Software Tools Subsystem Tutorial

T. Allen Akin
Terrell L. Countryman
Perry B. Flinn
Daniel H. Forsyth, Jr.
Jeanette T. Myers
Arnold D. Robbins
Peter N. Wan

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

September, 1984

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 9 of the Subsystem as released in September 1984.

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. In other words, as far as Primos is concerned, the Subsystem is a single program invoked by the user. If the user wishes to use the Subsystem, he or she must call it explicitly (it is possible to call the Subsystem automatically on login; we will discuss how to do so a little further on).

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 (User 15) logged in Friday, 06 Jul 84 14:22:07. (3)
Welcome to PRIMOS version 19.2.
Last login Friday, 06 Jul 84 14:06:32
OK, swt                                                 (4)
Password:                                               (5)
Enter terminal type: ti                                 (6)
]                                                       (7)
```

(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 (the key labelled "newline", "return", or "cr" on your terminal). Note: password checking on login, as of Rev. 19, is now a standard part of Primos.

(3) Primos acknowledges a successful login by typing your login name, your process number (in parentheses), and the current time and date. (Note: At Georgia Tech, the login acknowledgement will look somewhat different from what is shown here.)

(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 not issued and not prompted for by 'swt'.) After you receive the prompt, enter your Subsystem password. It will not be printed on the terminal.

(6) 'Swt' asks you to enter the type of terminal that you are using. Depending on your local configuration, you may or may not see this message. If you do see it, enter the type of terminal you are using. You may obtain the name of your terminal type by asking your system administrator, or you can enter a question mark ("?") and try to find your terminal type in the list that 'swt' will display for you.

(7) 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 (and for other mysterious reasons), the Subsystem always knows what kind of terminal you are using. You may have told it your terminal type when you entered the Subsystem, or your system administrator may have pre-assigned your terminal type. In any event, the Subsystem initially decides whether or not you are using an upper-case-only terminal from this terminal type.

You can find out what the Subsystem thinks about your terminal by entering the 'term' command:

```
] term
type tty  buffer 2
-erase BS  -escape ESC  -kill DEL
-retype DC2  -eof ETX  -newline LF
-echo  -lf  -xoff  -noinh  -nose  -novth  -nolcase
-break
]
```

If the Subsystem thinks you are using an upper-case-only terminal, you will see the entry "-nolcase" in the last line; otherwise, you will see "-lcase". If you see that you have mistakenly entered the wrong terminal type, you can use 'term' to change it. To list the possible terminal types for your installation, enter

```
] term ?
```

Then change your terminal type by entering

```
] term <new terminal type>
```

If you are using an upper-case-only terminal, the Subsystem converts all subsequent upper case letters you type to lower case, and converts all lower case letters sent to your terminal by the computer to upper case. Since your terminal is also missing a few other necessary characters, the Subsystem 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 the Subsystem thinks you have an upper-case-only terminal (i. e., you see "-nolcase" in the output from 'term'), 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 "-nolcase" 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 "-nolcase".

Finishing Up

When you're finished using the Subsystem, you have several options for getting out. The first two 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.

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 (User 15) logged out Friday, 06 Jul 84 15:30:00. (2)
Time Used: 01h 08m connect, 01m 06s CPU, 01m 10s I/O. (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.

Note the the 'bye' command is equivalent to exiting the Subsystem and executing the Primos LOGOUT command.

Automatically Running the Subsystem

With Primos Rev. 19, you can arrange to automatically run the Subsystem when you log in. Simply put the command 'swt' into a file named 'login.comi' in the directory to which you will be

attached when you log in.

Primos will execute the command(s) in this file automatically. Furthermore, if your profile directory is an ACL directory instead of a password directory, the Subsystem will not even ask you for a password, since the file system provides the protection automatically. (If this paragraph makes no sense to you at all, don't worry about it. It isn't all that important.)

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 shortcomings; 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 a message ending "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.

The File System and Related Utilities

Users spend much of their time creating, deleting, modifying and manipulating files. The utilities discussed in this section perform these tasks.

Creating Files

The most common way to create a file is to write the contents of a text editor to a new filename. Another common way (especially for creating small files) is to use the 'cat' command. Both of these methods are covered later in this guide. Right now, we prefer that you not be concerned with creating large, elaborate files or with knowing about more advanced features of the Subsystem. Instead, we will show you a simple method for creating one-line files. (Although you may not understand the command format at this point in time, don't worry because you will by the time you get through the tutorial).

You can use the command 'echo' to create files as in the examples below:

