so you are not limited to deleting files in your current directory.

The Subsystem Postal Service

In order to facilitate communication among users, the Subsystem supports a postal service in the form of the 'mail' command. 'Mail' can be used in either of two ways:

```
mail
```

which looks to see if you have been sent any mail, prints it on your terminal, and asks if you would like your mail to be saved, or

```
mail login_name
```

which accepts input from standard input and sends it to the mailbox of the user whose login name is "login_name". Used in this fashion, 'mail' reads until it sees an end-of-file. From the terminal, this means until you type a control-c in column 1. Your letter is postmarked with the day, date and time of mailing and with your login name.

Whenever you enter the Subsystem (by typing "swt") a check is made to see if you have received any mail. If you have, you are told so. When you receive your mail (by typing "mail"), you are asked if you want it to be saved. If you reply "n", the mail you have just received will be discarded. Otherwise, it is appended to the file "=mailfile=", which is located in your profile directory. (You can look at it with 'cat', print it with 'pr', or do anything else you wish to it, simply by giving its name to the proper command. For example,

```
] cat =mailfile=
```

would print all your saved mail on your terminal.)

The Subsystem News Service

Whereas 'mail' is designed for person to person communication, the Subsystem news service is intended for the publication of articles that appeal to a more general interest. The news service is implemented by three commands: 'subscribe', 'publish' and 'news'. The use of the first two should be obvious.

If you wish to subscribe to the new service, simply type

```
] subscribe
```

and then, whenever anyone publishes an article, a copy of it will be delivered to your news box. (You need subscribe to the news service only once; all subscriptions are perpetual.) Whenever you enter the Subsystem, as with mail, a check is made to see if there is anything in your news box; if there is, you are given a message to that effect.

Having gotten such a message, you may then read the news at your convenience by typing

```
] news
```

The news will be printed out on your terminal and then you will be asked whether or not you want to save it. If you say "yes", it will be left in your box and you may read it again at a later date; otherwise, it is discarded. There are other ways to use the 'news' command

that are fully explained by 'help'.

Now suppose you have a hot story that you want to publish. All you have to do is create a file (let's call it "article") whose first line is the headline, followed by the text of the story. Then you type

```
] publish article
```

and your story will be delivered to all subscribers of the news service. If you are a subscriber yourself, you can check this with the 'news' command. In addition, a copy is made in the news archives.

Input/Output

One of the most powerful features of the Software Tools Subsystem is its handling of input and output. As much as possible, the Subsystem has been designed to shield the user from having to be aware of any specific input or output medium; to present to him, instead, a standardized interface with his environment. This facilitates use of programs that work together, without the need for any esoteric or complicated programming techniques. The ability to combine programs as cooperating tools makes them more versatile; and the Software Tools Subsystem makes combining them easy.

Standard Input and Standard Output

Several files are always open and available to any program that runs under the Subsystem. Such files are generically referred to as 'standard ports'; those available for input are called 'standard inputs', and those available for output are called 'standard outputs'.

Standard inputs and standard outputs are initially assigned to your terminal, and revert back to those assignments after each program terminates. However, this can be changed by the command interpreter without any knowledge on the part of user programs. This facility is called "input/output redirection" (or "i/o redirection" for short).

I/O Redirection

As we mentioned, standard input and output are by default assigned to the terminal. Since this is not always desirable, the command interpreter allows them to be redirected (reassigned) to other media. Typically, they are redirected to or from disk files, allowing one program's output to be saved for later use perhaps as the input to another program. This opens the possibility for programs to co-operate with each other. What is more, when programs can communicate through a common medium such as a disk file, they can be combined in ways innumerable, and can take on functions easily and naturally that they were never individually designed for. A few examples with 'cat' below, will help to make this clear.

However, let us first examine the techniques for directing standard inputs and standard outputs to things other than the terminal. The command interpreter supports a special syntax (called a funnel) for this purpose:

```
pathname>       (read "from" pathname)
```

redirects standard input to come from the file named by "pathname";

```
>pathname       (read "toward" pathname)
```

redirects standard output to go to the file named by "pathname". For example, suppose you wanted a copy of your mail, perhaps to look at slowly with the editor. Instead of typing

```
mail
```

which would print your mail on the terminal, you would type

```
mail >mymail
```

which causes your mail to be written to the file named "mymail" in the current directory. It is important to realize that 'mail' does nothing special to arrange for this; it still thinks it is printing mail on the terminal. It is more important to realize that any program you write need not be aware of what file or device it is writing on or reading from.

A bit of terminology from Software Tools: programs which read only from standard input, process the data so gathered, and write only on standard output, are known as "filters." They are useful in many ways.

Examples of Redirected I/O Using 'Cat'

Now, 'cat' does not seem like a particularly powerful command; all it can do is concatenate files and do some peculiar things when it isn't given any arguments. But this behavior is designed with redirected i/o in mind. Look through the following examples and see if they make sense.

```
cat  file1  >file2
```

What this does is to copy "file1" into "file2". Note that since 'cat' sends its output to standard output, we have gained a copy program for free.

```
cat  file1  file2  file3  >total
```

This example concatenates "file1", "file2", and "file3" and places the result in the file named "total". This is probably the most common use of 'cat' besides the simple "cat filename".

```
cat  >test
```

This is an easy way to create small files of data. 'Cat' does not see any filenames for it to take input from, so it reads from standard input. Now, notice that where before, this simply caused lines to be echoed on the terminal as they were typed, each line is now placed in the file named "test". As always, end-of-file from the terminal is generated by typing a control-c in column 1.

One thing that is extremely important is the placement of blanks around i/o redirectors. A funnel (">") must not be separated from its associated file name, and the entire redirector must be surrounded by at least on blank at each end. For example, "file> cat" and "cat >file" are correct, but "file > cat", "cat > file", "file>cat" and "cat>file" are all incorrect, and may cause catastrophic results if used!

You can see that more complicated programs can profit greatly from this system of i/o. After all, from a simple file concatenator we have gained functions that would have to be performed by separate programs on other systems.

Using Primos from the Subsystem

Unfortunately, all the functions of Primos and its support programs have not been neatly bundled into the Subsystem. Three of the Subsystem's commands that attempt to address this problem are the topic of this section.

Executing Primos Commands from the Subsystem

In order to give Subsystem users access to Primos programs and commands without having to go through the work of leaving and re-entering the Subsystem, the 'x' command provides an "escape to Primos."

'X' may be used in either of two ways:

```
x  Primos-command
```

is the method of choice for executing a single Primos command, while

```
x
command-1
command-2
  ...
command-n
control-c
```

allows many Primos commands to be executed. Notice that, in this second mode, 'x' is reading the list of commands from standard input and storing it away. The list is terminated by end-of-file, at which point execution of the commands begins. Note also that because of this, a command such as

```
command_list> x
```

is perfectly legal.

Using Fortran from the Subsystem

The 'fc' command is the Subsystem's interface to Primos' Fortran compiler. As with 'x', it builds a file containing the Primos commands necessary to run the Fortran compiler, and then causes Primos to execute the commands in the file. Fortran has many options -- too many to describe here, or even to remember. 'Fc' automatically selects those options that are best for programs to run under the Subsystem. Others can be selected, of course, and complete information on what the available options are and how they are invoked can be found with the 'help' command.

The easiest and most frequent use of 'fc' goes something like this:

```
fc prog.f
```

This will cause the Fortran program in the file named "prog.f" to be compiled, with the binary output going to a file named "prog.b".

By default, 'fc' names the output file according to the following convention. If the name of the input file ends with ".f", 'fc' replaces the ".f" with ".b" and uses the resulting name as the binary output file. If the name of the input file does not end with ".f", 'fc' just appends the ".b" to it and uses that name for the output file. The following table illustrates this process:

Fortran input file	Binary output file
cat.f	cat.b
ftn_file	ftn_file.b
fortran.s	fortran.s.b

If this naming convention is unacceptable for your particular needs, there is a way to circumvent it; as usual, details are available from 'help'.

The Fortran compiler prints a one line summary for each program unit in the input file. The summary looks something like this:

```
nnnn ERRORS [<routine_name>FTN-REV17.3]
```

where "nnnn" is a four digit count of errors encountered, and "routine_name" is the name of

the program unit just compiled. In addition to this, a diagnostic (though cryptic) message consisting of line number and context is printed for each error encountered. For help in decoding these messages and, more importantly, correcting the errors, we suggest that you consult either a guru or the Fortran manual (Prime publication number PDR3057), whichever is most handy.

Fortran produces an intermediate binary code that is not directly executable. In order to turn it into something useful, Fortran's output must be processed with the Linking Loader, which we will take up next.

Using the Linking Loader from the Subsystem

As we mentioned above, Fortran's binary output must be processed by the Primos Linking Loader before it can be executed. This step in the program development process is necessary to make available to your program the library and system subroutines and functions that you are likely to have used, and to convert it into a form that can be executed.

Unaided use of the Loader can be a painful and finger-fraying experience; much typing and a good bit of otherwise useless knowledge is required. For this reason, the 'ld' command enjoys a prosperous existence.

'Ld' is to the Linking Loader what 'fc' is to Fortran, even down to the modus operandi; it builds a file containing the commands that are most frequently used to invoke and control the Loader, and then hands the file to Primos for execution. The format of the 'ld' command is (the items enclosed in square brackets are optional):

```
ld [-u] binary_input_file ... [-o executable_file]
```

If the "-u" option is specified, 'ld' instructs the Loader to tell you the names of any subroutines or functions that your program called that were not included in any of the binary_input_files. The "-o" option allows you to specify the name of the file in which to save the executable version of your program. If you don't use this option, 'ld' will construct a default name in a way similar to that used by 'fc': if the name of the first binary_input_file ends with ".b", 'ld' will replace the ".b" with ".o" and use the resulting name; otherwise, ".o" is appended to the first binary_input_file name and that name is used. It is very important to remember here that the output file associated with an 'ld' command must be located in the current directory. This restriction is imposed by the Loader and is thus beyond the Subsystem's control.

After you type an 'ld' command, you will see a stream of apparent garbage unfold before your eyes. This is the contents of the file that 'ld' built being echoed by Primos as it executes. Most of this stream is irrelevant. However, there is one important thing to look for: if you see a line that contains only

```
LOAD COMPLETE
```

this indicates that the operation was successful and that, barring any program logic errors, your program will execute correctly. Otherwise, your program has probably called one or more functions or subroutines that were not contained in any of the binary_input_files. (If you used the "-u" option of 'ld', the Loader has printed a list of names that it did not find.) The next section contains an example of the use of 'ld.'

'Ld' has other features that we have not mentioned, mainly because effective use of them requires some familiarity with the Loader. If you're curious, we refer you to 'help' for the additional capabilities of 'ld,' and to Prime's LOAD and SEG Reference Guide (IDR3524) or Fortran Programmer's Guide (FDR3057) for details on the Loader.

Warning

Occasionally, the Fortran compiler, the Loader, or a program called by 'x' will bomb out and leave you stranded outside the Subsystem. To cure this condition, type

```
swt
```

just like you did to enter the Subsystem orignally.

Program Development

One of the most important uses of the Software Tools Subsystem is program development. The Ratfor language presented in Software Tools is an elegant language for software developers, and is the foundation of the Subsystem; virtually all of the Subsystem is written in Ratfor.

Caveats for Subsystem Programmers

Since the Subsystem is exactly that, not an operating system but a sub-system, programs written for it must follow a few simple conventions, summarized below.

- To exit, a program running under the Subsystem should either use a "stop" statement (Ratfor programs only), or call the subroutine "swt". Specifically, the Primos routine "exit" should not be called to terminate a program.
- Whenever possible, Subsystem i/o and utility routines should be used instead of Primos routines, since the latter often do not handle errors in a way suitable for supporting the Subsystem. If, however, other i/o routines must be used, remember that they will not be able to take advantage of standard input and standard output or of any of the other i/o related features provided by the Subsystem.

The Subsystem Text Editor

The first program most users will see when they wish to create another program is 'ed', the Subsystem text editor. A complete description of 'ed' is beyond the scope of this tutorial, but a short list of commands and their formats, as well as an example, should help you get started. For more complete information, refer to Introduction to the Software Tools Text Editor and of course to Software Tools.

'Ed' is an interactive program used for the creation and modification of "text". "Text" may be any collection of characters, such as a report, a program, or data to be used by a program. All editing takes place in a "buffer", which is nothing more than 'ed's own private storage area where it can manipulate your text. 'Ed's commands have the general format

<line number>,<line number><command>

where, typically, both line numbers are optional. Commands are one letter, sometimes with optional parameters.

The symbol <line number> above can have several formats. Among them are:

- an integer, meaning the line with that number. For example, if the integer is 7, then the 7th line in the buffer;
- a period ("."), meaning the current line;
- a dollar sign ("$"), meaning the last line of the buffer;
- /string/, meaning the next line containing "string";
- ?string?, meaning the previous line containing "string";
- any of the above expression elements followed by "+" or "-" and another expression element.

All commands assume certain default values for their line numbers. In the list below, the defaults are in parentheses.

Command	Action
(.)a	Appends text from standard input to the buffer after the line specified. The append operation is terminated by a line containing only a period in column 1. Until that time, though, everything you type goes into the buffer.

(.,.)d	Deletes lines from the first line specified to the last line specified.
e filename	Fills the buffer from the named file.
(.,.)p	Prints lines from the first line specified to the last. 1,$p prints the entire buffer.
q	Causes 'ed' to return to the command interpreter. Note that unless you have given a "w" command (see below), everything you have done to the buffer is lost.
(.)r filename	Reads the contents of the named file into the buffer after the specified line.
(.,.)s/old/new/p	Substitutes the string "new" for the string "old". If the trailing p is included, the result is printed, otherwise 'ed' stays quiet.
(1,$)w filename	Writes the buffer to the named file. This command must be used if you want to save what you have done to the buffer.
?	Prints a longer description of the last error that occurred.

If 'ed' is called with a filename as an argument, it automatically performs an "e" command for the user.

'Ed' is extremely quiet. The only diagnostic message issued (except in a time of dire distress) is a question mark. Almost always it is obvious to the user what is wrong when 'ed' complains. However, a longer description of the problem can be had by typing "?" as the next command after the error occurs. The only commands for which 'ed' provides unsolicited information are the "e", "r", and "w" commands. For each of these, the number of lines transferred between the file and 'ed's buffer is printed.

It should be noted that specifying a line number without a command is identical to specifying the line number followed by a "p" command; i.e., print that line.

Creating a Program

Now that we have a basic knowledge of the editor, we should be able to use it to write a short program. As usual, user input is boldfaced.

```
          ] ed                                              (1)
          a                                                 (2)
          # now --- print the current time                  (3)

*         define(TIME_OF_DAY,2)                             (4)

             character now (10)                             (5)

*            call date (TIME_OF_DAY, now)                   (6)
             call print (STDOUT, "Now: *s*n.", now          (7)

             stop                                           (8)
             end                                            (9)
          .                                                 (10)
          w now.r                                           (11)
|         11                                                (12)
          q                                                 (13)
          ]                                                 (14)
```

(1) You invoke the editor by typing "ed" after the command interpreter's prompt. 'Ed,' in its usual soft-spoken manner, says nothing.

(2) 'Ed's "a" command allows text to be added to the buffer.

(3) Now you type in the text of the program. The sharp sign "#" introduces comments in Ratfor.

* (4) Ratfor's built-in macro processor is used to define a macro with the name "TIME_OF_DAY". Whenever this name appears in the program, it will be replaced by the text appearing after the comma in its definition. This technique is used to improve readability and allow quick conversions in the future.

(5) An array "now", of type character, length 10, is declared.

(6) The library routine "date" is called to determine the current time.

(7) The library routine "print" is called to perform formatted output to the program's standard output port.

(8) The "stop" statement causes a return to the Subsystem command interpreter when executed.

(9) The "end" statement marks the end of the program.

(10) The period alone on a line terminates the "a" command. Remember that this must be done before 'ed' will recognize any further commands.

(11) With the "w" command, 'ed' copies its buffer into the file named "now.r".

(12) 'Ed' responds by typing the number of lines written out.

(13) The "q" command tells ed to quit and return to the Subsystem's command interpreter.

(14) The Subsystem command interpreter prompts with a right bracket, awaiting a new command.

Now we are talking to the command interpreter again. We may now use the 'rp' command to change our program from Ratfor into Fortran, and hopefully compile and execute it.

```
] rp now.r                                              (1)
   8 (.main.): '<NEWLINE>' missing right parenthesis. (2)
]                                                       (3)
```

(1) 'Rp' is called. The argument "now.r" directs Ratfor to take its input from the file "now.r" and produce output on the file "now.f".

(2) 'Rp' has detected an error in the Ratfor program. 'Rp's error messages are of the form

line (program-element): 'context' explanation

In this case, a missing parenthesis was detected on line 8 in the main program.

(3) 'Rp' has returned to the Subsystem's command interpreter, which prompts with "]".

Looking back over the program, we quickly spot the difficulty and proceed to fix it with 'ed':

```
] ed now.r                                  (1)
ll                                          (2)
8                                           (3)
   call print (STDOUT, "Now: *s*n.", now    (4)
s/, now/, now)/p                            (5)
   call print (STDOUT, "Now: *s*n.", now)   (6)
w                                           (7)
ll                                          (8)
q                                           (9)
] rp now.r                                  (10)
]                                           (11)
```

(1) The editor is called as before. However, since we have given the name of a file, "now.r", to 'ed' as an argument, it automatically does an "e" command on that file, bringing it into the buffer.

(2) 'Ed' types the number of lines in the file.

(3) We type the line number 8, since that is the line that 'rp' told us had the error.

(4) 'Ed' responds by typing the line. (Remember that a line number by itself is the same as a "p" command of that line number.)

(5) We use 'ed's "s" command to add the missing parenthesis. Note the use of the "p" at the end of the command.

(6) 'Ed' makes the substitution, and since we have specified the "p", the result is printed.

(7) We now write the changed buffer back out to our file ('ed' remembers the file name "now.r" for us).

(8) 'Ed' prints the number of lines written.

(9) We exit from the editor with the quit command "q".

(10) We invoke Ratfor to process the program. Ratfor detects no errors. The output of the preprocessing is on file "now.f".

(11) The command interpreter prompts us for another command.

Now that the Ratfor program has been successfully preprocessed, it is time to compile the Fortran output, which was placed in the file "now.f". 'Fc,' which we covered in the previous section, should be used to compile Subsystem programs, since it selects several useful compiler options and standardizes the compilation process:

```
] fc now.f
OK, ftn 1/101707 60401
0000 ERRORS [<.MAIN.>FTN-REV17.3]
OK, C 1 2 3
OK, swtrtn -2
]
```

All of the garbage between the "fc" and the "]" prompt is stuff produced either by Primos or by the Fortran compiler and is mostly irrelevant at this point. The essential thing to recognize about it is that the number before "ERRORS" is zero.

Now that our program has compiled successfully, we bravely proceed to invoke the Linking Loader using 'ld,' which we also covered in the previous section. Remember that 'fc' has left the output of Fortran on the file "now.b". We will use 'ld's "-o" option to select the name of the executable file:

```
] ld now.b -o now
OK, swtseg
[SEG rev 17.3]
# vl //
$ co ab 4001
$ s/sy swt$cm 4036 100000
$ s/sy swt$tp 2030 150000
$ mi
$ s/lo now.b 0 4000 4000
$ s/li vswtlb 0 4000 4000
$ s/li 0 4000 4000
LOAD COMPLETE
$ sa
$ re
# sh
TWO CHARACTER FILE ID: //
CREATING //4000
# delete //
# QU
OK, cn //4000 now
OK, swtrtn -2
]
```

Again, all the noise between "ld" and "]" comes from Primos and the Loader. The important thing to notice here is the "LOAD COMPLETE" message, which indicates that linking is complete. If we did not get the "LOAD COMPLETE" message, we would re-link using the command "ld -u now.b -o now" and the loader would then list the undefined subprograms.

We now have an executable program in our directory. We can check this using 'lf':

```
] lf
now       now.b     now.f     now.r
]
```

Deciding we do not need the Fortran source file and the intermediate binary file hanging around, we remove them with 'del':

```
] del now.f now.b
] lf
now       now.r
]
```

And getting really brave, we try to run our newly created program:

```
] now
Now: 16:34:41
]
```

Hopefully the preceding example will be of some help in the development of your own (more important) programs. Even though it is simple, it shows almost all the common steps involved in creating and running a typical program.

Advanced Techniques

This section deals with several of the more advanced features of the Subsystem.

Command Files

As an illustration, let us take an operation that finds use quite frequently: making printed listings of all the Ratfor source code in a directory. Command language programs, or "shell programs," greatly simplify the automation of this process. Shell programs are files containing commands to be executed when human intervention is not required.

Suppose that we put the following commands in a file named "mklist" (note the use of i/o redirection here):

```
lf -c >temp1
temp1> find .r >temp2
temp2> change % "sp " >temp3
temp3> sh
del temp1 temp2 temp3
```

Then, whenever we want a listing of all the Ratfor source code in the current directory, we just type:

```
mklist
```

The only price we must pay for this convenience is to ensure that the names of all files containing Ratfor programs end in ".r". (If the "find", "change", and "sp" commands mystify you, 'help' can offer explanations.)

Pipes

Pipes are another handy feature of the Subsystem. A "pipe" between two programs simply connects the standard output of the first to the standard input of the second; and two or more programs connected in this manner form a "pipeline". With pipes, programs are easily combined as cooperating tools to perform any number of complex tasks that would otherwise require special-purpose programs.

The command interpreter provides a simple and intuitive way to specify these combinations:

```
prog1 | prog2
```

Essentially, two or more complete commands are typed on the same line, separated by vertical bars ("|"). (One or more spaces must appear on both sides of this symbol.) The command interpreter then does all the work in connecting them together so that whatever the program on the left of the bar writes on its standard output, the one on the right reads from its standard input.

Take our shell program to create listings as an example; that series of commands involved the creation of three temporary files. Not only is this distracting, in that it takes our attention away from the real work at hand, but it also leads to wasted storage space, since one all too frequently forgets to delete temporary files after they have served their function. Using pipes, we could just as easily have done the same thing like this:

```
lf -c | find .r | change % "sp " | sh
```

and the command interpreter would have taken care of all the details that before we had to attend to ourselves. In addition to being much cleaner looking, the pipeline's function is also more obvious.

Additional I/O Redirectors

The last advanced features of the Subsystem we will examine are the two remaining i/o redirection operators, represented by two variations of the double funnel (">>").

In the first variation,

```
>>xyz          (read "append to xyz")
```

causes standard output to be appended to the file named "xyz". Whereas

```
cat file1 >file2
```

would copy the contents of file1 into file2, destroying whatever was previously in file2,

```
cat file1 >>file2
```

would copy the contents of file1 to the end of file2, without destroying anything that was there to start with.

In the second variation, the double funnel is used without a file name

```
>>              (read "from command input")
```

to connect standard input to the current shell program. For example, suppose we wanted to make a shell program that extracted the first ten lines of a file, and deleted all the rest. The shell program might look something like this:

```
>>  ed file
11,$d
w
q
```

">>" is frequently used in this way for the editor to read commands from the shell program, without having to have a separate script file.

This is only a very small sample of the power made available by the features of the Subsystem. As is the case with any craft, given the proper tools and an hospitable environment in which to work, the only limit to the variety of things that can be done is the imagination and ingenuity of the craftsman himself.

Background

Ancient History

The Software Tools Subsystem, as it now exists, is in its seventh major revision. To give you an idea of its development, here is a short history of successive versions.

Version 1:

- Features: Basic utility commands, no redirection of input or output, low-level routines for performing file operations but no consistent input/output.
- Language: Fortran

Version 2:

- Features: Almost complete set of utility commands, redirection of input and output, all Software Tools i/o routines, Software Tools editor and Ratfor, improved reliability during information passing from one program to another.
- Language: Low level routines in Fortran, high level routines and programs in Ratfor

Version 3:

- Features: Same as version 2, but with Primos compatible i/o for speed; New shell added later greatly expanded program interaction
- Language: Almost entirely Ratfor

Version 4:

- Features: Same as version 3, plus: (1) ability to handle file names of up to 32 characters on new Primos file partitions; (2) much faster disk i/o (On an unloaded system, benchmarks show an improvement on the order of a factor of 20); (3) internal reorganization to speed up command searches; (4) Support for virtual mode programs and a shared command interpreter.
- Language: All higher-level routines in Ratfor. A few special routines in assembly language to provide capabilities not inherent in Fortran.

Version 5:

- Features: A new command interpreter supporting arbitrary networks of pipes, generalized command file handling, and dynamic command line structures was added. General reorganization of Subsystem directories on disk.
- Language: Ratfor and Assembler (PMA).

Version 6:

- Features: Shared libraries, maximal security under unmodified Primos, increased robustness.
- Language: Ratfor and Assembler (PMA)

Version 7:

- Features: Much faster disk I/O, extensions to pathnames to allow specification of non-file-system devices, new Ratfor preprocessor with significant extensions, some general cleanup of code and redundant tools, many additional tools.
- Language: Ratfor, Assembler (PMA), and some PL/I.

Authors and Origins

The principal authors of the Software Tools Subsystem are Allen Akin, Perry Flinn, Dan Forsyth, and Jack Waugh, of the Georgia Institute of Technology, aided by a cast of thousands.

The ultimate antecedent for the design of the Subsystem is the UNIX operating system, written by Dennis Ritchie and Ken Thompson of Bell Labs for the DEC PDP-11 computers.

The tremendous debt owed to Brian W. Kernighan and P. J. Plauger, the authors of Software Tools, cannot be overstated.

User's Guide to the Primos File System

Perry B. Flinn

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

March, 1980

TABLE OF CONTENTS

Introduction

One thing that you will almost certainly encounter frequently during your exploits in the Software Tools Subsystem is the Primos file system. Indeed, there is hardly anything you can do that does not in some way involve this ubiquitous beast. Recognizing this fact, we offer this guide as an attempt to acquaint you with everything you need to know to make effective use of the file system from within the Subsystem. Although we have tried to be thorough in our coverage of concepts and features, we have specifically avoided the details of the programmer's interface to the file system, and everything having to do with implementation. Should you find yourself in need of further information in either of these areas, let us direct your attention to section two of The Software Tools Subsystem Reference Manual and to the Reference Guide, File Management System (Prime publication number FDR3110).

What is a File?

A file is a named collection of information retained on some storage medium such as a disk pack. Just what kind of information a file contains isn't of much concern to us here; it may be ASCII character codes that form the text of a book or a program's source code, it may be arbitrary binary machine words to be used as input data for a program, or it may be the actual machine instructions of the program itself, to mention just a few. No matter what form the information in a file takes, as far as Primos is concerned it is just an ordered sequence of sixteen bit binary numbers. The interpretation of those numbers is left to other programs.

Entrynames

Since we mentioned that a file has a name, you might ask what names are acceptable. A file is known by something called its "entryname." An entryname is a sequence of 32 or fewer characters chosen from the letters of the alphabet, the decimal digits, and the following special characters:

```
# $ & - * . / _
```

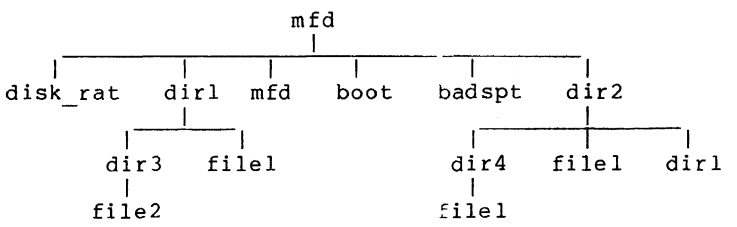
The first character in the entryname must not be a digit. Also, no distinction is made between upper- and lower-case letters. Thus "file_name" and "FILE_NAME" are the same.

Even though Primos allows you to use slashes (/) in entrynames, for reasons that will become apparent in the section on pathnames they must be treated specially when you are using the Subsystem. Because the slash is used to separate entrynames from one another in pathnames, if you want to use it in an entryname you have to "escape" it. By this we mean that you have to precede it with the "escape" character "@". The "@" simply tells the Subsystem to "treat the next character literally, no matter what special meaning it may have;" it is not taken as part of the entryname. It is important that you realize this caveat applies only when you are dealing with the Subsystem; if you try to put an "@" in an entryname when talking directly to Primos, you will get a rather impudent message.

Directories

The way that Primos makes the association between a file's entryname and its contents is through the use of "directories." Like a file, a directory has an entryname and contains some information; but it is different from ordinary files in that the information it contains is treated specially by Primos. The information in a directory is a series of "entries," each consisting of the entryname of some other file, that file's location on the disk pack, and some other stuff that we will cover in a later section. When a file's entryname and location appear in a directory, we say that the directory "contains" that file, or that the file "resides within" that directory. Either way you say it, every file in the system appears in exactly one directory.

Since a directory is so much like a file, there is really nothing to prevent us from having directories that contain other directories. This phenomenon is known as "nesting" and may be carried out to any depth, giving rise to a hierarchical structure:



At the topmost level of the hierarchy is a directory named "mfd", short for master file

directory. You will find this directory at the top level of every Primos file system. The MFD is special because it always begins at a fixed location on the disk pack, and because it always contains the following entries:

disk_rat
: The disk_rat (disk record availability table) is a file that catalogs all of the storage space on the disk pack that isn't already in use. It is always the first entry in the MFD and, like the MFD, always begins at a fixed location. This file may have any valid entryname; it doesn't have to be called "disk_rat". But whatever entryname is chosen, it is known as the "packname" for that disk pack.

mfd
: The MFD always has an entry describing itself.

boot
: The "boot" file, which also begins at a fixed location, contains the memory-image of a program that is loaded and executed whenever the computer is cold-started. This program is usually a single-user version of Primos.

badspt
: Although this file is not necessarily present on every disk pack, if it is it contains a list of faulty records that should not be used.

You may have noticed in the diagram that there are three occurrences of the entryname "file1", and two of "dir1". Each of these entrynames refers to a different file or directory. Even though each entryname must be unique among all those in a given directory, it is perfectly legal to use the same name repeatedly in different directories.

Logical Disks

Since Primos doesn't allow file systems that span multiple disk packs, it does the next best thing and allows you to have multiple file systems in the same installation. Each file system is called a "logical disk" and has exactly the structure described in the last section. Although each installation is virtually guaranteed to have at least one logical disk, the actual number may vary dynamically from 0 to 62. Each disk is uniquely identified by its "logical disk number," and though it is not required, it is extremely desirable for each disk to have a unique packname.

The "Current" and "Home" Directories

Now that we have described this wonderful hierarchy of directories and files just waiting to be used, you might wonder how it is that you go about getting to them. One concept that is central to the solution of this problem is that of the "current directory." From the time you log in to the time you log out, your terminal is having an ongoing relationship with some directory in the file system. When you first log in, this directory is typically the one whose entryname matches your login name and which resides in some MFD. But monogamy is not required; you are free to jump around from directory to directory upon the slightest whim. We say the "current directory" is the directory to which you are attached.

The current directory is important because all the files contained in it are directly accessible to you at the drop of an entryname. In fact, if you are using some of Prime's software, these are the only files accessible to you without changing your current directory. But there is a handy device called the "home directory" that takes some of the edge off of this restriction. Your home directory is the one to which you intend to return after an expedition into the wilds of the file system. In effect, it allows you to remember the location of some particular directory, and to later return there in one giant step, regardless of your (then) current location. Whenever you change your current directory, you get to choose whether to change your home directory as well or to leave it where it is.

Protection and Access Control

So that you can guard your files from unwanted perusal or alteration, the file system includes a basic access control mechanism that provides two levels of protection to each file. As part of this mechanism, each directory has associated with it a pair of six-character passwords, one called the "owner password," and the other called the "non-owner password." Normally, when a directory is created its owner password is blank and its non-owner password is zero; these are the default values. But if the passwords have other than default values, then before you can successfully attach to the directory, you must prove your worthiness to do so by citing one of them. If you cite the owner password, then you are attached to the directory with "owner status;" if it's the non-owner password that you cite, then you are attached with "non-owner status." If you fail to cite either password, then unless one of them has a default value your attempt will be in vain. Just what status you attain when attaching to a directory bears upon the kinds of things you may do to the files

it contains.

For the purposes of protection, there are three things you can do to a file: you can read from it, you can write into it, and you can truncate (shorten) or delete it. Now if you will recall that "other stuff" we mentioned a while back as being in a file's directory entry, part of it is two sets of "protection keys:" one for people attached to the containing directory with owner status, and the other for those with non-owner status. Each set of keys has a bit for each type of access: read, write and delete. If a bit is turned on, the associated type of access is permitted; otherwise, it is denied.

Pathnames

Unlike the Prime software we mentioned that only lets you manipulate files in your current directory, the Subsystem places no restrictions on the whereabouts of the files you can reference. Generally speaking, anywhere the name of a file is required you may use something called a "pathname." A pathname is a construct that allows you to uniquely specify any file in the system by describing a path to it from some known point. As we have seen, the current directory is one such point, and because of its fixed location, the MFD on each logical disk is another.

The syntax of a pathname is divided into two basic parts which we will call the "starting node," designating the particular known point at which the path starts, and the "directory path," designating the actual series of nested directories that leads to the desired file. Both parts, by the way, are optional: either one may stand alone, they may stand together, or they may both be omitted. But if both are present, they must be separated by a single slash (/).

The starting node of a pathname comes in two varieties. The first designates the MFD of a particular logical disk and consists of an initial slash followed by a packname, a logical disk number in octal, or a single asterisk (*):

```
/vol00
/7
/*
```

If the asterisk is used, the MFD of the logical disk containing the current directory is implied; the other two forms should be self-explanatory. The second variety of starting node refers to one of the current directory's ancestors in the hierarchy and consists of one or more backslashes (\). The number of backslashes indicates the number of nesting levels above the current directory at which the path begins. If the starting node is omitted altogether, then the path starts in the current directory.

Now the other half of a pathname, the directory path, is simply a series of one or more entrynames, each separated from the next by a single slash. The first entryname must be contained in the starting directory, and each subsequent entryname must reside in the directory designated by the preceding entryname. The very last entryname in the path is that of the target file. To illustrate,

```
src/lib/swt
extra
```

are proper directory paths. As you might expect, if the directory path is omitted, the target of the pathname is the starting directory. Thus, the pathname from which both the starting node and the directory path have been omitted (the empty pathname) refers to the current directory.

A couple of special cases are worth mentioning here: First, a pathname that begins with a slash and whose directory path is not omitted need not contain a packname or logical disk number. In this case an implicit search of the MFD on each logical disk is made for the first entryname in the directory path. The MFD on the lowest numbered logical disk in which that entryname is found is taken as the starting directory. Notice that such a pathname is easily recognizable because it begins with two slashes; the first one belongs to the starting node and the second separates it from the directory path:

```
//system
```

The second special case has to do with pathnames beginning with a backslash. Although we said that a slash must be used to separate a starting node from a directory path, when using backslashes the intervening slash is not required; indeed it is omitted more often than not.

Passwords in Pathnames

Thus far in discussing pathnames we have assumed that we may freely specify any valid sequence of directories in a directory path without regard to the passwords that may be associated with those directories. In fact, this is true only if the directories have at least one password with a default value. You see, the interpretation of a pathname involves temporarily attaching to each directory in the path; if this can't be done without a password then the pathname can't be interpreted. Furthermore, the set of access privileges (owner or non-owner) available to you with respect to the target file is determined by whether you are attached to its parent directory as an owner or a non-owner by the pathname interpreter. So, to let you deal effectively with passworded directories, the pathname syntax allows you to append a password to each directory entryname in the path, separated from the entryname by a colon:

```
entryname:passwd
```

If a password is so specified, the pathname interpreter will use it when attaching to the associated directory.

A password may contain arbitrary characters which are not necessarily legal in entrynames. So to avoid the ambiguity in interpreting a password containing a slash, as with entrynames, the slash must be "escaped" by preceding it with an "@". This also means that the "@" itself must be escaped if it is to appear literally in the password. Remember that the "@" used as an escape character is not included in the password; it merely turns off the special meaning of the character that follows.

The following set of examples contains an instance of just about every possible variation in the syntax of pathnames, along with an explanation of each. A formal summary of pathname syntax in BNF notation is included in Appendix B.

a_file
: A file in the current directory whose entryname is "a_file".

a_ufd/a_file
: A file whose entryname is also "a_file" and is contained in the subdirectory "a_ufd" of the current directory.

\
: The parent of the current directory.

\brother (or \/brother)
: The file or directory named "brother" that resides in the same directory that contains the current one.

/0/cmdnc0:secret
: The directory named "cmdnc0" (one of whose passwords is "secret") which resides in the MFD on logical disk 0.

/md
: The MFD on the logical disk whose packname is "md".

/*/boot
: The "boot" file on the current logical disk.

//spoolq/q.ctrl
: The file named "q.ctrl" in the "spoolq" directory on the lowest numbered logical disk that has one.

ki@/da:ad@/ik
: The directory residing in the current directory whose entryname is "ki/da" and one of whose passwords is "ad/ik". (Note the use of the "@" to turn off the special meaning of "/".)

<empty>
: The current directory.

Templates

In order to provide flexibility in the organization and placement of the directories and files used by the Subsystem, the pathname interpreter contains a primitive macro substitution facility, a feature that is loosely referred to as "templates." Templates provide a means for designating particular files or directories without having to know their exact location in the file system, and for constructing file names whose exact interpretation may vary with the context in which, or the user by whom they are used. A template simply consists of a name (made up of letters, digits and underscores, and beginning with a letter) enclosed in

equals bars (=). Unlike entrynames, upper- and lower-case letters are different in template names; "name" and "NAME" are not the same. Each defined template has an associated value which is an arbitrary character string. The effect of including a template in a pathname is the same as if its value had appeared instead.

There are two types of templates: static and dynamic. The value of a dynamic template varies depending upon who you are, how you are connected to the computer, or what time it is. The following list describes all of the available dynamic templates:

=date=
The current date in the format mmddyy.

=day=
The current day of the week; "monday", for example.

=passwd=
The owner password of the current user's profile directory. (This is the same password the Subsystem asked you for when you typed "swt".)

=pid=
The current user's process-id. This is a two-digit number in the range 01-63 that is unique to each logged-in user.

=time=
The current time in the format hhmmss.

=user=
The current user's login name.

These templates are particularly useful for constructing unique file names.

Static templates are those whose definitions are independent of the context in which they are used. These templates and their values come from two sources. The file whose name is the value of the template

=template=

contains template definitions that apply globally to all Subsystem users. In fact the definition of =template= itself is contained in this file, as are definitions for other important Subsystem files and directories. In addition to this file, you may have in your profile directory (named by the template =varsdir=) a file named ".template" that contains your own personal template definitions. Any templates that you define yourself preempt similarly-named system and dynamic templates, so you should exercise caution in choosing names. Also note that any new templates you place in your personal template file do not take effect until the next time you enter the Subsystem via 'swt'; this is the only time that the file is examined.

The format of both files is the same: a series of lines containing a name, followed by one or more blanks, and then a value. Blank lines are ignored, as are leading and trailing blanks on each line. Comments may be introduced with the sharp character (#); all characters from the sharp to the end of the line are ignored:

```
# example of a template definition
   template_name     template_value     # comment
```

A quick perusal of the contents of =template= should clear up any lingering questions you may have. Just for convenience, all the standard templates, along with an explanation of each, are listed in Appendix A.

If you look at the template definition file, you will notice that some of the definitions appear to contain templates themselves. This is perfectly legal, for after each template is expanded, the result is inspected for further templates until no others are found. This makes possible the definition of such templates as "=varsdir=", and generally enhances the utility of the mechanism.

Just one further remark about templates: Remember the trouble we had with "/" in passwords and entrynames? Well, we have a similar situation with "="; when should it be taken literally, and when should it indicate the beginning of a template? To solve this dilemma, any time the template expander sees a template with an empty name (that is, two consecutive equals bars), it supplies a single "=" as the replacement value and does not consider it to be the start of another template. So if you ever want a literal "=", in a password for example, just type "==" and you've got it.

Device Names

Up to this point, we have been talking only about disk files, and the pathnames we have described have corresponded exactly to some actual sequence of directories leading to a file. Although this is certainly the most common use of pathnames, there is one additional feature that significantly enhances their usefulness. If the "starting node" of a pathname is "/dev", the pathname doesn't necessarily refer to a disk file, but may instead refer to an arbitrary peripheral device, or to some special file that requires unusual processing. As with ordinary pathnames, the "directory path" provides more information about the target file or device.

Perhaps the most useful of these extended pathnames (or "device names," as they are usually called) is

```
/dev/lps
```

which refers to the line printer spooler. When this pathname is opened for writing, a special disk file is created and other processing is done so that when the file is closed, its contents will be written to the on-site line printer by the spooler and then deleted. Additional entrynames may be included after the "lps" to select various processing options specific to the spooling process. A complete list of these in included as Appendix C.

Another useful device name is

```
/dev/tty
```

which refers to your terminal device. There are also others which, when opened, yield file descriptors for the various standard input and output ports:

```
/dev/stdout          /dev/stdin
/dev/stdout1         /dev/stdin1
/dev/stdout2         /dev/stdin2
/dev/stdout3         /dev/stdin3
/dev/errout          /dev/errin
```

Finally, the device name

```
/dev/null
```

when opened yields a file descriptor which discards all data written to it and returns an end-of-file signal every time it is read. It is really just a fancy name for the proverbial bit bucket.

Georgia Tech Extensions

As many of you reading this guide will eventually come to know, using the standard Primos file system can be quite awkward, principally because of the constant necessity of typing passwords in pathnames. Relief from this burden comes only at the expense of security, which in many cases is a more important consideration than ease of use. So that we can have our cake and eat it too, we at Georgia Tech have made a few modifications to the standard protection mechanism that virtually eliminate the necessity for typing passwords in all but the rarest of circumstances. The Subsystem requires none of these modifications to operate properly, and in those cases where it behaves differently depending on the extant version of Primos, it does so completely transparently to the user.

In Georgia Tech Primos, if a directory's owner password is a valid entryname, it is assumed to be the login name of the user that "owns" that directory. In this case, the "owner password" is instead called the "owner name." When you attach to a directory whose owner name "matches" your login name, you automatically get owner status without having to cite a password. The word "match" is quoted in the last sentence because more is meant by it than just character-for-character equality. The underscore (_) is treated as a "wild" character: whenever it appears in a given position in either your login name or the directory's owner name, the two names are considered equal in that position. Thus, a directory whose owner name is "_____" can be attached with owner status by everyone. Likewise, if your login name is "_____" then you get automatic owner status in any directory with an owner name instead of an owner password. With judicious assignment of login names, this feature may be used to effect very convenient sharing of files by multiple users working on the same project. Just as icing on the cake, anyone logged in with the name "system" gets owner status in every directory, regardless of whether it has an owner name or an owner password.

The modifications described in the last paragraph constitute all of the substantive differences between the protection mechanism in Georgia Tech Primos and the standard mechanism. In all other situations, you can expect the standard behavior.

Appendix A - Standard Templates

The following list describes all of the templates that are provided either in the standard Subsystem template file or by the template interpreter.

=aux=
This Subsystem directory contains large files that are not absolutely necessary for the operation of the Subsystem.

=bin=
The standard Subsystem command directory.

=cmdnc0=
The directory to which the system console is normally attached.

=date=
The current date in the format mmddyy.

=day=
The current day of the week (e.g., "monday", "tuesday", etc.).

=dictionary=
A file containing English words, used by the spelling checker.

=doc=
The Subsystem documentation directory.