```
] echo xxxx >file_of_x                          (1)
] echo contents of myfile >myfile               (2)
```

(1) Creates a file named "file_of_x" containing "xxxx".

(2) Creates a file named "myfile" containing the line "contents of myfile".

In case you were wondering, you can only use letters, digits, underscores, and periods in file names. (You can actually use a few other characters in names, but that can get you into trouble.) The names must not start with a digit, and can be no longer than 32 characters.

Looking at the Contents of Files

There are several ways of looking at the contents of a file. One command that you can use is 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 a subsequent 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; but of course, you can delete only files that belong to you.

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.

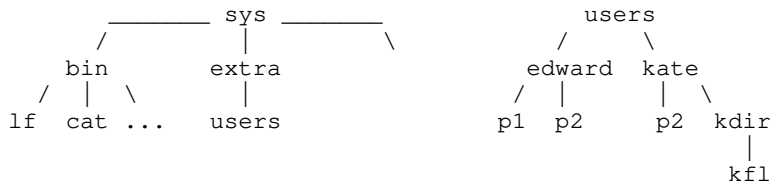
(2) List the names of all files in the current directory, including all information that is known about each file.

(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".

The Primos File System

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. This diagram shows a simple structure that we will use as an example:



In this example, the mfds are named "sys" and "users", while there are primary directories named "bin", "extra", "edward", and "kate".

The Subsystem allows you 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. "sys" in the diagram above could be referred to as "/0/" or "/sys/"). The named disk is the starting point for the search of the rest of the pathname. The disk name may be omitted, implying that all disks are to be searched. For example, "//edward" would cause a search for a primary directory named "edward" starting its search at "sys" and then "users" where "//edward" is found.

- When a pathname does not begin with a slash, the file search operation begins with your current directory. You can think of your current directory as your "location" in the file system at the time you use the pathname. For instance, if your current directory was "/users/edward" and you used the name "p2", you would get the file "p2" under "/users/edward"; however, if your current directory was "/users/kate" you would get the file "p2" under "/users/kate". Later, you will see how find out the name of your current directory and how to "move around" the file system by changing your 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. (For revisions of Primos below Rev 19, which have passworded directories, you may have to specify nodes as a name possibly followed by a colon (":") and a password.) For example

```
kdir
extra
sys:xxxxxx (pre-Rev 19 Primos)
```

  are nodes.

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

name." For example,

```
/sys/bin
```

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

```
//extra/users
```

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

```
p2
```

names the file "p2" in "/users/edward" if your current directory is "/users/edward" or the file "p2" in "/users/kate" if your current directory is "/user/kate".

```
kdir:pwd/kfl
```

names the file "kfl" in the directory "kdir" (with password "pwd"), in a pre-Rev 19 Primos file system, only if your current directory is "/user/kate".

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". which could be referred to on a standard system as "//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", "=bin=". which would correctly reference "bin" regardless of its location. 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.

A word on upper and lower case: The Primos file system does not distinguish between upper and lower case, thus "//BIN", "//Bin", and "//bin" are all the same. However, the Subsystem template mechanism does distinguish between upper and lower case, so "=BIN=", "=Bin=", and "=bin=" are three different templates. This can be a subtle trap for the unwary.

Directories

Directories can be created with the 'mkdir' ("make directory) command; e.g.