=extra=
A standard Subsystem directory containing miscellaneous files required for proper operation of the Subsystem.

=fmac=
The Subsystem directory containing all the text formatter macro definition files.

=GaTech=
This is a template having nothing to do with pathnames. Its value is "yes" at installations that run the Georgia Tech version of Primos, and "no" elsewhere. Programs that are sensitive to the operating system version use this template to determine their environment.

=gossip=
The directory containing user-to-user message files generated by the 'to' command.

=home=
The current user's login directory. Take note that this is not the same as his "home directory" as described in the section on "current" and "home" directories.

=incl=
The standard Subsystem directory containing files that are **included** by Ratfor programs.

=installation=
A file containing the name of the installation.

=lbin=
The standard Subsystem locally-supported-command directory.

=lib=
The Primos directory containing all library files that should be accessible to the loader.

=mail=
The Subsystem directory that contains per-user mail delivery files.

=mailfile=
The current user's mail storage file. This is where the 'mail' command deposits a letter after you have asked that it be saved.

=newbin=
The Subsystem directory into which newly-compiled commands are placed during a recompilation of the entire Subsystem.

=newlbin=
The Subsystem directory into which newly-compiled locally-supported-commands are placed during a recompilation of the entire Subsystem.

=newlib=
The Subsystem directory into which newly-compiled object code libraries are placed during a recompilation of the entire Subsystem.

=news=
The directory used by the Subsystem news service.

=newsfile=
The current user's news delivery file.

=passwd=
The password of the current user's profile directory. (This is the same password the Subsystem asked you for when you typed "swt".)

=pid=
The current user's process-id. This is a two-digit number in the range 01-63 that is unique to each logged-in user.

=src=
The Subsystem source code directory.

=srcloc=
A file associating each Subsystem library subroutine and command with the pathname(s) of its source code file(s).

=swtrtn=
The value of this template is the entryname of the 'swtrtn' program in the directory =cmdnc0=.

=system=
The Primos directory that contains the core-images of the various shared memory segments.

=temp=
The Subsystem directory in which all temporary files are created.

=template=
The system template definition file.

=termlist=
A file describing the location and type of each terminal connected to the computer.

=time=
The current time in the format hhmmss.

=user=
The current user's login name.

=userlist=
A file containing a list of all users authorized to use the computer.

=utemplate=
The current user's private template definition file.

=vars=
The Subsystem directory in which all per-user profile directories are contained.

=varsdir=
The current user's profile directory.

=varsfile=
The current user's shell variable storage file.

=vth=
The directory used by the Subsystem virtual terminal handler.

Appendix B - Pathname Syntax

For the grammar afficionados among you, here is a formal description of the syntax of pathnames. The notation used is an extended Backus-Naur Form (BNF) which is described in the introduction to the Software Tools Subsystem Reference Manual.

```
<pathname>        ::= <starting node>
                    | <directory path>
                    | <starting node>/<directory path>
                    | <empty>
<starting node>   ::= \{\}
                    | /<volume id>
<volume id>       ::= <packname>
                    | <octal integer>
                    | *
<packname>        ::= <entryname>
<directory path>  ::= <node>{/<node>}
<node>            ::= <entryname>[:<password>]
<entryname>       ::= <non-digit>{<valid char>}
<non-digit>       ::= <letter>  |  <special char>
<valid char>      ::= <non-digit>  |  <digit>
<letter>          ::= a | b | c |...| x | y | z
<digit>           ::= 0 | 1 | 2 |...| 7 | 8 | 9
<special char>    ::= # | $ | & | - | * | . | / | _
```

Appendix C - Spool Options

The entrynames that may be appended to the "/dev/lps" device name to control spooling options are summarized in the following list. These entrynames correspond exactly to the options that are accepted by the 'sp' command (see section one of the Subsystem reference manual).

a This option selects a specific location at which the file is to be printed. The immediately following entryname in the path is taken as the name of the destination printer.

b The file name that is printed on the banner page of the printout may be set arbitrarily with this option. The next entryname in the path is taken as the name to be printed. If this option is not used, the name "/dev/lps" is printed.

c This option specifies the number of copies of the file that are to be printed. The next entryname must be a decimal integer indicating the number of copies.

d Printing of the file may be deferred until a specific time of day using this option. The next entryname in the path must be a time of day in any reasonable format.

f If specified, this option indicates that the print file contains standard Fortran carriage control characters.

h This option causes the spooler to suppress the printing of the banner page that normally precedes each printout.

j Specifying this option causes the spooler to suppress the trailing page eject that it normally supplies at the end of each printout.

n This option causes the spooler to print a consecutive line number in front of each line of the print file.

p This option instructs the spooler that the print file is to be printed on a special type of paper. The name of the desired form should follow as the next entryname in the path.

r "Raw" forms control mode is selected by this option. No carriage control characters are recognized, nor is any pagination done when this mode is in effect.

s This option selects the standard Primos forms control mode. Under this mode, the printout is automatically paginated, and a header line is printed on each page.

Introduction to the Software Tools Text Editor

T. Allen Akin
Perry B. Flinn
Daniel H. Forsyth, Jr.

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

April, 1980

TABLE OF CONTENTS

Foreword

'Ed' is an interactive program that can be used for the creation and modification of "text." "Text" may be any collection of character data, such as a report, a program, or data to be used by a program.

This document is intended to provide the beginning user of 'ed' with a tutorial, an aid to becoming familiar with editing. It does not attempt to cover the editor in full; only the most frequently used aspects are mentioned. For details on advanced uses, a careful reading of Software Tools and the Software Tools Subsystem Reference Manual is recommended.

How To Use This Guide

This tutorial includes a step-by-step journey through an editing session. You should be sitting at a terminal and running the Software Tools Subsystem, so that you can perform the suggested exercises as you go.

Throughout the text of this guide are sample editing commands, which you can execute on your terminal to get a feel for their actual effect. If at any time your terminal session produces results different from those shown in the text, carefully re-check what you have typed, or consult someone in charge of your installation.

Tutorial

Starting an Editing Session

We assume that you have successfully logged in to your computer and are running the Software Tools Subsystem. If you need assistance, see the Software Tools Subsystem Tutorial. We further assume that you know how to use the character erase and line delete characters, so that you will have no trouble correcting typographical errors, and that you have some idea of what a "file" is.

Since you are in the Subsystem, the command interpreter should have just printed the prompt "]". To enter the text editor, type

```
] ed      (followed by a newline)
```

(Throughout this guide, boldface is used to indicate information that you should type in. Things typed by 'ed' are shown in the regular font.) You are now in the editor, ready to go. Note that 'ed' does not print any prompting information; this quiet behavior is preferred by experienced users. (If you would like a prompt, it can be provided; try the command "op/prompt/".)

At this point, 'ed' is waiting for instructions from you. You can instruct 'ed' by using "commands", which are single letters (occasionally accompanied by other information, which you will see shortly).

Entering Text - the Append Command

The first thing that you need is text to edit. Working with 'ed' is like working with a blank sheet of paper; you write on the paper, alter or add to what you have written, and either file the paper away for further use or throw it away. In 'ed's terminology, the blank sheet of paper you start with is called a "buffer." The buffer is empty when you start editing. All editing operations take place in the buffer; nothing you do can affect any file unless you make an explicit request to transfer the contents of the buffer to a file.

So the first problem reduces to finding a way to put text into the buffer. The "append" command is used to do this:

```
a
```

This command appends (adds) text lines to the buffer, as they are typed in.

To put text into the buffer, simply type it in, terminating each line with a newline:

```
The quick brown fox
    jumps over
   the lazy dog.
.
```

To stop entering text, you must enter a line containing only a period, immediately followed by a newline, as in the last line above. This tells 'ed' that you are finished writing on the buffer, and are ready to do some editing.

The buffer now contains:

```
The quick brown fox
    jumps over
   the lazy dog.
```

Neither the append command nor the final period are included in the buffer -- just the text you typed in between them.

Writing text on a file - the Write command

Now that you have some text in the buffer, you need to know how to save it. The write command "w" is used for this purpose. It is used like this:

```
w file
```

where "file" is the name of the file used to store what you just typed in. The write command copies the contents of the buffer to the named file, destroying whatever was previously in the file. The buffer, however, remains intact; whatever you typed in is still there. To indicate that the transfer of data was successful, 'ed' types out the number of lines written. In this example, 'ed' would type:

It is advisable to write the contents of the buffer out to a file periodically, to insure that you have an up-to-date version in case of some terrible catastrophe (like a system crash).

Finishing up - the Quit command

Now that you have saved your text in a file, you may wish to leave the editor. The "quit" command "q" is provided for this:

```
q
```

The next thing you see should be the "]" prompt from the Subsystem command interpreter. If you did not write out the contents of the buffer, the editor would respond:

```
?
(not saved)
```

This is to remind you to write out the buffer, so that the results of your editing session are not lost. If you intended that the buffer be discarded, just enter "q" again and 'ed' will throw away the buffer and terminate.

When you receive the "]" prompt from the Subsystem command interpreter, the buffer has been thrown away; there is absolutely no way to recover it. If you wrote the contents of the buffer to a file, then this is of no concern; if you did not, it may mean disaster.

To check if the text you typed in is really in the file you wrote it to, try the following command:

```
] cat file
```

where "file" is the name of the file given with the "w" command. ("Cat" is a Subsystem command that can be used to print files on the terminal. If, for example, you wished to print your file on the line printer, you could say:

```
] pr file
```

and the contents of "file" would be queued for printing.)

Reading files - the Enter command

Of course, most of the time you will not be entering text into the buffer for the first time. You need a way to fill the buffer with the contents of some file that already exists, so that you can modify it. This is the purpose of the "enter" command "e"; it enters the contents of a file into the buffer. To try out "enter," you must first get back into the editor:

```
] ed
```

"Enter" is used like this:

```
e file
```

"File" is the name of a file to be read into the buffer.

Note that you are not restricted to editing files in the current directory; you may also edit files belonging to other users (provided they have given you permission). Files belonging to other users must be identified by their full "pathname" (discussed fully in User's Guide to the Primos File System). For example, to edit a file named "document" belonging to user "tom," you would enter the following command:

```
e //tom/document
```

After the file's contents are copied into the buffer, 'ed' prints the number of lines it read. In our example, the buffer would now contain:

```
The quick brown fox
    jumps over
   the lazy dog.
```

If anything at all is present in the buffer, the "e" command destroys it before reading the named file.

As a matter of convenience, 'ed' remembers the file name specified on the last "e" command, so you do not have to specify a file name on the "w" command. With these provisions, a common editing session looks like

```
] ed
e file
{editing}
w
q
```

The "file" command ("f") is available for finding out the remembered file name. To print out the name, just type:

```
f
file
```

You might also want to check that

```
] ed file
```

is exactly the same as

```
] ed
e file
```

That is, 'ed' performs an "e" command for you if you give it a file name on the command line.

Errors - the Query command

Occasionally, an error of some kind is encountered. Usually, these are caused by misspelled file names, although there are other possibilities. Whenever an error occurs, 'ed' types

```
?
```

Although this is rather cryptic, it is usually clear what caused the problem. If you need further explanation, just enter "?" and 'ed' responds with a one-line explanation of the error. For example, if the last command you typed was an "e" command, 'ed' is probably saying that it could not find the file you asked for. You can find out for sure by entering "?":

```
e myfile
?
?
I can't open the file to read
```

Except for the messages in reponse to "?", 'ed' rarely gives other, more verbose error messages; if you should see one of these, the best course of action is to report it to whoever maintains the editor at your installation.

Printing text - the Print command

You are likely to need to print the text you have typed to check it for accuracy. The "print" command "p" is available to do this. "P" is different from the commands seen thus far; "e", "w", and "a" have been seen to work on the whole buffer at once. For a small file, it might be easiest to print the entire buffer just to check on some few lines, but for very large files this is clearly impractical. The "p" command therefore accepts "line numbers" that indicate which lines to print. Try the following experiment:

```
] ed file
3
1p
The quick brown fox
3p
   the lazy dog.
1,2p
The quick brown fox
    jumps over
1,3p
The quick brown fox
    jumps over
   the lazy dog.
```

"1p" tells 'ed' to print line 1 ("The quick brown fox"). "3p" says to print the third line ("the lazy dog."). "1,2p" tells 'ed' to print the first through the second lines, and "1,3p" says to print the first through the third lines.

Suppose we want to print the last line in the buffer, but we don't know what its number is. 'Ed' provides an abbreviation to specify the last line in the buffer:

```
$p
   the lazy dog.
```

The dollar sign can be used just like a number. To print everything in the buffer, we could type:

```
1,$p
The quick brown fox
    jumps over
   the lazy dog.
```

If for some reason you want to stop the printing before it is done, press the BREAK key on your terminal. If you receive no response from BREAK, 'ed' is waiting for you to enter a command. Otherwise, 'ed' responds with

```
?
```

and waits for your next command.

More Complicated Line Numbers

'Ed' has several ways to specify lines other than just numbers and "$". Try the following command:

```
p
   the lazy dog.
```

'Ed' prints the last line. Does 'ed' always print the last line when it is given an unadorned "p" command? No. The "p" command by itself prints the "current" line. The "current" line is the last line you have edited in any way. (As a matter of fact, the last thing we did was to print all the lines in the buffer, so the last line was edited by being printed.) 'Ed' allows you to use the symbol "." (read "dot") to represent the current line. Thus

```
.p
   the lazy dog.
```

is the same as

```
.,.p
   the lazy dog.
```

which is the same as just

```
p
   the lazy dog.
```

"." can be used in many ways. For example,

```
1,2p
The quick brown fox
    jumps over
1,.p
The quick brown fox
    jumps over
.,$p
    jumps over
   the lazy dog.
```

This example shows how to print all the lines up to the current line (1,.p) or all the lines from the current line to the end of the buffer (.,$p). If for some reason you would like to know the number of the current line, you can type

```
.=
3
```

and 'ed' displays the number. (Note that the last thing we did was to print the last line,

so the current line became line 3.)

"." is not particularly useful when used alone. It becomes much more important when used in "line-number expressions." Try this experiment:

```
.-1p
    jumps over
```

".-1" means "the line that is one line before the current line."

```
.+1p
   the lazy dog.
```

".+1" means "the line that is one line after the current line."

```
.-2,.-1p
The quick brown fox
    jumps over
```

".-2,.-1p" means "print the lines from two lines before to one line before the current line."

You can also use "$" in line-number expressions:

```
$-1p
    jumps over
```

"$-1p" means "print the line that is one line before the last line in the buffer, i.e., the next to the last line."

Some abbreviations are available to help reduce the amount of typing you have to do. Typing a newline by itself is equivalent to typing ".+1p"; typing a caret, "^", followed by a newline is equivalent to typing ".-1p"; and typing a line-number expression followed by a newline is equivalent to typing that line-number expression followed by "p". Examples:

```
{type a newline by itself}
   the lazy dog.
^
    jumps over
1
The quick brown fox
```

It might be worthwhile to note here that almost all commands expect line numbers of one form or another. If none are supplied, 'ed' uses default values. Thus,

```
w file
```

is equivalent to

```
1,$w file
```

and

```
a
```

is equivalent to

```
.a
```

(which means, append text <u>after</u> the current line.)

Deleting Lines

As yet, you have seen no way of removing lines that are no longer wanted or needed. To do this, use the "delete" command "d":

```
1,2d
```

deletes the first through the second lines. "D" expects line numbers that work in the same way as those specified for "p", deleting one line or any range of lines.

```
d
```

deletes only the current line. It is the same as ".d" or ".,.d".

After a deletion, the current line pointer is left pointing to the first line after the group of deleted lines, unless the last line in the buffer was deleted. In this case, the current line is the last line before the group of deleted lines.

Text Patterns

Frequently it is desirable to be able to find a particular "pattern" in a piece of text. For example, suppose that after proofreading a report you have typed in using 'ed' you find a spelling error. There must be an easy way to find the misspelled word in the file so it can be corrected. One way to do this is to count all the lines up to the line containing the error, so that you can give the line number of the offending line to 'ed'. Obviously, this way is not very fast or efficient. 'Ed' allows you to "search" for patterns of text (like words) by enclosing the pattern in slashes:

```
/jumps/
    jumps over
```

'Ed' looks for the pattern you specified, and moves to the first line which contains the pattern. Note that if we had typed

```
/jumped/
?
```

'ed' would inform us that it could not find the pattern we wanted.

'Ed' searches forward from the current line when it attempts to find the pattern you specified. If 'ed' reaches the last line without seeing the pattern, it "wraps around" to the first line in the file and continues searching until it either finds the pattern or gets back to the line where it started (line "."). This procedure ensures that you get the "next" occurrence of the pattern you were looking for, and that you don't miss any occurrences because of your current position in the file.

Suppose, however, that you do not wish to find the "next" occurrence of a word, but the previous one instead. Very few text editors provide this capability; however, 'ed' makes it simple. Just surround the pattern with backslashes:

```
\quick\
The quick brown fox
```

Remember: backslashes search backward. The backward search (or backscan, as it is sometimes called) wraps around the file in a manner similar to the forward search (or scan). The search begins at the line before the current line, proceeds until the first line of the file is seen, then begins at the last line of the file and searches up until the current line is encountered. Once again, this is to ensure that you do not miss any occurrences of a pattern due to your current position in the file.

'Ed' also provides more powerful pattern matching services than simply looking for a given string of characters. (A note to beginning users: this section may seem fairly complicated at first, and indeed you do not really need to understand it completely for effective use of the editor. However, the results you might get from some patterns would be mystifying if you were not provided with some explanation, so look this over once and move on.)

The pattern that may appear within slashes (or backslashes) is called a "regular expression." It contains characters to look for and special characters used to perform other operations. The following characters

```
%  ?  $  [  *  @  {
```

have special meaning to 'ed':

% Beginning of line. The "%" character appearing as the first element in a pattern matches the beginning of a line. It is most frequently used to locate lines with some string at the very beginning; for example,

```
/%The/
```

finds the next line that begins with the word "The". The percent sign has its special meaning only if it is the first element of the pattern; otherwise, it is treated as a literal percent sign.

? Any character. The question mark "?" in a regular expression matches any character (except a beginning-of-line or a newline). It can be used like this:

```
/a?b/
```

to find strings like

```
a+b
a-b
a b
arbitrary
```

However, "?" is most often used with the "closure" operator "*" (see below).

$ End of line. The dollar sign appearing as the last element of a pattern matches the newline character at the end of a line. Thus,

```
/today$/
```

can be used to find a line with the word "today" at the very end. Like the percent sign, the dollar sign has no special meaning in positions other than the end of a pattern.

[] Character classes. The square brackets are used to match "classes" of characters. For example,

```
/[A-Z]/
```

finds the next line containing a capital letter,

```
/%[abcxyz]/
```

finds the next line beginning with an a, b, c, x, y, or z, and

```
/[~0-9]/
```

finds the next line which contains a non-digit. Character classes are also frequently used with the "closure" operator "*".

* Closure. The asterisk is used to mean "any number of repetitions (including zero) of the previous pattern element (one character or a character class in brackets)." Thus,

```
/a?*b/
```

finds lines containing an "a" followed by any number of characters and a "b". For example, the following lines are matched:

```
ab
abnormal
Recording Media, by Dr. Joseph P. Gunchy
```

As another example,

```
/%=*$/
```

matches only those lines containing all equal-signs (or nothing at all). If you wish to ensure that only non-empty lines are matched, use

```
/%==*$/
```

Always remember that "*" (closure) matches zero or more repetitions of an element.

@ Escape. The "at" sign has special meaning to 'ed'. It is the "escape" character, which is used to prevent interpretation of a special character which follows. Suppose you wish to locate a line containing the string "a * b". You may use the following command:

```
/a @* b/
```

The "at" sign "turns off" the special meaning of the asterisk, so it can be used as an ordinary text character. You may have occasion to escape any of the regular expression metacharacters (%, ?, $, [, *, @, or {) or the slash itself. For example, suppose you wished to find the next occurrence of the string "1/2". The command you need is:

```
/1@/2/
```

{} Pattern tags. As seen in the next section, it is sometimes useful to remember what part of a line was actually matched by a pattern. By default, the string matched by the entire pattern is remembered. It is also possible to remember a string that was matched by only a part of a pattern by enclosing that part of the pattern in braces. Hence to find the next line that contains a quoted string and remember the text between the quotes, we might use

```
/"{?*}"/
```

If the line thus located looked like this

```
This is a line containing a "quoted string".
```

then the text remembered as matching the tagged part of the pattern would be

```
quoted string
```

The last important thing you need to know about patterns is the use of the "default" pattern. 'Ed' remembers the last pattern used in any command, to save you the trouble of retyping it. To access the remembered pattern, simply use an "empty" string. For example, the following sequence of commands could be used to step through a file, looking for each occurrence of the string "ICS":

```
/ICS/
//
//
(and so on)
```

One last comment before leaving pattern searching. The constructs

```
/pattern/
\pattern\
```

are not separate commands; they are components of line number expressions. Thus, to print the line after the next line containing "tape", you could say

```
/tape/+1p
```

Or, to print a range of lines from one before to one after a line with a given pattern, you could use

```
/pattern/-1,/pattern/+1p
```

Making Substitutions - the Substitute command

This is one of the most used editor commands. The "substitute" command "s" is used to make small changes within lines, without retyping them completely. It is used like this:

```
starting-line,ending-line s /pattern/new-stuff/
```

For instance, suppose our buffer looks like this:

```
1,$p
The quick brown fox
    jumps over
   the lazy dog.
```

To change "jumps" to "jumped,"

```
2s/jumps/jumped/p
    jumped over
```

Note the use of the trailing "p" to print the result. If the "p" had been omitted, the change would have been performed (in the buffer) but the changed line would not have been printed out.

If the last string specified in the substitute command is empty, then the text matching the pattern is deleted:

```
s/jumped//p
    over
s/% */    jumps /p
    jumps over
```

Recalling that a missing pattern means "use the last pattern specified," try to explain what the following commands do:

```
s///p
jumps over
s//    /p
    jumps over
```

(Note that, like many other commands, the substitute command assumes you want to work on the current line if you do not specify any line numbers.)

What if you want to change "over" into "over and over"? You might use

```
s/over/over and over/p
    jumps over and over
```

to accomplish this. There is a shorthand notation for this kind of substitution that was alluded to briefly in the last section. (Recall the discussion of "tagged" paterns.) By default, the part of a line that was matched by the whole pattern is remembered. This string can then be included in the replacement string by typing an ampersand ("&") in the desired position. So, instead of the command in the last example,

```
s/over/& and &/
```

could have been used to get the same result. If a portion of the pattern had been tagged, the text matched by the tagged part in the replacement could be reused by typing "@1":

```
s/jump{?*}/vault@1/p
    vaults over and over
```

It is possible to tag up to nine parts of a pattern using braces. The text matched by each tagged part may then be used in a replacement string by typing

```
@n
```

where n corresponds to the nth "{" in the pattern. What does the following command do?

```
s/{[~ ]*} {?*}/@2 @1/
```

Final words on substitute: the slashes are known as "delimiters" and may be replaced by any other character except a newline, as long as the same character is used consistently throughout the command. Thus,

```
s#vaults#vaulted#p
    vaulted over and over
```

is legal. Also, note that substitute changes only the first occurrence of the pattern that it finds; if you wish to change all occurrences on a line, you may append a "g" (for "global") to the command, like this:

```
s/ /*/gp
****vaulted*over*and*over
```

Line Changes and Insertions

Two "abbreviation" commands are available to shorten common operations applying to changes of entire lines. These are the "change" command "c" and the "insert" command "i".

The change command is a combination of delete and append. Its format is

```
starting-line,ending-line c
```

This command deletes the given range of lines, and then goes into append mode to obtain text to replace them. Append mode works exactly the same way as it does for the "a" command; input is terminated by a period standing alone on a line. Examine the following editing session to see how change might be used:

```
1,$c
Ed is an interactive program used for
the creation and modification of "text.
.
c
the creation and modification of "text."
"Text" may be any collection of character
data.
.
```

As you can see, the current line is set to the last line entered in append mode.

The other abbreviation command is "i". "I" is very closely related to "a"; in fact, the following relation holds:

```
starting-line i
```

is the same as

```
starting-line - 1 a
```

In short, "i" inserts text before the specified line, whereas "a" inserts text after the specified line.

Moving Text

Throughout this guide, we have concentrated on what may be called "in-place" editing. The other type of editing commonly used is often called "cut-and-paste" editing. The move command "m" is provided to facilitate this kind of editing, and works like this:

```
starting-line,ending-line m after-this-line
```

If you wanted to move the last fifty lines of a file to a point after the third line, the command would be

```
$-49,$m3
```

Any of the line numbers may, of course, be full expressions with search strings, arithmetic, etc.

You may, if you like, append a "p" to the move command to cause it to print the last line moved. The current line is set to the last line moved.

Global Commands

The "global" command "g" is used to perform an editing command on all lines in the buffer that match a certain pattern. For example, to print all the lines containing the word "editor", you could type

```
g/editor/p
```

If you wanted to correct some common spelling error, you would use

```
g/old-stuff/s//new-stuff/gp
```

which makes the change in all appropriate lines and prints the resulting lines. Another example: deleting all lines that begin with an asterisk could be done this way:

```
g/%@*/d
```

"G" has a companion command "x" (for "exclude") that performs an operation on all lines in the buffer that do not match a given pattern. For example, to delete all lines that do not begin with an asterisk, use

```
x/%@*/d
```

"G" and "x" are very powerful commands that are essential for advanced usage, but are usually not necessary for beginners. Concentrate on other aspects of 'ed' before you move on to tackle global commands.

Marking Lines

During some types of editing, especially when moving blocks of text, it is often necessary to refer to a line in the buffer that is far away from the current line. For instance, say you want to move a subroutine near the beginning of a file to somewhere near the end, but you aren't sure that you can specify patterns to properly locate the subroutine. One way to solve this problem is to find the first line of the subroutine, then use the command ".=":

```
/subroutine/
   subroutine think
.=
47
```

and write down (or remember) line 47. Then find the end of the subroutine and do the same thing:

```
/end/
   end
.=
71
```

Now you move to where you want to place the subroutine and enter the command

```
47,71m.
```

which does exactly what you want.

The problem here is that absolute line numbers are easily forgotten, easily mistyped, and difficult to find in the first place. It is much easier to have 'ed' remember a short "name" along with each line, and allow you to reference a line by its name. In practice, it seems convenient to restrict names to a single character, such as "b" or "e" (for "beginning" or "end"). It is not necessary for a given name to be uniquely associated with one line; many lines may bear the same name. In fact, at the beginning of the editing session, all lines are marked with the same name: a single space.

To return to our example, using the 'k' command, we can mark the beginning and ending lines of the subroutine quite easily:

```
/subroutine/
   subroutine think
kb
/end/
   end
ke
```

We have now marked the first line in the subroutine with "b" and the second line with "e".

To refer to names, we need more line number expression elements: ">" and "<". Both work in line number expressions just like "$" or "/pattern/". The symbol ">" followed by a single character mark name means "the line number of the first line with this name when you search forward". The symbol "<" followed by a single character mark name means "the line number of the first line with this name when you search backward". (Just remember that '<' points backward and '>' points forward.)

Now in our example, once we locate the new destination of the subroutine, we can use "<b" and "<e" to refer to lines 47 and 71, respectively (remember, we marked them). The "move" command would then be

```
<b,<em.
```

Several other features pertaining to mark names are important. First, the 'k' command does not change the current line '.'. You can say

```
$kx
```

(which marks the last line with "x") and "." will not be changed. If you want to mark a range of lines, the 'k' command accepts two line numbers. For instance,

```
5,10ka
```

marks lines 5 through 10 with "a" (i.e., gives each of lines 5 through 10 the markname "a").

The 'n', '!' and apostrophe commands also deal with marks. The 'n' command performs two functions. If it is invoked without a mark name following it, like

```
$n
```

it prints the mark name of the line. In this case, it would print the mark name of the last line in the file. If the 'n' command is followed by a mark name, like

```
4nq
```

it marks the line with that mark name, and erases the marks on any other lines with that name. In this case, line 4 is marked with "q" and it is guaranteed that no other line in the file is marked with "q".

The '!' and apostrophe commands are both global commands that deal with mark names. The apostrophe command works very much like the 'g' command: the apostrophe is followed by a mark name and another command; the command is performed on every line marked with that name. For instance,

```
'as/fox/rabbit/
```

changes the first "fox" to "rabbit" on every line that is named "a". The '!' command works in the same manner, except that it performs the command on those lines that are not marked with the specified name. For example, to delete all lines not named "k", you could type

```
!kd
```

Undoing Things -- the Undo Command

Unfortunately, Murphy's Law guarantees that if you make a mistake, it will happen at the worst possible time and cause the greatest possible amount of damage. 'Ed' attempts to prevent mistakes by doing such things as working with a copy of your file (rather than the file itself) and checking commands for their plausibility. However, if you type

```
d
```

when you really meant to type

```
a
```

'ed' must take its input at face value and do what you say. It is at this point that the "undo" command 'u' becomes useful. "Undo" allows you to "undelete" the last group of lines that was deleted from the buffer. In the last example, some inconvenience could be avoided by typing

```
^ud
```

which restores the deleted line. (By default "undo" replaces the specified line by the last group of lines deleted. Specifying the "d", as in "ud", causes the group to be inserted after the specified line instead.)

The problem that arises with "undo" is the answer to the question: "What was the last group of lines deleted?" This answer is very dependent on the implementation of 'ed' and in some cases is subject to change. After many commands, the last group of lines deleted is well-defined, but unspecified. It is not a good idea to use the "undo" command after anything other than 'c', 'd', or 's'. After a 'c' or 'd' command,

```
ud
```

places the last group of deleted lines after the current line. After an 's' command (which by the way, deletes the old line, replacing it by the changed line),

```
u
```

deletes the current line and replaces it by the last line deleted -- it exactly undoes the effects of the 's' command. But beware! If the 's' command covered a range of lines, 'u' can only restore the last of the lines in which a substitution was made; the others are gone forever.

You should be warned that while "undo" works nicely for repairing a single 'c', 'd', or 's' command, it cannot repair the damage done by one of these commands under the control of a global prefix ('g', 'x', '!' and apostrophe). Since the global prefixes cause their command to be performed many times, only the very last command performed by a global prefix can be repaired.

Summary

This concludes our tour through the world of text editing. In the section that follows, you will find a brief introduction to the Software Tools Subsystem screen editor 'se', which supports all of the line-oriented commands of 'ed' as well as full screen editing capabilities, while giving you a "window" into your edit buffer. Following that, we have included for your convenience a short summary of all available line editing commands supported by 'ed' and 'se', many of which were not discussed in this introduction, but which you will undoubtedly find useful.

The Subsystem Screen Editor

The screen editor, 'se', is an extended version of the Subsystem line editor, 'ed'. Although 'se' contains a number of additional features, it accepts all 'ed' commands (almost without exception), and is therefore easily used by anyone familiar with 'ed'. This section outlines the differences between 'ed' and 'se'.

Invoking the Screen Editor

Calling 'se' is slightly more complicated than calling 'ed', since 'se' must also be told the type of terminal on which it is to run. Three invocation methods are available, the simplest of which is to type

```
] se
```

or

```
] se myfile
```

just as if you were calling 'ed'. 'Se' then asks you questions (to be answered "y" or "n") until it determines your terminal type. For example, if you are using an ADDS Regent 100 terminal, the conversation might look like this:

```
] se
Is your terminal any model Beehive? n
ADDS Consul 980? n
DEC GT40 with Waugh terminal program? n
Perkin-Elmer 'Fox'? n
ADDS Regent 100? y
```

The second method of invoking 'se' is to specify a mnemonic for the terminal type on the command line with the "-t" option. The mnemonics for some of the supported terminals are as follows:

```
b150     Beehive B150
b200     Beehive B200
consul   ADDS Consul 980
regent   ADDS Regent 40, 100
fox      Perkin-Elmer 1100
ibm      IBM 3101
```

(For a complete list of currently supported terminals, use the command "help se".) You specify the terminal mnemonic on a command line of the following format:

```
se -t <mnemonic>
```

or

```
se -t <mnemonic> myfile
```

Again, if you are using the an ADDS Regent 100 terminal, you could type

```
] se -t regent
```

Use of the third method depends on how your installation is set up and is usually practical only when terminals are directly connected to the computer, or are all of the same model. If this is the case in your installation, as it is at Georgia Tech, you can use the 'e' command to invoke 'se' with the proper terminal type. All you need do is type

```
] e myfile
```

and 'se' will appear.

Using 'Se'

Once 'se' knows your terminal type, it clears the screen, draws in its margins, and begins reading your file (if you specified one). The screen it draws looks something like this. (The parenthesized numerals are not part of the screen layout, but are there to aid in the following discussion.)

```
(1) (2)              (3)
A     |
B    *|   integer a
C     |
.  -> |   for (a = 1; a <= 12; a = a + 1)
E     |      call putch (NEWLINE, STDOUT)
F     |   stop
$     |   end
cmd>      (4)
11:39  myfile ....(5).................................
```

The display is divided into five parts: (1) the line number area, (2) the mark name area, (3) the text area, (4) the command line, and (5) the status line. The current line (remember ".") is indicated by the symbol "." in the line number area of the screen. In addition, a rocket ("->") is displayed to make the current line more obvious. The current mark name of each line is shown in the markname area just to the left of the vertical bar. Other information, such as the number of lines read in, the name of the file, and the time of day, are displayed in the status line.

The cursor is positioned at the beginning of the command line, showing you that 'se' awaits your command. You may now enter any of the 'ed' commands and 'se' will perform them, while making sure that the current line is always displayed on the screen. There are only a few other things that you need know to successfully use 'se'.

- 'Se' always recognizes BS (control-h) and DEL as the erase and kill characters, regardless of your Subsystem erase and kill character settings.

- If you make an error, 'se' displays an error message in the status line (you need not request it). It also leaves your command line intact so that you may change it using in-line editing commands (we'll get to this a little later). If you don't want to bother with changing the command, just hit DEL and 'se' will erase it.

- The "p" command has a different meaning than in 'ed'. When used with line numbers, it displays as many of the lines in the specified range as possible (always including the last line). When used without line numbers, "p" displays the previous page.

- The ":" command positions a specified line at the top of the screen (e.g., "12:" positions the screen so that line 12 is at the top). If no line number is specified, ":" displays the next page.

- The "v" command can be use to modify an entire line rather than just add to the end of it. Also, if you use "v" over a range of lines and find you want to terminate the command before all lines have been considered, the control-f key is used instead of a period.

Keeping these few differences in mind, you will see that 'se' can perform all of the funtions of 'ed', while giving the advantage of a "window" into the edit buffer.

Extended Line Numbers

'Se' has a number of features that take advantage of the window display to minimize keystrokes and speed editing. In the line number area of the screen, 'se' always displays for each line a string that may be used in a command to refer to that line. Normally, it displays a capital letter for each line, but in "absolute line number" mode (controlled by the "oa" command; see the section on options for more details), it displays the ordinal number of the line in the buffer.

The line number letters displayed by 'se' may be used in any context requiring a line number. For instance, in the above example, a change to the first line on the screen could be specified as

```
As/%/# my new program/
```

You could delete the line before the first line on the screen by typing

```
A-1d
```

Case Conversion

When 'se' is displaying upper-case letters for line numbers, it accepts command letters only in lower case. For those who edit predominately upper-case text this is somewhat inconvenient; for those with upper-case only terminals this is a disaster. For this reason, 'se'

offers several options to alleviate this situation.

First of all, typing a control-z causes 'se' to invert the case of all letters (just like the alpha-lock key on some terminals). Upper-case letters are converted to lower-case, lower-case letters are converted to upper-case, and all other characters are unchanged. You can type control-z at any time to toggle the case conversion mode. When case inversion is in effect, 'se' displays the word "CASE" in the status line.

One drawback to this feature is that 'se' still expects line numbers in upper case and commands in lower case, so you must shift to type the command letter -- just the reverse of what you're used to. A more satisfactory solution is to specify the "c" option. Just type

```
oc
```

on the command line and 'se' toggles the case conversion mode, and completely reverses its interpretation of upper and lower case letters. In this mode, 'se' displays the line number letters in lower case and expects its command letters in upper case. Unshifted letters from the terminal are converted to upper case and shifted letters to lower case.

Tabs

In the absence of tabs, program indentation is very costly in keystrokes. So 'se' gives you the ability to set arbitrary tab stops using the "ot" command. By default, 'se' places a stop at column 1 and every third column thereafter. Tabs corresponding to the default can be set by enumerating the column positions for the stops:

```
ot 1 4 7 10 13 16 19 22 25 28 31 34  ...
```

This is almost as bad as typing the blanks on each line. For this reason, there is also a shorthand for such repetitive specifications.

```
ot +3
```

sets a tab stop at column 1 and at every third column thereafter. Fortran programmers may prefer the specification

```
ot 7 +3
```

to set a stop at column 7 and at every third thereafter.

Once the tab stops are set, the control-i and control-e keys can be used to move the cursor from its current position forward or backward to the nearest stop, respectively.

Full-Screen Editing

Full screen editing with 'se' is accomplished through the use of control characters for editing functions. A few, such as control-h, control-i, and control-e have already been mentioned. Since 'se' supports such a large number of control functions, the mnemonic value of control character assignments has dwindled to almost zero. About the only thing mnemonic is that most symmetric functions have been assigned to opposing keys on the keyboard (e.g., forward and backward tab to control-i and control-e, forward and backward space to control-g and control-h, skip right and left to control-o and control-w, and so on). We feel pangs of conscience about this, but can find no more satisfactory alternative. If you feel the control character assignments are terrible and you can find a better way, you may change them by modifying the definitions in 'se' and recompiling. If they work well, we'd like to hear about them.

Except for a few special purpose ones, control characters can be used anywhere, even on the command line. (This is why erroneous commands are not erased -- you may want to edit them.) Most of the functions work on a single line, but in overlay mode (controlled by the "v" command) the cursor may be positioned anywhere in the buffer.

Horizontal Cursor Motion

There are quite a few functions for moving the cursor. You've probably used at least one (control-h) to backspace over errors. None of the cursor motion functions erase characters, so you may move forward and backward over a line without destroying it. Here are several of the more frequently used cursor motion characters:

control-q Move forward one column.

control-h Move backward one column.

control-i Move forward to the next tab stop.

control-e Move backward to the previous tab stop.

control-o Move to the first column beyond the end of the line.

control-w Move to column 1.

Vertical Cursor Motion

'Se' provides two control keys, control-d and control-k, to move the cursor up and down, respectively, from line to line through the edit buffer. The exact function of each depends on 'se's current mode: in command mode they simply move the current line pointer without affecting the cursor position or the contents of the command line; in overlay mode (viz. the "v" command) they actually move the cursor up or down one line within the same column; finally, in append move, these keys are ignored. Regardless of the mode, the screen is adjusted when necessary to insure that the current line is displayed.

control-d Move the cursor up one line.

control-k Move the cursor down one line.

Character Insertion

Of course the next question is: "Now that I've moved the cursor, how do I change things?" If you want to retype a character, just position the cursor over it, and type the desired character; the old one is replaced. You may also insert characters at the current cursor position instead of merely overwriting what's already there. Typing a control-c inserts a single blank before the character under the cursor and moves the remainder of the line one column to the right; the cursor remains in the same column over the newly-inserted blank. Typing a control-x inserts enough blanks at the current cursor position to move the character that was there to the next tab stop. This can be handy for aligning items in a table, for example. As with control-c, the cursor remains in the same column.

A more general way of handling insertions is to type control-a. This toggles "insert mode" -- the word "INSERT" appears on the status line, and all characters typed from this point are inserted in the line (and characters to the right are moved over). Typing control-a again turns insert mode off. Here is a summary of these control characters:

control-a Toggle insert mode.

control-c Insert a blank to the left of the cursor.

control-x Insert blanks to the next tab stop.

Character Deletion

There are many ways to do away with characters. The most drastic is to type DEL; 'se' erases the current line and leaves the cursor in column 1. Typing control-t causes 'se' to delete the character under the cursor and all those to its right. The cursor is left in the same column which is now just beyond the new end of the line. Similarly, control-y deletes all the characters to the left of the cursor (not including the one under it). The remainder of the line is moved to the left, leaving the cursor over the same character, but now in column 1. Control-r deletes the character under the cursor and closes the gap from the right, while control-u does the same thing after first moving the cursor one column to the left. These last two are most commonly used to eat characters out of the middle of a line.

DEL Erase the entire line.

control-t Erase the characters under and to the right of the cursor.

control-y Erase the characters to the left of the cursor.

control-r Erase the character under the cursor.

control-u Erase the character immediately to left of the cursor.

Terminating a Line

After you have edited a line, there are two ways of terminating it (having it entered or executed). The most commonly used is newline (or carriage-return) which deletes all characters under and to the right of the cursor and terminates the line. To terminate a line without deleting any characters, there is control-v.

NEWLINE Erase characters under and to the right of the cursor and terminate.

control-v Terminate.

Non-printing Characters

'Se' displays a non-printing character as a blank (or other user-selectable character; see the description of "ou" in the section on options). Non-printing characters (such as 'se's control characters), or any others for that matter, may be entered by hitting the ESC key followed immediately by the key to generate the desired character. Note, however, that the character you type is taken literally, exactly as it is generated by your terminal, so case conversion does not apply.

ESC Accept the literal value of the next character, regardless of its function.

Screen Editor Options

Options for 'se' can be specified in two ways: with the "o" command or on the Subsystem command line that invokes 'se'. To specify an option with the "o" command, just enter "o" followed immediately by the option letter and its parameters. To specify an option on the command line, just use "-" followed by the option letter and its parameters. With this second method, if there are imbedded spaces in the parameter list, the entire option should be enclosed in quotes. For example, to specify the "a" (absolute line number) option and tab stops at column 8 and every fourth thereafter with the "o" command, just enter

```
oa
ot 8 +4
```

when 'se' is waiting for a command. To enter the same options on the invoking command line, you might use

```
se -t regent myfile -a "-t 8 +4"
```

The following table summarizes the available 'se' options:

Option	Action
a	causes absolute line numbers to be displayed in the line number area of the screen. The default behavior is to display upper-case letters with the letter "A" corresponding to the first line in the window.
c	inverts the case of all letters you type (i.e., converts upper-case to lower-case and vice versa). This option causes commands to be recognized only in upper-case and alphabetic line numbers to be displayed and recognized only in lower-case.
d[<dir>]	selects the placement of the current line pointer following a "d" (delete) command. <dir> must be either ">" or "<". If ">" is specified, the default behavior is selected: the line following the deleted lines becomes the new current line. If "<" is specified, the line immediately preceding the deleted lines becomes the new current line. If neither is specified, the current value of <dir> is displayed in the status line.
f	selects Fortran oriented options. This is equivalent to specifying both the "c" and "t7 +3" (see below) options.
l[<lop>]	sets the line number display option. Under control of this option, 'se' continuously displays the value of one of three symbolic line numbers. <lop> may be any of the following: . display the current line number # display the number of the top line on the screen $ display the number of the last line in the buffer If <lop> is omitted, the line number display is disabled.
t[<tabs>]	sets tab stops according to <tabs>. <tabs> consists of a series of numbers indicating columns in which tab stops are to be set. If a number is preceded by a plus sign ("+"), it indicates that the number is an increment; stops are set at regular intervals separated by that many columns, beginning with the most recently specified absolute column number. If no such number precedes the first increment specification, the stops are set relative to column 1. By default, tab stops are set in every third column starting with column 1, corresponding to a <tabs> specification of "+3". If <tabs> is omitted, the current tab spacing is displayed in the status line.
u[<chr>]	selects the character that 'se' displays in place of unprintable characters. <chr> may be any printable character; it is initially set to blank. If <chr> is omitted, 'se' displays the current replacement character on the status line.
v[<col>]	sets the default "overlay column". This is the column at which the cursor is initially positioned by the "v" command. <Col> must be a positive integer, or a dollar sign ($) to indicate the end of the line. If <col> is omitted, the current overlay column is displayed in the status line.

w[<col>] sets the "warning threshold" to <col> which must be a positive integer. Whenever the cursor is positioned at or beyond this column, the column number is displayed in the status line and the terminal's bell is sounded. If <col> is omitted, the current warning threshold is displayed in the status line. The default warning threshold is 74, corresponding to the first column beyond the right edge of the screen on an 80 column crt.

-[<lnr>] splits the screen at the line specified by <lnr> which must be a simple line number within the current window. All lines above <lnr> remain frozen on the screen, the line specified by <lnr> is replaced by a row of dashes, and the space below this row becomes the new window on the file. Further editing commands do not affect the lines displayed in the top part of the screen. If <lnr> is omitted, the screen is restored to its full size.

Screen Editor Control Characters

Character Action

control-a Toggle insert mode. The status of the insertion indicator is inverted. Insert mode, when enabled, causes characters typed to be inserted at the current cursor position in the line instead of overwriting the characters that were there previously. When insert mode is in effect, "INSERT" appears in the status line.

control-b Scan right and erase. The current line is scanned from the current cursor position to the right margin until an occurrence of the next character typed is found. When the character is found, all characters from the current cursor position up to (but not including) the scanned character are deleted and the remainder of the line is moved to the left to close the gap. The cursor is left in the same column which is now occupied by the scanned character. If the line to the right of the cursor does not contain the character being sought, the terminal's bell is sounded. 'Se' remembers the last character that was scanned using this or any of the other scanning keys; if control-b is hit twice in a row, this remembered character is used instead of a literal control-b.

control-c Insert blank. The characters at and to the right of the current cursor position are moved to the right one column and a blank is inserted to fill the gap.

control-d Cursor up. The effect of this key depends on 'se's current mode. When in command mode, the current line pointer is moved to the previous line without affecting the contents of the command line. If the current line pointer is at line 1, the last line in the file becomes the new current line. In overlay mode (viz. the "v" command), the cursor is moved up one line while remaining in the same column. In append mode, this key is ignored.

control-e Tab left. The cursor is moved to the nearest tab stop to the left of its current position.

control-f "Funny" return. The effect of this key depends on the editor's current mode. In command mode, the current command line is entered as-is, but is not erased upon completion of the command; in append mode, the current line is duplicated; in overlay mode (viz. the "v" command), the current line is restored to its original state and command mode is reentered (except if under control of a global prefix).

control-g Cursor right. The cursor is moved one column to the right.

control-h Cursor left. The cursor is moved one column to the left. Note that this does not erase any characters; it simply moves the cursor.

control-i Tab right. The cursor is moved to the next tab stop to the right of its current position.

control-k Cursor down. As with the control-d key, this key's effect depends on the current editing mode. In command mode, the current line pointer is moved to the next line without changing the contents of the command line. If the current line pointer is at the last line in the file, line 1 becomes the new current line. In overlay mode (viz. the "v" command), the cursor is moved down one line while remaining in the same column. In append mode, control-K has no effect.

control-l Scan left. The cursor is positioned according to the character typed immediately after the control-l. In effect, the current line is scanned, starting from the current cursor position and moving left, for the first occurrence of this character. If none is found before the beginning of the line is reached, the scan resumes with the last character in the line. If the line does not contain the character being looked for, the message "NOT FOUND" is printed in the status line. 'Se' remembers the last character that was scanned for using this key; if the control-l is hit twice in a row, this remembered character is searched for instead of a literal control-l. Apart from this, however, the character typed after control-l is taken literally, so 'se's case conversion feature does not apply.

control-m Newline. This key is identical to the NEWLINE key described below.

control-n Scan left and erase. The current line is scanned from the current cursor position to the left margin until an occurrence of the next character typed is found. Then that character and all characters to its right up to (but not including) the character under the cursor are erased. The remainder of the line, as well as the cursor are moved to the left to close the gap. If the line to the left of the cursor does not contain the character being sought, the terminal's bell is sounded. As with the control-b key, if control-n is hit twice in a row, the last character

scanned for is used instead of a literal control-n.

control-o Skip right. The cursor is moved to the first position beyond the current end of line.

control-p Interrupt. If executing any command except "a", "c", "i" or "v", 'se' aborts the command and reenters command mode. The command line is not erased.

control-q Fix screen. The screen is reconstructed from 'se's internal representation of the screen.

control-r Erase right. The character at the current cursor position is erased and all characters to its right are moved left one position.

control-s Scan right. This key is identical to the control-l key described above, except that the scan proceeds to the right from the current cursor position.

control-t Kill right. The character at the current cursor position and all those to its right are erased.

control-u Erase left. The character to the left of the current cursor position is deleted and all characters to its right are moved to the left to fill the gap. The cursor is also moved left one column, leaving it over the same character.

control-v Skip right and terminate. The cursor is moved to the current end of line and the line is terminated.

control-w Skip left. The cursor is positioned at column 1.

control-x Insert tab. The character under the cursor is moved right to the next tab stop; the gap is filled with blanks. The cursor is not moved.

control-y Kill left. All characters to the left of the cursor are erased; those at and to the right of the cursor are moved to the left to fill the void. The cursor is left in column 1.

control-z Toggle case conversion mode. The status of the case conversion indicator is inverted; if case inversion was on, it is turned off, and vice versa. Case inversion, when in effect, causes all upper case letters to be converted to lower case, and all lower case letters to be converted to upper case. Note, however, that 'se' continues to recognize alphabetic line numbers in upper case only, in contrast to the "case inversion" option (see the description of options above). When case inversion is on, "CASE" appears in the status line.

control-_ Insert newline. A newline character is inserted before the current cursor position, and the cursor is moved one position to the right. The newline is displayed according to the current non-printing replacement character (see the "u" option).

control-\ Tab left and erase. Characters are erased starting with the character at the nearest tab stop to the left of the cursor up to but not including the character under the cursor. The rest of the line, including the cursor, is moved to the left to close the gap.

control-^ Tab right and erase. Characters are erased starting with the character under the cursor up to but not including the character at the nearest tab stop to the right of the cursor. The rest of the line is then shifted to the left to close the gap.

NEWLINE Kill right and terminate. The characters at and to the right of the current cursor position are deleted, and the line is terminated.

DEL Kill all. The entire line is erased, along with any error message that appears in the status line.

ESC Escape. The ESC key provides a means for entering 'se's control characters literally as text into the file. In fact, any character that can be generated from the keyboard is taken literally when it immediately follows the ESC key. If the character is non-printing (as are all of 'se's control characters), it appears on the screen as the current non-printing replacement character (normally a blank).

Editor Command Summary

Range	Syntax	Function
.	a[:text]	Append Inserts text after the specified line. Text is inserted until a line containing only a period and a newline is encountered. In 'se', if the command is followed immediately by a colon, then whatever text follows the colon is inserted without entering "append" mode. The current line pointer is left at the last line inserted.
.,.	c[:text]	Change Deletes the lines specified and inserts text to replace them. Text is inserted until a line containing only a period and a newline is encountered. In 'se', if the command is followed immediately by a colon, then whatever text follows the colon is inserted without entering "append" mode. The current line pointer is left at the last line inserted.
.,.	d[p]	Delete Deletes all lines between the specified lines, inclusive. The current line pointer is left a the line after the last one deleted. If the "p" is included, the new current line is printed.
none	e [filename]	Enter Loads the specified file into the buffer and prepares for editing. Automatically invoked if a filename is specified as an argument on the command line used to invoke the editor. The current line pointer is positioned at the first line in the buffer.
none	f [filename]	File Print or change the remembered file name. If a name is given, the remembered file name is set to that value; otherwise, the remembered file name is printed.
.,$	g/pat/command	Global on pattern Performs the given command on all lines in the specified range that match a certain pattern.
none	h[stuff]	Help In 'se', provides access to online documentation on the screen editor. "Stuff" may be used to select which information is displayed.
.	i[:text]	Insert Inserts text before the specified line. Text is inserted until a line containing only a period and a newline is encountered. In 'se', if the command is immediately followed by a colon, then whatever text follows is inserted without entering "append" mode. The current line pointer is left at the last line inserted.
^,.	j/stuff/[p]	Join The specified lines are joined into a single line. The newlines that previously separated the lines are replaced by "stuff". If the "p" option is used, the resulting line (which becomes the new current line) is printed.
.,.	kc	marK The specified lines are marked with 'c' which may be any single character other than a newline. If 'c' is not present, the lines are marked with the default name of blank. The current line pointer is never changed.
.,.	m<line>[p]	Move Moves the specified block of lines after <line>. <Line> may not be omitted. The current line pointer is left at the last line moved. If the "p" is specified, the new current line is also printed.
.,.	n[c]	Name If 'c' is present, the last line in the specified range is marked with it and all other lines having that mark name are given the default mark name (" "). In 'ed', if 'c' is not present, the mark name of each line in the range is printed; in 'se' the names of all lines in the range are cleared.

none	o[stuff]	Option Editing options may be queried or set. "Stuff" determines which options are affected.
.,.	p	Print Prints all the lines in the given range. In 'se', as much as possible of the range is displayed, always including the last line; if no range is given, the previous page is displayed. The current line pointer is left at the last line printed.
none	q	Quit Exit from the editor.
.	r [filename]	Read Insert the contents of the given file after the specified line. The current line pointer is left at the last line read.
.,.	s/pat/sub/[g][p]	Substitute Substitutes "sub" for each occurrence of the pattern "pat". If the optional "g" is specified, all occurrences in each line are changed; otherwise, only the first occurrence is changed. The current line pointer is left at the last line in the range in which a substitution was made. This line is also printed if the "p" is used.
.,.	t/from/to/[p]	Transliterate The range of characters specified by 'from' is transliterated into the range of characters specified by 'to'. The last line on which something was transliterated is printed if the "p" option is used. The last line in the range becomes the new current line.
.	u[d][p]	Undo The specified range of lines is replaced by the last range of lines deleted. If the "d" is used, the restored text is inserted after the last line in the specified range. The current line pointer is set at the last line that was restored; this line is also printed if the "p" is specified.
.,.	v	oVerlay In 'ed', each line in the given range is printed without its terminating newline and a line of input is read and added to the end of the line. If the first and only character on the input line is a period, no further lines are printed. In 'se', "overlay mode" is entered and the control characters may be used to modify text anywhere in the buffer. A control-f may be used to restore the current line to its original state and terminate the command.
1,$	w [filename]	Write Writes the portion of the buffer specified to the named file. The current line pointer is not changed.
1,$	x/pat/command	eXclude on pattern Performs the command on all lines in the given range that do not match the specified pattern.
.,.	y<line>[p]	copY Makes a copy of all the lines in the given range, and inserts the copies after <line>. As with the "m" command, <line> may not be omitted. The current line pointer is set to the new copy of the last line in the range; this line is printed if the "p" is present.
.	=[p]	Equals The number of the specified line is printed. The line itself is also printed if the "p" option is used. The current line pointer is not changed.
none	?	Query In 'ed' only, a verbose description of the last error encountered is printed.
1,$	!ccommand	Exclude on markname Similar to the 'x' prefix except that 'command' is performed for all lines in the range that do not have the mark name 'c'.

1,$	'ccommand	Global on markname Similar to the 'g' prefix except that 'command' is performed for all lines in the range that have the mark name 'c'.
.	:	Print next page In 'ed', 23 lines beginning with the current line are printed (equivalent to ".,.+23p"). In 'se', the next page of the buffer is displayed and the current line pointer is placed at the top of the window.

Elements of Line Number Expressions

Form	Value
integer	value of the integer (e.g., 44).
.	number of the current line in the buffer.
$	number of the last line in the buffer.
^	number of the previous line in the buffer (same as .-1).
/pattern/	number of the next line in the buffer that matches the given pattern (e.g., /February/); the search proceeds to the end of the buffer, then wraps around to the beginning and back to the current line.
\pattern\	number of the previous line in the buffer that matches the given pattern (e.g., \January\); search proceeds in reverse, from the current line to line 1, then from the last line back to the current line.
>name	number of the next line having the given markname (search wraps around, like //).
<name	number of the previous line having the given markname (search proceeds in reverse, like \\).
expression	any of the above operands may be combined with plus or minus signs to produce a line number expression. Plus signs may be omitted if desired (e.g., /parse/-5, /lexical/+2, /lexical/2, $-5, .+6, .6).

Summary of Pattern Elements

Element	Meaning
%	Matches the null string at the beginning of a line. However, if not the first element of a pattern, is treated as a literal percent sign.
?	Matches any single character other than newline.
$	Matches the newline character at the end of a line. However, if not the last element of a pattern, is treated as a literal dollar sign.
[<ccl>]	Matches any single character that is a member of the set specified by <ccl>. <Ccl> may be composed of single characters or of character ranges of the form <c1>-<c2>. If character ranges are used, <c1> and <c2> must both belong to the digits, the upper case alphabet or the lower case alphabet.
[~<ccl>]	Matches any single character that is not a member of the set specified by <ccl>.
*	In combination with the immediately preceeding pattern element, matches zero or more characters that are matched by that element.
@	Turns off the special meaning of the immediately following character. If that character has no special meaning, this is treated as a literal "@".
{<pattern>}	Tags the text actually matched by the sub-pattern specifed by <pattern> for use in the replacement part of a substitute command.
&	Appearing in the replacement part of a substitute command, represents the text actually matched by the pattern part of the command.
@<digit>	Appearing in the replacement part of a substitute command, represents the text actually matched by the tagged sub-pattern specified by <digit>.

User's Guide for the Software Tools Subsystem Command Interpreter (The Shell)

T. Allen Akin
Perry B. Flinn
Daniel H. Forsyth, Jr.

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

April, 1980

TABLE OF CONTENTS

Introduction

The Software Tools Subsystem is a set of program development tools based on the book Software Tools by Brian W. Kernighan and P. J. Plauger. It was originally developed for use on the Prime 400 computer in 1977 and 1978 in the form of several cooperating user programs. The present Subsystem, the seventh version, is a powerful tool that aids in the effective use of computing resources.

The command interpreter, also referred to as the "shell," is a vital part of the Subsystem. It is a program which accepts commands typed by the user on his terminal and converts them into more primitive directions to the computer itself. The user's instructions are expressed in a special medium called the "command language." The greatest part of this document is involved with describing the command language and giving examples of how it is used.

Three areas will be covered in the following pages. First, there is a tutorial on the use of the command language. New Subsystem users should read this chapter first. Some minimal knowledge of terminal usage is assumed; if you are unsure of yourself in this area, see Prime's published documentation and the Software Tools Subsystem Tutorial for help. Second, there is a summary of the syntax and semantics of the command language. Experienced users should find this chapter valuable as a reference. Finally, there is a selection of application notes. This chapter is a good source of useful techniques and samples of advanced usage. Experienced users and curious beginners should find it well worthwhile.

Tutorial

Getting Started

After you have logged in to the computer, you must start up the Subsystem. To do this, type the command "swt":

```
OK, swt
]
```

The Subsystem fires up and the command interpreter prompts you for input by typing a right bracket.

Typographical Conventions

The rules for correcting typographical errors under the Subsystem may be different from the usual ones on your system. (You can change the rules, if you like; see the Subsystem Tutorial for more information.) Usually, the Subsystem expects a backspace (control-h) to be used for deleting single characters that are in error, and a DEL (RUBOUT on some terminals) to delete entire lines that are in error.

Throughout this chapter, references will be made to input lines that are terminated with a "newline." You should use the "newline" key on a terminal only if it lacks a "return" key. They do not necessarily have the same effect.

Commands

Input to the command interpreter consists of "commands." Commands, in turn, consist of a "command name," usually made up of letters and digits, and additional pieces of information, often called "parameters" or "arguments." (Note that a command may or may not have arguments, depending on its function.) The command name and any arguments are separated from each other by spaces.

For example:

```
] echo Hello world!
Hello world!
]
```

The command name is "echo". 'Echo' is not a very involved command; it simply types back its parameters, whatever they may be.

Here is another command, one that is a bit more useful:

```
] lf
adventure       ee              guide           m6800
shell           shell.doc       subsys          time_sheet
words           zunde
]
```

'Lf' is used to list the names of your files. ('Lf' may be used with arguments, as well as without them; for more information, see the Software Tools Subsystem Reference Manual, or try typing the command "help lf".)

Special Characters and Quoting

Some characters have special meaning to the command interpreter. For example, try typing this command:

```
] echo Alas, poor Yorick
Alas
poor:  not found
]
```

This strange behavior is caused by the fact that the comma is used for dark mysterious purposes elsewhere in the command language. (The comma actually represents a null I/O connection between nodes of a network. See the section on pipes and networks for more information.) In fact, all of the following characters are potential troublemakers:

```
, ; # @ > | { } [ ] ( ) blank
```

* The way to handle this problem is to use quotes. You may use either single or double quotes,

but be sure to match each with another of the same kind. Try this command now:

```
] echo "Alas, poor Yorick; I knew him well."
Alas, poor Yorick; I knew him well.
]
```

You can use quotes to enclose other quotes:

```
] echo 'Quoth the raven: "Nevermore!"  '
Quoth the raven: "Nevermore!"
]
```

A final word on quoting: Note that anything enclosed in quotes becomes a single argument. For example, the command

```
] echo "Can I use that in my book?"
```

has only one argument, but

```
] echo Can I use that in my book?
```

has seven.

Command Files

Suppose you have a task which must be done often enough that it is inconvenient to remember the necessary commands and type them in every time. For an example, let's say that you have to print the year-end financial reports for the last five years. If the "print" command is used to print files, your command might look like:

```
] print year74 year75 year76 year77 year78 year79
```

If you use a text editor to make a file named "reports" that contains this command, you can then print your reports by typing

```
] reports
```

No special command is required to perform the operations in this "command file;" simply typing its name is sufficient.

Any number of commands may be placed in a command file. It is possible to set up groups of commands to be repeated or executed only if certain conditions occur. See the Applications Notes for examples.

It is one of the important features of the command interpreter that command files can be treated exactly like ordinary commands. As shown in later sections, they are actually programs written in the command language; in fact, they are often called "shell programs." Many Subsystem commands ('e', 'fos', and 'rfl', for example) are implemented in this manner.

Doing Repetitive Tasks --- Iteration

Some commands can accept only a single argument. One example of this is the 'fos' command. "Fos" stands for "format, overstrike, and spool." It is a shorthand command for printing "formatted" documents on the line printer. (A "formatted" document is one prepared with the help of a program called a "text formatter," which justifies right margins, indents paragraphs, etc. This document was prepared by the Software Tools text formatter 'fmt.') If you have several documents to be prepared, it is inconvenient to have to type the 'fos' command for each one. A special technique called "iteration" allows you to "factor out" the repeated text. For example,

```
] fos (file1 file2 file3)
```

is equivalent to

```
] fos file1
] fos file2
] fos file3
```

The arguments inside the parentheses form an "iteration group." There may be more than one iteration group in a command, but they must all contain the same number of arguments. This is because each new command line produced by iteration must have one argument from each group. As an illustration of this,

```
] (echo print fos) file(1 2 3)
```

is equivalent to

```
] echo file1
] print file2
] fos file3
```

Iteration is performed by simple text substitution; if there is no space between an argument and an iteration group in the original command, then there is none between the argument and group elements in the new commands. Thus,

```
file(1 2 3)
```

is equivalent to

```
file1
file2
file3
```

Iteration is most useful when combined with function calls, which will be discussed later.

Sources and Destinations of Data

Control of the sources and destinations of data is a very basic function of the command interpreter, yet one that deserves special attention. The concepts involved are not new, yet they are rarely employed to the extent that they have been used in the Subsystem. The best approach to learning these ideas is to experiment. Get on a terminal, enter the Subsystem, and try the examples given here until they seem to make sense. Above all, experiment freely; try anything that comes to mind. The Subsystem has been designed with the idea that users are intelligent human beings, and their freedom of expression is the most valuable of tools. Use your imagination; if it needs tweaking, take a look at the Application Notes in the last chapter.

Every program and command in the Subsystem can gather data from certain well-known places and deliver it to other well-known places. Data sources are known as "standard input ports;" data destinations are called "standard output ports." Programs are said to "read from standard input" and "write on standard output." The key point here is that programs need not take into account how input data is made available or what happens to output data when they are finished with it; the command interpreter is in complete control of the standard ports.

A command we will use frequently in this section is 'copy'. 'Copy' does exactly what its name implies; it copies data from one place to another. In fact, it copies data from its first standard input port to its first standard output port.

The first point to remember is that by default, standard ports reference the terminal. Try 'copy' now:

```
] copy
```

After you have entered this command, type some random text followed by a newline. 'Copy' will type the same text back to you. (When you tire of this game, type a control-c; this causes an end-of-file signal to be sent to 'copy', which then returns to the command interpreter. Typing control-c to cause end-of-file is a convention observed by all Subsystem programs.) Since you did not say otherwise, standard input and standard output referred to the terminal; input data was taken from the terminal (as you typed it) and output data was placed on the terminal (printed by 'copy').

Obviously, 'copy' would not be of much use if this was all it could do. Fortunately, the command interpreter can change the sources and destinations of data, thus making 'copy' less trivial.

Standard ports may be altered so as to refer to disk files by use of a "funnel." The greater-than sign (>) is used to represent a funnel. Conventionally, the ">" points in the direction of data flow. For example, if you wished to copy the contents of file "ee" to file "old_ee", you could type

```
] ee> copy >old_ee
```

The greater-than sign must always be immediately next to its associated filename; no intervening blanks are allowed. At least one blank must separate the '>' from any command name or arguments. This restriction is necessary to insure that the command language can be * interpreted unambiguously.

The construct "ee>" is read "from ee"; ">old_ee" is read "toward old_ee." Thus, the command above can be read "from ee copy toward old_ee," or, "copy from ee toward old_ee." The process of changing the file assignment of a standard port by use of a funnel is called "I/O redirection," or simply "redirection."

It is not necessary to redirect both standard input and standard output; either may be redirected independently of the other. For example,

```
] ee>  copy
```

can be used to print the contents of file "ee" on the terminal. (Remember that standard output, since it was not specifically redirected, refers to the terminal.) Not surprisingly, the last variation of 'copy',

```
] copy  >old_ee
```

is also useful. This command causes input to be taken from the terminal (until an end-of-file is generated by typing a control-c) and placed on the file "old_ee". This is a quick way of creating a small file of text without using a text editor.

It is important to realize that all Subsystem programs behave uniformly with regard to redirection. It is as correct to redirect the output of, say, 'lf'

```
] lf  >file_list
```

as it is to redirect the output of 'copy'.

Although the discussion has been limited to one input port and one output port up to this point, more of each type are available. In the current implementation, there are a total of six; three for input and three for output. The highest-numbered output port is generally used for error messages, and is often called "ERROUT"; you can "capture" error messages by redirecting this output port. For example, if any errors are detected by 'lf' in this command

```
] lf  3>errors
```

then the resulting error messages will be placed on the file "errors".

Final words on redirection: there are two special-purpose redirection operators left. They are both represented by the double funnel ">>". The first operator is called "append:"

```
] lf  >>list
```

causes a list of files to be placed at the end of (appended to) the file named "list". The second operator is called "from command input." It is represented as just ">>" with no file name, and causes standard input to refer to the current source of commands. It is useful for running programs like the text editor from "scripts" of instructions placed in a command file.

Pipes and Networks

The last section discussed I/O redirection, the process of making standard ports refer to disk files, rather than just to the terminal. This section will take that idea one step further. Frequently, the output of one program is placed on a file, only to be picked up again later and used by another program. The command interpreter simplifies this process by eliminating the intermediate file. The connection between programs that is so formed is called a "pipe," and a linear array of programs communicating through pipes is called a "pipeline."

Suppose that you maintain a large directory, containing drafts of various manuals. Each draft is in a file with a name of the form "MANxxxx.rr", where "xxxx" is the number of the manual and "rr" is the revision number. You are asked to produce a list of the numbers of all manuals at the first revision stage. The following command will do the job:

```
] lf -c | find .01
```

"lf -c" lists the names of all files in the current directory. The "pipe connection" (vertical bar) causes this listing to be passed to the 'find' command, which selects those lines containing the string ".01" and prints them. Thus, the pipeline above will print all filenames matching the conventional form of a first-revision manual name.

The ability to build special purpose commands cheaply and quickly from available tools using pipes is one of the most valuable features of the command interpreter. With practice, surprisingly difficult problems can be solved with ease. For further examples of pipelines, see the Applications Notes.

Combinations of programs connected with pipes need not be linear. Since multiple standard ports are available, programs can be and often are connected in non-linear networks. (Some networks cannot be executed if the programs in the network are not executed concurrently. The command interpreter detects such networks, and prints a warning message if they cannot be performed.) Further information on networks can be found in both the reference and applications chapters of this Guide.

Compound Nodes

It is sometimes necessary to change the standard port environment of many commands at one time, for reasons of convenience or efficiency. The "compound node" (a set of networks surrounded by curly braces) can be used in these situations.

As an example of the first case, suppose that you wish to generate a list of manual names (see the last example) in either the first or the second stage of revision. One way to do this is to generate the list for the first revision stage, place it on a file using a funnel, then generate a list for the second revision stage and place it on the end of the same file using an "append" redirector. A compound node might simplify the procedure thusly:

```
] { lf -c | find .01;  lf -c | find .02 }  >list
```

The first network finds all manuals at the first revision stage, and the second finds all those at the second stage. The networks will execute left-to-right, with the output of each being placed on the file "list," thus generating the desired listing. With iteration, the command can be collapsed even farther:

```
] { lf -c | find .0(1 2) }  >list
```

This combination of iteration and compound nodes is often useful.

Efficiency becomes a consideration in cases where successive long streams of data are to be copied onto a file; if the "append" redirector is used each time, the file must be reopened and repositioned several times. Using a compound node, the output file need be opened only once:

```
] { (file1 file2 file3)> copy }  >all_files
```

This complex example copies the contents of files "file1," "file2," and "file3" into the file named "all_files."

Function Calls

Programs in the Subsystem receive information through their command-line arguments as well as their standard ports, so it is sometimes useful to deliver the output of a program to the command interpreter for further processing, rather than to a pipe. The "function call" mechanism is available for this purpose. For example, recall the situation illustrated in the section on pipes and networks; suppose it is necessary to actually print the manuals whose names were found. This is how the task could be done:

```
] print [lf -c | find .01]
```

The function call is composed of the pipeline "lf -c | find .01" and the square brackets enclosing it. The output of the pipeline within the brackets is passed to 'print' as a set of arguments, which it accesses in the usual manner. Specifically, all the lines of output from the pipeline are combined into one set of arguments, with spaces provided where multiple lines have been collapsed into one line.

'Print' accepts multiple arguments; however, suppose it was necessary to use a program like 'fos', that accepts only one argument. Iteration can be combined with a function call to do the job:

```
] fos ([lf -c | find .01])
```

This command formats and prints all manuals in the current directory with revision numbers "01".

Function calls are frequently used in command files, particularly for accessing arguments passed to them. Since the sequence "lf -c | find pattern" occurs very frequently, it is a good candidate for replacement with a command file; it is only necessary to pass the pattern to be matched from the argument list of the command file to the 'find' command with a function call. The following command file, called 'files', will illustrate the process:

```
lf -c | find [arg 1]
```

"arg 1" retrieves the first command file argument. The function call then passes that argument to 'find' through its argument list. 'Files' may then be used anywhere the original network was appropriate:

```
] files .01
] print [files .01]
] fos ([files .01])
```

Variables

It has been claimed that the command language is a programming language in its own right. One facet of this language that has not been discussed thus far is the use of its variables. The command interpreter allows the user to create variables, with scope, and assign values to them or reference the values stored in them.

Certain special variables are used by the command interpreter in its everyday operation. These variables have names that begin with the underline (_). One of these is '_prompt', which is the prompt string the command interpreter prints when requesting a command. If you object to "]" as a prompt, you can change it with the "set" command:

```
] set _prompt = "OK, "
OK, set _prompt = "% "
% set _prompt = "] "
]
```

You may create and use variables of your own. To create a variable in the current scope (level of command file execution), use the "declare" comand:

```
] declare i j k sum
```

Values are assigned to variables with the 'set' command. The command interpreter checks the current scope and all surrounding scopes for the variable to be set; if found, it is changed, otherwise it is declared in the current scope and assigned the specified value.

Variables behave like small programs that print their current values. Thus the value of a variable can be obtained by simply typing its name, or it can be used in a command line by enclosing it in brackets to form a function call. The following command file (which also illustrates the use of 'if', 'eval', and 'goto') will count from 1 to the number given as its first argument:

```
declare i
set i = 1
:loop
   if [eval i ">" [arg 1]]
      goto exit
   fi
   i
   set i = [eval i + 1]
   goto loop
:exit
```

Note the use of the "eval" function, which treats its arguments as an arithmetic expression and returns the expression's value. This is required to insure that the string "i + 1" is interpreted as an expression rather than as a character string. Also note that 'fi' terminates the 'if' command.

Conclusion

This concludes the tutorial chapter of this document. Despite the fact that a good deal of material has been presented, much detail has been omitted. The next chapter is a complete summary of the capabilities of the command interpreter. It is written in a rather technical style, and is recommended for reference rather than self-teaching. The last chapter is a set of examples that may prove helpful. As always, the best approach is simply to sit down at a terminal and try out whatever you wish to do. Should you have difficulty, further tutorials are available, and the 'help' command can be consulted for quick reference.

Summary of Syntax and Semantics

This section is the definitive document for the syntax and corresponding semantics of the Software Tools Subsystem Command Interpreter. It is composed of several sub-sections, each covering some major area of command syntax, with discussions of the semantic consequences of employing particular constructs. It is not intended as a tutorial, nor is it intended to supply multitudinous examples; the other sections of this document are provided to fill those needs.

Commands

<command> ::= [<net> { ; <net> }] <newline>

The "command" is the basic unit of communication between the command interpreter and the user. It consists of any number of networks (described below) separated by semicolons and terminated by a newline. The networks are executed one at a time, left-to-right; should an error occur at any point in the parse or execution of a network, the remainder of the <command> is ignored. The null command is legal, and causes no action.

The command interpreter reads commands for interpretation from the "command source." This is initially the user's terminal, although execution of a command file may change the assignment. Whenever the command source is the terminal, and the command interpreter is ready for input, it prompts the user with the string contained in the shell variable '_prompt'. Since this variable may be altered by the user, the prompt string is selectable on a per-user basis.

Networks

<net> ::= <node>
{ <node separator> { <node separator> } <node> }

<node separator> ::= , | <pipe connection>

<pipe connection> ::= [<port>] '|' [<node number>] [.<port>]

<port> ::= <integer>

<node number> ::= <integer> | $ | <label>

A <net> generates a block of (possibly concurrent) processes that are bound to one another by channels for the flow of data. Typically, each <node> corresponds to a single process. (<Node>s are described in more detail below.) There is no predefined "execution order" of the processes composing a <net>; the command interpreter will select any order it sees fit in order to satisfy the required input/output relations. In particular, the user is specifically enjoined not to assume a left-to-right serial execution, since some <net>s cannot be executed in this manner.

Input/output relations between <node>s are specified with the <node separator> construct. The following discussion may be useful in visualizing the data flows in a <net>, and clarifing the function of the components of the <node separator>.

The entire <net> may be represented as a directed graph with one vertex for each <node> (typically, equivalent to each process) in the net. Each vertex may have up to n arcs terminating at it (representing "input data streams"), and m arcs originating from it (representing "output data streams"). An arc between two vertices indicates a flow of data from one <node> to another, and is physically implemented by a pipe.

Each of the n possible input points on a <node> is assigned an identifier consisting of a unique integer in the range 1 to n. These identifiers are referred to as the "port numbers" for the "standard input ports" of the given <node>. Similarly, each of the m possible output points on a <node> is assigned a unique integer in the range 1 to m, referred to as the port numbers for the "standard output ports" of the given <node>.

Lastly, the <node>s themselves are numbered, starting at 1 and increasing by 1 from the left end of the <net> to the right.

Clearly, in order to specify any possible input/output connection between any two <node>s, it is sufficient to specify:

. The number of the "source" <node>.

- The number of the "destination" <node>.
- The port number of the standard output port on the source <node> that is to be the source of the data.
- The port number of the standard input port on the destination <node> that is to receive the data.

The syntax for <node separator> includes the specifications for the last three of these items. The source <node> is understood to be the node that immediately preceeds the <node separator> under consideration. The special <node separator> "," is used to separate <node>s that do not participate in data sharing; it specifies a null connection. Thus, the <node separator> provides a means of establishing any possible connection between two <node>s of a given <net>.

The full flexibility of the <node separator> is rarely needed or desirable. In order to make effective use of the capabilities provided, suitable defaults have been designed into the syntax. The semantics associated with the defaults are as follows:

- If the output port number (the one to the left of the vertical bar) is omitted, the next unassigned output port (in increasing numerical order) is implied. This default action takes place only after the entire <net> has been examined, and all non-defaulted output ports for the given node have been assigned. Thus, if the first <node separator> after a <node> has a defaulted output port number, port 1 will be assigned if and only if no other <node separator> attached to that <node> references output port 1. It is an error for two <node separators> to reference the same output port. (This particular behavior may be changed in the future to allow "forking" output streams, which would be copied to more than one destination.)
- If the destination <node> number is omitted, then the next node in the <net> (scanning from left to right) is implied. Occasionally a null <node> is generated at the end of a <net> because of the necessity for resolving such references.
- If the destination <node>'s input port number is omitted, then the next unassigned input port (in increasing numerical order) is implied. As with the defaulted output port, this action takes place only after the entire <net> has been examined. The comments under (1) above also apply to defaulted input ports.

In addition to the defaults, specifying input/output connections between widely separated <node>s is aided by alternative means of giving <node> numbers. The last <node> in a <net> may be referred to by the <node number> $, and any <node> may be referred to by an alphanumeric <label>. (<Node> labelling is discussed in the section on <node> syntax, below.) If the first <node> of a <net> is labelled, the <net> may serve as a target for the 'goto' command; see the Applications Notes for examples.

As will be seen in the next section, further syntax is necessary to completely specify the input/output environment of a <node>; the reader should remember that <node separator>s control only those flows of data between processes.

A few examples of the syntax presented above may help to clarify some of the semantics. Since the syntax of <node> has not yet been discussed, <node>s will be represented by the string "node" followed by a digit, for uniqueness and as a key to <node number>s.

A simple linear <net> of three <node>s without defaults:

```
node1  1|2.1  node2  1|3.1  node3
```

(Data flows from output port 1 of node1 to input port 1 of node2 and output port 1 of node2 to input port 1 of node3.)

The same <net>, with defaults:

```
node1 | node2 | node3
```

(Note that the spaces around the vertical bars are mandatory, so that the lexical analysis routines of the command interpreter can parse the elements of the command unambiguously.)

A simple cycle:

```
node1 |1.2
```

(Data flows from output port 1 of node1 to input port 2 of node1. Other data flows are unspecified at this level.)

A branching <net> with overidden defaults:

```
node1  |$  node2  |.1  node3
```

(Data flows from output port 1 of node1 to input port 2(!) of node3 and output port 1 of node2 to input port 1 of node3.)

Nodes

```
<node> ::=  {:<label>}  [ <simple node> | <compound node> ]

<simple node> ::= { <i/o redirector> }
                  <command name>
                  { <i/o redirector> | <argument> }

<compound node> ::= { <i/o redirector> }
                    '{' <net> { <net separator> <net> } '}'
                    { <i/o redirector> }

<i/o redirector> ::= <file name> '>'  [ <port> ]    |
                     [ <port> ]  '>'  <file name>   |
                     [ <port> ]  '>>' <file name>   |
                                 '>>' [ <port> ]

<net separator> ::=  ;

<command name> ::= <file name>

<label> ::= <identifier>
```

The <node> is the basic executable element of the command language. It consists of zero or more labels (strings of letters, digits, and underscores, beginning with a letter), optionally followed by one of two additional structures. Although, strictly speaking, the syntax allows an empty node, in practice there must be either a label or one of the two additional structures present.

The first option is the <simple node>. It specifies the name of a command to be performed, any arguments that command may require, and any <i/o redirector>s that will affect the data environment of the command. (<I/o redirectors will be discussed below.) The execution of a simple node normally involves the creation of a single process, which performs some function, then returns to the operating system.

The second option is the <compound node>. It specifies a <net> which is to be executed according to the usual rules of <net> evaluation (see the previous subsection), and any <i/o redirector>s that should affect the environment of the <net>. The <compound node> is provided for two reasons. One, it is occasionally useful to alter default port assignments for an entire <net> with <i/o redirector>s, rather than supplying <i/o redirector>s for each <node>. Two, use of compound nodes containing more than one <net> gives the user some control over the order of execution of his processes. These abilities are discussed in more detail below.

Since it is the more basic construct, consider the <simple node>. It consists of a <command name> with <argument>s, intermixed with <i/o redirector>s. The <command name> must be a filename, usually specifying the name of an object code file to be loaded. The command interpreter locates the command to be performed by use of a user-specified "search rule." The search rule resides in the shell variable "_search_rule", and consists of a series of comma-separated elements. Each element is either a template in which ampersands (&) are replaced by the <command name> or a flag instructing the command interpreter to search one of its internal tables. The flag "^int" indicates that the command interpreter's repertoire of "internal" commands is to be checked. (An internal command is implemented as a subroutine of the command interpreter, typically for speed or because of a need to access some private data base.) The flag "^var" causes a search of the user's "shell variables" (see below for further discussion of variables and functions). The following search rule will cause the command interpreter to search for a command among the internal commands, shell variables, and the directory "=bin=", in that order:

```
"^int,^var,=bin=/&"
```

The purpose of the search rule is to allow optimization of command location for speed, and to admit the possibility of restricting some users from accessing "privileged" commands. (For example, the search rule

```
"^var,//project/library/&"
```

would restrict a user to accessing his variables and those commands in the directory

"//project/library". He could not alter this restriction, since he does not have access to the (internal) 'set' command; the "^int" flag is missing from his search rule.)

<Argument>s to be passed to the program being readied for execution are gathered by the command interpreter and placed in an area of memory accessible to the library routine 'getarg'. They may be arbitrary strings, separated from the command name and from each other by blanks. Quoting may be necessary if an <argument> could be interpreted as some other element of the command syntax. Either single or double quotes may be used. The appearance of two strings adjacent to one another without blanks implies concatenation. Thus,

```
"quoted "string
```

is equivalent to

```
"quoted string"
```

or to

```
quoted' string'
```

Single quotes may appear within strings delimited by double quotes, and vice versa; this is the only way to include quotes within a string. Example:

```
"'quoted string'"
'"Alas, poor Yorick!"'
```

Arguments are generally unprocessed by the command interpreter, and so may contain any information useful to the program being invoked.

In the previous section, it was shown that streams of data from "standard ports" could be piped from program to program through the use of the <pipe connection> syntax. It is also possible to redirect these data streams to files, or to use files as sources of data. The construct that makes this possible is the <i/o redirector>. The <i/o redirector> is composed of filenames, port numbers (as described in the last section), and one or two occurrences of the "funnel" (>).

The two simplest forms take input from a file to a standard port or output from a standard port to a file. In the case of delivering output to a file, the file is automatically created if it did not exist, and overwritten if it did. In the case of taking input from a file, the file is unmodified. Example:

```
documentation>1
```

causes the data on the file "documentation" to be passed to standard input port 1 of the node;

```
1>results
```

causes data written to standard output port 1 of the node to be placed on the file "results".

If no <i/o redirector> is present for a given port, then that port automatically refers to the user's terminal.

If port numbers are omitted, an assignment of defaults is made. The assignment rule is identical to that given above for <pipe connections>: the first available port after the entire <net> has been scanned is used. <I/O redirector>s are evaluated left-to-right, so leftmost defaulted redirectors are assigned to lower-numbered ports than those to their right. For example,

```
data> requests>  trans  2>summary 3>errors | sp
```

is the same as

```
data>1 requests>2 trans 2>summary 3>errors 1|2.1 sp
```

where all defaults have been elaborated. 'Trans' might be some sort of transaction processor, accepting data input and update requests, and producing a report (here printed off-line by being piped to a spooler program), a summary of transactions, and an error listing.

In addition to the <i/o redirector>s mentioned above, there are two lesser-used redirectors that are useful. The first appends output to a file, rather than overwriting the file. The syntax is identical to the other output redirector, with the exception that two funnels '>>' are used, rather than one. For example,

```
6>>stuff
```

causes the data written to output port 6 to be appended to the file "stuff". (Note the lack of spaces around the redirector; a redirector and its parameters are never separated from one another, but are always separated from surrounding arguments or other text. This restriction is necessary to insure unambiguous interpretation of the redirector.) The second redirector causes input to be taken from the current command source file. It is most useful in conjunction with command files. The syntax is similar to the input redirector mentioned above, but two funnels are used and no filename may be specified. As an example, the following segment of a command file uses the text editor to change all occurrences of "March" to "April" in a given file:

```
>> ed file
g/March/s//April/
w
q
```

When the editor is invoked, it will take input directly from the command file, and thus it will read the three commands placed there for it.

The "command source" and "append" redirectors are subject to the same resolution of defaults as the other redirectors and <pipe connection>s. Thus, in the example immediately above,

```
>> ed file
```

is equivalent to

```
>>1 ed file
```

Now that the syntax of <node> has been covered, just two further considerations remain. First, the nature of an executable program must be defined. Second, the problem of execution order must be clarified.

In the vast majority of cases, a <node> is executed by bringing an object program into memory and starting it. However, the <command name> may also specify an internal command, a shell variable, or a command file. Internal commands are executed within the command interpreter by the invocation of a subroutine. When a shell variable is used as a command, the net effect is to print the value of the variable on the first output port, followed by a newline. If the filename specified is a text file rather than an object file, the command interpreter "guesses" that the named file is a file of commands to be interpreted one at a time. In any case, command invocation is uniform, and any <i/o redirector> or <pipe connection> given will be honored. Thus, it is allowable to redirect the output of a command file just as if it were an object program, or copy a shell variable to the line printer by connecting it to the spooler through a pipe.

As mentioned in the section on <net>s, the execution order of nodes in a <net> is undefined. That is, they may be executed serially in any order, concurrently, or even simultaneously. The exact method is left to the implementor of the command interpreter. In any case, the flows of data described by <pipe connection>s and <i/o redirector>s are guaranteed to be present. There are times when it would be preferable to know the order in which a <net> will be evaluated; to help with this situation, <compound node>s may be used to effect serialization of control flow within a network. <Net>s separated by semicolons or newlines are guaranteed to be executed serially, left-to-right, otherwise the command interpreter would exhibit unpredictable behavior as the user typed in his commands. Suppose it is necessary to operate four programs; three may proceed concurrently to make full use of the multiprogramming capability of the computer system, but the fourth must not be executed until the second of the three has terminated. For simplicity, we will assume there are no input/output connections between the programs. The following command line meets the requirements stated above:

```
program1, {program2; program4}, program3
```

(Recall that the comma represents a null i/o connection.) Suppose that we have a slightly different problem: the fourth program must run after all of the other three had run to completion. This, too, can be expressed concisely:

```
program1, program2, program3; program4
```

Thus, the user has fairly complete control over the execution order of his <net>s. (The use of commas and semicolons in the command language is analogous to their use for collateral and serial elaboration in Algol 68.)

This completes the discussion of the core of the command language. The remainder of the features present in the command interpreter are provided by a built-in preprocessor, which handles function calls, iteration, and comments. The next few sections deal with the preprocessor's capabilities.

Comments

Any good command language should provide some means for the user to comment his code, particularly in command files that may be used by others. The command interpreter has a simple comment convention: Any text between an unquoted sharp sign (#) and the next newline is ignored. A comment may appear at the beginning of a line, like this:

```
# command file to preprocess, compile, and link edit
```

Or after a command, like this:

```
file.r> rp   # Ratfor's output goes to the terminal
```

Or even after a label, for identification of a loop:

```
:loop     # beginning of daily cycle
```

As far as implications in other areas of command syntax, the comment is functionally equivalent to a newline.

Variables

<variable> ::= <identifier>

<set command> ::= set [<variable>] = [<argument>]

<declare command> ::= declare { <variable [= <argument>] }

<forget command> ::= forget <variable> { <variable> }

The command interpreter supports named string storage areas for miscellaneous user applications. These are called variables. Variables are identified by a name, consisting of letters of either case, digits, and underscores, not beginning with a digit. Variables have two attributes: value and scope. The value of a variable may be altered with the 'set' command, discussed below. The scope of a variable is fixed at the time of its creation; simply, variables declared during the time when the command interpreter is taking input from a command file are active as long as that file is being used as the command source. Variables with global scope (those created when the command interpreter is reading commands from the terminal) are saved as part of the user's profile, and so are available from terminal session to terminal session. Other variables disappear when the execution of the command file in which they were declared terminates.

Variables may be created with the 'declare' command. 'Declare' creates variables with the given names at the current lexical level (within the scope of the current command file). The newly-created variables are assigned a null value, unless an initialization string is provided.

Variables may be destroyed prematurely with the 'forget' command. The named variables are removed from the command interpreter's symbol table and storage assigned to them is released to the system. Note that variables created by operations within a command file are automatically released when that command file ceases to execute. Also note that the only way to destroy variables at the global lexical level is to use the 'forget' command.

The value of a variable may be changed with the 'set' command. The first argument to 'set' is the name of the variable to be changed. If absent, the value that would have been assigned is printed on 'set's first standard output. The last argument to 'set' is the value to be assigned to the variable. It is uninterpreted, that is, treated as an arbitrary string of text. If missing, 'set' reads one line from its first standard input, and assigns the resulting string. If the variable named in the first argument has not been declared at any lexical level, 'set' declares it at the current lexical level.

Variables are accessed by name, as with any command. (Note that the user's search rule must contain the flag "^var" before variables will be evaluated.) The command interpreter prints the value of the variable on the first standard output. This behavior makes variables useful in function calls (discussed below). In addition, the user may obtain the value of a variable for checking simply by typing its name as a command.

Iteration

<iteration> ::= '(' <element> { <element> } ')'

Iteration is used to generate multiple command lines each differing by one or more substrings. Several iteration elements (collectively, an "iteration group") are placed in

parentheses; the command interpreter will then generate one command line for each element, with successive elements replacing the instance of iteration. Iteration takes place over the scope of one <net>; it will not extend over a <net separator>. (If iteration is applied to a <compound node>, it will, of course, apply to the entire <node>; not just to the first <net> within that <node>.)

Multiple iterations may be present on one command; each iteration group must have the same number of elements, since the command interpreter will pick one element from each group for each generated command line. (Cross-products over iteration groups are not implemented.)

An example of iteration:

```
] fos part(1 2 3)
```

is equivalent to

```
] fos part1; fos part2; fos part3
```

and

```
] cp (intro body summary) part(1 2 3)
```

is equivalent to

```
] cp intro part1; cp body part2; cp summary part3
```

Function Calls

<function call> ::= '[' <net> { <net separator> <net> } ']'

Occasionally it is useful to be able to pass the output of a program along as arguments to another program, rather than to an input port. The "function call" makes this possible. The output appearing on each of the first standard output ports of the <net>s within the function call is copied into the command line in place of the function call itself. Line separators (newlines) present in the <net>'s output are replaced by blanks. No quoting of <net> output is performed, thus blank-separated tokens will be passed as separate arguments. (If quoting is desired, the filter 'quote' can be used or the shell variable "_quote_opt" may be set to the string "YES" to cause automatic quotation.)

A <net> may of course be any network; all the syntax described in this document is applicable. In particular, the name of a variable may appear with the brackets; thus, the value of a variable may be substituted into the command line.

Conclusion

This concludes the description of command syntax and semantics. The next, and final, chapter contains actual working examples of the full command syntax, along with suggested applications; it is highly recommended for those who wish to gain proficiency in the use of the command language.

Application Notes

This section consists mostly of examples of current usage of the command interpreter. Extensive knowledge of some Subsystem programs may be necessary for complete understanding of these examples, but basic principles should be clear without this knowledge.

Basic Functions

In this section, some basic applications of the command language will be discussed. These applications are intended to give the user a "feel" for the flow of the language, without being explicitly pedagogical.

One commonly occurring task is the location of lines in a file that match a certain pattern. The 'find' command performs this function:

```
] file> find pattern >lines_found
```

Since the lines to be checked against the pattern are frequently a list of file names, the following sequence occurs often:

```
] lf -c directory | find pattern
```

Consequently, a command file named 'files' is available to abbreviate the sequence:

```
] cat =bin=/files
lf -c [args 2] | find [arg 1]
```

('Cat' is used here only to print the contents of the command file.) The internal command 'arg' is used to fetch the first argument on the command line that invoked 'files'. Similarly, the internal command 'args' fetches the second through the last arguments on the command line. The command file gives the external appearance of a program 'files' such that

```
] files pattern
```

is equivalent to

```
] lf -c | find pattern
```

and

```
] files pattern directory
```

is equivalent to

```
] lf -c directory | find pattern
```

Once a list of file names is obtained, it is frequently processed further, as in this command to print Ratfor source files on the line printer:

```
] pr [files .r$ | sort]
```

'Files' produces a list of file names with the ".r" suffix, which is then sorted by 'sort'. 'Pr' then prints all the named files on the line printer.

One problem arises when the pattern to be matched contains command language metacharacters. When the pattern is substituted into the network within 'files', and the command interpreter parses the command, trouble of some kind is sure to arise. There are two solutions: One, the filter 'quote' can be used to supply a layer of quotes around the pattern:

```
lf -c [args 2] | find [arg 1 | quote]
```

Two, the shell variable "_quote_opt", which controls automatic function quotation by the command interpreter, can be set to the string "YES":

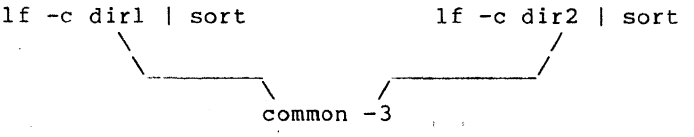
```
declare _quote_opt = YES
lf -c [args 2] | find [arg 1]
```

This latter solution works only because 'args' prints each argument on a separate line; the command interpreter always generates separate arguments from separate lines of function output. In practice, the first solution is favored, since the non-intuitive quoting is made more evident.

One common non-linear command structure is the so-called "Y" structure, where two streams of data join together to form a third (after some processing). This situation occurs because of the presence of dyadic operations (especially comparisons) in the tools available under the Subsystem. As an example, the following command compares the file names in two directories and lists those names that are present in both:

```
] lf -c dir1 | sort |$ lf -c dir2 | sort | common -3
```

Visualize the command in this way:



The two 'lf' and 'sort' pairs produce lists of file names that are compared by 'common', which produces a list of those names common to both input lists.

Command files tend to be used not only for oft-performed tasks but also to make life easier when typing long, complex commands. Quite often these long command lines make use of line continuation -- a newline preceded immediately by an underscore is ignored. The following command file is used to create a keyword-in-context index from the heading lines of the Subsystem Reference Manual. Although it is not used frequently, it does a great deal of work and is illustrative of many of the features of the command interpreter.

```
# make_cmd.k --- build permuted index of commands
   files .d$ -f sl _
      | change % "find %.hd -o 1" _
      | sh _
      | change '%.hd  *{[~ ]*} [ "]*{[~"]*}?*' '@1: @2' _
      | kwic -d =aux=/spelling/discard _
      | sort -d | unrot -w [width]  >cmd.k
```

First a few words on how Subsystem documentation is stored: The documentation for Subsystem commands resides in a subdirectory named "sl". The documentation for each command is in a separate file with the name "<command>.d". The heading line in each file can be identified by the characters ".hd" at the beginning of the line.

The entire command file consists of a single network. The 'files' command produces a list of the full path names (the -f option is passed on to 'lf') of the files in the subdirectory "sl" that have path names ending with the characters ".d". The next 'change' command generates a 'find' command for each documentation file to find the heading line. These command lines are passed back to the shell ('sh') for execution. The outputs of all of these 'find' commands, namely the heading lines from all the documentation files, are passed back on the first standard output of 'sh'. The second 'change' command uses tagged patterns to isolate the command name and its short description from the header line and to construct a suitable entry for the kwic index generator. Finally, 'kwic', 'sort', and 'unrot' produce the index on the file "cmd.k".

To this point, only serially-executed commands have been discussed, however sophisticated or parameterized. Control structures are necessary for more generally useful applications. The following command file, 'ssr', shows a useful technique for parameter-setting commands. Like many APL system commands, 'ssr' without arguments prints the value it controls (in this case, the user's command search rule), while 'ssr' with an argument sets the search rule to the argument given, then prints the value for verification. 'Ssr' looks like this:

```
# ssr --- set user's search rule and print it

   if [nargs]
      set _search_rule = [arg 1 | quote]
   fi

   _search_rule
```

The 'if' command conditionally executes other commands. It requires one argument, which is interpreted as "true" if it is present, not null, and non-zero. If the argument is true, all the commands from the 'if' to the next unmatched 'else' or 'fi' command are executed. If the argument is false, all the commands from the next unmatched 'else' command (if one is present) to the next unmatched 'fi' command are executed. In 'ssr' above, the argument to 'if' is a function call invoking 'nargs', a command that returns the number of arguments passed to the command file that is currently active. If 'nargs' is zero, then no arguments were specified, and 'ssr' does not set the user's search rule. If 'nargs' is nonzero, then 'ssr' fetches the first argument, quotes it to prevent the command interpreter from evaluating special characters, and assigns it to the user's search rule variable '_search_rule'.

'If' is useful for simple conditional execution, but it is often necessary to select one among several alternative actions instead of just one from two. The 'case' command is available to perform this function. One example of 'case' is the command file 'e', which is used to invoke either the screen editor or the line editor depending on which terminal is being used (as well as remembering the name of the file last edited):

```
# e --- invoke the editor best suited to a terminal

   if [nargs]
      set f = [arg 1 | quote]
   fi

   case [line]
      when 10
         se -t consul [se_params] [f]
      when 11
         se -t b200   [se_params] [f]
      when 15
         se -t b150   [se_params] [f]
      when 17
         se -t gt40   [se_params] [f]
      when 18
         se -t b200   [se_params] [f]
      when 25
         se -t b150   [se_params] [f]
      out
         ed [f]
   esac
```

The first 'if' command sets the remembered file name (stored in the shell variable 'f') in the same fashion that 'ssr' was used to set the search rule (above). The 'case' command then selects from the terminals it recognizes and invokes the proper text editor. The argument of 'case' is compared with the arguments of successive 'when' commands until a match occurs, in which case the group of commands after the 'when' is executed; if no match occurs, then the commands after the 'out' command will be executed. (If no 'out' command is present, and no match occurs, then no action is taken as a result of the 'case'.) The 'esac' command marks the end of the control structure. In 'e', the 'case' command selects either 'se' (the screen editor) or 'ed' (the line editor), and invokes each with the proper arguments (in the case of 'se', identifying the terminal type and specifying any user-dependent personal parameters).

The 'goto' command may be used to set up a loop within a command file. For example, the following command file will count from 1 to 10:

```
# bogus command file to show computers can count

   declare i = 1

   :loop
      i
      set i = [eval i + 1]
      if [eval i <= 10]
         goto loop
      fi
```

In actual experience, little need has been found for loops within command files; thus the need for this contrived example.

Shell Control Variables

Several special shell variables are used to control the operation of the command interpreter. The following table identifies these variables and gives a short explanation of the function of each.

Variable	Function
_ci_name	This variable is used to select a command interpreter to be executed when the user enters the Subsystem. It should be set to the full pathname of the command interpreter desired. The default value is "=bin=/sh".
_erase	This variable may be set to a single character or an ASCII mnemonic for a character to be used as the "erase," or character delete, control character for Subsystem terminal input processing. The change in erase character becomes effective only after the Subsystem is re-entered and the initialization routines read the shell variable storage file.

| _escape | This variable may be set to a single character or an ASCII mnemonic for a character to be used as the "escape" control character for Subsystem terminal input processing. The change in escape character becomes effective only after the Subsystem is re-entered and the initialization routines read the shell variable storage file.

| _hello | This variable, if present, is used as the source of a command to be executed whenever the user enters the Subsystem. It is frequently used to implement memo systems, supply system status information, and print pleasing messages-of-the-day.

| _kill | This variable may be set to a single character or an ASCII mnemonic for a character to be used as the "kill," or line delete, control character for Subsystem terminal input processing. The change in kill character becomes effective only after the Subsystem is re-entered and the initialization routines read the shell variable storage file.

| _prompt | This variable contains the prompt string printed by the command interpreter before any command read from the user's terminal. The default value is a right bracket (]).

| _quote_opt | This variable, if set to the value "YES", causes automatic quotation of each line of program output used in a function call. It is mainly provided for compatibility with an older version of the command interpreter, which performed the quoting automatically. The program 'quote' may be used to explicitly force quotation.

| _retype | This variable may be set to a single character or an ASCII mnemonic for a character to be used as the "retype" control character for Subsystem terminal input processing. The change in retype character becomes effective only after the Subsystem is re-entered and the initialization routines read the shell variable storage file.

| _search_rule | This variable contains a sequence of comma-separated elements that control the procedure used by the command interpreter to locate the object code for a command. Each element is either (1) the flag "^int", meaning the command interpreter's table of internal commands, (2) the flag "^var", meaning the user's shell variables, or (3) a template containing the character ampersand (&), meaning a particular directory or file in a directory. In the last case, the command name specified by the user is substituted into the template at the point of the ampersand, hopefully providing a full pathname that locates the
* object code needed.

Symbiotic Commands

There are several commands that, in effect, live symbiotically with the Shell. In the | following sections, some of the more useful of these will be reviewed. For further | information, consult the Software Tools Subsystem Reference Manual.

| Argument Fetching. Four internal commands are frequently used in shell programs to | fetch arguments given on the command line. 'Arg' fetches a single argument, 'args' fetches | several, 'argsto' fetchs a specified group, and 'nargs' returns the number of available arguments.

arg <position> [<level>]

'Arg' prints on its first standard output the argument which appeared in the | <position>th position in the command line invoking the shell program containing 'arg'. Position zero refers to the command name, position one to the first argument, etc. If an illegal position is specified, 'arg' prints nothing. The optional second argument, <level>, specifies the number of lexic levels to ascend in order to reach the desired argument list. The entry of any command file or function call constitutes a new lexic level; thus, an 'arg' command used in a function call to fetch an argument to the command file containing the function call needs a <level> of 1 (to escape the lexic level in which the function is evaluated). In fact, this is the most common use of 'arg', so the default value for <level> is 1. The following three commands, when placed in a command file, would cause that command file's first argument to be printed three times on standard output one:

```
echo [arg 1]
echo [arg 1 1]
arg 1 0
```

args <first> [<last> [<level>]]

'Args' prints on its first standard output the arguments specified on the command file <level> lexic levels above the current level. <First> is the position on the command line of the first argument to be printed; <last> is the position of the last argument to be printed. If <last> is omitted, the final argument on the command line is assumed. <Level> has the same meaning as for 'arg' above.

argsto <delim> [<number> [<start> [<level>]]]

'Argsto' prints a group of arguments delimited by arguments consisting of <delim>. <Number> is an integer that controls which group of arguments is printed. If <number> is 0 or omitted, arguments up to the first occurence of <delim> are printed; if <number> is 1, arguments between the first occurrence of <delim> and the second occurrence of <delim> are printed, and so on. <Start> is an integer indicating the argument at which the scan is to begin; if <start> is omitted (or is 1), the scan begins at the first argument. <Level> has the same meaning as for 'arg' above.

nargs [<level>]

'Nargs' prints on its first standard output the number of arguments passed to the command file <level> lexic levels above the current level. <Level> has the same meaning as for 'arg' above.

Shell Tracing. The 'shtrace' command is useful for tracing the operation of the shell. Although primarily intended for debugging the command interpreter itself, it also finds use in monitoring and debugging shell files. To turn the trace on, enter

```
shtrace on
```

To turn the trace off, enter

```
shtrace
```

Many other options are available. Consult the Software Tools Subsystem Reference Manual for details.

Shell Variable Utilities. The following commands (in addition to 'declare', 'set', and 'forget' discussed earlier) have been found useful in dealing with shell variables. Further information can, as usual, be found in the Software Tools Subsystem Reference Manual.

vars

'Vars' lists the names (and optionally the values) of the user's shell variables. Various options control the listing format, the number of lexic levels scanned, and whether or not shell control variables are listed. The most common form is probably

```
vars -alv
```

which lists all variables at all lexic levels along with their values.

save

'Save' forces the user's lexic level zero shell variables to be written to the user's private shell variable save file. This operation normally takes place only when the shell exits to Primos after a 'stop' or control-c, but it is frequently convenient (and essential from some command files) to save variables without leaving the shell.

Conclusion

This concludes the Application Notes section of the guide. Hopefully it has presented some ideas that will make the use of the command interpreter more productive and enjoyable.

Messages from the Shell

Listed here are messages with obscure meanings that are produced by the Shell; several indicate dire internal problems that should not occur during normal operation. In the interest of saving paper, self-explanatory messages are not included.

<command>: not found
The list of elements in the search rule was exhausted, but the command had not been located.

<command>: too many ci files
The nesting depth of command files has been exceeded. This is usually caused by an infinitely recursive call on a command file. The maximum nesting depth (currently 10) is a compile time option of the shell and may be increased at the expense of additional table space.

continue?
This message occurs after each network when the "single_step" shell trace option is set. A line beginning with anything other than an upper or lower case letter "n" will cause the shell to execute the next network. A response beginning with "n" * will cause the shell to return to command level.

illegal destination node spec
The destination node specifier must be a defined label or a number between 1 and the number of nodes in the network.

illegal port number
A port number must be a number between 1 and the maximum number of standard ports defined (currently 3).

missing command name
Although an empty net is allowable, redirectors must not be specified without a | command name.

| missing pathname in redirector
| A greater-than sign was encountered without a pathname on either side.

net is not serially executable
Because multiple processes per user are not supported, each node of a net must be executed serially. Therefore, nets which have pipe connections that form a complete cycle cannot be executed.

overflow (save_state): <level>
The nesting depth of command files has been exceeded. This is usually caused by an infinitely recursive call on a command file. The maximum nesting depth (currently 10) is a compile time option of the shell and may be increased at the expense of additional table space.

pipe destination not found
The destination node of a pipe is not in the range of the current net.

state save stack overflow
The nesting depth of command files has been exceeded. This is usually caused by an infinitely recursive call on a command file. The maximum nesting depth (currently 10) is a compile time option of the shell and may be increased at the expense of additional table space.

unbalanced iteration groups
Because of the semantics of iteration, each iteration group in the same net must contain the same number of arguments.

unexpected EOF on variable save file
End of file has been encountered on the shell variable save file when a value has been expected. The shell variables have been corrupted. To recover what might be | left, exit the Subsystem with a <break> or control-P and consult your system | administrator.

whitespace required around pipe connector
A pipe connector and its associated port numbers and destination label must be surrounded by spaces.

whitespace required around i/o redirector
An i/o redirector and its associated i/o redirector must be surrounded by spaces.

User's Guide for the Ratfor Preprocessor

Second Edition

Daniel H. Forsyth, Jr.
T. Allen Akin
Perry B. Flinn

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

April, 1980

TABLE OF CONTENTS

Ratfor Language Guide

Ratfor Language Reference

Ratfor Programming Under the Subsystem

Appendixes

Preface

Ratfor ("Rational Fortran") is an extension of Fortran-66 that serves as the basis for the Software Tools Subsystem. It provides a number of enhancements to Fortran that facilitate structured design and programming, as well as enhance program readability and ease the burden of program coding.

This guide is intended to explain and demonstrate the use of Ratfor as a programming language within the Software Tools Subsystem. In addition, applications notes are provided to help users build on the experience of others.

Ratfor Language Guide

What is Ratfor?

The Ratfor ("Rational Fortran") language was introduced in the book Software Tools by Brian W. Kernighan and P. J. Plauger (Addison-Wesley, 1976). There, the authors use it as the medium for the development of programs that may be used as cooperating tools. Ratfor offers many extensions to Fortran that encourage and facilitate structured design and programming, enhance program readability and ease the burden of coding. Through some very simple mechanisms, Ratfor helps the programmer to isolate machine and implementation dependent sections of his code.

Among the many programs developed in Software Tools is a Ratfor preprocessor -- a program for converting Ratfor into equivalent ANSI-66 Fortran. 'Rp', the preprocessor described in this guide, is an original version based on the program presented in Software Tools.

Differences Between Ratfor and Fortran

As we mentioned, Ratfor and Fortran are very similar. Perhaps the best introduction to their differences is given by Kernighan and Plauger in Software Tools:

> "But bare Fortran is a poor language indeed for programming or describing programs. . . . Ratfor provides modern control flow statements like those in PL/I, Cobol, Algol, or Pascal, so we can do structured programming properly. It is easy to read, write and understand, and readily translates into Fortran. . . . Except for a handful of new statements like **if - else**, **while**, and **repeat - until**, Ratfor is Fortran."

Source Program Format

Case Sensitivity. In most cases, the format of Ratfor programs is much less restricted than that of Fortran programs. Since the Software Tools Subsystem encourages use of terminals with multi-case capabilities, 'rp' accepts input in both upper and lower case. 'Rp' is case sensitive. Keywords, such as **if** and **select**, must appear in lower case. Case is significant in identifiers; they may appear in either case, but upper case letters are not equivalent to lower case letters. For example, the words "blank" and "Blank" do not represent the same identifier. For circumstances in which case sensitivity is a bother, 'rp' accepts a command line option ("-m") that instructs it to ignore the case of all identifiers and keywords. See the applications notes or the 'help' command for more details.

Blank Sensitivity. Unlike most Fortran compilers, 'rp' is very sensitive to blanks. 'Rp' requires that all words be separated by at least one blank or special character. Words containing imbedded blanks are not allowed. The best rule of thumb is to remember that if it is incomprehensible to you, it is probably incomprehensible to 'rp.' (Remember, we humans normally leave blank spaces between words and tend not to place blanks inside words. Such things make text difficult to understand.)

As a bad example, the following Ratfor code is incorrect and will not be interpreted properly:

```
subroutineexample(a,b,c)
integera,b,c

repeatx=x+1
   until(x>1)
```

A few well placed blanks will have to be added before 'rp' can understand it:

```
subroutine example(a,b,c)
integer a,b,c

repeat x=x+1
   until(x>1)
```

You should note that extra spaces are allowed (and encouraged) everywhere except inside words and literals. Extra spaces make a program much more readable by humans:

```
subroutine example (a, b, c)
integer a, b, c

repeat x = x + 1
   until (x > 1)
```

Card Columns. As should be expected of any interactive software system, 'rp' is completely insensitive to "card" columns; statements may begin and end at any position in a line. Lines may be of any length, but identifiers and quoted strings may not be longer than 100 characters. 'Rp' will output all statements beginning in column 7, and automatically generate continuation lines for statements extending past column 72. All of the following are valid Ratfor statements, although such erratic indentation is definitely frowned upon.

```
integer i, j
   i = 1
      j = 2
stop
   end
```

Multiple Statements per Line. 'Rp' also allows multiple statements per line, although indiscriminate use of this feature is not encouraged. Just place a semicolon between statements and 'rp' will generate two Fortran statements from them. You will find

```
integer i
real a
logical l
```

to be completely equivalent to

```
integer i; real a; logical l
```

Statement Labels and Continuation. You may wonder what happens to statement labels and continuation lines, since 'rp' pays no attention to card columns. It turns out that statement labels and continuation lines are not often necessary. While 'rp' minimizes the need for statement labels (except on **format** statements) and is quite intelligent about continuation lines, there are conventions to take care of those situations where a label is required or the need for a continuation line is not obvious to 'rp.'

A statement may be labeled simply by placing the statement number, starting in any column, before the statement. Any executable statement, including the Ratfor control statements, may be labeled, and 'rp' will place the label correctly in the Fortran output. It is wise to refrain from using five-digit statement numbers; 'rp' uses these statement labels to implement the Ratfor control statements, and consequently will complain if it encounters them in a source program. As examples of statement labels,

```
2     read (1, 10) a, b, c
            10  format (3e10.0)
      write (1, 20) a, b, c; 20 format (3f20.5)
         go to 2
```

all show statement numbers in use. You should note that with proper use of Ratfor and the Software Tools Subsystem support subroutines, statement labels are almost never required.

As for continuation lines, 'rp' is usually able to recognize when the current line needs to be continued. A line ending with a comma, unbalanced parentheses in a condition, or a missing statement (such as at the end of an **if**) are all situations in which 'rp' correctly anticipates a continuation line:

```
integer a, b, c, d,
   e, f, g

if (a == b & c == d & e == f &
    g == h & i == j & k == l)  call eql

if (a == b)

   c = -2
```

If an explicit continuation is required, such as in a long assignment statement, 'rp' can be made to continue a line by placing a trailing underscore ("_") at the end of the line. This underscore must be preceded by a space. You should note that the underscore is placed on the end of line to be continued, rather than on the continuation line as in Fortran. If

you are unsure whether Ratfor will correctly anticipate a continuation line, go ahead and place an underscore on the line to be continued -- 'rp' will ignore redundant continuation indicators.

Identifiers may not be split between lines; continuation is allowed only between tokens. If you have an extremely long string constant that requires continuation, you can take advantage of the fact that 'rp' always concatenates two adjacent string constants. Just close the first part of the literal with a quote, space, and underscore, and begin the second part on the next line with a quote. 'Rp' will ignore the line break (because of the trailing underscore) and concatenate the two literals.

The following are some examples of explicit line continuations:

```
i = i + j + k + l + m + n + o + p + q + r + _
   s + t + u + v

1 format ("for inputs of ", i5, " and ", i5/ _
      "the expected output should be ", i5)

string heading   _
   "-------------------------------------------"
   "-------------------------------------------"  _
```

Comments. Comments, an important part of any program, can be entered on any line; a comment begins with a sharp sign ("#") and continues until the end of the line. In addition, blank lines and lines containing only comments may be freely placed in the source program. Here are some appropriate and (correct but) inappropriate uses of Ratfor comments:

```
if (i > 48)
   # do this only if i is greater than 48
   j = j + 1

data array / 1,     # element 1
             2,     # element 2
             3,     # element 3
             4/     # element 4

integer cnt,          # counter for controlling the
                      #    outer loop
        total_errs,   # total number of errors
                      #    encountered
        last_pass     # flag for determining the
                      #    last pass; init = 0
```

Identifiers

A major difference between Ratfor and Fortran is Ratfor's acceptance of arbitrarily long identifiers. A Ratfor identifier may be up to 100 characters long, beginning with a letter, and may contain letters, digits, dollar signs, and underscores. However, it may not be a Ratfor or Fortran keyword, such as **if**, **else**, **integer**, **real**, or **logical**. Underscores are allowed in identifiers only for the sake of readability, and are always ignored. Thus, "these_tasks" and "the_set_asks" are equivalent Ratfor identifiers.

'Rp' guarantees that an identifier longer than six characters will be transformed into a unique Fortran identifier. Normally, the process of transforming Ratfor identifiers into Fortran identifiers is transparent; you need not be concerned with how this transformation is accomplished. The one notable exception is the effect on external symbols (i.e. subroutine and function names, common block names). When the declaration of a subprogram and its invocation are preprocessed together, in the same run, no problems will occur. However, if the subprogram and its invocation are preprocessed separately, there is no guarantee that a given Ratfor name will be transformed into the same Fortran name in the two different runs. This situation can be avoided in either of three ways: (1) use the **linkage** statement decribed in the next section, (2) use six-character or shorter identifiers for subprogram names, or (3) preprocess subprograms and their invocations in the same run.

Just for pedagogical reasons, here are a few correct and incorrect Ratfor identifiers:

Correct

```
long_name_1
long_name_2
prwf$$
I_am_a_very_long_Ratfor_name_that_is_perfectly_correct
a_a     # You should note that 'a_a', 'a__a', and 'aa'
a__a    # are all absolutely identical in Ratfor --
aa      # underscores are always ignored in identifiers,
AA      # but 'AA' is very different.
```

Incorrect

```
123_part    # starts with a digit
_part1      # starts with an underscore
part 2      # contains a blank
a*b         # contains an asterisk
```

* The following paragraph contains a description of exactly how Ratfor identifiers are transformed into Fortran identifiers. You need not know how this transformation is accomplished to make full use of Ratfor; hence, you probably need not read the next paragraph.

If a Ratfor identifier is longer than six characters or contains an upper case letter, it is made unique by the following procedure:

(1) The identifier is padded with 'a's or truncated to five characters. Remaining characters are mapped to lower case.
(2) The first character is retained to preserve implicit typing.
(3) The sixth character is changed to a "uniquing character" (normally a zero).
(4) If necessary, the second, third, fourth, and fifth characters are altered to make sure there is no conflict with a previously used identifier.

'Rp' also examines six-character identifiers containing the uniquing character in the sixth position, to ensure that no conflicts arise.

Integer Constants

Since it is sometimes necessary to use other than decimal integer constants in a program, 'rp' accepts integers in bases 2 through 16. Integers consisting of only digits are, of course, considered decimal integers. Other bases can be indicated with the following notation:

<base>r<number>

where <base> is the base of the number (in decimal) and <number> is number in the desired base (the letters 'a' through 'f' are used to represent the digits '10' through '15' in bases greater than 10). For example, here are some Ratfor integer constants and the decimal values they represent:

Number	Decimal Value
8r77	63
16rff	255
-2r11	-3
7r13	10

Some care must be exercised when using this form of constant to generate bit-masks with the high-order bit set. For example, to set the high-order bit in a 16-bit word, one might be tempted to use one of the constants

16r8000 or 8r100000

Either of these would cause incorrect results, because the value that they represent, in decimal, is 65536. This number, when encountered by Prime Fortran, is converted to a 32-bit constant (with the high order bit in the second word set). This is probably not the desired result. The only solutions to this problem (which occurs when trying to represent a negative twos-complement number as a positive number) are (1) use the correct twos-complement representation (-32768 in this case), or (2) fall back to Prime Fortran's octal constants (e.g. :100000).

String Constants

Under the Software Tools Subsystem, character strings come in various flavors. Because various internal representations are used for character strings, Fortran Hollerith constants are not sufficient to easily provide all the different formats required.

All types of Ratfor string constants consist of a string body followed by a string format indicator. The body of a string constant consists of strings of characters bounded by pairs of quotes (either single or double quotes), possibly separated by blanks. All the character strings in the body (not including the bounding quotes) are concatenated to give the value of the string constant. For example, here are three string constant bodies that contain the same string:

```
"I am a string constant body"
"I" ' am ' "a" ' string ' "constant" ' body'
"I am a string "'constant body'
```

The string format indicator is an optional letter that determines the internal format to be used when storing the string. Currently there are five different string representations available:

omitted — Fortran Hollerith string. When the string format indicator is omitted, a standard Fortran Hollerith constant is generated. Characters are left-justified, packed in words (two characters per word on the Prime), and unused positions on the right are filled with blanks.

c — Single character constant. The 'c' string format indicator causes a single character constant to be generated. The character is right-justified and zero-filled on the left in a word. Only one character is allowed in the body of the constant. Since it is easy to manipulate and compare characters in this format, it is the preferred format for all single characters in the Software Tools Subsystem.

p — Packed (Hollerith) period-terminated string. The 'p' format indicator causes the generation of a Fortran Hollerith constant containing the characters in the string body followed by a period. In addition, all periods in the string body are preceded by an escape character ("@"). The advantage of a "p" format string over a Fortran Hollerith string is that the length of the "p" format string can be determined at run time.

v — PL/I character varying string. For compatibility with Prime's PL/I and because this data format is required by some system calls, the "v" format indicator will generate Fortran declarations to create a PL/I character varying string. The first word of the constant contains the number of characters; subsequent words contain the characters of the string body packed two per word. "V" format string constants may only be used in executable statements.

s — EOS-terminated unpacked string. The "s" string format indicator causes 'rp' to generated declarations necessary to construct an array of characters containing each character in the string body in a separate word, right-justified and zero-filled (each character is in the same format as is generated by the "c" format indicator). Following the characters is a word containing a value different from any character value that marks the end of the string. This ending value is defined as the symbolic constant EOS. EOS-terminated strings are the preferred format for multi-character strings in the Subsystem, and are used by most Subsystem routines dealing with character strings. "S" format string constants may only be used in executable statements.

Here are some examples of strings and the result that would be generated for Prime Fortran. On a machine with a different character set or word length, different code might be generated.

String Constant	Resulting Code
'v'c	the integer constant 246
"=doc="s	an integer array of length 6 containing 189, 228, 239, 227, 189, -2
"a>b c>d"v	an integer array containing 7, "a>", "b ", "c>", "d "
".main."p	the constant 9h@.main@..
"Hollerith"	the constant 9hHollerith

Logical and Relational Operators

Ratfor allows the use of graphic characters to represent logical and relational operators instead of the Fortran ".EQ." and such. While use of these graphic characters is encouraged, it is not incorrect to use the Fortran operators. The following table shows the equivalent syntaxes:

Ratfor	Fortran	Function
>	.GT.	Greater than
>=	.GE.	Greater or equal
<	.LT.	Less than
<=	.LE.	Less or equal
==	.EQ.	Equal to
~=	.NE.	Not equal to
~	.NOT.	Logical negation
&	.AND.	Logical conjunction
|	.OR.	Logical disjunction

Note than the digraphs shown in the table must appear in the Ratfor program with no imbedded spaces.

For example, the two following **if** statements are equivalent in every way:

```
if (a .eq. b .or. .not. (c .ne. d .and. f .ge. g))

if (a == b | ~ (c ~= d & f >= g))
```

In addition to graphics representing Fortran operators, two additional operators are available in any logical expression parsed by 'rp' (i.e. anywhere but assignment statements). These operators, '&&' ("and if") and '||' ("or if") perform the same action as the logical operators '&' and '|', except that they guarantee that the expression is evaluated from left to right, and that evaluation is terminated when the truth value of the expression is known. They may appear within the scope of the '~' operator, but they may not grouped within the scope of '&' and '|'.

These operators find use in situations in which it may be illegal or undesirable to evaluate the right-hand side of a logical expression based on the truth value of the left-hand side. For example, in

```
while (i > 0 && str (i) == ' 'c)
   i = i - 1
```

it is necessary that the subscript be checked before it is used. The order of evaluation of Fortran logical expressions is not specified, so in this example, it would be technically illegal to use '&' in place of '&&'. If the value of 'i' were less than 1, the illegal subscript reference might be made regardless of the range check of the subscript. The Ratfor short-circuited logical operators prevent this problem by insuring that "i > 0" is evaluated first, and if it is false, evaluation of the expression terminates, since its value (false) is known.

Assignment Operators

Ratfor provides shorthand forms for the Fortran idioms of the form

```
<variable> = <variable> <operator> <expression>
```

In Ratfor, this assignment can be simplified to the form

```
<variable> <assignment operator> <expression>
```

with the use of assignment operators. The following assignment operators are available:

Operator	Use	Result
+=	<v> += <e>	<v> = <v> + (<e>)
-=	<v> -= <e>	<v> = <v> - (<e>)
*=	<v> *= <e>	<v> = <v> * (<e>)
/=	<v> /= <e>	<v> = <v> / (<e>)
%=	<v> %= <e>	<v> = mod (<v>, <e>)
&=	<v> &= <e>	<v> = and (<v>, <e>)
|=	<v> |= <e>	<v> = or (<v>, <e>)
^=	<v> ^= <e>	<v> = xor (<v>, <e>)

The Ratfor assignment operators may be used wherever a Fortran assignment statement is allowable. Regrettably, the assignment operators provide only a shorthand for the programmer; they do not affect the efficiency of the object code.

The assignment operators are especially useful with subscripted variables; since a complex subscript expression need appear only once, there is no possiblity of mistyping or forgeting to change one. Here are some examples of the use of assignment operators

```
i += 1
fact *= i + 10
subs (2 * i - 2, 5 * j - 23) -= 1
int %= 10 ** j
mask &= 8r12
```

For comparison, here are the same assignments without the use of assignment operators:

```
i = i + 1
fact = fact * (i + 10)
subs (2*i-2, 5*j-23) = subs (2*i-2, 5*j-23) - 1
int = mod (int, (10 ** j))
mask = and (mask, 8r12)
```

Incompatibilities

Even with the great similarities between Fortran and Ratfor, an arbitrary Fortran program is not necessarily a correct Ratfor program. Several areas of incompatibilities exist:

- In Ratfor, blanks are significant -- at least one space must separate adjacent identifiers.
- The Ratfor **do** statement, as we shall soon see, does not contain the statement number following the "do". Instead, its range extends over the next (possibly compound) statement.
- Two word Fortran key phrases such as **double precision, block data,** and **stack header** must be presented as a single Ratfor identifier (e.g. "blockdata" or "block_data").
- Fortran statement functions must be preceded by the Ratfor keyword **stmtfunc.** To assure that they will appear in the correct order in the Fortran, they should immediately precede the **end** statement for the program unit.
- Hollerith literals (i.e. 5HABCDE) are not allowed anywhere in a Ratfor program. Instead, 'rp' expects all Hollerith literals to be enclosed in single or double quotes (i.e. "ABCDE" or 'ABCDE'). 'Rp' will convert the quoted string into a proper Fortran Hollerith string.
- 'Rp' does not allow Fortran comments. In Ratfor, comments are introduced by a sharp sign ("#") appearing anywhere on a line, and continue to the end of the line.
- 'Rp' does not accept the Fortran continuation convention. Continuation is implicit for any line ending with a comma, or any conditional statement containing unbalanced parentheses. Continuation between arbitrary words may be indicated by placing an underscore, preceded by at least one space, at the end of the line to be continued.
- 'Rp' does not ignore text beyond column 72.
- Fortran and Ratfor keywords may not be used as identifiers in a Ratfor program. Their use will result in unreasonable behavior.

Ratfor Text Substitution Statements

'Rp' provides several text substitution facilities to improve the readability and maintainability of Ratfor programs. You can use these facilities to great advantage to hide tedious implementation details and to assist in writing transportable code.

Define

The Ratfor **define** statement bears a vague similarity to the non-standard Fortran **parameter** declaration, but is much more flexible. In Ratfor, any legal identifier may be defined as almost any string of characters. Thereafter, 'rp' will replace all occurrences of the defined identifier with the definition string. In addition, identifiers may be defined with a formal parameter list. Then, during replacement, actual paramters specified in the invocation are substituted for occurrences of the formal parameters in the replacement text.

Defines find their principle use in helping to clarify the meaning of "magic numbers" that appear frequently. For example,

```
while (getlin (line, -10) ~= -1)
   call putlin (line, -11)
```

is syntactically correct, and even does something useful. But what? The use of **define** to hide the magic numbers not only allows them to be changed easily and uniformly, but also gives the program reader a helpful hint as to what is going on. If we rewrite the example, replacing the numbers by defined identifiers, not only are the numbers easier to change uniformly at some later date, but also, the reader is given a little bit of a hint as to what is intended.

```
define (EOF, -1)
define (STANDARD_INPUT, -10)
define (STANDARD_OUTPUT, -11)

while (getlin (line, STANDARD_INPUT) ~= EOF)
   call putlin (line, STANDARD_OUTPUT)
```

The last example also shows the syntax for definitions without formal parameters.

Often there are situations in which the replacement text must vary slightly from place to place. For example, let's take the last situation in which the programmer must supply "STANDARD_INPUT" and "STANDARD_OUTPUT" in calls to the line input and output routines. Since this occurs in a large majority of cases, it would be more convenient to have procedures named, say "getl" and "putl" that take only one parameter and assume "STANDARD_INPUT" or "STANDARD_OUTPUT". We could, of course, write two new procedures to fill this need, but that would add more code and more procedure calls. Two **define** statements will serve the purpose very well:

```
define (STANDARD_INPUT, -10)
define (STANDARD_OUTPUT, -11)
define (getl (ln), getlin (ln, STANDARD_INPUT))
define (putl (ln), putlin (ln, STANDARD_OUTPUT))

while (getl (line) ~= EOF)
   call putl (line)
```

In this case, when the string "getl (line)" is replaced, all occurrences of "ln" (the formal parameter) will be replaced by "line" (the actual parameter). This example will give exactly the same results as the first, but with a little less typing when "getl" and "putl" are called often.

The full syntax for a **define** statement follows:

```
define (<identifier> [(<formal params>)], <replacement>)
```

When such a **define** statement is encountered, <replacement> is recorded as the value of <identifier>. At any later time, if <identifier> is encountered in the text, it is replaced by the text of <replacement>. If the original **define** contained a formal parameter list, the list of actual parameters following <identifier> is collected, and the actual parameters are substituted for the corresponding formal parameters in <replacement> before the replacement is made.

There is a file of "standard" definitions used by all Subsystem programs called "=incl=/swt_def.r.i". The **define** statements in this file are automatically inserted before each source file (unless 'rp' is told otherwise by the "-f" command line option). For

information on the exact contents of this file, see Appendix D.

There are also a few other facts that are helpful when using **define**:

- The <replacement> may be any string of characters not containing unbalanced parentheses or unpaired quotes

- <Formal parameters> must be identifiers.

- <Actual parameters> may be any string of characters not containing unbalanced parentheses, unpaired quotes, or commas not surrounded by quotes or parentheses.

- Formal parameter replacement in <replacement> occurs even inside of quoted strings. For example,

```
define (assert (cond), {
   if (~(cond))
      call error ("assertion cond not valid"p)}
assert (i < j)
```

would generate

```
{
   if (~(i < j))
      call error ("assertion i < j not valid"p)}
```

- During replacement of an identifier defined without a formal parameter list, an actual parameter list will never be accessed. For example,

```
define (ARRAYNAME, table1)
ARRAYNAME (i, j) = 0
```

would generate

```
table1 (i, j) = 0
```

- The number of actual and formal parameters need not match. Excess formal parameters will be replaced by null strings; excess actual parameters will be ignored.

- A **define** statement affects only those identifiers following it. In the following example, STDIN would **not** be replaced by -11, unless a **define** statement for STDIN had occurred previously:

```
l = getlin (buf, STDIN)
define (STDIN, -11)
```

- A **define** statement applies to all lines following it in the input to 'rp', regardless of subroutine, procedure, and source file boundaries.

- After replacement, the substituted text itself is examined for further defined identifiers. This allows such definition sequences as

```
define (DELCOMMAND, LETD)
define (LETD, 100)
```

to result in the desired replacement of "100" for "DELCOMMAND". Actual parameters are not reexamined until the entire replacement string is reexamined.

- Identifiers may be redefined without error. The most recent definition supersedes all previous ones. Storage space used by superseded definitions is reclaimed.

Here are a few more examples of how defines can be used:

Before Defines Have Been Processed:

```
define (NO, 0)
define (YES, 1)
define (STDIN, -11)
define (EOF, -2)
define (RESET (flag), flag = NO)
define (CHECK_FOR_ERROR (flag, msg),
   if (flag == YES)
      call error (msg)
   )
define (FATAL_ERROR_MESSAGE,
   "Fatal error -- run terminated"p)
define (PROCESS_LINE,
   count = count + 1
   call check_syntax (buf, count, error_flag)
   )

while (getlin (buf, STDIN) ~= EOF) {
   RESET (error_flag)
   PROCESS_LINE
   CHECK_FOR_ERROR (error_flag, FATAL_ERROR_MESSAGE)
   }
```

After Defines Have Been Processed:

```
while (getlin (buf, -11) ~= -2) {
   error_flag = 0
   count = count + 1
   call check_syntax (buf, count, error_flag)
   if (error_flag == 1)
      call error ("Fatal error -- run terminated"p)
   }
```

Undefine

The Ratfor **undefine** statement allows termination of the range of a **define** statement. The identifier named in the **undefine** statement is removed from the define table if it is present; otherwise, no action is taken. Storage used by the definition is reclaimed. For example, the statements

```
define (xxx, a = 1)
xxx
undefine (xxx)
xxx
```

would produce the following code:

```
a = 1
xxx
```

Include

The Ratfor **include** statement allows you to include arbitrary files in a Ratfor program (much like the COBOL **copy** verb). The syntax of an **include** statement is as follows:

```
include "<file name>"
```

If the file name is six or fewer characters in length and contains only alphanumeric characters, the quotes may be omitted. For the sake of uniformity, we suggest that the quotes always be used.

When 'rp' encounters an **include** statement, it begins taking input from the file specified by <file name>. When the end of the included file is encountered, 'rp' resumes reading the preempted file. Files named in **include** statements may themselves contain **include** statements; this nesting may continue to an arbitrary depth (which, by the way, is arbitrarily limited to five).

For an example of **include** at work, assume the existence of the following files:

```
f1:
     include "f2"
     i = 1
     include "f3"

f2:
     include "f4"
     m = 1

f3:
     j = 1

f4:
     k = 1
```

If "f1" were the original file, the following text is what would actually be processed:

```
k = 1
m = 1
i = 1
j = 1
```

Ratfor Declarations

There are several declarations available in Ratfor in addition to those usually supported in Fortran. They provide a way of conveniently declaring data structures not available in Fortran, assist in supporting separate compilation, allow declaration of local variables within compound statements, and allow the declaration of internal procedures. Declarations in Ratfor may be intermixed with executable statements.

String

The **string** statement is provided as a shorthand way of creating and naming EOS-terminated strings. The structure and use of an EOS-terminated string is described in the section on Subsystem Conventions. Here it is sufficient to say that such a string is an integer array containing one character per element, right justified and zero filled, and ending with a special value (EOS) designating the "end of string." Since Fortran has no construct for specifying such a data structure, it must either be declared manually, as a Ratfor string constant, or by the Ratfor **string** statement.

The **string** statement is a declaration that creates a named string in an integer array using a Fortran **data** statement. The syntax of the **string** statement is as follows:

```
string <name> <quoted string>
```

where <name> is the Ratfor identifier to be used in naming the string and <quoted string> specifies the string's contents. As you might expect, either single or double quotes may be used to delimit <quoted string>. In either case, only the characters between the quotes become part of the string; the quotes themselves are not included.

String statements are quite often used for setting up constant strings such as file names or key words. For instance,

```
string file_name "//mydir/myfile"
string change_command "change"
string delete_command "delete"
```

define such character arrays.

Stringtable

The **stringtable** statement creates a rather specialized data structure -- a marginally indexed array of variable length strings. This data structure provides the same ease of access as an array, but it can contain entries of varying sizes. A **stringtable** declaration defines two data items: a marginal index and a table body. The marginal index is an integer array containing indices into the table body. The first element of the marginal index is the number of entries following in the marginal index. Subsequent elements of the marginal index are pointers to the beginning of items in the table body. Since the beginning of the table body is always the beginning of an item, the second entry of the marginal index is always 1.

The syntax of a **stringtable** declaration is as follows:

```
string_table <marginal index>, <table body>,
   [ / ] <item> { / <item> }
```

<Marginal index> and <table body> are identifiers that will be declared as the marginal index and table body, respectively. <Item> is a comma-separated list of single-character constants (with a "c" string format indicator), integers, or EOS-terminated character strings (with no string format indicator -- a little inconsistency here). The values contained in an <item> are stored contiguously in <table body> with no separator values (save for an EOS at the end of each EOS-terminated string). An entry is made in the marginal index containing the position of the first word of each <item>.

For example, assume that you have a program in which you wish to obtain one of three integer values based on an input string. You want to allow an arbitrary number of synonyms in the input (like "add", "insert", etc.).

```
string_table cmdpos, cmdtext,
   / ADD,      "add"
   / ADD,      "insert"
   / CHANGE,   "change"
   / CHANGE,   "update"
   / DELETE,   "delete"
   / DELETE,   "remove"
```

This declaration creates a structure something like the following:

```
          cmdpos                    cmdtext

1:          6
2:          1          1:  ADD,     'a'c, 'd'c, 'd'c, EOS
3:          6          6:  ADD,     'i'c, 'n'c, 's'c, 'e'c,
                                     'r'c, 't'c, EOS
4:         14         14:  CHANGE,  'c'c, 'h'c, 'a'c, 'n'c,
                                     'g'c, 'e'c, EOS
5:         22         22:  CHANGE,  'u'c, 'p'c, 'd'c, 'a'c,
                                     't'c, 'e'c, EOS
6:         29         29:  DELETE,  'd'c, 'e'c, 'l'c, 'e'c,
                                     't'c, 'e'c, EOS
7:         36         36:  DELETE,  'r'c, 'e'c, 'm'c, 'o'c,
                                     'v'c, 'e'c, EOS
```

There are several routines in the Subsystem library that can be used to search for strings in one of these structures. You can find details on the use of these procedures in the reference manual/'help' entries for 'strlsr' and 'strbsr'.

Linkage

The sole purpose of the **linkage** declaration is to circumvent problems with transforming Ratfor identifiers to Fortran identifiers when compiling program modules separately. To relax the restriction that externally visible names (subroutine, function, and common block names) must contain no more than six characters, each separately compiled module must begin with an identical **linkage** declaration containing the names of all external symbols -- subroutine names, function names, and common block names (the identifiers inside the slashes -- not the variable names). Except for text substitution statements, the **linkage** declaration must be the first statement in each module. The order of names in the statement is significant -- as a general rule, you should **include** the same file containing the **linkage** declaration in each module.

Linkage looks very much like a Fortran type declaration:

```
linkage identifier1, identifier2, identifier3
```

Each of the identifiers is an external name (i.e. subroutine, function, or common block name). If this statement appears in each source module, with the identifiers in exactly the same order, it is guaranteed that in all cases, each of these identifiers will be transformed into the same unique Fortran identifier. For Subsystem-specific information on the mechanics of separate compilation, you can see the section in the applications notes devoted to this topic.

Local

With the **local** declaration, you can indicate that certain variables are "local" to a particular compound statement (or block) just as in Algol. **Local** declarations are most often used inside internal procedures (which are described later), but they can appear in any compound statement.

The syntax for declaring local variables was devised primarily with ease-of-implemention in mind. When 'rp' becomes more aware of Fortran syntax, declarations inside of compound statements will be implicitly local to the block. Now, however, the type declarations for local variables must be preceded by a **local** declaration containing the names of all variables that are to be local to the block:

```
local i, j, a

integer i, j
real a
```

The **local** statement must precede the first appearance of a variable inside the block.

Scope rules similar to those of most block-structured languages apply to nested compound statements: A local variable is visible to all blocks nested within the block in which it is declared. Declaration of a local variable obscures a variable by the same name declared in an outer block.

There are several cautions you must observe when using local variables. 'Rp' is currently not well-versed in the semantics of Fortran declarations and therefore cannot diagnose the incorrect use of **local** declarations. Misuse can then result in semantic errors in the Fortran output that are often not caught by the Fortran compiler. If the declaration of a variable within a block appears before the variable is named in a **local** declaration, 'rp' will not detect the error, and an "undeclared variable" error will be generated in the Fortran. External names (i.e. function, subroutine, and common block names) must never be named in a **local** declaration, unless you want to declare a local variable of the same name. Finally, the formal parameters of internal procedures should never appear in a **local** declaration in the body of the procedure, again, unless you want to declare a local variable of the same name.

Here is an example showing the scopes of variables appearing in a **local** declaration:

```
### level 0
subroutine test

integer i, j, k

{  ### level 1
   local i, m; integer i, m
   # accessible: level 0 j, k; level 1 i, m
   {  ### level 2
      local m, k; real m, k
      # accessible: level 0 j; level 1 i; level 2 m, k
      }
   }

end
```

Ratfor Control Statements

As was said by Kernighan and Plauger in Software Tools, except for the control structures, "Ratfor is Fortran." The additional control structures just serve to give Fortran the capabilities that already exist in Algol, Pascal, and PL/I.

Compound Statements

Ratfor allows the specification of a compound statement by surrounding a group of Ratfor statements with braces ("{}"), just like **begin - end** in Algol or Pascal, or **do - end** in PL/I. A compound statement may appear anywhere a single statement may appear, and is considered to be equivalent to a single statement when used within the scope of a Ratfor control statement.

There is normally no need for a compound statement to appear by itself -- compound statements usually appear in the context of a control structure -- but for completeness, here is an example of a compound statement.

```
{     # end of line -- set to beginning of next line
   line = line + 1
   col = 1
   end_of_line = YES
   }
```

If - Else

The Ratfor **if** statement is much more flexible than its Fortran counterpart. In addition to allowing a compound statement as an alternative, the Ratfor **if** includes an optional **else** statement to allow the specificatin of an alternative statement. Here is the complete syntax of the Ratfor **if** statement:

```
if (<condition>) <statement1>
[else <statement2>]
```

<Condition> is an ordinary Fortran logical expression. If <condition> is true, <statement1> will be executed. If <condition> is false and the **else** alternative is specified, <statement2> will be executed. Otherwise, if <condition> is false and the **else** alternative has not been specified, no action occurs.

Both <statement1> and <statement2> may be compound statements or may be further **if** statements. In the case of nested **if** statements where one or more **else** alternatives are not specified, each **else** is paired with the most recently occuring **if** that has not already been paired with an **else**.

Although deep nesting of **if** statements hinders understanding, one situation often occurs when it is necessary to select one and only one of a set of alternatives based on several conditions. This can be nicely represented with a chain of **if - else if - else if . . . else** statements. For example,

```
if (color == RED)
   call process_red
else if (color == BLUE | color == GREEN)
   call process_blue_green
else if (color == YELLOW)
   call process_yellow
else
   call color_error
```

* could be used to select a routine for processing based on color.

While

The Ratfor **while** statement allows the repetition of a statement (or compound statement) as long as a specified condition is met. The Ratfor **while** loop is a "test at the top" loop exactly like the Pascal **while** and the PL/I **do while**. The **while** statement has the following syntax:

```
while (<condition>)
   <statement>
```

If <condition> is false, control passes beyond the loop to the next statement in the program; if <condition> is true, <statement> is executed and <condition> is retested. As should be

expected, if <condition> is false when the **while** is first entered, <statement> will be executed zero times.

The **while** statement is very handy for controlling such things as skipping blanks in strings:

```
while (str (i) == BLANK)
   i = i + 1
```

And of course, <statement> may also be a compound statement:

```
while (getlin (buf, STDIN) ~= EOF) {
   call process (buf)
   call output (buf)
   }
```

Repeat

The Ratfor **repeat** loop allows repetitive execution of a statement until a specified condition is met. But, unlike the **while** loop, the test is made at the bottom of the loop, so that the controlled statement will be executed at least once. The **repeat** loop has syntax as follows:

```
repeat
   <statement>
   [until (<condition>)]
```

When the **repeat** statement is encountered, <statement> is executed. If <condition> is found to be false, <statement> is reexecuted and the <condition> is retested. Otherwise control passes to the statement following the **repeat** loop. If the **until** portion of the loop is omitted, the loop is considered an "infinite repeat" and must be terminated within <statement> (usually with a **break** or **return** statement). Pascal users should note that the scope of the Ratfor **repeat** is only a single <statement> (which of course may be compound).

Repeat loops, as opposed to **while** loops, are used when the controlled statement must be evaluated at least once. For example,

```
repeat
   call get_next_token (token)
   until (token ~= BLANK_TOKEN)
```

The "infinite repeat" is often useful when a loop must be terminated "in the middle:"

```
repeat {
   call get_next_input (inp)
   call check_syntax (inp, error_flag)
   if (error_flag == NO)
      return
   call syntax_error (inp)     # go back and get another
   }
```

Do

Ratfor provides access to the Fortran **do** statement. The Ratfor **do** statement is identical to the Fortran do except that it does not use a statement label to delimit its scope. The Ratfor **do** statement has the following syntax:

```
do <limits>
   <statement>
```

<Limits> is the normal Fortran notation for the limits of a **do**, such as "i = 1, 10" or "j = 5, 20, 2". The same restrictions apply to <limits> as apply to the limits in the Fortran **do**. <Statement> is any Ratfor statement (which may be compound).

The Ratfor **do** statement is just like the standard Fortran one-trip **do** loop -- <statement> will be executed at least once, regardless of the limits. Also, the value of the do control variable is not defined on exit from the loop.

The **do** loop can be used for array initialization and other such things that can never require "zero trips", since it produces slightly more efficient object code than the **for** statement (which we will get to next).

```
do i = 1, 10
   array (i) = 0
```

One slight irregularity in the Ratfor syntax occurs when <statement> appears on the same line as the **do**. Since 'rp' knows very little about Fortran, it assumes that the <limits> continue until a statement delimiter. This means that the <limits> must be followed by a semicolon if <statement> is to begin on the same line. This often occurs when a compound statement is to be used:

```
do i = 1, 10; {
   array_1 (i) = 0
   array_2 (i) = 0
   }
```

For

The Ratfor **for** statement is an all-purpose looping construct that takes the best features of both the **while** and **do** statements, while allowing more flexibility. The syntax of the **for** statement is as follows:

```
for (<initialize>; <condition>; <reinitialize>)
   <statement>
```

When the **for** is executed, the statement represented by <initialize> is executed. Then, if <condition> is true, <statement> is executed, followed by the statement represented by <reinitialize>. Then, <condition> is retested, etc. Any or all of <initialize>, <condition>, or <reinitialize> may be omitted; the semicolons, however, must remain. If <initialize> or <reinitialize> is omitted, no action is performed in their place. If <condition> is omitted, an "infinite loop" is assumed. (Both <initialize> or <reinitialize> may be compound statements).

As you can see, the **for** loop with <initialize> and <reinitialize> omitted is identical to the **while** loop. With the addition of <initialize> and <reinitialize>, a zero-trip **do** loop can be constructed. For instance,

```
for (i = 1; i <= 10; i += 1) {
   array_1 (i) = 0
   array_2 (i) = 0
   }
```

is identical to the last **do** example, but given a certain combination of limits, the **for** loop would execute <statement> zero times while the **do** loop would execute it once.

The **for** loop can do many things not possible with a **do** loop, since the **for** loop is not constrained to the ascending incrementation of an index. As an example, assume a list structure in which "list" contains the index of the first item in a list, and the first position in each list item contains the index of the next. The **for** statement could be used to serially examine the list:

```
for (ptr = list; ptr ~= NULL; ptr = array (ptr)){
   [ examine the item beginning at array (ptr + 1) ]
   }
```

Break

The **break** statement allows the early termination of a loop. The statement

```
break [<level>]
```

will cause the immediate termination of <level> loops, where <level>, if specified, is an integer in the range 1 to the depth of loop nesting at the point the **break** statement appears. Where <level> is omitted, only the innermost loop surrounding the **break** is terminated.

In the following example, the **break** statement will cause the termination of the inner **for** loop if a blank is encountered in 'str':

```
while (getlin (str, STDIN) ~= EOF) {
   for (i = 1; str (i) ~= EOS; i += 1)
      if (str (i) == BLANK)
         break

   str (i) = EOS        # output just the first word
   call putlin (str, STDOUT)
   call putch (NEWLINE, STDOUT)
   }
```

Replacing the **break** statement with "break 1" would have exactly the same effect. However, replacing it with "break 2" would cause termination of both the inner **for** and outer **while** loops. Unless this fragment is nested inside other loops, a value greater than 2 would be an error.

Next

The **next** statement is very similar to the **break** statement, except that a statement of the form

```
next [<level>]
```

causes termination of <level> - 1 nested loops (zero when <level> is omitted). Execution then resumes with the next iteration of the innermost active loop. <Level>, if specified, is again an integer in the range 1 to the depth of loop nesting that specifies which loop (from inside out) is to begin its next iteration.

In this example, the **next** statement will cause the processing to be skipped when an array element with the value "UNUSED" is encountered.

```
for (i = 1; i <= 10; i += 1)
   for (j = 1; j <= 10; j += 1) {
      if (array (i, j) == UNUSED)
         next

      # process array (i, j)

      }
```

When an array element with the value "UNUSED" is encountered, execution of the **next** statement causes the <reinitialize> portion of the innermost **for** statement, "j += 1", to be executed before the next iteration of the inner loop begins. You should note that when used with a **for** statement, **next** always skips to the <reinitialize> part of the appropriate **for** loop.

If the statement "next 2" had been used in place of "next", the inner **for** loop would have been terminated, and the "i += 1" of the outer **for** loop would have been executed in preparation for its next iteration.

Return

The Ratfor **return** statement normally behaves exactly like the Fortran **return** statement in all but one case. In this case, Ratfor allows a parenthesized expression to follow the keyword **return** inside a function subprogram. The value of this expression is then assigned to the function name as the value of the function before the return is executed. This is just another shorthand and does not provide any additional functionality.

Normally in a Fortran function subprogram, you place an assignment statement that assigns a value to the function name before the **return** statement, like this:

```
integer function calc (x, y, z)
...
calc = x + y - z
return
...
```

If you like, Ratfor allows you to express the same actions with one line less code:

```
integer function calc (x, y, z)
...
return (x + y - z)
...
```

This segment performs exactly the same function as the preceding segment.

Select

The Ratfor **select** statement allows the selection of a statement from several alternatives, based either on the value of an integer variable or on the outcome of several logical conditions. A **select** statement of the form

```
select
   when (<expression list 1>)
      <statement 1>
   when (<expression list 2>)
      <statement 2>
   ...
   when (<expression list n>)
      <statement n>
[ifany
   <statement n+1>]
[else
   <statement n+2>]
```

(where <expression list> is a comma-separated list of logical expressions) performs almost the same function as a chain of **if - else if . . . else** statements. Each <logical expression> is evaluated in turn, and when the first true is encountered, the corresponding statement is executed. If any **when** alternative is selected, the statement in the **ifany** part is executed. If none of the **when** alternatives are selected, the statement in the **else** part is executed.

Although its function is very similar to an **if - else** chain, a **select** statement has two distinct advantages. First, it allows the "ifany" alternative -- a way to implement a rather frequently encountered control structure without repeated code or procedure calls. Second, it places all the logical expressions in the same basic optimization block, so that even a dumb Fortran compiler can optimize register loads and stores.

For example, assume that we want to check to see if the variable 'color' contains a valid color, namely 'RED', 'YELLOW', 'BLUE', or 'GREEN'. If it does, we want to executed one of the three subroutines 'process_red', 'process_yellow', or 'process_blue_green' and set the flag 'color_valid' to YES. Otherwise, we want to set the 'color_valid' to NO. A **select** statement performs this trick nicely, with no repeated code:

```
select
   when (color == RED)
      call process_red
   when (color == YELLOW)
      call process_yellow
   when (color == BLUE, color == GREEN)
      call process_blue_green
ifany
   color_valid = YES
else
   color_valid = NO
```

The second variant of the select statement allows the selection of a statement based on the value of an integer (or character) expression. It has almost exactly the same syntax as the logical variant:

```
select (<integer expression>)
   when (<expression list 1>)
      <statement 1>
   when (<expression list 2>)
      <statement 2>
   ...
   when (<expression list n>)
      <statement n>
[ifany
   <statement n+1>]
[else
   <statement n+2>]
```

Using this variant, a statement is selected when one of its corresponding integer expressions has the same value as the <integer expression> following the 'select'. The **ifany** and **else** clause behave as they do in the logical variant. The most visible difference, though, is that the order of evaluation of the integer expressions is not specified. If two values in two expression lists are identical, it is difficult to say which of the statements will be executed; it can only be said that one and only one will be executed.

The integer variant offers one further advantage. If elements in the expression lists are integer or single-character constants, 'rp' will generate Fortran computed **goto** statements, rather than Fortran if statements, where possible. This code is usually considerably faster and more compact than the code generated by **if** statements.

The example given for the logical variant of **select** would really be much more easily done with the integer variant:

```
select (color)
   when (RED)
      call process_red
   when (YELLOW)
      call process_yellow
   when (BLUE, GREEN)
      call process_blue_green
ifany
   color_valid = YES
else
   color_valid = NO
```

As a final example of **select**, the following program fragment selects an insert, update, delete, or print routine based on the input codes "i", "u", "d" or "p":

```
while (getlin (buf, STDIN) ~= EOF)

   select (buf (1))
      when ('i'c, 'I'c)     # insert record
         call insert_record
      when ('u'c, 'U'c) {   # update record
         call delete_record
         call insert_record
         }
      when ('d'c, 'D'c)     # delete record
         call delete_record
      when ('p'c, 'P'c)     # print record
         ;
   ifany                    # always print after command
      call print_record
   else                     # illegal input
      call command_error
```

This example shows the use of both a compound statement within an alternative (the "update" action deletes the target record and then inserts a new version), and a null statement consisting of a single semicolon.

Procedure

Procedures are a convenient and useful structuring mechanism for programs, but in Fortran there often reasons for restricting the unbridled use of procedures. Among these reasons are (1) the run-time expense of procedure calls, and argument and common block addressing; (2) external name space congestion; and (3) difficulty in detecting errors in parameter and common-block correspondence. Ratfor attempts to address these problems by allowing declaration of procedures within Fortran subprograms that are inexpensive to call (an assignment and two **gotos**), are not externally visible, and allow access to global variables. In addition, when correctly declared, Ratfor internal procedures can call each other recursively without requiring recursive procedures in the host Fortran.

Currently, Ratfor internal procedures do not provide the same level of functionality as Fortran subroutines and functions: internal procedure parameters must be scalars and are passed by value, internal procedures cannot be used as functions (they cannot return values), and no automatic storage is available with recursive integer procedures. But even with these restrictions, internal procedures can significantly improve the readability and modularity of Ratfor code.

Internal procedures are declared with the Ratfor **procedure** statement. Internal procedures may be declared anywhere in a program, but a declaration must appear before any of its calls. Here is an example of a non-recursive procedure declaration:

```
# putchar --- put a character in the output string
   procedure putchar (ch) {

      character ch

      str (i) = ch
      i += 1
      }
```

This procedure has one parameter, "ch", which must appear in a type declaration inside the procedure.

Internal procedures always exit by falling through the end of the compound statement. A **return** statement in an internal procedure will return from the Fortran subprogram in which the internal procedure is declared.

After the above declaration, "putchar" can be subsequently called in one of two ways:

```
putchar ('='c)

-or-

call putchar ('='c)
```

The second form is preferable, so that a procedure can be converted to a subroutine, and vice-versa. The number of parameters in the call must always match the number of parameters in the declaration. If parameter list is omitted in the declaration, then it also must be omitted in its calls.

If "putchar" were recursive, the declaration would be

```
procedure putchar (ch) recursive 128
```

The value "128" is an integer constant that is the maximum number of recursive calls to "putchar" outstanding at any one time.

Since internal procedures may be mutually recursive, and since they must be declared textually before they are used, procedures may be declared "forward" by separating the procedure declaration from its body. Here is "putchar" declared using a "forward" declaration:

```
procedure putchar (ch) forward

...

# putchar --- put a character in the output string
   procedure putchar {

      character ch

      str (i) = ch
      i += 1
      }
```

As you can see, the parameters must appear in the "forward" declaration; they may appear in the body declaration, but are ignored. For maximum efficiency, all internal procedures should be presented in a "forward" declaration. The procedure bodies should then be declared after the final **return** or **stop** statement in the body of the Fortran subprogram (then the program never has to jump around the procedure body).

In general, a **procedure** declaration contains five parts: the word "procedure", the procedure name, an optional list of formal parameters, an optional "recursive <integer>" part, and either a compound statement or the word "forward". An internal procedure call consists of three parts: optionally the word "call", the procedure name, and an optional parameter list.

Ratfor Language Reference

This section contains a summary of the Ratfor syntax and source program format. In addition to serving as a reference for Ratfor, it can also be used by someone who is familiar with Fortran and wants to quickly gain a reading knowledge of Ratfor.

Differences Between Ratfor and Fortran

Source Program Format

- 'Rp' is sensitive to letter case. Keywords must appear in lower case. Case is significant in identifiers.
- 'Rp' is blank sensitive in that words (sequences of letters, digits, dollar signs, and underscores) must be separated by special characters or blanks.
- 'Rp' is not sensitive to card columns. Statements may begin at any position on a line.
- 'Rp' allows multiple statements per line by separating the statements with semicolons.
- A Ratfor statement may be labeled by placing the numeric label in front of the statement. The label must be separated from the statement by at least one space.
- 'Rp' will expect a continuation line if it encounters a line ending with a trailing comma, a condition with unbalanced parentheses, a missing statement following a control statement, or a line ending with a trailing underscore.
- Any line may contain a comment. Comments begin with a sharp sign ("#") and continue until the end of the line.

Identifiers

Ratfor identifiers consist of letters, digits, underscores, dollar signs, and may be up to 100 characters long. An identifier must begin with a letter. Underscores may be included for readability, but are completely ignored. An identifier may not be the same as a Fortran or Ratfor keyword. 'Rp' transforms all long Ratfor identifiers into unique Fortran identifiers.

Integer Constants

'Rp' allows integer constants of the form "<base>r<number>" where <base> is an integer between 2 and 16. The letters "a" - "f" are used for digits in bases greater than 10.

String Constants

String constants in Ratfor consist of a string body and a string format indicator. The string body is a group of strings, bounded by quotes, and possibily separated by blanks. The string format indicator designates the data representation to be used for the characters in the string body. It has one of the following values:

omitted — Fortran Hollerith string. A standard Fortran Hollerith constant is generated. Characters are left-justified, packed in words (two characters per word on the Prime), and unused positions on the right are filled with blanks.

c — Single character constant. A single character constant is generated. The character is right-justified and zero-filled on the left in a word. Only one character is allowed in the body of the constant. This is the preferred format for all single characters in the Software Tools Subsystem.

p — Packed (Hollerith) period-terminated string. The 'p' format indicator causes the generation of a Fortran Hollerith constant. All periods in the string body are preceded by an escape character ("@").

v — PL/I character varying string. Fortran declarations are generated to create a PL/I

character varying string. "V" format string constants may only be used in executable statements.

s EOS-terminated unpacked string. Fortran declarations are generated to construct an array in which each element contains one character of the string body, right-justified and zero-filled (each character is in the same format as is generated by the "c" format indicator). Following the characters is a word containing the value EOS. EOS-terminated strings are the preferred format for multi-character strings in the Subsystem. "S" format string constants may only be used in executable statements.

Logical and Relational Operators

Ratfor allows the use of graphic characters to represent logical and relational operators instead of the Fortran ".EQ." and such. These characters will be replaced by their Fortran equivalents during preprocessing. The following table shows the equivalent syntaxes:

Ratfor	Fortran	Function
>	.GT.	Greater than
>=	.GE.	Greater or equal
<	.LT.	Less than
<=	.LE.	Less or equal
==	.EQ.	Equal to
~=	.NE.	Not equal to
~	.NOT.	Logical negation
&	.AND.	Logical conjunction
|	.OR.	Logical disjunction
&&	(none)	Short-circuited conjunction
||	(none)	Short-ciructied disjunction

Note that the digraphs shown in the table must appear in the Ratfor program with no imbedded spaces. The short-circuited operators may appear only in the <condition> part of Ratfor control statements.

Assignment Operators

Assignment operators provide a shorthand for the common Fortran idiom "<v> = <v> <op> <expr>". Assignment operators may appear anywhere a Fortran assignment statement may appear. The following assignment operators are available in Ratfor:

Operator	Use	Result
+=	<v> += <e>	<v> = <v> + (<e>)
-=	<v> -= <e>	<v> = <v> - (<e>)
*=	<v> *= <e>	<v> = <v> * (<e>)
/=	<v> /= <e>	<v> = <v> / (<e>)
%=	<v> %= <e>	<v> = mod (<v>, <e>)
&=	<v> &= <e>	<v> = and (<v>, <e>)
|=	<v> |= <e>	<v> = or (<v>, <e>)
^=	<v> ^= <e>	<v> = xor (<v>, <e>)

Incompatibilities

Even with the great similarities between Fortran and Ratfor, an arbitrary Fortran program is not necessarily a correct Ratfor program. Several areas of incompatibilities exist:

- Blanks are significant -- at least one space or special character must separate adjacent keywords and identifiers.
- The Ratfor **do** statement does not contain a statement number following the "do". Its range always extends over the next statement.
- Two word Fortran key phrases such as **double precision** must be presented as a single Ratfor identifier (e.g. "doubleprecision" or "double_precision").

- Fortran statement functions must be preceded by the Ratfor keyword **stmtfunc**. To assure that they will appear in the correct order in the Fortran, they should immediately precede the **end** statement of the program unit.

- Hollerith literals (i.e. 5HABCDE) are not allowed anywhere in a Ratfor program. Instead, 'rp' expects all Hollerith literals to be enclosed in single or double quotes (i.e. "ABCDE" or 'ABCDE').

- 'Rp' does not allow Fortran comments. Ratfor comments must be introduced by a sharp sign ("#").

- 'Rp' does not accept the Fortran continuation convention. Continuation is implicit for any line ending with a comma, or any conditional statement containing unbalanced parentheses. Continuation between arbitrary words may be indicated by placing an underscore, preceded by at least one space, at the end of the line to be continued.

- 'Rp' does not ignore text beyond column 72.

- Fortran and Ratfor keywords may not be used as identifiers in a Ratfor program. Their use will result in unreasonable behavior.

Ratfor Text Substitution Statements

define (<identifier> [(<formal params>)], <replacement text>)

When a **define** statement is encountered in a source program, <replacement text> is recorded as the replacement for <identifier>. If <identifier> is encountered later in the program, it will be replaced by <replacement text>. If <formal params> was present in the definition of <identifer>, and the subsequent occurence of <identifier> is followed by a parenthesized, comma-separated list of strings, occurrences of the formal parameters in <replacement text> will be replaced by the corresponding strings in the actual parameter list.

<Identifier> must be an alphabetic Ratfor identifier, while <replacement text> may contain any characters except unmatched quotes or parentheses. <Formal params> must be a comma-separated list of identifiers; corresponding actual parameters may contain any characters except unmatched quotes, unbalanced parentheses, or unnested commas. During replacement, <replacement text> is also examined for occurences of **defined** identifiers. Formal parameter replacement occurs on identifiers in <replacement text>, even if the identifiers are surrounded by quotes or parentheses. Redefinition of an <identifier> causes the new <replacement text> to replace the old.

undefine (<identifier>)

The **undefine** statement removes the definition of <identifier> from the list of defined identifiers. Subsequent occurrences of <identifier> in the program will not be replaced unless <identifier> appears in a subsequent **define** statement.

include '<path name>'

An **include** statement instructs 'rp' to begin taking input from the file specified by <path name>. When the end of the file is reached, 'rp' resumes taking input from the file containing the **include** statement. The path name may be surrounded by either single or double quotes. The file specified by <path name> may contain further **include** statements, up to a maximum depth of 5.

Ratfor Declarations

linkage <identifier> { , <identifier> }

The **linkage** declaration is used to guarantee that long external names are transformed into the same unique Fortran name. Names are transformed as they are presented in the **linkage** declaration. The same **linkage** statement should appear as the first statement of each separately compiled source module, and should contain the names of all subroutines, functions, and common blocks in the program.

local <identifier> { , <identifier> }

The **local** declaration allows the declaration of variables with names local to the scope of a compound statement (block). The **local** declaration should appear inside a compound statement and must precede all occurences of the identifiers to be declared local to the block. All identifiers appearing in a **local** declaration must subsequently appear in a type declaration in the same compound statement.

string <name> <quoted string>

The **string** statement generates declarations to produce an EOS-terminated string in the integer array <name>. <Quoted string> must be surrounded by either single or double quotes.

stringtable <index>, <body>, [/] <item> { / <item }

The **stringtable** declaration creates a marginally indexed array of integers and character strings. <Index> and <body> are variables to be declared as the index and body arrays respectively. <Body> is a one-dimensional array in which the values generated by the <item>s are stored consecutively. The first element of <index> contains the number of remaining elements in <index>; subsequent elements each contain the index in <body> of the first position of the corresponding <item>.

<Item>s are comma-separated lists of integers, single-character constants, and strings (with no string format indicators). Integers and EOS-terminated strings are generated and stored consecutively in <body>. The first position of each <item> in <body> is stored in the corresponding entry of <index>.

Ratfor Control Statements

break [<integer>]

The **break** statement allows the user to terminate the execution of a for, while, or repeat loop and resume control at the first statement following the loop. The <integer> specifies the number of loops to terminate; if absent, 1 is assumed (only the innermost loop is terminated). If the integer is N, then the N innermost loops currently active are terminated.

do <limits>; <statement>

The **do** statement provides a means of accessing the local Fortran do-statement. <Limits> includes whatever parameters are necessary to satisfy Fortran, minus the statement number of the last statement to be performed, which is generated by Ratfor. The semicolon must not be used if the statement to be iterated does not appear on the same line as the **do**.

for '(' <init>; <condition>; <reinit> ')' <statement>

The **for** statement is a very general looping construct. <init> is a statement to be executed before loop entry; it is frequently used to initialize a counter. <Condition> is a condition to be satisfied for every iteration; the condition is tested at the top of the loop. <Condition> becoming false is the most often used method of terminating the loop. <Reinit> is a statement to be executed at the bottom of the loop, just before a jump is made to the top to test the <condition>. <Reinit> is usually used to increment or decrement a counter. <Statement> may be any legal Ratfor statement.

if '(' <condition> ')' <statement> [else <statement>]

If is a generalization of the Fortran logical-if statement. If the condition is true, the first <statement> is executed. If the optional **else** clause is missing, control is then passed to the statement following the **if**; otherwise, the <statement> following the **else** is executed before passing control.

next [<integer>]

The **next** statement complements the **break** statement. It is used to force the next iteration of a for, repeat or while loop to occur. The parameter <integer> specifies the number of levels of nested loops to jump out; if omitted, the innermost loop is continued; otherwise, for <integer> = 2, the next-to-innermost loop is continued, etc.

```
procedure <procid> [ '(' <id> {, <id> } ')' ]
      [recursive <integer>]
      ( forward | <compound statement> )

[call] <procid> [ '(' <expr> {, <expr> } ')' ]
```

The **procedure** declaration allows the declaration of internal Ratfor procedures. <Procid> is the name of the internal procedure. Formal parameters (scalar, pass-by-value) are declared following the <procid>. Formal parameters must appear in a type declaration in the body of the procedure. If the procedure is to be called recursively, the **recursive** <integer> clause must be included; <integer> is the maximum number of recursive calls in process at any given time. Following the heading, either a compound statement or the word **forward** must appear. If the **forward** option is used, a **procedure** declaration containing <compound statement> must follow at some point in the program unit. Formal parameters specified on the second declaration may be present, but are ignored.

A <procid> must be defined before it is referenced by a call. The call can appear exactly as a Fortran call, or the word **call** can be omitted. Actual parameters must correspond in number to formal parameters. If the formal parameters list is omitted in the declaration, no actual parameter list may be present.

```
repeat <statement> [until '(' <condition> ')']
```

The **repeat** statement is used to generate a loop with the iteration test at the bottom. The <statement> is performed, then the <condition> checked; if false, the <statement> is repeated. If true, control passes to the statement following the **until**. If the **until** is omitted, the loop is repeated indefinitely, and must be terminated with a **stop**, **break**, or **goto**.

```
return ['(' <expression> ')']
```

The **return** statement behaves exactly like its Fortran counterpart, except that if the optional parenthesized expression is included inside a function subprogram, the value of <expression> will be assigned to the function name as the function value before the return is executed.

```
select
    {when '(' <condition> {, <condition>} ')' <statement> }
    [ifany <statement>] [else <statement>]

select '(' <integer expr> ')'
    {when '(' <integer expr> {, <integer expr>} ')' <statement>
    [ifany <statement>] [else <statement>]
```

Select is a generalization of the **if** statement. In its first alternative, the **when** <conditions>s are evaluated in order; the <statement> associated with the first one found to be true is executed. If any <condition> is found true, the <statement> assocated with **ifany** is executed; if none are found true, the <statement> assocated with **else** is executed.

Similarly, in the second alternative, the <integer expr> associated with **select** is evaluated. The result is then compared to the <integer expr>s associated with the **when** parts in an unspecified order. When an equal comparison is made, the <statement> following the corresponding **when** is executed. If an equal comparison is made, the <statement> following **ifany** is executed; if no equal comparison is made, the <statement> following **else** is executed.

```
while '(' <condition> ')' <statement>
```

The **while** statement is the basic test-at-the-top loop. The <condition> is evaluated; if true, the <statement> is executed and the loop is repeated, otherwise control passes to the statement following the loop.

Ratfor Programming Under the Subsystem

This chapter describes the use of Ratfor in the programming environment provided by the Software Tools Subsystem. In addition to demonstrating use of the Ratfor preprocessor, Fortran compiler, and linking loader, the programming conventions necessary for the use of the Subsystem support subprograms are described.

In this chapter, a number of programming conventions are presented. Since very few of the conventions can be enforced by the Subsystem, adherence to these conventions must be left to up to the programmer. Many conventions, such as those dealing with indentation and comment placement, are shown because they assist in producing readable, maintainable programs. Violation of these conventions, while not critical, may result in unmaintainable programs and extended debugging times. Other conventions, such as those dealing with character string representations and input/output, are crucial to the proper operation of the Subsystem and its support subprograms. Violation of these conventions can and will cause undesirable results.

Requirements for Ratfor Programs

The Software Tools Subsystem is not an operating system. Rather, it is a collection of cooperating user programs. To run successfully under the Subsystem, a program must cooperate with it. Several things are required of Subsystem programs:

* - The program must terminate with a **stop** statement, or a call to the routine "error". The program must not "call exit" or invoke any of the Primos error reporting subroutines with the the "immediate return" key. A program's failure to terminate properly will also cause the Subsystem command interpreter to be terminated, leaving its user face-to-face with Primos.

- The program should not have initialized common blocks (i.e. **block data**). Initialize the common areas with executable statements. (To link a program that must have initialized common, see appendix b.)

- Local variables in a subprogram are placed on the stack unless they appear in a **data** or **save** declaration. The value of variables not appearing in one of these declarations is not defined on entry to a subprogram.

Several conventions apply to the file containing the Ratfor source statements:

- The file name should end with the suffix ".r".

- Any number of program units (main program, functions, and subroutines) may be included in the file, but the main program must be first.

- All variables and functions must be declared in type statements (the Primos Fortran compiler enforces this restriction, except in the case of function names).

- Each program unit must end with an **end** statement.

- Since **defines** apply globally to all subsequent program units, a main program and all of its associated subprograms can be contained in the same file. Only one copy of definitions need be included at the beginning of the source file.

Running Ratfor Programs Under the Subsystem

Three steps are required to obtain an executable program from Ratfor source statements. The first step, preprocessing, produces ANSI Fortran statements from the Ratfor source statements. The second step, compilation, results in a relocatable binary module, which lacks all of the Primos, Fortran and Subsystem subroutines. The last step, linking, produces an executable object program by linking the relocatable binary module with the Primos, Fortran and Subsystem support routines necessary for its execution. The object program produced during linking may then be executed.

Preprocessing

In the preprocessing step, the Ratfor preprocessor, 'rp,' is used to tranlate Ratfor source statements into semantically equivalent ANSI Fortran statements acceptable to the Primos Fortran compiler. The Ratfor preprocessor is invoked with a command line of the following syntax:

```
rp [-o <output file>] <input file> . . .
```

If you do not want a conventionally named output file, you may specify the option "-o <output file>", where <output file> is the name you want given to the Fortran output. If you do not include a "-o <output file>" option, 'rp' will name the output file by appending ".f" to the name of the first <input file>. If the name of the first <input file> ends in ".r", the ".r" will be replaced by the ".f".

Finally comes a list of the files containing Ratfor source statements to be preprocessed. 'Rp' reads the files in the order specified on the command line and treats the contents as if they were together in one big file. This means that **defines** in each input file apply to all subsequent input files.

In spite of all this complicated stuff, the 'rp' preprocessor is quite easy to use if you follow the recommended naming conventions for files. For instance, if you have a Ratfor program in a file called "prog.r", you can have it preprocessed by just typing

```
rp prog.r
```

This command will cause the program contained in "prog.r" to be preprocessed, and the Fortran output to be produced on the file "prog.f" (which is exactly what the Fortran compiler expects).

Here are some more examples to show other ways in which 'rp' can be called:

```
# preprocess the files "p1.r", "p2.r", and "p3.r"
#    and produce Fortran output on "p1.f"

  rp p1.r p2.r p3.r


# preprocess the files "p1.r", "p2.r", and "p3.r"
#    and produce Fortran output on "ftn_out"

  rp p1.r p2.r p3.r -o ftn_out


# preprocess the file "p1.r", produce the Fortran
#    on "ftn_out" and include code to produce
#    subprogram level trace

  rp -t p1.r -o ftn_out
```

Compiling

After turning your Ratfor source code into Fortran with the preprocessor, the next step is to compile the Fortran code. Since the Subsystem uses the Primos Fortran compiler, the 'fc' command just produces a sequence of Primos commands to cause the compilation The following command will call the Fortran compiler for a compilation:

```
fc [<options>] <input> [+b <binary>] [+l <listing>]
```

The Fortran source code must be in the file <input>. The relocatable binary output will be placed in the file <binary>, unless "+b <binary>" is omitted. Then, following Subsystem conventions, the binary file name is constructed by appending the input file name with ".b"; if the input file ends with ".f", the "f" will be replaced by the "b". Normally no listing is produced; however, if one is requested, it will appear on the file <listing>, or if the listing file name is omitted, the name will be constructed by appending the ".l" to the input file name; again, if the input file name ends in ".f", the "f" will be replaced with the "l".

<Options> is a series of single letter options that specify how the compiler is to generate the object code. Since there are too many options to completely describe here, we will only mention a few of the more important ones. For those who wish to make full use of the Fortran compiler, or for those just curious, the Software Tools Subsystem Reference Manual, or the 'help' command will give complete information.

Here are brief descriptions of the options of interest:

+a Generate pseudo-assembly code describing the object code produced.

+i Unless otherwise specified, consider all integers to be "long" (32-bit) rather than "short" (16-bit). (This is useful for programs ported from machines with longer word lengths.)

+l Produce a listing of the Fortran program with imbedded diagnostics.

+t Insert code to produce a statement-level trace during execution.

-u Do not flag undeclared variables.

Of course, more than one of these options may be specified.

Again, even though all of this looks very complicated, it is really very simple, if you have used the Subsystem file naming conventions. If you have your Fortran code in a file named "prog.f" (remember where Ratfor put its output), you may compile it, using the default options, by just entering

```
fc prog.f
```

The command will call the Fortran compiler to produce binary output in the file "prog.b". Just for completeness, here are some other examples of 'fc' commands:

```
# Compile "p1.f" to produce the binary "p1.b" and
#    and a listing on "p1.l"

  fc +l p1.f


# Compile "p1.f" to produce the binary "bin" and
#    the listing on "list"

  fc +l p1.f +b bin +l list


# Compile "p3.f", produce a pseudo-assembly code
#    listing, do not flag undeclared variables,
#    and default to 32-bit integers

  fc +l +a +i -u p3.f
```

One problem you may encounter when using 'fc' is that the Primos Fortran compiler pays no attention to i/o redirection when it is writing error messages to the terminal. This is a problem common to all Primos commands called from the Subsystem. If you want to record the terminal output of the Fortran compiler, you must use the Primos command output facility. This facility is accessed through the Subsystem 'como' command; for details, see the **Software Tools Subsystem Reference Manual** or use the 'help' command.

Linking

The last step in preparing the program for execution is linking. The linking step fixes the memory locations of the Subsystem common areas; assigns the binary module for each subprogram to an absolute memory location; and links in the required Subsystem support routines, Fortran run-time routines, and Primos system calls. The memory image file produced by this step may then be executed. It should be noted here that programs linked under the Subsystem can run only under the Subsystem; they may not run without it.

The 'ld' command is used to invoke the Primos loader to to do the linking. Its syntax is as follows:

```
ld [-u] <binary file> . . . [-l <library file>] . . .
     [-t -m] [-o <output file>]
```

This is not the entire syntax accepted by 'ld,' but a complete discussion requires detailed knowledge of the Primos loaders. For more information, see the Subsystem reference manual.

The "-u" option causes the loader to print a list of undefined subprograms. Any number of binary files to be included may be listed. The only restriction is that the main program must be the first binary subprogram encountered -- it must be the first program unit in a binary file, and that binary file must be the first <binary file> to appear on the command line. Any number of libraries (residing in "=lib=") may then be specified with the "-l"

option. The "-t -m" options cause a load map to be produced on a file with the name as the output file (or first <binary file>, if an output file is not specified) with ".m" appended. If the file name ends with ".b", the ".b" is replaced by the ".m". The "-o" option specifies the name of the output file. If the "-o" option is omitted, the output file will have the same name as the first <binary file>, with ".o" appended. If the name of the first <binary file> ends in ".b", the ".b" will be replaced by the ".o".

Even though linking is a mysterious process, it need not be traumatic. Most of the time, you will be linking a single binary file with no additional libraries. For instance, if you had a binary file named "prog.b," you could produce an object program by just typing the command

```
ld prog.b
```

The Primos loader would be invoked, and after a great deal of garbage was printed on the terminal, the executable program "prog.o" would be produced.

The only thing that you must do is look for the message "LOAD COMPLETE" lurking somewhere near the end of this garbage. If you find this message, it means that all of the external references in your program (subroutine and function calls) have been satisfied, and linking is complete. If you don't find this message, there are unsatisfied references in your program. You may then call 'ld' with the "-u" option and the loader will print the names of the unsatisfied references on the terminal. You will probably then find that these references are caused by misspelled subprogram names, missing subprograms, or undimensioned arrays (remember, the Fortran compiler treats undimensioned arrays as functions calls, so you may not always get an error message from the compiler).

Again, for completeness, here are some examples of 'ld' at work:

```
# link the binary files "p1.b", "p2.b", and "p3.b"
#    to produce "p1.o" as output

  ld p1.b p2.b p3.b


# link the binary file "nprog.b",
#    include the library "vpatlb",
#    and produce the output file "nprog"

  ld nprog.b -l vpatlb -o nprog


# link the binary files "np1" and "np2",
#    produce a load map,
#    and output "my_new_prog"

  ld np1 np2 -t -m -o my_new_prog
```

The Primos loader also pays no attention to i/o redirection. If you want to catch its terminal output, you must use the Primos 'como' commands. For details, see the reference manual or use the 'help' command.

Executing

Executing a Subsystem program is the easiest step of all. All you have to do to execute it is to type its name. For instance, if your object program was named "prog.o", all you need type is

```
prog.o
```

to make it go. Because the shell also looks in your current directory for executable programs, "prog.o" is now a full-fledged Subsystem command. You may give it arguments on its command line, redirect its standard inputs and outputs, include it in pipelines, or use it as a function. Of course to be able to do all of these things properly, it must observe the Subsystem conventions and use the Subsystem I/O routines.

Shortcuts

There are several shortcuts that speed things up and save typing when developing programs.

Shell Programs. Shell programs can be a great help when performing repetitive tasks. Quite often one of these tasks is preprocessing, compiling, and linking a program during its

development. A simple shell program can save a great deal of typing in this situation. For instance, let's say we are writing a Ratfor program that is in the file "np.r". We are in the process of adding new features to "np" and will probably compile and test it several times. We can make a very simple shell program that will keep us from having to type 'rp,' 'fc,' and 'ld' commands every time we want to make a test run. All we have to do make a file containing these three commands with 'cat':

```
] cat >cnp
rp np.r
fc np.f
ld -u np.b -o np
<control-c>
]
```

Now the file "cnp" contains the following text:

```
rp np.r
fc np.f
ld -u np.b -o np
```

All we need do now to preprocess, compile, and link our program is just type the name of the shell program as a command:

```
cnp
```

and the shell will execute all of the commands contained in it.

The 'Rfl' Command. Of course, it is so common to preprocess, compile, and link a program, there is an already-built shell program that works nicely in most cases. 'Rfl' contains the necessary commands to preprocess, compile and link a Ratfor program contained in a file whose name ends with ".r". All you have to do is type

```
rfl np.r
```

and 'rfl' will execute the necessary commands to produce an executable file named "np". (note that the executable file is named "np" and not "np.o"!) 'Rfl' can also do some other handy things that you can find out about in the Subsystem reference manual.

Storing Source Programs Separately. When you write fairly large programs or test modules independently, it is often convenient to store the programs in separate files. If this is the case, creating an executable program is just a little bit more complicated. The easiest solution is to just name all of the programs on the 'rp' command line, like this:

```
rp p1.r p2.r p3.r
```

'Rp' will preprocess all of the files together and produce output on the file "p1.f". The **define** statements in "p1.r" will still be in effect when "p2.r" is preprocessed, etc. so "p1.r", "p2.r", and "p3.r" might just as well be together in one file.

Compiling Programs Separately. A little bit harder, but sometimes much faster, is to preprocess and compile the modules separately and then combine them during linking. There are two things that you have to watch. The first problem with separate compilation is that **define** statements in one file cannot affect subprograms in the other files. For a large program that would benefit from separate compilation, this nastiness can be avoided by placing all of the **defines** together in one file and placing an **include** for that file at the beginning of each of the files containing the program. The **defines** will then be applied uniformly to all parts of the program.

The second thing is that since Ratfor chooses unique Fortran names in the order it is presented with "long" Ratfor names, it cannot guarantee that a long name in one file will be transformed into exactly the same Fortran name as the same long name in a second file (although the probability is quite high). To avoid problems, either subprogram names that are cross-referenced in the separate binary files should be given six-character or shorter names, or a **linkage** declaration containing the names of all subroutines, function, and common blocks should be inserted at the beginning of each module. It is usually easiest to handle the **linkage** declaration just like the **define** statements: put it in a separate file, and add an **include** statement for it at the beginning of each module.

Then, the program units in each file may be preprocessed and compiled separately. The binary files from the separate compilations are linked together by just listing the names of all of the files on the 'ld' command:

```
ld p1.b p2.b p3.b
```

The only restriction is that the main program must appear first. The object file from this example would be named "p1.o", but this could have been overridden by including the "-o

<output file>" option.

When compiling parts of a program separately, you should be aware that incorrect use of the **linkage** declaration can cause totally irrational behavior of the program with no other indication of error. Since no checking is done on the **linkage** declaration, you must be certain that every external name appears in the statement. More importantly, when you add a subroutine, function, or common block, you must remember to change the **linkage** declaration. In addition, if you do not add the name to the very end of the declaration, you must immediately recompile all modules! If you compile separately, and are confronted with a situation in which your program is misbehaving for no apparent reason, re-check the **linkage** declaration and recompile all the modules.

Debugging

Debugging unruly programs under Primos is at best a grueling task, as currently there is almost no run-time debugging support. Except for a couple of machine-language level debuggers, you'll get very little help from Primos (except for some nasty error messages) while debugging programs. This means that such techniques as top-down design, reading other programmers' code, and reasonably careful desk checking will pay off in the long run. But even with all the care in the world, some bugs will creep through (especially on an unfamiliar system). The next few paragraphs will be devoted to techniques for exterminating these stubborn bugs.

For an experienced user, a load map, the Primos DMSTK command, and VPSD (the V-mode symbolic debugger) can very quickly isolate the location, if not the cause, of a bug. With more complicated programs that are dependent on the internal structure of the machine and operating system, such machine level debugging cannot always be avoided. If you find yourself in such as position, you can begin to learn some of these things by examining the following reference manuals:

MAN 1671 System Reference Manual, Prime 100-200-300

MAN 2798 System Reference Manual, Prime 400

FDR 3059 The PMA Programmer's Guide

FDR 3057 User Guide for the Fortran Programmer

Most often, the bug can be found by one or more of the following techniques:

(1) Inserting 'print' calls to display the intermediate results within the program.

(2) Using the Ratfor subroutine trace.

(3) Using the Fortran statement number and assignment trace.

It is usually quickest to use the Ratfor subroutine trace (by including the "-t" option on the 'rp' command line). Although this trace lists only subroutine nesting, it will narrow down where a program is blowing up to a single subprogram. If the program is very modular and contains mostly small subprograms, quite often, the error can be spotted.

If the Ratfor trace fails to pinpoint the problem, the Fortran statement and assignment trace will give a great deal more information (possibly hundreds of pages). The Fortran trace can be produced by specifying the "+t" option on the 'fc' command. The Fortran code produced by 'rp' must be examined to locate the statement numbers, but given the large number of statement labels generated by 'rp,' study of this trace can isolate the problem practically to within one statement.

The above debugging methods are quick and easy to use when the program contains a catastrophic error that causes an error termination or an infinite loop. While this is sometimes the case, more often a subtle error is the problem. In finding these errors, there is no substitute for carefully inserted debugging code (such as calls to 'print') at critical points in the program.

The rest of this section is devoted to a brief description of many of the terminal errors that may do away with programs (and the Subsystem). Most terminal errors cause the Subsystem command interpreter to be terminated along with the user's delinquent program. You can tell that you've been booted into Primos by the appearance of the "OK," or "ER!" prompt. All error messages that cause an exit to Primos are briefly explained in appendix A-4 of the Prime Fortran Programmer's Guide (FDR3057). Some very common programming errors can cause cryptic error messages with explanations that are close to unintelligible. Hopefully, most of these messages are described below.

Many Primos error messages are dead giveaways of program errors. Messages that begin with four asterisks are from the Fortran runtime packages -- they usually indicate such things as division by zero or extraction of the square root of a negative number. For example,

```
**** SQRT -- ARGUMENT < 0
OK,
```

results from extracting the square root of a number less than zero.

Other, more mysterious, error messages can also be caused by simple program errors.

```
Error: condition "POINTER_FAULT$" raised at <addr>
```

can be caused by referencing a subprogram which has not been included in the object file. An obvious indication of a missing subprogram is the failure to get the

```
LOAD COMPLETE
```

message from 'ld'. (Note that the Fortran compiler treats references to undimensioned arrays as function calls!) A more insidious cause of the "POINTER FAULT" message is a reference to an unspecified argument in a subprogram; i.e. the calling routine specifies three arguments and the called routine expects four. The error occurs when the unspecified argument is referenced in the subprogram, not during the subprogram call.

```
Error: condition "ACCESS_VIOLATION$" raised at <addr>
Error: condition "RESTRICTED_INST$" raised at <addr>
Error: condition "ILLEGAL_SEGNO$" raised at <addr>
Error: condition "ARITH$" raised at <addr>
Program halt at <addr>
```

all can result from a subscript exceeding its bounds. Because the program may have destroyed parts of its code, the memory addresses sometimes given may well be meaningless. Even so, you may locate the routine in which the program blew up by using the Primos DMSTK command and a load map. For instance, given the following scenario (ellipsis indicate irrelevant information),

```
Error: condition "POINTER_FAULT$" raised at 3.4000.001000.
OK, dmstk
...
Stack Segment is 6002.

   6) 001464: Condition Frame for "POINTER_FAULT$"; ...
      Raised at 3.4000.017202; LB= 0.4000.017402, ...

   7) 001374: Fault Frame; fault type= 000064
      Returns to 3.4000.017202; LB= 0.4000.017402, ...
      Fault code= 100000, Fault addr= 3.4000.017204
      Registers at time of fault:
...
```

you can at least attempt to tell where the program was executing. The numbers following "LB=" on the underlined portion of the stack dump show the address of the data area of the procedure executing when the fault occurred. The segment number portion of this address (the four-digit part) tells who the routine belongs to:

Segment	Use
0000 - 0023	Operating System
2030	Software Tools Shell
2035	Software Tools Library
2050	Fortran Library
4000 - 4037	User Program
6001	Fortran Library

If the executing routine is not part of your program, you can trace back the stack (see below) until you find which of your subprograms made the call. If the segment number begins with "4", you need only look down the right-most two columns of the load map (see the 'ld' command) for the two numbers (4000 17402 in this case). If you get an exact match, just look across to the name on the left -- this is the subprogram that was executing. Otherwise, if none of the numbers match then either the program has clobbered itself and jumped into nowhere, you left off an argument to a library subprogram, or one of the library routines has caused an exception trap with no fault vector.

Subsequent entries in the stack dump (following the information in the last scenario) can be used to find what procedure calls were in process when the error occurred. The

entries are of the following form:

```
     Stack Segment is 4035.

        8) 002222: Owner=  (LB= 0.4000.017402).
           Called from 3.4000.017700; returns to 3.2035.017702.

        9) 002156: Owner=  (LB= 0.4000.013026).
           Called from 3.4000.013442; returns to 3.2030.013450.
     ...
```

Each entry on the Subsystem stack (segment 4035) represents a procedure call in process. You can use the numbers following the "LB=" and the load map to trace back through the "stack" of procedure calls, just with the "fault frame" mentioned above.

If you find yourself at a complete and total loss at finding why your program is blowing up, here is a list of some of the errors that have caused us great anguish:

- Subscript out of range. This error can cause any number of strange results.

- Undefined subprogram. This error can be detected by the lack of a "LOAD COMPLETE" message from the 'ld' command.

- Too few arguments passed. This error almost always causes a "POINTER_FAULT$" when the missing argument is referenced.

- Code and initialized local data requires more that one segment (64K words). The load map shows how much space is allocated. No linkage or procedure frame should appear in any segment other than 4000.

- Delimiter character is missing in a packed string. This includes periods in packed strings passed to 'print' and 'input'. This error causes the program to run wild, writing all over the place.

- Type declaration is missing for a function. This error can causes failure of routines such as 'open' which return an integer result. The Primos Fortran compiler does not flag undeclared functions. This error may also cause an erratic real-to-integer conversion error or cause the program to take an exception trap.

- A subprogram is changing the value of a constant. If you pass a single constant as a function or subroutine argument, and the subprogram changes the corresponding parameter, the values of all occurrences of that constant in the calling program will be changed. With this error, it is quite possible for the constant 12 to have the value -37 at some time during execution.

Performance Monitoring

In most cases, it is very difficult to determine how much processing time is required by different parts of a program. Since it is nearly impossible to determine which parts of a program are "inefficient", especially before the program is written, it often more effective to write a program in the most simple and straightforward manner, and then use performance monitoring tools to find where the program is spending its time. It has many times been our experience to find even though parts of a program are coded inefficiently, only a very small amount of time is wasted.

There are two available methods for obtaining an execution time "profile" of a Ratfor program. The first method provides statistics on the number of calls to and the amount of time spent in each subprogram. The second method provides a count of the number of times each statement in the program is executed.

To invoke the subroutine profile, just preprocess (in one run) all the subprograms to be profiled. Add the "-p" option to the 'rp' command line when the programs are preprocessed. Then compile, link and execute the program normally. When the program terminates (it must execute a **stop** statement, and not call "error"), type the command

```
profile
```

'Profile' accesses the files "timer_dictionary" (output by 'rp') and "_profile" (output by your program) and prints the subroutine profile to standard output.

To invoke the statement count profile, put all the subprograms to be profiled (you must also include the main program) in a single file. Then preprocess the file with 'rp' and the "-c" option. Compile, link, and execute the program. When the program terminates normally, type the command

```
st_profile myprog.r
```

(Of cource, assuming your source file name is "myprog.r".) A listing of the program with execution count for each line will be printed.

When running a profile, there are several things to keep in mind. First, the program with the profiling code can be more than twice as large as the original program. Second, the program can run an order of magnitude more slowly. Third, there can be a considerable delay between the execution of the **stop** statement and the actual end of the program. Finally, you should remember that the main program and all subprograms to be profiled must be preprocessed at the same time.

Conditional Compilation

Conditional compilation is a handy trick for inserting debugging code or setting compile-time options for programs. Conditional compilation can be approximated in Ratfor by defining an identifier, such as "DEBUG" to a sharp sign or null (for off and on respectively). Lines in the Ratfor program beginning with the identifier "DEBUG" (i.e. debugging code) are not compiled if "DEBUG" is defined to be "#", but are compiled normally if "DEBUG" is defined as a null string.

For instance, the following example shows how conditional compilation can be used to "turn off" print statements at compile time:

```
define (DEBUG, #)

   fd = open (fn, READ)
DEBUG  call print (ERROUT, "fd returned:*i*n"s, fd)
...
   len = getlin (str, fd)
DEBUG  call print (ERROUT, "str read: *s"s, str)
```

In this example, all lines beginning with "DEBUG" are ignored, unless the **define** statement is replaced with

```
define (DEBUG, )
```

Then, all lines beginning with "DEBUG" will be compiled normally.

Source Program Format Conventions

After considering many program formatting styles, we have concluded that the convention used by Kernighan and Plauger in Software Tools is the most expedient in terms of clarity and ease of modification. As a consequence, we have tried to be consistent in the use of this convention throughout the Subsystem to provide uniformly readable and modifiable code. We present the convention here in the hope that you can use it to the same advantage.

Statement Placement

The placement of statements in program units is perhaps the most important part of the formatting convention. Through uniform placement of statements, many documents can be produced directly directly from the source code. For instance, the skeleton for Section 2 of the Subsystem Reference Manual was produced originally from the subprogram headers of the Subsystem library subprograms. Then the detail was filled in using the text editor.

The order of a program unit (including a main program) should be as follows:

1. A comment line of the following format:

   # <program name> --- <one-line description>

2. The **subroutine** or **function** statement (or nothing if it is a main program).

3. The declarations of all arguments passed to the subprogram, if any.

4. A blank line

5. Declarations for all local variables in the program unit.

6. A blank line.

7. Executable program statements.

8. The **end** statement.

9. Three blank lines.

Of course, extra blank lines should be used freely to separate different logical groups of declarations and different logical blocks of executable statements.

As an example, here is the source code for the subroutine "cant" taken directly from the Subsystem library:

```
# cant --- print "can't open" file message
   subroutine cant (str)
   character str (ARB)

   call putlin (str, ERROUT)
   call error (": Cannot open.")
   return
   end
```

Indentation

The indentation convention is very simple. It is based on the idea that a statement should be indented three spaces to the right of the innermost statement controlling it. Braces are placed as unobtrusively as possible, without affecting the ease of adding or deleting statements.

Statements, with the exception of the program heading comment, are placed three spaces to the right of the left margin. All statements are placed in this position, unless they are subordinate to a control statement. In this case, they are placed three spaces to the right of the beginning of the controlling statement.

Braces do not affect the placement of statements. An opening brace is placed on the line with the controlling statement. A closing brace is placed on a separate line three spaces to the right of the beginning of the controlling statement.

Multiple statements per line are forbidden, except when a chain of **if - else if . . . else** statements is used to implement a case structure. In this event, the **else if** is considered a single statement, appearing on the same line, and subsequent lines are indented only three spaces to the right.

If all of this seems terribly confusing, here are some examples that show the identation convention in action (the bars are just to show you the matching of braces):

```
for (i = 1; str (i) ~= EOS; i += 1) {
|  if (str (i) == 'a'c) {
|  |  j = ctoi (str (2), i)
|  |  select (j)
|  |  |  when (1)
|  |  |  |  call alt1
|  |  |  when (2)
|  |  |  |  call alt2
|  |  |  when (3) {
|  |  |  |  call alt1
|  |  |  |  call alt2
|  |  |  |  }
|  |  else
|  |     call error ("number must be >= 1 and <= 3.")
|  ---}
|  else if (str (i) == 's'c)
|     repeat {
|     |  j = ctoi (str (2), i)
|     |  status = getnext (j)
|     ---} until (status == EOF)
|  else {
|  |  call clean_up
|  |  stop
|  ---}
---}
```

Subsystem Definitions

The use of the **define** statement plays a large part in producing readable, maintainable programs. Hiding implementation details with **define** statements not only produces more readable code, but allows changes in the implementation details to be made without necessitating changes in applications programs. The development of a large part of the Subsystem would have been greatly hindered if it had not been possible to redefine the constant "STDIN" from "1" to "-11", with no more than recompilation.

The Subsystem definitions file, "=incl=/swt_def.r.i" exists primarily to hide the dirty details of the Subsystem support routines from Ratfor programmers. We sincerely believe that the character string "EOF" is inherently more meaningful than the string "-1". (Would you believe that after three years of using the Subsystem, the author of this section had to look up the value assigned to "EOF" in order to write the preceding sentence?)

Of course, the use of the Subsystem definitions also allow the developers to change these values when necessary. Of course, these changes force recompilation of all existing programs, but we feel that this is a small price to pay for the availability of more advanced features. All users of the Subsystem support routines are therefore warned that the values of the Subsystem definitions may change between versions of the Subsystem. (At Georgia Tech, this may be daily.) Programs that depend on the specific values of the symbolic constants may well cease to function when a new version of the Subsystem is installed.

Appendix D contains specific information about (but not specific specific values for) the standard Subsystem definition file. As a general rule, all symbolic constants mentioned in Section 2 of the Subsystem Reference Manual can be found in "=incl=/swt_def.r.i".

Using the Subsystem Support Routines

Many of the capabilities available to a Subsystem programmer are provided through the Subsystem support routines. The Subsystem support routines consist of well over one hundred Ratfor and PMA subprograms that either perform common tasks, insulate the user from Primos and Fortran, or conceal the internal mechanisms of the Subsystem. By default, the library containing all of these routines ("=lib=/vswtlb") is included in the linking of all Subsystem programs. Therefore, no special actions need be taken to call these routines.

If you notice that there are some "holes" in the functionality of the Subsystem library, you are probably quite correct. The Subsystem library has grown to its present size through the effort of many of its users. The instance often arises that a routine is required to fill a specific function. In keeping with the Software Tools methodology, instead of writing a very specific routine, we ask that the author write a slightly more general routine that can be used in a variety of instances. The routine can then be documented and placed in the Subsystem library for the benefit of all users. Many of the support routines, including the dynamic storage management routines, have come from just such instances. The "holes" in the Subsystem library are just waiting for someone to fill them; if you need a routine that isn't there, please write it for us.

Initialization and Termination

The two most important routines are those that perform the initialization and termination of Subsystem programs.

Init. The subprogram 'init' performs the initialization necessary when execution a Subsystem program. The statement "call init" must be the first executable statement in any program that is to run under the Subsystem. As a convenience, Ratfor automatically includes such a call in every main program.

Swt. The subprogram 'swt' terminates the program and causes a return to the Subsystem command interpreter. Any Subsystem files left open by the program are closed. Ratfor automatically inserts a "call swt" any time it encounters a Fortran **stop** statement. All Ratfor programs should **stop** rather than "call exit". Fortran and PMA programs should invoke 'swt' to terminate.

Character Strings

Most of the support routines use characters that are unpacked, one per word (i.e. integer variable), right-justified with zero fill, rather than the Fortran default, two characters per word, left-justified, with blank fill (for an odd last character). In addition to the simplicity of manipulating unpacked strings, the unpacked format represents characters as small, positive integers. Thus, character values can be used in comparisons and as indexes without conversion.

Most of the support routines that manipulate character strings expect them to be stored in an integer arrray, one character per word, right-justified and zero-filled, and terminated with a word containing the symbolic constant 'EOS'. Strings of this format are usually called EOS-terminated strings.

Support for the use of unpacked characters is provided in several ways: (1) the Subsystem I/O routines perform conversion to and from unpacked format, (2) single-character constants 'a'c, 'b'c, ','c, etc. are provided for use in place of single-character Hollerith literals, and (3) the Ratfor **string** statement is provided to initialize EOS-terminated strings.

In a few cases, it is more convenient to use a Hollerith literal instead of an EOS-terminated string. Since it is impossible to tell the length of a Hollerith literal at run time, Hollerith literals used with the Subsystem are required to contain a delimiter character (usually a period) as the last character. Hollerith literals or integer arrays that contain Hollerith-format characters and end with a delimiter character are referred to as packed strings.

Following are brief descriptions for the most generally useful character manipulation routines. For specific information, see the Software Tools Subsystem Reference Manual.

Equal. 'Equal' is an integer function that takes two EOS-terminated strings as arguments. If the two strings are identical, 'equal' returns YES; otherwise it returns NO. For example,

```
string dash_x "-x"
integer equal
...
if (equal (argument, dash_x) == YES)
   call cross_ref
```

Index. 'Index' is used to find the position of a character in an EOS-terminated string. If the character is in the string, its position is returned, otherwise zero is returned. 'Index' is very similar to the built-in function of the same name in PL/I. Example:

```
string options "acx"
integer ndx
integer index
...
ndx = index (options, opt_character)
select (ndx)
   when (1)
      call list_all
   when (2)
      call list_common
   when (3)
      call cross_reference
else
   call remark ("illegal option"p)
```

This example selects one of a number of subroutines to be executed depending on a single-character option specifier. Of course, this particular example could be done with just **select** alone. 'Index' is also useful in character transliteration and conversion from character to binary integer.

Length. 'Length' is an integer function that returns the length of an EOS-terminated string. The length of a string is zero if and only if its first character is an EOS; it is the number of characters before the EOS in all other cases. 'Length' is often useful in deciding where to start appending additional text, as in the following example:

```
integer len
integer length
...
len = length (str)
call scopy (new_str, 1, str, len + 1)
```

Mapdn and Mapup. These functions accept a single character as an argument and if the character is alphabetic, force it to lower or upper case, respectively. 'Mapdn' and 'mapup' quite often find use in mapping option letters to a single case before comparison. Since non-alphabetic characters are not modified, these routines may be used safely even if non-alphabetic characters appear. In addition, these routines provide a very good place to isolate character set dependencies. For example,

```
character c
character mapdn
...
if (mapdn (c) == 'a'c) {
   # handle 'a' option
...
else if (mapdn (c) == 'l'c) {
   # handle 'l' option
```

Mapstr. 'Mapstr' provides case mapping for alphabetic characters in EOS-terminated strings. As arguments 'Mapstr' takes a string and the symbolic constant 'LOWER' or 'UPPER'. Alphabetic characters in the string are then forced to lower or upper case, depending on the constant specified.

Scopy. The subroutine 'scopy' is used for copying EOS-terminated strings. It requires four arguments: the source string, the position from which to start copying, the destination string, and the position at which filling begins in the destination string. Since Ratfor provides no string assignment, 'scopy' is normally used to provide the capability. The simple movement of a string from one place to another is coded as

```
character strl (MAXLINE), str2 (MAXLINE)
...
call scopy (strl, 1, str2, 1)
```

'Scopy' is also capable of appending one string to another, as in the following example:

```
character strl (MAXLINE), str2 (MAXLINE)
...
call scopy (strl, 1, str2, length (str2) + 1)
```

Note that 'scopy' makes no attempt to avoid writing past the end of 'str2'!

Type. 'Type' is another of the routines that is intended to isolate character dependencies. Type is a function that takes a single character as an argument. If that character is a letter, 'type' returns the constant 'LETTER'; if the character is a digit, 'type' returns the constant 'DIGIT'; otherwise, 'type' returns the character. 'Type' often finds use in a lexical analyzer:

```
character c
character type

if (type (c) == LETTER) {
   # collect identifier
...
else if (type (c) == DIGIT) {
   # collect integer
...
else {
   # handle special character
```

File Access

File access is one of the more important aspects of the Subsystem. It is through the Subsystem i/o routines that device indepedence and i/o redirection are accomplished; moreover, the Subsystem routines provide a much less complicated interface than comparable Primos routines.

The basic method of access to a Subsystem file is through the contents of an integer variable called a **file descriptor.** File descriptors can be set by one of several routines or they can be set to one of the six standard descriptors representing the six standard ports provided to all Subsystem programs.

Quite often, the standard ports provide all of the file access required by a program. Values for the standard port descriptors can be accessed from **defines** contained in "=incl=/swt_def.r.i" ('Rp' automatically includes this file in each run). The following table gives the symbolic names for the three standard input and three standard output ports available:

Input Ports	Output Ports
STDIN1 (or STDIN)	STDOUT1 (or STDOUT)
STDIN2	STDOUT2
STDIN3 (or ERRIN)	STDOUT3 (or ERROUT)

These constants may be used wherever a file descriptor is required by a Subsystem i/o routine.

Other files may be accessed or created through the routines 'open', 'create', and 'mktemp' that are described later. At the moment, it is sufficient to say that these routines are functions that return a file descriptor that may be used in other Subsystem i/o calls.

Once a file descriptor has been obtained, the file it references may be read with the routines 'getlin', 'getch', or 'input'; written with the routines 'putlin', 'putch', or 'print'; positioned with the routines 'wind' or 'rewind'; or closed with the routines 'close' or 'rmtemp'.

Open and Close. 'Open' takes an EOS-terminated path name and a mode (one of the constants READ, WRITE, or READWRITE) as arguments and returns the value of a file descriptor or the symbolic constant ERR as a function value. 'Open' is normally used to make a file available for processing in the specified mode. If the mode is READ, 'open' will open the file for reading; if the file doesn't exist or cannot be read (i.e. no read permission), 'open' will return ERR. If the mode is WRITE or READWRITE, 'open' will open an existing file or create a new file for writing or reading and writing, if possible; otherwise it will return ERR. If 'open' opens an existing file, it will never destroy the contents, even if mode is WRITE. To be certain that a "new" file is empty, use 'create' instead of 'open'.

'Close' takes a file descriptor as its argument; it closes and releases the file attached to the descriptor. If 'close' is called with a standard port, it takes no action.

Opening and closing a file is really very easy. This example opens a file named "=extra=/news/index" and returns the file descriptor in 'fd'. If the file can't be opened, the program will terminate with a call to 'cant'.

```
file_des fd
integer open
string fn "=extra=/news/index"

fd = open (fn, READ) # open "=extra=/news/index"
if (fd == ERR)
   call cant (fn)

<process the contents of =extra=/news/index>

call close (fd)      # release the file
stop
```

If the file can't be opened, 'cant' will print the message

```
=extra=/news/index: can't open
```

and terminate the program.

Create. 'Create' takes the same arguments as 'open', but also truncates the file (makes it empty) to be sure that there are no remnants of its previous contents.

Mktemp and Rmtemp. Quite often, programs need temporary files for their internal use only. 'Mktemp' and 'rmtemp' allow the creation of unique temporaries in the directory "=temp=". 'Mktemp' requires only a mode (READ, WRITE, or READWRITE) as an argument and returns a file descriptor as its function value. 'Rmtemp' takes a file descriptor as its argument and destroys and closes the temporary file. (One should use caution, for if a descriptor for a permanent file is passed to 'rmtemp', that file will also be destroyed.)

Typical use of 'mktemp' and 'rmtemp' usually involves the writing and reading of an intermediate file:

```
file_des fd
integer mktemp

fd = mktemp (READWRITE) # create a temporary file

<code to write the intermediate file>

call rewind (fd)        # reposition the temporary

<code to read the intermediate file>

call rmtemp (fd)        # close and destroy the temporary
```

Wind and Rewind. The subroutines 'wind' and 'rewind' allow the positioning of an open file to its end and beginning, respectively. Both take a file descriptor as an argument. Usually, 'rewind' is used when a program creates a file and then wishes to read it back; 'wind' is often used when a program wants to add to the end of an existing file.

A program wishing to extend a file would make a call to 'wind' just after successfully opening the file to be extended:

```
file_des fd
integer open
string fn "myfile"

fd = open (fn, READWRITE)
if (fd == ERR)
   call cant (fn)
call wind (fd)     # file is now positioned at the
                   #    end, ready for appending.
```

Trunc. 'Trunc' truncates an open file. Truncating a file means releasing all of its disk space, hence making it empty, but retaining its name and attributes. 'Trunc' takes a file descriptor as its argument.

Remove. 'Remove' removes a file by name, deleting it from the disk directory. It takes an EOS-terminated string as its argument, and returns the constant OK or ERR, depending on whether or not it could remove the file. ('Remove' will also delete a Primos segment directory without complaining.)

Cant. 'Cant' is a handy routine for handling exceptions when opening files. For its argument, 'cant' takes an EOS-terminated string containing a file name. It prints the message

```
<file name>: can't open
```

and then terminates the program.

Getlin. All Subsystem character input is done through 'getlin'. 'Getlin' takes a character array (at least MAXLINE long) and a file descriptor and returns a line of input in the array as an EOS-terminated string. Although the last character in the string is normally a NEWLINE character, if the line is longer than MAXLINE, no NEWLINE will be present and the rest of the line will be obtained on the next call to 'getlin'. For its function value, 'getlin' returns the length of the line delivered, (including the NEWLINE, if any) or the constant EOF if end-of-file was encountered.

Most line-oriented i/o is done with 'getlin'. For instance, using 'getlin' with its analog 'putlin', a program to select only those lines beginning with the letter "a" can be written very quickly:

```
character buf (MAXLINE)
integer getlin

while (getlin (buf, STDIN) ~= EOF)
   if (buf (1) == 'a'c)
      call putlin (buf, STDOUT)
```

'Getlin' is guaranteed to never return a line longer than the symbolic constant MAXLINE (including the terminating EOS).

If needed, there are a number of routines that you can call to convert the character string returned by 'getlin' into other formats, such as integer and real. Most of these routines are described later in the section on "Type Conversion".

Getch. 'Getch' returns one character at a time from a file; it requires a character variable and a file descriptor as arguments; it returns the character obtained, or the constant EOF, in the supplied argument and as the function value. Calls to 'getch' and 'getlin' may be interleaved; 'getlin' will pick up the rest of a line not read by 'getch'.

'Getch' is very useful in lexical analyzers or just when counting characters. For instance, the following routine counts both characters and lines at the same time:

```
character c
integer c_count, l_count
integer getch

c_count = 0
l_count = 0
while (getch (c, STDIN) ~= EOF) {
   c_count = c_count + 1
   if (c == NEWLINE)
      l_count = l_count + 1
   }
```

This example assumes that since each line ends with a NEWLINE character, lines can be counted by counting the NEWLINEs.

Input. 'Input' is a rather general routine created to provide easy access to both interactive and file input. For interactive input, 'input' will prompt at the terminal, accept input, and call the proper conversion routines to produce the desired data formats. In case of unexpected input (like letters in an integer), it will ask for a line to be retyped. For file input, 'input' recognizes that its input is not coming from a terminal (even if from a standard port) by turning off all prompting. It will then accept fixed or variable-length fields from the file under control of the format string.

'Input' requires a variable number of arguments: a file descriptor, a format string, and as many destination fields as required by the format string. It returns the constant EOF as its function value if it encountered end-of-file; otherwise it returns OK.

The file descriptor passed to 'input' describes the file to be read. All prompting output (if any) always appears on the terminal. The format string passed to 'input' indicates what prompting information is to be output and what data format to expect as input. Prompts to be output are specified as literal characters; i.e. to output "Input X:", the characters "Input X:" would appear in the format string. Prompting characters may only appear at the beginning of the string and immediately after "skip-newline" ("*n") format codes. Data items to be input are described by an asterisk followed by optionally one or two numbers and a letter. For instance the code to input a decimal integer would be "*i" and the code to input a double precision floating point number would be "*d".

When a call to 'input' is executed, the format string is interpreted from left to right. When leading literal characters are encountered, they are output as a prompt. When the first format code is encountered, a line is read from the file, the corresponding item is obtained from the input line, and the item is placed in the next item in the argument list. More items are removed from the input line until the end of the format string is reached or a newline appears in the input. If the end of the format string is encountered, the rest of the input line is discarded, and 'input' returns OK. Otherwise, if a newline is encountered in the input, fields designated by the format are filled with empty strings, blanks, or zeroes, until the format string is exhausted, or a code ("*n") to skip the NEWLINE and read a new line is encountered.

The format string must contain exactly as many input indicators as there are receiving data items in the call. In any case, the maximum number of input items per call is 10.

Before we go any further, here is an example of an 'input' call to obtain three integers:

```
call input (STDIN, "Type i:  *i*nType j:  *i*nType k:  *i"s,
   i, j, k)
```

If this statement were executed the following might appear at the terminal (user input is boldfaced):

```
Type i:  22 <newline>
Type j:  476 <newline>
Type k:  1 <newline>
```

We could also type all three integers on the same line, and 'input' would omit the prompting for the second and third numbers:

Type i: **22 476 1 <newline>**

There are a number of input indicators available for use in the format string. Since there are a large number of them with many available options, only a few are mentioned in the following table. For further information, see the Subsystem reference manual.

Item	Data Type	Input Representation
*n	skip newline	If there is a NEWLINE at the current position, skip over it and read another line. Otherwise do nothing. ('Input' will never read more than one line per call, unless this format code is present.
*i	16 bit integer	Input an integer with optional plus or minus sign, followed by a string of digits, delimited by a blank or newline. Leading blanks are ignored. The input radix can be changed by preceding the number with "<radix>r" (e.g. octal should be expressed by "8r").
*l	32 bit integer	Same as "*i".
*r	32 bit real	Input a real number with optional plus or minus sign, followed by a possible empty string of digits, optionally followed by a decimal point and a possibly empty string of digits. Scaling by a power of 10 may be indicated by an "e" followed by an optional plus or minus sign, followed by a string of digits. The number is delimited by a blank; leading blanks are ignored.
*d	64 bit real	Same as "*r".
*s	string	Input a string of characters delimited by a blank or newline. No more than MAXLINE characters will be delivered, regardless of input size.

Fixed size input fields can be requested by placing the desired field size immediately following the asterisk in the format code. For instance, to read three integers requiring five spaces each, you can use the following format string:

```
"*5i*5i*5i"
```

You can also change the delimiting character of a field from its default value of a blank. Just place two commas followed by the new delimiter immediately after the asterisk. For instance, two strings delimited by slashes can be input with the following format string:

```
*,,/s*,,/s
```

Regardless of the delimiter setting, a newline is always treated as a delimiter. One caution: if the delimiter is not a blank, leading blanks in strings are not ignored.

Readf. You can use 'readf' to read binary (memory-image) files that were created with 'writef'. 'Readf' is the fastest way to read files, since no data conversion is performed. However, use of 'readf' and 'writef' tend to make a program dependent on machine word size, and hence, non-portable.

'Readf' takes three arguments: a receiving data array, the maximum number of words to be read, and a Subsystem file descriptor. When called, 'readf' attempts to read the number of words requested; if there are not that many in the file, it returns all that are left. If there are no words left in the file at all, 'readf' returns EOF as its function value; otherwise, it returns the number of words actually read as its function value.

Putlin. 'Putlin' is the primary output routine of the Subsystem. It takes an EOS-terminated string and a file descriptor as arguments, and writes the characters in the string on the file specified by the descriptor. There is no restriction on the length of the input string; 'putlin' will write characters until it sees an EOS. 'Putlin' **does not** supply a newline character at the end of the line; if one is to be written, it must appear in the string. For a simple example, see the description of 'getlin'.

Putch. A single character can be output to a file with 'putch'; it takes a character and a file descriptor as arguments and writes the character on the file specified by the descriptor. Calls to 'putch' and 'putlin' can be interleaved as desired.

Print. 'Print' is a general output routine that accepts a format string and up to ten output data items. Interpreting the format string, 'print' calls the appropriate type conversion routines to produce character data, and outputs the characters as directed by the format string. 'Print' requires several arguments: a file descriptor; an EOS-terminated format string; and zero to ten output data arguments, depending on how many are required by the format string.

The format string contains two kinds of items: literal items which are output when they are encountered, and output items, which cause the next data argument to be converted to character format and output. Literal items are just characters in the string; i.e. to output "X =", the format string would contain "X =". Output items consist of an asterisk, followed by two optional numbers, followed by a letter. For instance an output item for an integer is "*i" and an output item for single precision floating point is "*r". The next example shows the output of three integers:

```
call print (STDOUT, "i = *i, j = *i, k = *i*n"s,
   i, j, k)
```

If this call were executed, the following might be the result:

```
i = 342, j = 1, k = -3382
```

Some of the more useful output items are described in the following table:

Item	Data Representation
*i	short (16 bit) integer
*l	long (32 bit) integer
*r	single precision (32 bit) real
*d	double precision (64 bit) real
*p	packed, period-terminated string
*s	EOS-terminated string
*c	single character
*n	newline

It is possible to exert much more control over the format of output using 'print'; for more information, see the Subsystem reference manual.

Writef. 'Writef' is the companion routine to 'readf'; it writes words to a binary (memory-image) file. It is the fastest of the output routines, since it performs no data conversion. It is called with three arguments: a data array containing the words to be written, the number of words to write, and a Subsystem file descriptor. Here is an example fast file-to-file copy using 'readf' and 'writef' together.

```
integer l, buf (1024)
integer readf
file_des in_fd, out_fd

repeat {
   l = readf (buf, 1024, in_fd)
   if (l == EOF)
      break
   call writef (buf, l, out_fd)
   }
```

Fcopy. 'Fcopy' is a very simple routine that copies files. You open and position the input and output files and call 'fcopy' with the input and output file descriptors. It then copies lines from the input file to the output file. 'Fcopy' uses a great deal of "secret knowledge" of the workings of the Subsystem input-output routines, and as a consequence, it copies disk-file to disk-file very quickly (even when the descriptors are of standard ports).

Markf and Seekf. 'Markf' and 'seekf' are companion routines that implement random access on disk files. 'Markf' takes a file descriptor as argument and returns a "file_mark" (currently a 32-bit integer). 'Seekf' takes the file mark along with a file descriptor and sets the file pointer so that the file is positioned at the same place as when the "mark" was taken.

To be used portably, "markf" and "seekf" may only be used between calls to "readf" and "writef", or immedately after input or output of a newline character (i.e. at the ends of lines). In addition, a call to 'putlin' or 'putch' on a file effectively (although not actually) destroys information following the current position of the file. For example, if you want to write a line in a file, go off and do other operations on the file, and then be able to re-read the line later, you can use 'markf' and 'seekf':

```
file_mark fm
file_mark markf
file_des fd
character line (MAXLINE)

fm = markf (fd)
call putlin (line, fd)

### perform other operations on 'fd'

call seekf (fm, fd)
call getlin (line, fd)  # get 'line' back
```

Non-portably, you can assume that a "file mark" is a zero-relative word number within the file -- to get word number 12 in the file, just execute

```
call seekf (intl (12), fd)
call readf (word, 1, fd)
```

(Remember: file marks are 32 bits, not 16! We use 'intl' here to make "12" into a 32 bit integer.) Keep in mind that this "secret knowledge" is useful only with "readf" and "writef", not with any other input or output routine. Blank compression is used in line oriented files, so the position of a line is dependent not only on length of previous lines, but also on their content. This usually makes the position of a line in a file quite unpredictable.

Getto. 'Getto' exists primarily to interface with the Primos file system calls. 'Getto' takes a path name (in an EOS-terminated string) as its first argument. It follows the path and sets the current directory to that specified for the file in the path name. It then packs the file name into its second argument, a 16 word array (with blank padding), ready for a call to the Primos file system. It fills its 3-word third argument with the password of the last node of the path (if there was one). Its fourth argument, an integer, is set to YES if 'getto' changed the attach point, and NO otherwise.

'Getto' often finds use when functions other than those supported by Subsystem routines need to be performed, such as setting the passwords on a directory:

```
integer pfn (16), opw (3), npw (3), pw (3), att
integer getto
string fn "=vars=/system"

if (getto (fn, pfn, pw, att) == ERR)
   call print (ERROUT, "can't get to *s*n.", fn)
call spas$$ (pfn, 32, opw, npw) # set passwords
if (att == YES)
   call follow (EOS, 0)     # attach back to home
```

Type Conversion

There are a very large number of type conversion routines available to convert most data types into character strings and back. Because keeping up with all the conversion routine names and calling sequences can be quite a chore, two routines 'decode' and 'encode' exist to handle conversion details in a consistent format. These two routines are described at the end of this section.

Most of the "character-to-something" routines require at least two arguments. The first argument is usually the character string, and the second is an integer variable indicating the first of the characters to be converted. The result of conversion is then returned as the function value, and the position variable is updated to indicate the first position past the characters used in the conversion.

For example, the simplest "character-to-integer" routine, 'ctoi' requires the two arguments mentioned above. Since it skips leading blanks, but stops at the first non-digit character, it can be called several times in succession to grab several blank-separated integers on a line:

```
character str (MAXLINE)
integer i, k (4), pos
integer ctoi
...
pos = 1
do i = 1, 4
   k (i) = ctoi (str, pos)
if (str (pos) ~= EOS)
   call remark ("illegal character in input"p)
```

This routine will assume unspecified values to be zero, but complain if non-numeric, non-blank characters are specified.

Here is a list of all of the currently supported "character-to-something" routines.

ctoc Character-to-character; copies character strings and pays attention to the maximum length parameter.

ctod Character-to-double precision real; handles general floating point input.

ctoi Character-to-integer (16 bit); does not handle plus and minus signs; decimal only.

ctop Character-to-packed-string; converts to packed format with no delimiter character.

ctor Character-to-single precision real; handles general floating point input.

ctov Character-to-PL/I-character-varying; converts to PL/I character varying format.

gctoi Generalized-character-to-integer (16 bit); handles plus and minus signs; in addition to program-specified radix, accepts an optional user-specified radix from 2-16.

gctol Generalized-character-to-long-integer (32 bit); handles plus and minus signs; in addition to program-specified radix, accepts an optional user-specified radix from 2-16.

In addition to the "character-to-something" routines, there are the "something-to-character" routines. Most of these routines require three arguments: the value to be converted, the destination string, and the maximum size allowable. They return the length of the string produced as the function value. An EOS is always placed in the position following the last character in the destination string, but the EOS is not included when the size of the returned string is calculated.

Since the functions will accept a sub-array reference for the output string, you may place several objects in the same string. For example, using the "integer-to-character" conversion routine 'itoc', you can place the four integers in the array 'k' into 'str' in character format:

```
character str (MAXLINE)
integer i, k(4), pos
integer itoc
...
pos = 1
do i = 1, 4; {
   pos = pos + itoc (k (i), str (pos), MAXLINE - pos)
   if (pos >= MAXLINE - 1) # there's no room for any more
      break
   str (pos) = BLANK
   pos = pos + 1
   }
str (pos) = EOS   # cover up the last blank
```

This code will place the four integers in 'str', separated by a single blank. Although all conversion routines leave an EOS in the string, we have to replace it here because we clobber it with the blank.

It's worth noting that the maximum size parameter always includes the EOS -- the conversion routine will never touch any more characters than are specified by this parameter.

Here is a list of all available "something-to-character" conversion routines:

ctoc Character-to-character; copies character strings and pays attention to the

maximum length parameter.

dtoc — Double-precision-real-to-character; handles general floating point conversions in Basic or Fortran formats.

gitoc — Generalized-integer-to-character (16 bit); handles integer conversions; program-specified radix.

gltoc — Generalized-long-integer-to-character (32 bit); handles long integer conversion; program specified radix.

itoc — Integer-to-character (16 bit); handles integer conversion; decimal only.

ltoc — Long-integer-to-character (32 bit); handles long integer conversion; decimal only.

ptoc — Packed-string-to-character; accepts arbitrary delimiter character; will unpack fixed length strings if delimiter is set to EOS and maximum is set to (length + 1).

rtoc — Single-precision-real-to-character; handles general real conversion in Basic or Fortran formats.

vtoc — PL/I-character-varying-to-character; converts PL/I character varying format to character.

Decode. 'Decode' handles conversion from character strings to all other formats. It is written to be used in concert with 'getlin' and other such routines, and as such, has a rather odd calling sequence. It requires a minimum of five arguments: the usual string, and string index; a format string; a format string index and an argument string index. Following are receiving arguments, depending on the data types specified in the format string. In almost all cases, you should just supply variables with a values of 1 for the format index and the argument index. The string index behaves just as it does in all other character-to-something routine -- on successful conversion, it points to the EOS in the string. The specifics of the format string and receiving fields are identical to 'input'. The only differences are that 'decode' returns with OK in the situations in which 'input' would read another line of input, and EOF otherwise, and that all characters in the format string that are not format codes are ignored.

Encode. 'Encode' is a companion routine to 'decode': it can access all of the something-to-character conversion routines in a consistent way. For arguments it takes a character string, maximum length of the string, a format string, and a varying number of source arguments, depending on the format string. 'Encode' behaves exactly like 'print', except that it puts the converted characters into the string, rather than putting them onto a file.

Argument Access

Programs often find it necessary to access arguments specified on the command line. These arguments can be obtained as EOS-terminated strings, ready for processing or passing to a routine such as 'open'.

Getarg. 'Getarg' is the only routine that retrieves arguments from the shell's argument buffer. It is called with three arguments: an integer describing the position of the argument desired, a character array to receive the argument, and an integer describing the maximum size of the receiving array. 'Getarg' tries to retrieve the argument in the specified position; if it can, it returns the length of the string placed in the array; if it can't, it returns the constant EOF. 'Getarg' will never write farther in the character array than the size specified in the third argument.

Arguments are numbered 0 through the maximum specified on the command line. Argument 0 is the name of the command, argument 1 is the first argument specified, and so on. The number of arguments present on the command line can be determined by the point at which 'getarg' returns EOF.

As a short example, here is a program fragment that attempts to delete all files specified as arguments on its command line:

```
character file (MAXLINE)
integer i
integer remove, getarg

i = 1
while (getarg (i, file, MAXLINE ~= EOF)) {
   if (remove (file) == ERR)
      call print (ERROUT, "*s: cannot remove*n"s,
         file)
   i = i + 1
   }
```

Dynamic Storage Management

Dynamic storage subroutines reserve and free variable size blocks from an area of memory. In this implementation, the area of memory is a one-dimensional array. Each block consists of consecutive words of that array.

The dynamic storage routines assume that you have included the following declaration in your main program and in any subprograms that reference dynamic storage:

```
DS_DECL (mem, MEMSIZE)
```

where 'mem' is an array name that can be used to reference the dynamic storage area. You must also define MEMSIZE to an integer value between 6 and 32767 inclusive. This number is the maximum amount of space available for use by the dynamic storage routines. In estimating for the amount of dynamic storage required, you must allow for two extra 'overhead' words for each block allocated. Three other overhead words are required for a pointer to the first available block of memory and to store the value of MEMSIZE.

Dsinit. The call

```
call dsinit (MEMSIZE)
```

initializes the storage structure's pointers and sets up the list of free blocks. This call must be made before any other references to the dynamic storage area are made.

Dsget. 'Dsget' allocates a block of words in the storage area and returns a pointer (array index) to the first useable word of the block. It takes one argument -- the size of the block to be allocated (in words).

After a call to 'dsget', you may then fill consecutive words in the 'mem' array beginning at the pointer returned by 'dsget' (up to the number of words you requested in the block) with whatever information called for by your application. If you should write more words to the block than you allocated, the next block will be overwritten. Needless to say, if this happens you may as well give up and start over.

If 'dsget' finds that there is not enough contiguous storage space to satisfy your request, it prints an error message, and if you desire, calls 'dsdump' to give you a dump of the contents of the dynamic storage array.

Dsfree. A call to 'dsfree' with a pointer to a block of storage (obtained from a call to 'dsget') deallocates the block and makes it available for later use by 'dsget'. 'Dsfree' will warn you if it detects an attempt to free an unallocated block and give you the option of terminating or continuing the program.

Dsdump. The dynamic storage routines cannot check for correct usage of dynamic storage. Because block sizes and pointers are also stored in 'mem' it is very easy for a mistake in your program to destroy this information. 'Dsdump' is a subroutine that can print the dynamic storage area in a semi-readable format to assist in debugging. It takes one argument: the constant LETTER for an alphanumeric dump, or the constant DIGIT for a numeric dump.

The following example shows the use of the dynamic storage routines and uses 'dsdump' to show the changes in storage that result from each call.

```
define (MEMSIZE, 35)

pointer pos1, pos2   # pointer is a subsystem defined type
pointer dsget
DS_DECL (mem, MEMSIZE)

call dsinit (MEMSIZE)
call dsdump (LETTER) # first call

pos1 = dsget (4)
call scopy ("aaa"s, 1, mem, pos1)
call dsdump (LETTER) # second call

pos2 = dsget (3)
call scopy ("bb"s, 1, mem, pos2)
call dsdump (LETTER) # third call

call dsfree (pos2)
call dsdump (LETTER) # fourth call

stop
end
```

The first call to 'dsdump' (after 'init') produces the following dump:

```
* DYNAMIC STORAGE DUMP *
    1   3 words in use
    4   32 words available
* END DUMP *
```

The first three words are used for overhead, and 32 (MEMSIZE - 3) words are available starting at word four in 'mem'.

The second call to 'dsdump' (after the first write to dynamic storage) produces the following:

```
* DYNAMIC STORAGE DUMP *
    1   3 words in use
    4   26 words available
   30   6 words in use
          aaa
* END DUMP *
```

Note that only four characters were written, three a's and an EOS (an EOS is a nonprinting character), but two extra control words are required for each block. That block is comprised of words 30 - 35 in the array 'mem'.

The third call to 'dsdump' (after the second 'scopy') produces the following:

```
* DYNAMIC STORAGE DUMP *
    1   3 words in use
    4   21 words available
   25   5 words in use
          bb
   30   6 words in use
          aaa
* END DUMP *
```

The final call to 'dsdump' produces:

```
* DYNAMIC STORAGE DUMP *
    1   3 words in use
    4   26 words available
   30   6 words in use
          aaa
* END DUMP *
```

As you can see, the second block of storage that began at word 25 has been returned to the list of available space.

Symbol Table Manipulation

Symbol table routines allow you to index tabular data with a character string rather than an integer subscript. For instance, in the following table, the information contained in "field1", "field2", and "field3" can obtained by specifying a certain key value (e.g.

"firstentry").

key	field1	field2	field3
firstentry	10268	data	u
secondentry	27043	moredata	a

All Subsystem symbol table routines use dynamic storage. Therefore, the declarations and initialization required for dynamic storage are also required for the symbol table routines; namely:

```
DS_DECL (mem, MEMSIZE)
...
call dsinit (MEMSIZE)
```

where 'mem' is an array name that can be used to reference the dynamic storage area, and MEMSIZE is a user-defined identifer describing how many words are to be reserved for items in dynamic storage. MEMSIZE must be a integer value between 6 and 32767 inclusive. For a discussion on how to estimate the amount of dynamic storage space needed in a program, you can refer back to the section on the dynamic storage routines.

A symbol table entry consists of two parts: an identifier and its associated data. The identifier is a variable length character string; it is dynamically created when the symbol is entered into a symbol table. The data associated with the symbol is treated as a fixed-length array of words to be stored or modified when the assocated symbol is entered in the table and returned when the symbol is looked up. The size of the data is fixed for each symbol table -- each entry in a table must have associated data of the same size, but different symbol tables may have different lengths of data.

Mktabl. A symbol table is created by a call to the pointer function 'mktabl' with a single integer argument giving the size of the associated data array or the "node size". 'Mktabl' returns a pointer to the symbol table in dynamic storage. This returned pointer identifies the symbol table -- you must pass it to the other symbol table routines to identify which table you want to reference. A symbol table is relatively small (each table requires about 50 words, not counting the symbols stored in it), so you may create as many of them as you like (as long as you have room for them).

In the table above, if "field1" and "field3" require one word each, and "field2" requires no more than 9 words, then you can create the symbol table with the following call:

```
pointer extable
...
extable = mktabl (11)
```

The argument to 'mktabl' is 11 -- the total length of the data to be associated with each symbol.

Enter. To enter a symbol in a symbol table, you must provide two items: an EOS-terminated string containing the identifier to be placed in the table, and an array containing the data to be associated with the symbol. Of course this array must be at least as large as the "nodesize" declared when the particular symbol table was created. A call to the subroutine 'enter' with the identifier, the data array, and the symbol table pointer will make an entry in the symbol table. However, if the identifier is already in the table, its associated data will be overwritten by that you've just supplied. It is not possible to have the same identifier in the same symbol table twice.

Now, continuing our example, to enter the first row of information in the table, you can use the following statements:

```
info (1) = 10268
call scopy ("data"s, 1, info, 2)
info (11) = 'u'c
call enter ("firstentry"s, info, extable)
```

Lookup. Once you've made an entry in the symbol table, you can retrieve it by supplying the identifier in an EOS-terminated string, an empty data array, and the symbol table pointer to the function 'lookup'. If 'lookup' can find the identifier in the table, it will fill in your data array with the data it has stored with the symbol and return with YES for its function value. Otherwise, it will just return with NO as its function value.

In our example, to access the data associated with the "firstenty" we can make the following call:

```
foundit = lookup ("firstentry"s, info, extable)
```

After this call (assuming that "firstentry" was in the table), "foundit" would have the value YES, "info (1)" would have the value for "field1", "info (2)" through "info (10)" would have the value for "field2", and "info (11)" would have the value for "field3".

Delete. If you should want to get rid of an entry in a symbol table, you can make a call to the subroutine 'delete' with identifier you want to delete in an EOS-terminated string and the symbol table pointer. If the identifier you pass is in the table, 'delete' will delete it and free its space for later use. If the identifier is not in the table, then 'delete' won't do anything.

Using our example again, if you want to delete 'firstentry' from the table, you can just make the call

```
call delete ("firstentry"s, extable)
```

and "firstentry" will be removed from the table.

Rmtabl. When you are through with a table and want to reclaim all of its storage space, you pass the table pointer to 'rmtabl'. 'Rmtabl' will delete all of the symbols in the table and release the storage space for the table itself. Of course, after you remove a table, you can never reference it again.

To complete our example, we can get rid of our symbol table by just calling 'rmtabl':

```
call rmtabl (extable)
```

Sctabl. So far, the routines we've talked about have been sufficient for dealing with symbol tables. It turns out that there is one missing operation: getting entries from the table without knowing the identifiers. The need for this operation arises under many circumstaces. Perhaps the most common is when we want to print out the contents of a symbol table for debugging.

To use 'sctabl' to return the contents of a symbol table, you first need to initialize a pointer with the value zero. We'll call this the position pointer from now on. Then you call 'sctable' repeatedly, passing it the symbol table pointer, a character array for the name, a data array for the associated data, and the position pointer. Each time you call it, 'sctabl' will return another entry in the table: it will fill in the character string with the entry's identifier, fill in your data array with the entry's data, and update position in the position pointer. When there are no more entries to return in the table, 'sctabl' returns EOF as its function value.

There are two things you have to watch when using 'sctabl'. First, if you don't keep calling 'sctabl' until it returns EOF, you must call 'dsfree' with the position pointer to release the space. Second, you may call 'enter' to modify the value of a symbol while scanning a table, but you cannot use 'enter' to add a new symbol or use 'delete' to remove a symbol. If you do, 'sctabl' may lose its place and return garbage, or it may not return at all!

Here is a subroutine that will dump the contents of our example symbol table:

```
# stdump --- print the contents of a symbol table
   subroutine dptabl (table)
   pointer table

   integer posn
   integer sctabl
   character symbol (MAXSTR)
   untyped info (11)

   call print (ERROUT, "*4xSymbol*12xInfo*n"s)

   posn = 0
   while (sctabl (table, symbol, info, posn) ~= EOF)
      call print (ERROUT, "*15s|*6i|*9s|*c*n"s,
             symbol, info (1), info (2), info (9))

   return
   end
```

If make a call to 'stdump' after made the entry for "firstentry", it would print the following:

```
|              Symbol            Info
|           firstentry     | 10268|data      |u
```

Other Routines

There are a number of miscellaneous routines that provide often needed assistance. The following table gives their names and a brief description. For full information on their use, see the Subsystem reference manual:

	date	Obtain date, time, process id, login name
*	error	Print an error message and terminate
*	follow	Follow a path and set the current and/or home directories
	remark	Print a string followed by a newline
	tquit$	Check if the break key was hit
	wkday	Determine the day of the week of any date

Appendixes

Appendix A -- Implementation of Control Statements

This appendix contains flowcharts of the code produced by the Ratfor control statements along with actual examples of the code Ratfor produces.

In different contexts, a given sequence of Ratfor control statements can generate slightly different code. First, where possible, statement labels are not produced when they are not referenced. For instance, a **repeat** loop containing no **break** statements will have no "exit" label generated, since one is not needed. Second, **continue** statements are generated only when two statement numbers must reference the same statement. Finally, internally generated **goto** statements are omitted when control can never pass to them; e.g. a **when** clause ending with a **return** statement.

These code generation techniques make no fundamental difference in the control-flow of a program, but can make the code generated by very similar instances of a control statement appear quite different. Please keep in mind that the examples of Fortran code generated by 'rp' are included for completeness, and are not necessarily character-for-character descriptions of the code that would be obtained from preprocessing. Rather, they are intended to illustrate the manner in which the Ratfor statements are implemented in Fortran.

Break

Syntax:

```
break [<levels>]
```

Function:

Causes an immediate exit from the referenced loop.

Example:

```
for (i = length (str); i > 0; i = i - 1)
   if (str (i) ~= ' 'c)
      break
```

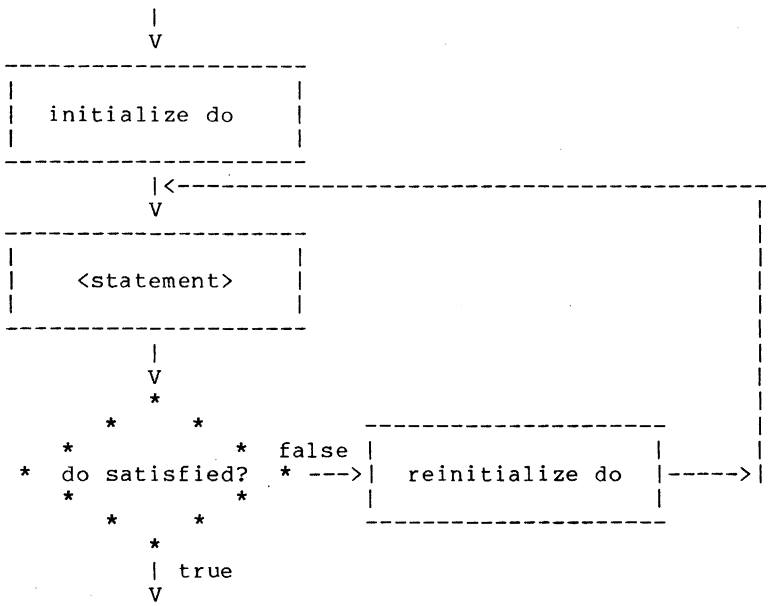
```
      i=length(str)
      goto 10002
10000 i=i-1
10002 if((i.le.0))goto 10001
        if((str(i).eq.160))goto 10003
          goto 10001
10003 goto 10000
10001 continue
```

Do

Syntax:

```
do <limits>
   <statement>
```

Function:



Example:

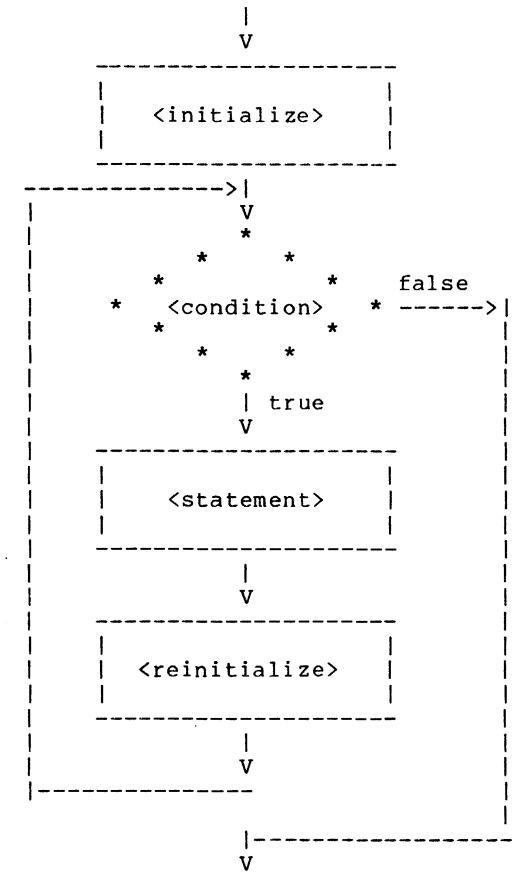
```
do i = 1, 10
   array (i) = 0


      do 10000 i=1,10
10000 array(i)=0
```

For

Syntax:

```
for ([<initialize>]; [<condition>]; [<reinitialize>])
   <statement>
```

Function:



Example:

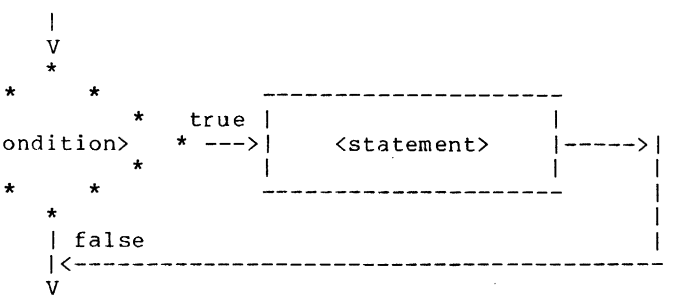
```
for (i = limit - 1; i > 0; i = i - 1) {
   array_1 (i) = array_1 (i + 1)
   array_2 (i) = array_2 (i + 1)
   }


      i=limit-1
      goto 10002
10000 i=i-1
10002 if((i.le.0))goto 10001
        array1(i)=array1(i+1)
        array2(i)=array2(i+1)
      goto 10000
10001 continue
```

If

Syntax:

```
if (<condition>)
   <statement>
```

Function:



Example:

```
if (a == b) {
   c = 1
   d = 1
   }
```

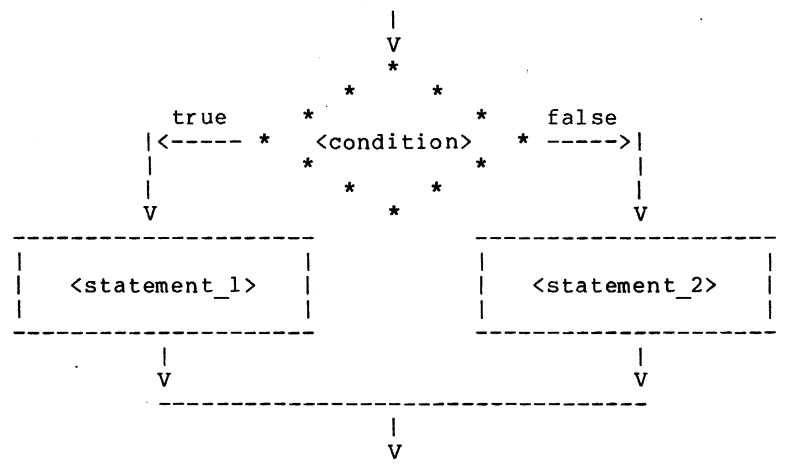
```
      if((a.ne.b))goto 10000
        c=1
        d=1
10000 continue
```

If - Else

Syntax:

```
if (<condition>)
   <statement_1>
else
   <statement_2>
```

Function:



Example:

```
if (i >= MAXLINE)
   i = 1
else
   i = i + 1


      if((i.lt.102))goto 10000
        i=1
        goto 10001
10000   i=i+1
10001 continue
```

Next

Syntax:

```
next [<levels>]
```

Function:

All loops nested within the loop specified by <levels> are terminated. Execution resumes with the next iteration of the loop specified by <levels>.

Example:

```
  # output only strings containing no blanks
for (i = 1; i <= LIMIT; i = i + 1) {
   for (j = 1; str (j, i) ~= EOS; j = j + 1)
      if (str (j, i) == ' 'c)
         next 2
   call putlin (str (1, i), STDOUT)
   }
```

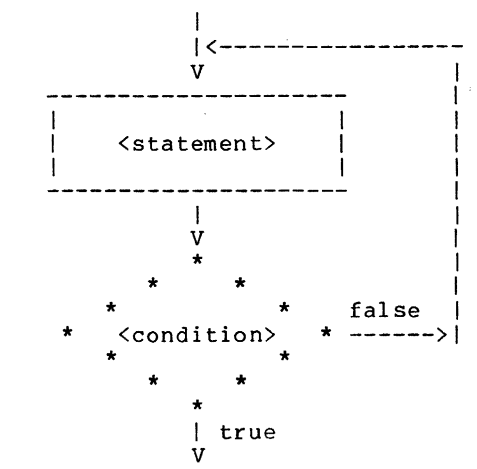
```
      i=1
      goto 10002
10000 i=i+1
10002 if((i.gt.50))goto 10001
        j=1
        goto 10005
10003   j=j+1
10005   if((str(j,i).eq.-2))goto 10004
          if((str(j,i).ne.160))goto 10006
            goto 10000
10006   goto 10003
10004   call putlin(str(1,i),-11)
      goto 10000
10001 continue
```

Repeat

Syntax:

```
repeat
   <statement>
   [until (<condition>)]
```

Function:



Example:

```
repeat {
   i = i + 1
   j = j + 1
   } until (str (i) ~= ' 'c)


10000   i=i+1
        j=j+1
      if((str(i).eq.160))goto 10000
```

Return

Syntax:

```
return [ '(' <expression ')' ]
```

Function:

Causes <expression> (if specified) to be assigned to the function name, and then causes a return from the subprogram.

Example:

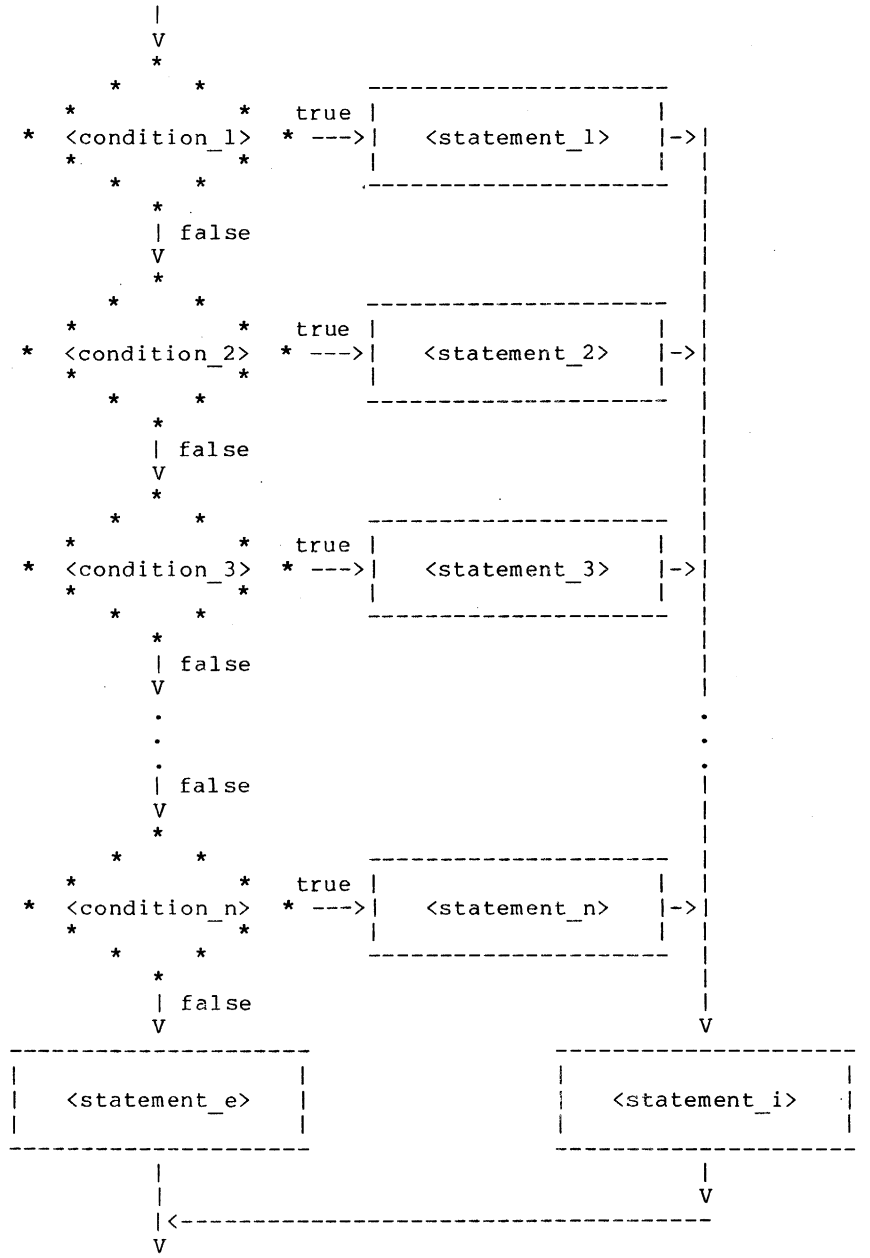
```
integer function fcn (x)
...
return (a + 12)

      integer function fcn (x)
      ...
      fcn=a+12
      return
```

Select

Syntax:

```
select
   when (<condition_1>)
      <statement_1>
   when (<condition_2>)
      <statement_2>
   when (<condition_3>)
      <statement_3>
         .
         .
         .
   when (<condition_n>)
      <statement_n>
   }
[ifany
   <statement_i>]
[else
   <statement_e>]
```

Function:



Example:

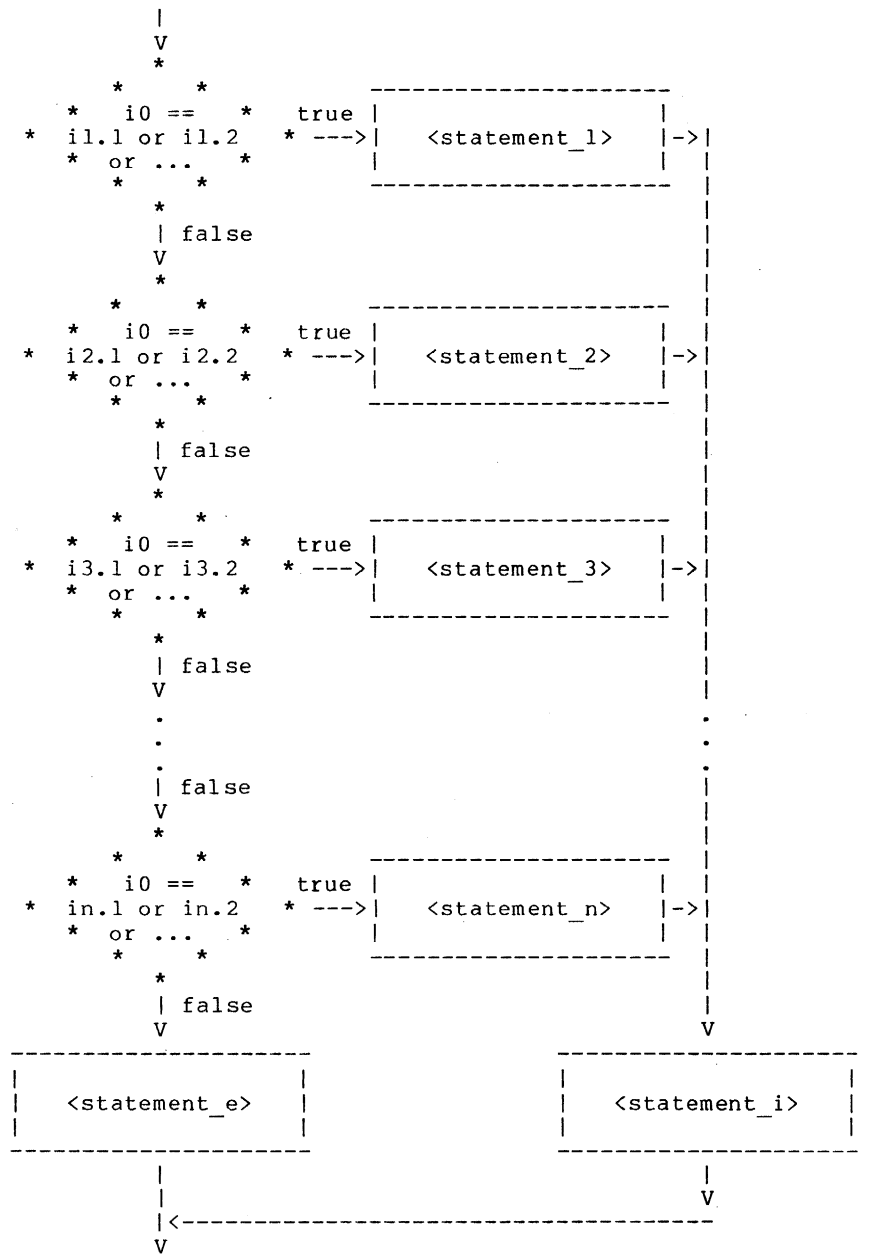
```
      select
         when (i == 1)
            call add_record
         when (i == 2)
            call delete_record
      else
         call code_error


            goto 10001
      10002   call addre0
            goto 10000
      10003   call delet0
            goto 10000
      10001 if((i.eq.1))goto 10002
            if((i.eq.2))goto 10003
              call codee0
      10000 continue
```

Select (<integer expression>)

Syntax:

```
select (<i0>)
   when (<i1.1>, <i1.2>, ...)
      <statement_1>
   when (<i2.1>, <i2.2>, ...)
      <statement_2>
   when (<i3.1>, <i3.2>, ...)
      <statement_3>
        .
        .
        .
   when (<in.1>, <in.2>, ...)
      <statement_n>
   }
[ifany
   <statement_i>]
[else
   <statement_e>]
```

Function:


i0 == i1.1 or i1.2 or ...
true
<statement_1>
false
i0 == i2.1 or i2.2 or ...
true
<statement_2>
false
i0 == i3.1 or i3.2 or ...
true
<statement_3>
false
i0 == in.1 or in.2 or ...
true
<statement_n>
false
<statement_e>
<statement_i>

Example:

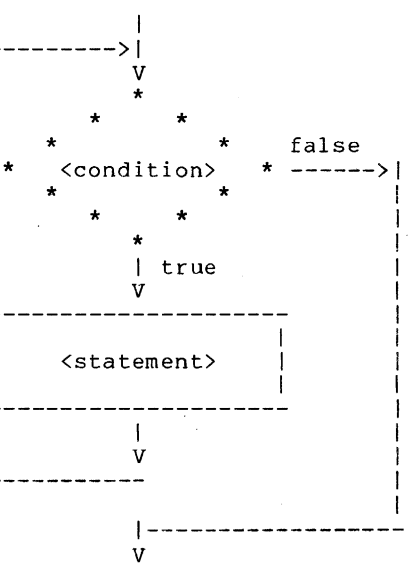
```
select (i)
   when (4, 6, 3003)
      call add_record
   when (2, 12, 5000)
      call delete_record
else
   call code_error


      integer aaaaa0,aaaab0
      ...
      aaaaa0=i
      goto 10001
10002   call addre0
      goto 10000
10003   call delet0
      goto 10000
10001 aaaab0=aaaaa0-1
      goto(10003,10004,10002,10004,10002,
     *     10004,10004,10004,10004,10004,
     *     10003),aaaab0
      if(aaaaa0.eq.3003)goto 10002
      if(aaaaa0.eq.5000)goto 10003
10004 continue
10000 continue
```

While

Syntax:

```
while (<condition>)
   <statement>
```

Function:



Example:

```
while (str (i) ~= EOS)
   i = i + 1
```

```
10000 if((str(i).eq.-2))goto 10001
        i=i+1
      goto 10000
10001 continue
```

Appendix B -- Linking Programs With Initialized Common

The Subsystem link procedure makes the assumption that all common areas are uninitialized to allow programs to access up to 27 64K word segments of data space. A program which uses initialized common areas requires a slightly modified link procedure to execute correctly. The program is also restricted to one segment (64K words) for both code and data space. If this limit is exceeded, no warning will be given, and unpredictable results will occur during execution. If more than 64K words of space is required, the common areas must be initialized at run time, or the program cannot be run under the Subsystem.

The modification to the link procedure is as follows: the option string "-s 'co ab 4000'" must appear on the 'ld' command line before the first binary file. For instance, if the file "prog.b" contained a program with **block data** statements, an 'ld' command to link it might appear as follows:

```
ld -s 'co ab 4000' prog.b
```

The executable program would be placed in the file "prog.o".

Appendix C -- Requirements for Subsystem Programs

This appendix gives the technical specifications of requirements for programs that run under the Subsystem. It is included to allow non-Ratfor programs to run under the Subsystem.

32S and 16S addressing modes

- There is no support for the execution of these addressing modes.

64R & 32R addressing modes

- The 64R mode library routines cannot access the Subsystem common areas, so 32R and 64R mode programs cannot execute under the current version of the Subsystem. However, in the near future, this restriction will be lifted.

64V addressing mode

- Segments '4035 and '4036 may not be disturbed.
- When a Subsystem program is executed, the stack is already constructed in segment '4035. However, the executing program may rebuild it if desired.
- The program must make a call to the Subsystem routine 'init' at the beginning of execution. A "call init" is inserted automatically at the beginning of each Ratfor main program.
- The program must terminate with a call to the Subsystem routine 'swt' at the end of its execution or its main program must return to its caller. A **stop** statement in Ratfor will be transformed into a call to 'swt'.
- The program must not tamper with any file units already open by the Subsystem. It should always use a Subsystem or Primos call to obtain an unused file unit.
- The program must be in a P300 format runfile. These runfiles may be produced by the 'seg' SHARE command.
- The program must have been loaded by the modified version of the segmented loader, 'swtseg', or the entry control block for the main program must be at location '1000 in segment '4000.
- The runfile must not expect any segment other than '4000 to be initialized before execution. Initialization of any other segment is the responsibility of the object program.
- The default load sequence produced by 'ld' will correctly link programs requiring up to 64K words of procedure (code) and linkage (initialized local data) frames. Up to 27 64K word segments may be used for uninitialized common blocks. Up to 64K words of local data may be allocated on the stack.

32I addressing mode

- No support is available for 32I mode programs (unless someone gives us a machine that supports 32I mode)

Appendix D -- The Subsystem Definitions

The file "=incl=/swt_def.r.i" contains Ratfor **define** statements for all the symbolic contants required to use the routines in the Subsystem support library. This appendix describes the more frequently used constants and the constraints placed on them.

Characters

Ascii Mnemonics. Character definitions for the Ascii control characters NUL, SOH, STX, ..., GS, RS, US, as well as SP and DEL.

Control characters. Character definitions for the Ascii control characters CTRL_AT, CTRL_A, CTRL_B, ..., CTRL_LBRACK, CTRL_BACKSLASH, CTRL_RBRACK, CTRL_CARET, and CTRL_UNDERLINE.

BACKSPACE Synonym for Ascii BS.

TAB Synonym for Ascii HT.

BELL Synonym for Ascii BEL.

RHT Relative horizonal tab character (used for blank compression in Primos text files).

RUBOUT Synonym for Ascii DEL.

Data Types

bits Bit strings (16 bit items).

bool Boolean (logical) values: .true. and .false. (16 bit items).

character Single right-justified zero-filled character (scalar), or a string of these characters terminated by an EOS (array).

file_des File descriptor returned 'open', 'create', etc.

file_mark File position returned by 'seekf'.

longint Double precision (32 bit) integer.

longreal Double precision (64 bit) floating point.

pointer Pointer for use with dynamic storage and symbol table routines.

Macro Subroutines

fpchar (<packed array>, <index>, <character>) Fetches <character> from <packed array> at character position <index> and increments <index>. The first character in the array is position zero.

spchar (<packed array>, <index>, <character>) Stores <character> in <packed array> at character position <index> and increments <index>. The first character in the array is position zero.

getc (<char>) Behaves exactly like 'getch', except the character is always obtained from STDIN.

putc (<char)) Behaves exactly like 'putch', except the character is always placed on STDOUT.

SKIPBL (<character array>, <index>) Increments <index> until the corresponding position in the character array is non-blank.

DS_DECL (<ds array name>, <ds array size>) Declares the dynamic storage array with the name <ds array name> with size <ds array size>.

Lanuage Extensions

ARB Used when dimensioning array parameters in subprograms (since their length is determined by the calling program, not the subprogram).

FALSE Represents the Fortran logical constant .false.

IS_DIGIT (<char>) Logical expression yielding TRUE if <char> is a digit.

IS_LETTER (<char>) Logical expression yielding TRUE if <char> is an upper or lower case letter.

IS_UPPER (<char>) Logical expression yielding TRUE if <char> is an upper case letter.

IS_LOWER (<char>) Logical exparession yielding TRUE if <char> is a lower case letter.

SET_OF_UPPER_CASE Sequence of 26 character constants representing the upper case letters for use in the **when** parts of **select** statements.

SET_OF_LOWER_CASE Sequence of 26 character constants representing the lower case letters for use in **when** parts of **select** statements.

SET_OF_LETTERS Sequence of 52 character constants representing the upper and lower case letters for use in **when** parts of **select** statements.

SET_OF_DIGITS Sequence of 10 character constants representing the digits for use in **when** parts of **select** statements.

SET_OF_CONTROL_CHAR Sequence of 32 character contants representing the first 32 Ascii control characters for use in **when** parts of **select** statements.

TRUE Represents the Fortran logical constant .true.

Limits

```
CHARS_PER_WORD Maximum number of packed characters per machine word.
MAXINT     Largest 16-bit integer.
MAXARG     Maximum length of a command line argument (EOS-terminated character string).
MAXCARD    Maximum input line length (excluding the EOS).
MAXDECODE  Maximum size of string processed by 'decode'.
MAXLINE    Maximum input line length.
MAXPAT     Maximum size of a pattern array.
MAXPATH    Maximum size of a Subsystem pathname.
MAXPRINT   Maximum number of character that can be output by a single call to 'print'.
MAXTREE    Maximum number of characters in a Primos tree name.
MAXFNAME   Maximum number of characters in a simple file name.
```

Standard Ports

```
STDIN      Standard input 1.
STDIN1     Standard Input 1.
STDIN2     Standard input 2.
ERRIN      Standard input 3.
STDIN3     Standard input 3.
STDOUT     Standard output 1.
STDOUT1    Standard output 1.
STDOUT2    Standard output 2.
ERROUT     Standard output 3.
STDOUT3    Standard output 3.
```

Argument and Return Values

```
ABS        Request absolute positioning ('seekf').
REL        Request relative positioning ('seekf').
DIGIT      Character is a digit ('type').
LETTER     Character is a letter ('type').
UPPER      Map to upper case ('mapstr').
LOWER      Map to lower case ('mapstr').
READ       Open file for reading.
WRITE      Open file for writing.
READWRITE  Open file for reading and writing.
EOF        End of file (guaranteed distinct from all characters and from OK and ERR).
OK         No error (guaranteed distinct from all characters and from EOF and ERR).
ERR        Error occurred (guaranteed distinct from all characters and from EOF and OK).
EOS        End of string (guaranteed distinct from all characters).
LAMBDA     Null pointer (guaranteed distinct from all pointer values).
YES        Affirmative response (guaranteed distinct from NO).
NO         Negative response (guaranteed distinct from YES).
```

Appendix E -- 'Rp' Reserved Words

The following identifiers are reserved keywords in Ratfor and cannot be used as identifiers. 'Rp' will not diagnose the use of reserved keywords as identifiers; results of misuse will be unreasonable behavior such as misleading error messages and mis-ordered Fortran code.

```
blockdata
break
call
case
common
complex
continue
data
define
dimension
do
doubleprecision
else
end
equivalence
external
for
forward
function
goto
if
ifany
implicit
include
integer
linkage
local
logical
next
parameter
procedure
real
recursive
repeat
return
save
select
shortcall
stackheader
stmtfunc
stop
string
stringtable
subroutine
trace
undefine
until
when
while
```

Software Tools Text Formatter
User's Guide

Perry B. Flinn

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

March, 1980

TABLE OF CONTENTS

Foreword

'Fmt' is a program designed to facilitate the preparation of neatly formatted text. It provides many features, such as automatic margin alignment, paragraph indentation, hyphenation and pagination, that are designed to greatly ease an otherwise tedious job.

It is the intent of this guide to familiarize the user with the principles of automatic text formatting in general and with the capabilities and usage of 'fmt' in particular.

Basics

Usage

'Fmt' takes as input a file containing text with interspersed formatting instructions. It is invoked by a command with various optional parameters, discussed below. The resultant output is appropriately formatted text suitable for a printer having backspacing capabilities. The output of 'fmt' is made available on its first standard output port, and so may be placed in a file, sent to a line printer, or changed in any of a number of ways, simply by applying standard Software Tools Subsystem I/O redirection.

When 'fmt' is invoked from the Subsystem, there are several optional parameters that may be specified to control its operation. The full command line syntax is

```
fmt [ -s ] [ -p<first>[-<last>] ] { <file name> }
```

A brief explanation of the cryptic notation: the items enclosed within square brackets ("[]") are optional -- they may or may not be specified; items enclosed between braces ("{}") may occur any number of times, including zero; items enclosed in angle brackets ("<>") designate character strings whose significance is suggested by the text within the brackets; everything else should be taken literally.

And now for an explanation of what these parameters mean:

-s — If this option is selected, 'fmt' will pause at the top of each page, ring the bell or buzzer on your terminal, and wait for a response. This feature is for the benefit of people using hard-copy terminals with paper not having pin-feed margins. The correct response, to be entered after the paper is mounted, is a control-c (hold the 'control' key down and type 'c').

-p ... — This option allows selection of which pages of the formatted document will actually be printed. Immediately following the "-p", without any intervening spaces, should be a number indicating the first page to be printed. Following this, a second number may be specified, separated from the first by a single dash, which indicates the last page to be printed. If this second number is omitted, all remaining pages will be produced.

<file> — Any number of file names may be specified on the command line. 'Fmt' will open the files in turn, formatting the contents of each one as if they constituted one big file. When the last named file is processed, 'fmt' terminates. If no file names are specified, standard input number one is used. In addition, standard input may be specified explicitly on the command line by using a dash as a file name.

Commands and Text

'Fmt', like almost every other text formatter ever written, operates on an input stream that consists of a mixture of text and formatting commands. Each command starts at the beginning of a line with a 'control character', usually a period, followed by a two character name, in turn followed by some optional 'parameters'. There must not be anything else on the line. For example, in

```
.ta 11 21 31 41
```

the control character is a period, the command name is ta, and there are four parameters: "11", "21", "31" and "41". Notice that the command name and all the parameters must be separated from each other by one or more blanks. Anything not recognizable as a command is treated as text.

Filling and Margin Adjustment

Filled Text

'Fmt' collects as many words as will fit on a single output line before actually writing it out, regardless of line boundaries in its input stream. This is called 'filling' and is standard practice for 'fmt'. It can, however, be turned off with the 'no-fill' command

```
.nf
```

and lines thenceforth will be copied from input to output unaltered. When you want to turn filling back on again, you may do so with the 'fill' command

```
.fi
```

and 'fmt' will resume its normal behavior.

If there is a partially filled line that has not yet been written out when an **nf** command is encountered, the line is forced out before any other action is taken. This phenomenon of forcing out a partially filled line is known as a 'break' and occurs implicitly with many formatting commands. To cause one explicitly, the 'break' command

```
.br
```

is available.

Hyphenation

If, while filling an output line, it is discovered that the next word will not fit, an attempt is made to hyphenate it. Although 'fmt' is usually quite good in its choice of where to split a word, it occasionally makes a gaffe or two, giving reason to want to turn the feature off. Automatic hyphenation can be disabled with the 'no-hyphenation' command

```
.nh
```

long enough for a troublesome word to be processed, and then reenabled with the 'hyphenate' command

```
.hy
```

Neither command causes a break.

Margin Adjustment

After filling an output line, 'fmt' inserts extra blanks between words so that the last word on the line is flush with the right margin, giving the text a "professional" appearance. This is one of several margin adjustment modes that can be selected with the 'adjust' command

```
.ad <mode>
```

The optional parameter <mode> may be any one of four single characters: "b", "c", "l" or "r". If the parameter is "b" or missing, normal behavior will prevail -- both margins will be made even by inserting extra blanks between words. This is the default margin adjustment mode. If "c" is specified, lines will be shifted to the right so that they are centered between the left and right margins. If the parameter is "l", no adjustment will be performed; the line will start at the left margin and the right margin will be ragged. If "r" is specified, lines will be moved to the right so that the right margin is even, leaving the left margin ragged.

The 'no-adjustment' command

```
.na
```

has exactly the same effect as the following 'adjust' command:

```
.ad l
```

No adjustment will be performed, leaving the left margin even and the right margin ragged. In no case does a change in the adjustment mode cause a break.

Centering

Input lines may be centered, without filling, with the help of the 'center' command

```
.ce N
```

The optional parameter N is the number of subsequent input lines to be centered between the left and right margins. If the parameter is omitted, only the next line of input text is centered. Typically, one would specify a large number, say 1000, to avoid having to count lines; then, immediately following the lines to be centered, give a 'center' command with an parameter of zero. For example:

```
.ce 1000
more lines
than I care
to count
.ce 0
```

It is worth noting the difference between

```
.ce
```

and

```
.ad c
```

When the former is used, an implicit break occurs before each line is printed, preventing filling of the centered lines; when the latter is used, each line is filled with as many words as possible before centering takes place.

Sentence Punctuation

By default, 'fmt' adds an extra blank after punctuation at the end of a sentence; specifically, after periods, colons, exclamation points and question marks. This may not be desirable, particularly when abbreviations or a person's initials are involved. Thus, it can be turned on and off at will. The 'single-blank' command

```
.sb
```

turns the mode off, while the 'extra-blank' command

```
.xb
```

turns it back on again. As with hyphenation, neither command causes a break.

Summary - Filling and Margin Adjustment

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.ad <mode>	"b"	"b"	no	Set margin adjustment mode.
.br	-	-	yes	Force a break.
.ce N	N=0	N=1	yes	Center N input text lines.
.fi	on	-	no	Turn on fill mode.
.hy	on	-	no	Turn on automatic hyphenation.
.na	-	-	no	Turn off margin adjustment.
.nf	-	-	yes	Turn off fill mode. (Also inhibits adjustment.)
.nh	-	-	no	Turn off automatic hyphenation.
.sb	off	-	no	Single blank after end of sentence.
.xb	on	-	no	Extra blank after end of sentence.

Spacing and Page Control

Line Spacing

'Fmt' usually produces single-spaced output, but this can be changed, without a break, using the 'line-spacing' command

```
.ls N
```

The parameter N specifies how many lines on the page a single line of text will use; for double spacing, N would be two. If N is omitted, the default (single) spacing is reinstated.

Blank lines may be produced with the 'space' command

.sp N

The parameter N is the number of blank lines to produce; if omitted, a value of one is assumed. The **sp** command first causes a break; this not only causes a partially filled line to be output, but if the current line spacing is more than one, the break will cause the extra blank lines to be output as well. Then the blank lines generated by **sp** are output. Thus, if output is being double-spaced and the command

.sp 3

is given, four blank lines will be generated: one from the double-spacing that is in effect, and three from the **sp** command. If the value of N calls for more blank lines than there are remaining on the current page, any extra ones are discarded. This ensures that, normally, each page begins at the same distance from the top of the paper.

Page Division

'Fmt' automatically divides its output into pages, leaving adequate room at the top and bottom of each page for running headings and footings. There are several commands that facilitate the control of page divisions when the normal behavior is inadequate.

The 'begin-page' command

.bp ±N

causes a break and a skip to the top of the next page. If a parameter is given, it serves to alter the page number and so it must be numeric with an optional plus or minus sign. If the parameter is omitted, the page number is incremented by one. If the command occurs at the top of a page before any text has been printed on it, the command is ignored, except perhaps to set the page number. This is to prevent the random occurrence of blank pages.

The optionally signed numeric parameter is a form of parameter used by many formatting commands. When the sign is omitted, it indicates an absolute value to be used; when the sign is present, it indicates an amount to be added to or subtracted from the current value.

The page number may be set independently of the 'begin-page' command with the 'page-number' command

.pn ±N

The next page after the current one, when and if it occurs, will be numbered ±N. No break is caused.

The length of each page produced by 'fmt' is normally 66 lines. This is standard for eleven inch paper printed at six lines per inch. However, if non-standard paper is used, the printed length of the page may easily be changed with the 'page-length' command

.pl ±N

which will set the length of the page to ±N lines without causing a break.

Finally, if it is necessary to be sure of having enough room on a page, say for a figure or a graph, use the 'need' command

.ne N

'Fmt' will cause a break, check if there are N lines left on the current page and, if so, will do nothing more. Otherwise, it will skip to the top of the next page where there should be adequate room.

'No-space' Mode

'No-space' mode is a feature that assists in preventing unwanted blank lines from appearing, usually at the top of a page. When in effect, certain commands that cause blank lines to be generated, such as **bp**, **ne** and **sp**, are suppressed. For the most part, 'no-space' mode is managed automatically; it is turned on automatically at the top of each page before the first text has appeared, and turned off again automatically when a line of output is generated. This accounts for the suppression of **bp** commands at the top of a page and the discarding of excess blank lines in **sp** commands.

'No-space' mode may be turned on explicitly with the 'no-space' command

```
.ns
```

and turned off explicitly with the 'restore-spacing' command

```
.rs
```

Neither command causes a break.

Summary - Spacing and Page Control

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.bp ±N	N=1	next	yes	Begin a new page.
.ls N	N=1	N=1	yes	Set line spacing.
.ne N	-	N=1	yes	Express a need for N contiguous lines.
.ns	on	-	no	Turn on 'no-space' mode.
.pl ±N	N=66	N=66	no	Set page length.
.pn ±N	N=1	ignored	no	Set page number.
.rs	-	-	no	Turn off 'no-space' mode.
.sp N	-	N=1	yes	Put out N blank lines.

Margins and Indentation

Margins

All formatting operations are performed within the framework of a page whose size is defined by four margins: top, bottom, left and right. The top and bottom margins determine the number of lines that are left blank at the top and bottom of each page. Likewise, the left and right margins determine the first and last columns across the page into which text may be placed.

Top and Bottom Margins

Both the top and the bottom margins consist of two sub-margins that fix the location of the header and footer lines. For the sake of clarity, the first and second sub-margins of the top margin will be referred to as 'margin 1' and 'margin 2', and the first and second sub-margins of the bottom margin, 'margin 3' and 'margin 4'.

The value of margin 1 is the number of lines to skip at the top of each page before the header line, plus one. Thus, margin 1 includes the header line and all the blank lines preceding it from the top of the paper. By default, its value is three. Margin 2 is the number of blank lines that are to appear between the header line and the first text on the page. Normally, it has a value of two. The two together form a standard top margin of five lines, with the header line right in the middle. It is easy enough to change these defaults if they prove unsatisfactory; just use the 'margin-1' and 'margin-2' commands

```
.m1 ±N
.m2 ±N
```

to set either or both sub-margins to ±N.

The bottom margin is completely analogous to the top margin, with margin 3 being the number of blank lines between the last text on a page and the footer line, and margin 4 being the number of lines from the footer to the bottom of the paper (including the footer). They may be set using the 'margin-3' and 'margin-4' commands

```
.m3 ±N
.m4 ±N
```

which work just like their counterparts in the top margin; none cause a break.

Left and Right Margins

The left and right margins define the first and last columns into which text may be printed. They affect such things as adjustment and centering. The left margin is normally set at column one, though this is easily changed with the 'left-margin' command

.lm ±N

The right margin, which is normally positioned in column sixty, can be set similarly with the 'right-margin' command

.rm ±N

To ensure that the new margins apply only to subsequent text, each command causes a break before changing the margin value.

Indentation

It is often desirable to change the effective value of the left margin for indentation, without actually changing the margin itself. For instance, all of the examples in this guide are indented from the left margin in order to set them apart from the rest of the text. Indentation is easily arranged using the 'indent' command,

.in ±N

whose parameter specifies the number of columns to indent from the left margin. The initial indentation value, and the one assumed if no parameter is given, is zero (i.e. start in the left margin).

For the purpose of margin adjustment, the current indentation value is added to the left margin value to obtain the effective left margin. In this respect, the **lm** and **in** commands are quite similar. But, whereas the left margin value affects the placement of centered lines produced by the **ce** command, indentation is completely ignored when lines are centered.

Paragraph indentation poses a sticky problem in that the indentation must be applied only to the first line of the paragraph, and then normal margins must be resumed. This can't be done conveniently with the 'indent' command, since it causes a break. Therefore, 'fmt' has a 'temporary-indent' command

.ti ±N

whose function is to cause a break, alter the current indentation value by ±N until the next line of text is produced, and then reset the indentation to its previous value. So to begin a new paragraph with a five column indentation, one would say

.ti +5

Page Offset

As if control over the left margin position and indentation were not enough, there is yet a third means for controlling the position of text on the page. The concept of a page offset involves nothing more than prepending a number of blanks to each and every line of output. It is primarily intended to allow output to be easily positioned on the paper without having to adjust margins and indentation (with all their attendant side effects) and without having to physically move the paper. Although the page offset is initially zero, other arrangements may be made with the 'page-offset' command

.po ±N

which causes a break.

Margin Characters

It is common practice in the revision of technical literature to indicate parts of the text that are different from previous versions of the same document. Such changes are usually indicated by "revision bars" which are vertical lines in the left margin of lines that are new or revised. 'Fmt' provides for this capability with two formatting commands. The 'margin-offset' command,

.mo ±N

without causing a break, specifies that ±N columns are to be reserved between the 'page-

offset' columns and the 'left-margin' column for revision bars or other marginal characters. The margin offset starts out at zero, and reverts to that value if no parameter is specified.

Once a non-zero margin offset has been set, any arbitrary character may be placed in the leftmost column of the area with the 'margin-character' command:

.mc <char>

Initially, and when <char> is omitted, this character has blank as its value. For revision bars, <char> would be specified as "|". Whatever character is specified, it is placed next to the left margin on every line of output as long as the margin offset is non-zero.

Summary - Margins and Indentation

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.in ±N	N=0	N=0	yes	Indent left margin.
.lm ±N	N=1	N=1	yes	Set left margin.
.m1 ±N	N=3	N=3	no	Set top margin before and including page heading.
.m2 ±N	N=2	N=2	no	Set top margin after page heading.
.m3 ±N	N=2	N=2	no	Set bottom margin before page footing.
.m4 ±N	N=3	N=3	no	Set bottom margin including and after page footing.
.mc <char>	BLANK	BLANK	no	Set margin character.
.mo ±N	N=0	N=0	no	Set margin offset.
.po ±N	N=0	N=0	yes	Set page offset.
.rm ±N	N=60	N=60	yes	Set right margin.
.ti ±N	N=0	N=0	yes	Temporarily indent left margin.

Headings, Footings and Titles

Three Part Titles

A three part title is a line of output consisting of three segments. The first segment is left-justified, the second is centered between the left and right margins, and the third is right-justified. For example

left part center part right part

is a three part title whose first segment is "left part", whose second segment is "center part", and whose third segment is "right part".

To generate a title at the current position on the page, the 'title' command is available:

.tl /left part/center part/right part/

In fact, this command was used to generate the previous example. The parameter to the title command is made up of the text of the three parts, with each segment enclosed within a pair of delimiter characters. Here, the delimiter is a slash, but any other character may be used as long as it is used consistently within the same command. If one or more segments are to be omitted, indicate this with two adjacent delimiters at the desired position. Thus,

.tl ///Page 1/

specifies only the third segment and would produce something like this:

Page 1

It is not necessary to include the trailing delimiters.

To facilitate page numbering, you may include the sharp character ("#") anywhere in the text of the title; when the command is actually performed, 'fmt' will replace all occurrences of the "#" with the current page number. To produce a literal sharp character in the title, it should be preceded by an "@"

```
@#
```

so that it loses its special meaning.

The first segment of a title always starts at the left margin as specified by the **lm** command. While the third segment normally ends at the right margin as specified by the **rm** command, this can be changed with the 'length-of-title' command:

```
.lt ±N
```

which changes the length of subsequent titles to +N, still beginning at the left margin. Note that the title length is automatically set by the **lm** and **rm** commands to coincide with the distance between the left and right margins.

Page Headings and Footings

The most common uses for three part titles are page headings and footings. The header and footer lines are initially blank. Either one or both may be set at any time, without a break, by using the 'header' command

```
.he /left/center/right/
```

to set the page heading, and the 'footer' command

```
.fo /left/center/right/
```

to set the page footing. The change will become manifest the next time the top or the bottom of a page is reached. As with the **tl** command, the "#" may be used to access the current page number.

It is often desirable when producing text to be printed on both sides of a page to have different headings and footings on odd- and even-numbered pages. Although the **he** and **fo** commands affect the headings and footings on all pages, it is possible to set up independent headings and footings for odd- and even-numbered pages. For odd-numbered pages, the 'odd-header' and 'odd-footer' commands are available:

```
.oh /left/center/right/
.of /left/center/right/
```

while the 'even-header' and 'even-footer' commands are provided for even-numbered pages:

```
.eh /left/center/right/
.ef /left/center/right/
```

As an illustration, the following commands were used to generate the page headings and footings for this guide:

```
.eh /Text Formatter User's Guide///
.oh ///Text Formatter User's Guide/
.fo //- # -//
```

Summary - Headings, Footings and Titles

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.ef /l/c/r/	blank	blank	no	Set even-numbered page footing.
.eh /l/c/r/	blank	blank	no	Set even-numbered page heading.
.fo /l/c/r/	blank	blank	no	Set running page footing.
.he /l/c/r/	blank	blank	no	Set running page heading.
.lt ±N	N=60	N=60	no	Set length of header, footer and titles.
.of /l/c/r/	blank	blank	no	Set odd-numbered page footing.
.oh /l/c/r/	blank	blank	no	Set odd-numbered page heading.
.tl /l/c/r/	blank	blank	yes	Generate a three part title.

Tabulation

Tabs

Just like any good typewriter, 'fmt' has facilities for tabulation. When it encounters a special character in its input called the 'tab character' (analogous to the TAB key on a typewriter), it automatically advances to the next output column in which a 'tab stop' has been previously set. Tab stops are always measured from the effective left margin, that is, the left margin plus the current indentation or temporary indentation value. Whatever column the left margin may actually be in, it is always assumed to be column one for the purpose of tabulation.

Originally, a tab stop is set in every eighth column, starting with column nine. This may be changed using the 'tab' command

```
.ta <col> <col> ...
```

Each parameter specified must be a number, and causes a tab stop to be set in the corresponding output column. All existing stops are cleared before setting the new ones, and a stop is set in every column beyond the last one specified. This means that if no columns are specified, a stop is set in every column.

By default, 'fmt' recognizes the ASCII TAB, control-i, as the 'tab character'. But since this is an invisible character and is guaranteed to be interpreted differently by different terminals, it can be changed to any character with the 'tab-character' command:

```
.tc <char>
```

While there is no restriction on what particular character is specified for <char>, it is wise to choose one that doesn't occur too frequently elsewhere in the text. If you omit the parameter, the tab character reverts to the default.

When 'fmt' expands a tab character, it normally puts out enough blanks to get to the next tab stop. In other words, the default 'replacement' character is the blank. This too may easily be changed with the 'replacement-character' command:

```
.rc <char>
```

As with the **tc** command, <char> may be any single character. If omitted, the default is used.

A common alternate replacement character is the period, which is frequently used in tables of contents. The following example illustrates how one might be constructed:

```
.ta 52
.tc \
Section Name\Page
.rc .
.sp
.nf
.ta 53
Basics\1
Filling and Margin Adjustment\2
Spacing and Page Control\5
.sp
.fi
```

The result should look about like this:

```
Section Name                                          Page

Basics.....................................................1
Filling and Margin Adjustment..............................2
Spacing and Page Control...................................5
```

A final word on tabs: Since the default replacement character is a blank you might think that, in the process of adjusting margins (i.e., when the adjustment mode is "b"), 'fmt' might throw in extra blanks between words that were separated by the tab character. Since this is definitely not the expected or desired behavior, 'fmt' uses what is called a "phantom blank" as the default replacement character. The phantom blank prints as an ordinary blank, but is not recognized as one during margin adjustment.

Summary - Tabulation

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.ta N ...	9 17 ...	all	no	Set tab stops.
.tc c	TAB	TAB	no	Set tab character.
.rc c	BLANK	BLANK	no	Set tab replacement character.

Miscellaneous Commands

Comments

It is rare that a document survives its writing under the pen of just one author or editor. More frequently, several different people are likely to put in their two cents worth concerning its format or content. So, if the author is particularly attached to something he has written, he is well advised to say so. Comments are an ideal vehicle for this purpose and are easily introduced with the 'comment' command

```
.# <commentary text>
```

Everything after the # up to and including the next newline character is completely ignored by 'fmt'.

Boldfacing and Underlining

'Fmt' makes provisions for **boldfacing** and underlining lines or parts thereof with two commands:

```
.bf N
```

boldfaces the next N lines of input text, while

```
.ul N
```

underlines the next N lines of input text. In both cases, if N is omitted, a value of one is assumed. Neither command causes a break, allowing single words or phrases to be boldfaced or underlined without affecting the rest of the output line.

It is also possible to use the two in combination. For instance, the heading at the beginning of the table of contents was produced by a sequence of commands and text similar to the following:

```
.bf
.ul
TABLE OF CONTENTS
```

As with the 'center' command, these two commands are often used to bracket the lines to be affected by specifying a huge parameter value with the first occurrence of the command and a value of zero with the second:

```
.bf 1000
.ul 1000
lots of lines
to be
boldfaced
and
underlined
.bf 0
.ul 0
```

Control Characters

As mentioned in the first section, command lines are distinguished from text by the presence of a 'control character' in column one. In all the examples cited thus far, a period has been used to represent the control character. It is possible to select any character for this purpose. In fact, several occasions arose in the writing of this guide which called for use of an alternate control character, particularly in the construction of the command summaries at the end of each section. The 'control-character' command may be used anywhere to select a new value:

```
.cc <char>
```

The parameter <char>, which may be any single character, becomes the new control character. If the parameter is omitted, the familiar period is reinstated.

It has been shown that many commands automatically cause a break before they perform their function. When this presents a problem, it can be altered. If instead of using the basic control character the 'no-break' control character is used to introduce a command, the automatic break that would normally result is suppressed. The standard no-break control character is the grave accent ("`"), but may easily be changed with the following command:

```
.c2 <char>
```

As with the **cc** command, the parameter may be any single character, or may be omitted if the default value is desired.

Prompting

Brief, one-line messages may be written directly to the user's terminal using the 'prompt' command

```
.er <brief, one-line message>
```

The text that is actually written to the terminal starts with the first non-blank character following the command name, and continues up to, but not including, the next newline character. If a newline character should be included in the message, the escape sequence

```
@n
```

may be used. Leading blanks may also be included in the message by preceding the message with a quote or an apostrophe. 'Fmt' will discard this character, but will then print the rest of the message verbatim. For instance,

```
.er '          this is a message with 10 leading blanks
```

would write the following text on the terminal, leaving the cursor or carriage at the end of the message

```
          this is a message with 10 leading blanks
```

For a multiple-line message, try

```
.er multiple@nline@nmessage@n
```

The output should look like this:

```
multiple
line
message
```

Prompts are particularly useful in form letter applications where there may be several pieces of information that 'fmt' has to ask for in the course of its work. The next section describes how 'fmt' can dynamically obtain information from the user.

Premature Termination

If 'fmt' should ever encounter an 'exit' command

```
.ex
```

in the course of doing its job, it will cause a break and exit immediately to the Subsystem.

Summary - Miscellaneous Commands

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.#	-	-	no	Introduce a comment.
.bf N	N=0	N=1	no	Boldface N input text lines.
.c2 c	`	`	no	Set no-break control character.
.cc c	.	.	no	Set basic control character.
.er text	-	ignored	no	Write a message to the terminal.
.ex	-	-	yes	Exit immediately to the Subsystem.
.ul N	N=0	N=1	no	Underline N input text lines.

Input Processing

Input File Control

Up to this point, it has been assumed that 'fmt' reads only from its standard input file or from files specified as parameters on the command line. It is also possible to dynamically include the contents of any file in the midst of formatting another. This aids greatly in the modularization of large, otherwise unwieldy documents, or in the definition of frequently used sequences of commands and text.

The 'source' command is available to dynamically include the contents of a file:

```
.so <file>
```

The parameter <file> is mandatory; it may be an arbitrary file system pathname, or, as with file names on the command line, a single dash ("-") to specify standard input number one.

The effect of a 'source' command is to temporarily preempt the current input source and begin reading from the named file. When the end of that file is reached, the original source of input is resumed. Files included with 'source' commands may themselves contain other 'source' commands; in fact, this 'nesting' of input files may be carried out to virtually any depth.

'Fmt' provides one additional command for manipulating input files. The 'next file' command

```
.nx <file>
```

may be used for either one of two purposes. If you specify a <file> parameter, all current

input files are closed (including those opened with **so** commands), and the named file becomes the new input source. You can use this for repeatedly processing the same file, as, for example, with a form letter. If you omit the <file> parameter, 'fmt' still closes all of its current input files. But instead of using a file name you supply with the **nx** command, it uses the next file named on the command line that invoked 'fmt'. If there is no next file, then formatting terminates normally.

Neither the **so** command nor the **nx** command causes a break.

Functions, Variables and Special Characters

Whenever 'fmt' reads a line of input, no matter what the source may be, there is a certain amount of 'pre-processing' done before any other formatting operations take place. This pre-processing consists of the interpretation of 'functions', 'variables' and 'special characters'. A 'function' is a predefined set of actions that produces a textual result, possibly based on some user supplied textual input. For example, one hypothetical function might be named 'time', and its result might be a textual representation of the current time of day:

```
17:22:43
```

A 'variable' is simply one of 'fmt's internal parameters, such as the current page length or the current line-spacing value; the name of each variable is the same as the two-character name of the corresponding command to set the value of that parameter. The result of a variable is just a textual representation of that value.

A 'special character' is like a function or variable, but its result is a single character that cannot be conveniently generated from the keyboard.

From the standpoint of a user, functions, variables and special characters are all very similar. In fact, they are invoked identically by enclosing the appropriate name, plus any text to be used as arguments, in square brackets:

```
[bf This text to be boldfaced]
[ls]
[alpha 5]
```

Such a construct is known as a "function call."

When 'fmt' sees a function call in an input line, it excises everything in between the brackets, including the brackets themselves, and inserts the results in its place. Naturally, anything not recognizable is left alone. If by chance you want the name of a function, variable or special character enclosed in square brackets included literally as part of the text, you can inhibit evaluation by preceding the left bracket with the escape character:

```
@[time]
```

It is also possible to "nest" function calls so that the results of one may be used as arguments to another:

```
[bf [ldate]]
```

Number Registers

The 'number registers' are a group of 64 accumulators (numbered 1-64) on which simple arithmetic operations may be performed. They find their greatest use in the preparation of documents with numbered sections and paragraphs. Number registers are accessed and manipulated by a special set of functions. The 'set' function

```
[set reg value]
```

assigns the integer 'value' to the register 'reg' and yields the empty string as its result. The 'add' function

```
[add reg value]
```

adds the integer 'value' (which, by the way may be positive or negative) to the register 'reg'. This function too yields an empty result. Finally, the 'num' function

```
[num reg]
```

yields the current value of the register 'reg' as its result. In addition, 'reg' may either

be prefixed or postfixed by a plus or minus sign. If the sign appears before the register number, the register is incremented or decremented (according to the sign) by one before the function's result is yielded. If the sign follows the register number, though, the register's current value is yielded and then the register is incremented or decremented.

Functions

The following table summarizes the available functions:

add	Add constant to number register
bf	Boldface the arguments on output
cu	Output the arguments with a continuous underline
date	Current date; e.g., 04/30/80
day	Current day of the week; e.g., Wednesday
ldate	Current date: e.g., April 30, 1980
num	Output value of number register with optional pre- or post-incrementation or decrementation
RN	Convert the argument to a lower-case Roman numeral
rn	Convert the argument to an upper-case Roman numeral
set	Set number register to value
sub	Output the arguments as a subscript (requires post-processor)
sup	Output the arguments as a superscript (requires post-processor)
time	Current time of day; e.g., 17:22:45
ul	Underline the arguments on output

Variables

The formatting parameters whose values are available through function calls are summarized in the following table:

cc	Current basic control character
c2	Current no-break control character
in	Current indentation value
lm	Current left margin value
ln	Current line number on the page
ls	Current line-spacing value
ml	Current macro invocation level
m1	Current margin 1 value
m2	Current margin 2 value
m3	Current margin 3 value
m4	Current margin 4 value
pl	Current page length value
pn	Current page number
po	Current page offset value
rm	Current right margin value
tc	Current tab character
ti	Current temporary indentation value
tcpn	Current page number, right justified in 4 character field

Special Characters

The following table summarizes the available special characters. In each case, a positive integer may be included as an argument following the name to produce multiple instances of the character. For example, "[bl 5]" yields five contiguous phantom blanks. NOTE: in order for the Greek letters and mathematical symbols to be printed correctly, a post-processor such as 'dprint' (see Section 3 of the Software Tools Subsystem Reference Manual) and/or special printing equipment is required.

bl	Phantom blank
bs	Backspace
alpha	Lower-case Greek alpha
beta	Lower-case Greek beta
delta	Lower-case Greek delta
DELTA	Upper-case Greek delta
epsilon	Lower-case Greek epsilon
eta	Lower-case Greek eta
gamma	Lower-case Greek gamma
GAMMA	Upper-case Greek gamma
infinity	Infinity symbol
integral	Integral symbol
lambda	Lower-case Greek lambda
LAMBDA	Upper-case Greek lambda
mu	Lower-case Greek mu

	nabla	Upside-down upper-case Greek delta (APL "DEL")
	not	Not symbol
	nu	Lower-case Greek nu
	omega	Lower-case Greek omega
	OMEGA	Upper-case Greek omega
	partial	Partial derivative symbol
	phi	Lower-case Greek phi
	PHI	Upper-case Greek phi
	psi	Lower-case Greek psi
	PSI	Upper-case Greek psi
	pi	Lower-case Greek pi
	PI	Upper-case Greek pi
	rho	Lower-case Greek rho
	sigma	Lower-case Greek sigma
	SIGMA	Upper-case Greek sigma
	tau	Lower-case Greek tau
	theta	Lower-case Greek theta
	THETA	Upper-case Greek theta
	xi	Lower-case Greek xi
	zeta	Lower-case Greek zeta

Summary - Input Processing

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.nx file	-	next arg	no	Move on to the next input file.
.so file	-	ignored	no	Temporarily alter the input source.

Macros

Macro Definition

A macro is nothing more than a frequently used sequence of commands and/or text that have been grouped together under a single name. This name may then be used just like an ordinary command to invoke the whole group in one fell swoop.

The definition (or redefinition) of a macro starts with a 'define' command

```
.de xx
```

whose parameter is a one or two character string that becomes the name of the macro. The macro name may consist of any characters other than blanks, tabs or newlines; upper and lower letters are distinct. The definition of the macro continues until a matching 'end' command

```
.en xx
```

is encountered. Anything may appear within a macro definition, including other macro definitions. The only processing that is done during definition is the interpretation of variables and functions (i.e. things surrounded by square brackets). Other than this, lines are stored exactly as they are read from the input source. To include a function call in the definition of a macro so that its interpretation will be delayed until the macro is invoked, the opening bracket should be preceded by the escape character "@". For example,

```
.# tm --- time of day
.de tm
@[time]
.en tm
```

would produce the current time of day when invoked, whereas

```
.# tm --- time of day
.de tm
[time]
.en tm
```

would produce the time at which the macro definition was processed.

Macro Invocation

Again, a macro is invoked like an ordinary command: a control character at the beginning of the line immediately followed by the name of the macro. So to invoke the above 'time-of-day' macro, one might say

```
.tm
```

As with ordinary commands, macros may have parameters. In fact, anything typed on the line after the macro name is available to the contents of the macro. As usual, blanks and tabs serve to separate parameters from each other and from the macro name. If it is necessary to include a blank or a tab within a parameter, it may be enclosed in quotes. Thus,

```
"parameter one"
```

would constitute a single parameter and would be passed to the macro as

```
parameter one
```

To include an actual quotation mark within the parameter, type two quotes immediately adjacent to each other. For instance,

```
"""quoted string"""
```

would be passed to the macro as the single parameter

```
"quoted string"
```

Within the macro, parameters are accessed in a way similar to functions and variables: the number of the desired parameter is enclosed is square brackets. Thus,

```
[1]
```

would retrieve the first parameter,

```
[2]
```

would fetch the second, and so on. As a special case, the name of the macro itself may be accessed with

```
[0]
```

Assume there is a macro named "mx" defined as follows:

```
.# mx --- macro example
.de mx
Macro named '[0]', invoked with two arguments:
'[1]' and '[2]'.
.en mx
```

Then, typing

```
.mx "param 1" "param 2"
```

would produce the same result as typing

```
Macro named 'mx', invoked with two arguments:
'param 1' and 'param 2'.
```

Macros are quite handy for such common operations as starting a new paragraph, or for such tedious tasks as the construction of tables like the ones appearing at the end of each section in this guide. For some examples of frequently used macros, see the applications notes in the following pages.

Summary - Macros

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.de xx	-	ignored	no	Begin definition or redefinition of a macro.
.en xx	-	ignored	no	End macro definition.

Applications Notes

This section will illustrate the capabilities of 'fmt' with some actual applications. Most of the examples are macros that assist in common formatting operations, but attention has also been given to table construction. All of the macros presented here are available for general use in the file '/extra/fmacro/report', which may be named on the command line invoking 'fmt' or may be included with a 'source' command as follows:

```
.so =fmac=/report
```

Paragraphs

One standard way of beginning a new paragraph is to skip a line and indent by a few spaces, as was done throughout this guide. This can be done by giving an **sp** command followed by a **ti** command. A better way is to define a macro. This allows procrastination on deciding the format of paragraphs and facilitates change at some later date without a major editing effort.

Here is the paragraph macro used in this document:

```
.# pp --- begin paragraph
.de pp
.sp
.ne 2
.ti @[in]
.ti +5
.ns
.en pp
```

First a line is skipped via the 'space' command. Then, after checking that there is room on the current page for the first two lines of the new paragraph, a temporary indentation is set up that is five columns to the rignt of the running indentation with the two **ti** commands. Finally, no-space mode is turned on to suppress unwanted blank lines.

Sub-headings

Sub-headings such as the ones used here may be easily produced with the following macro:

```
.# sh --- sub-heading
.de sh
.sp 2
.ne 4
.ti @[in]
.bf
[1]
.pp
.en sh
```

First, two blank lines are put out. Then it is determined if there are four contiguous lines on the current page: one for the heading itself, one for the blank line after the heading, and two for the first two lines of the next paragraph. The temporary indentation value is then set to coincide with the current indentation value. Next, the first parameter passed to the macro (the text of the sub-heading) is boldfaced and a new paragraph is begun. The "pp" macro will insert the blank line after the heading.

Major Headings

Each section of this guide is introduced by a major heading that is boldfaced, underlined and centered on the page. The macro used to produce these headings is the following:

```
.# mh --- major heading
.de mh
.sp 3
.ne 5
.ce
.ul
.bf
[1]
.sp
.pp
.en mh
```

This is similar to the sub-heading macro: three blank lines are put out; a check for enough room is made; the parameter is centered, underlined and boldfaced; another blank line is put out; and a new paragraph is begun.

Quotations

Lengthy quotations are often set apart from other text by altering the left and right margins to narrow the width of the quoted text. Here is a pair of macros that may be used to delimit the beginning and end of a direct quotation:

```
.# bq --- begin direct quote
.de bq
.sp
.ne 2
.in +5
.rm -5
.lt +5
.en bq


.# eq --- end direct quote
.de eq
.sp
.in -5
.rm +5
.en eq
```

Notice the **lt** command in the first macro. To avoid affecting page headings and footings, the left margin is not adjusted; rather, an additional indentation is applied. But to increase the right margin width, there is no other alternative but to use the **rm** command. The 'title-length' command is thus necessary to allow headings and footings to remain unaffected by the interim right margin.

Italics

Since most printers can't easily produce italics, they are frequently simulated by underlining. The following macro 'italicizes' its parameter by underlining it.

```
.# it --- italicize (by underlining)
.de it
.ul
[1]
.en it
```

Boldfacing

While 'fmt' has built-in facilities for boldfacing, their use may be somewhat cumbersome if there are many short phrases or single words that need boldfacing; each phrase or word requires two input lines: one for the **bf** command and one for the actual text. The following macro cuts the overhead in half by allowing the command and the text to appear on the same line.

```
.# bo --- boldface parameter
.de bo
.bf
[1]
.en bo
```

Examples

This guide is peppered with examples, each one set apart from other text by surrounding blank lines and additional indentation. The next two macros, used like the "bq" and "eq" macros, facilitate the production of examples.

```
.# bx --- begin example text
.de bx
.sp
.ne 2
.nf
.in +10
.en bx


.# ex --- end example text
.de ex
.sp
.fi
.in -10
.en ex
```

Note that the definition of the "ex" macro causes the **ex** command to become inaccessible.

Table Construction

One example of table construction (for a table of contents) has already been mentioned in the section dealing with tabs. Another type of table that occurs frequently is that used in the command summaries in this guide. Each entry of such a table consists of a number of 'fields', followed on the right by a body of explanatory text that needs to be filled and adjusted.

The easiest way to construct a table like this involves using a combination of tabs and indentation, as the following series of commands illustrates:

```
.in +40
.ta 14 24 34 41
.tc \
```

The idea is to set a tab stop in each column that begins a field, and one last one in the column that is to be the left margin for the explanatory text. The extra indentation moves the effective left margin to this column. To begin a new entry, temporarily undo the extra indentation with a **ti** command, and then type the text of the entry, separating the fields from one another with a tab character:

```
.ti -40
field 1\field 2\field 3\field 4\Explanatory text
```

The first line of the entry will start at the old left margin. Then all subsequent lines will be filled and adjusted between column forty-one and the right margin (inclusive).

Summary of Commands Sorted Alphabetically

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.#	-	-	no	Introduce a comment.
.ad c	both	both	no	Set margin adjustment mode.
.bf N	N=0	N=1	no	Boldface N input text lines.
.bp ±N	N=1	next	yes	Begin a new page.
.br	-	-	yes	Force a break.
.c2 c	`	`	no	Set no-break control character.
.cc c	.	.	no	Set basic control character.
.ce N	N=0	N=1	yes	Center N input text lines.
.de xx	-	ignored	no	Begin definition or redefinition of a macro.
.ef /l/c/r/	blank	blank	no	Set even-numbered page footing.
.eh /l/c/r/	blank	blank	no	Set even-numbered page heading.
.en xx	-	ignored	no	End macro definition.
.er text	-	ignored	no	Write a message to the terminal.
.ex	-	-	yes	Exit immediately to the Subsystem.
.fi	on	-	no	Turn on fill mode.
.fo /l/c/r/	blank	blank	no	Set running page footing.
.he /l/c/r/	blank	blank	no	Set running page heading.
.hy	on	-	no	Turn on automatic hyphenation.
.in ±N	N=0	N=0	yes	Indent left margin.
.lm ±N	N=1	N=1	yes	Set left margin.
.ls N	N=1	N=1	no	Set line spacing.
.lt ±N	N=60	N=60	no	Set length of header, footer and titles.
.ml ±N	N=3	N=3	no	Set top margin before and including page heading.
.m2 ±N	N=2	N=2	no	Set top margin after page heading.
.m3 +N	N=2	N=2	no	Set bottom margin before page footing.
.m4 ±N	N=3	N=3	no	Set bottom margin including and after page footing.
.mc <char>	BLANK	BLANK	no	Set margin character.
.mo ±N	N=0	N=0	no	Set margin offset.
.na	-	-	no	Turn off margin adjustment.
.ne N	-	N=1	yes	Express a need for N contiguous lines.
.nf	-	-	yes	Turn off fill mode. (Also inhibits adjustment.)
.nh	-	-	no	Turn off automatic hyphenation.
.ns	on	-	no	Turn on 'no-space' mode.
.nx file	-	next arg	no	Move on to the next input file.

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.of /l/c/r/	blank	blank	no	Set odd-numbered page footing.
.oh /l/c/r/	blank	blank	no	Set odd-numbered page heading.
.pl ±N	N=66	N=66	no	Set page length.
.pn ±N	N=1	ignored	no	Set page number.
.po ±N	N=0	N=0	yes	Set page offset.
.rc c	BLANK	BLANK	no	Set tab replacement character.
.rm ±N	N=60	N=60	yes	Set right margin.
.rs	-	-	no	Turn off 'no-space' mode.
.sb	off	-	no	Single blank after end of sentence.
.so file	-	ignored	no	Temporarily alter the input source.
.sp N	-	N=1	yes	Put out N blank lines.
.ta N ...	9 17 ...	all	no	Set tab stops.
.tc c	TAB	TAB	no	Set tab character.
.ti ±N	N=0	N=0	yes	Temporarily indent left margin.
.tl 'l'c'r'	blank	blank	yes	Generate a three part title.
.ul N	N=0	N=1	no	Underline N input text lines.
.xb	on	-	no	Extra blank after end of sentence.