```
] mkdir /users/edward
```

will create the directory "edward" under the master file directory "users". The command

```
] mkdir edward
```

will create the directory "edward" in the current directory.

As mentioned above, the 'lf' command can be used to list information about directories and the files and subdirectories contained therein; e.g.,

```
] lf /users/edward
] lf edward
```

Finally, directories, like files, can be deleted with 'del'. However, unlike files, directories cannot be deleted until all the files and subdirectories contained in them have been deleted. If "edward" is an empty directory it can be deleted with the command

```
] del edward
```

If "edward" is not an empty directory then it can be deleted with the command

```
] del -ds edward
```

| where the the "-ds" specifies to delete the contents of the
* directory, then the directory itself.

Moving Around in the File System

You can change your current directory with the 'cd' (change directory) command. Simply type 'cd' followed by the pathname of the directory to which you wish to move and, as long as its a valid directory name, you will be promptly deposited there; e.g.

```
] cd /users/edward
] cd kdir
```

Note that in the second example, since the pathname 'kdir' is not preceded by slashes, your current directory must be "/users/kate" for it to work.

You can move "up" in the file system with

```
] cd \
```

For instance, if you were in "/users/kate/kdir" and you typed "cd \", your current directory would then be "/user/kate".

Finally, if you get lost, you can find out where you are with the command

```
] cd -p
```

It will print the full name of your current directory.

Subsystem Communication Services

Communication utilities are becoming increasingly important in today's computer systems. The Subsystem, in keeping up with the times, offers as its most important communication facilities a postal and news service and a real-time communication system.

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.)

If you have declared the shell variable "_mail_check", (but not set it), the shell will check your mail file every 60 seconds, to see if it has increased in size. If it has, the shell will tell you, "You have new mail." You may then read your mail with the 'mail' program. If you want it to check you mail more frequently, or less frequently, you may set it to the number of seconds between checks. For instance:

```
declare _mail_check = 300  # check mail every five minutes
```

By default, "_mail_check" will not be set for new users, so the shell will only check your mail once, when the Subsystem is first cranked up. (See the User's Guide for the Software Tools Subsystem Command Interpreter for a more detailed discussion of the use of shell variables.

Due to the nature of the file system, setting "_mail_check" to less than four will be no different than setting it to four. At Georgia Tech, the mail directory is shared among several machines, so, since the shell has to go across Primenet, you should set "_mail_check" to a fairly large value, say 300, for once every five minutes.

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.

If you find that you have published the wrong article or if you want to remove an outdated one, you can do a

] **retract <article number>**

to remove the article, where <article number> is the sequence number obtained from the news index ("news -i" will give you such an index). A retraction notice will be delivered to all subscribers who have seen the article, and the article will simply be removed from the news boxes of subscribers who have not yet seen it. If you are only removing an outdated article, then using

] **retract -q <article number>**

will quietly remove all traces of the article, leaving no retraction notices behind to disturb those who have seen it.

Subsystem Real-Time Communications

As if 'mail' and 'news' were not enough, the Subsystem offers still another way to communicate with your fellow user, by means of the 'to' command. 'To' allows you to communicate with other logged-in users on a real-time basis; messages that are sent to another user by the command

] **to login_name <message>**

will be retrieved by the user whose login name is "login_name" the next time his shell is ready for a command. Contrast this behavior to that of 'mail', where the message must be retrieved by an action on the part of the addressee. If <message> contains any of the shell's metacharacters, it must be enclosed in quotes, as in:

] **to allen "Where are you, and what are you doing?"**

If you want to send a multi-line message, 'to' will read your message from standard input (just like most other Subsystem programs), so that the only argument you would specify in this case would be the login_name. As always, a control-c in column 1 will generate an end-of-file to terminate your input.

Messages are only retrieved when the shell is ready for the next command, so a user who is running a long program may not see your messages until long after you have sent them. If he logs out before he sees your messages, they will sit there, waiting to be retrieved until the next time he logs in.

To alleviate this somewhat, the Subsystem screen editor, 'se', will notify you if there is a message waiting for you. See the "om" command in the help on 'se' for details.

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; it presents 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

Programs in the Subsystem do not have to be written to read and write to specific devices. In fact, most commands are written to read from "anything" and write to "anything." Only when the command is executed do you specify what "anything" is, which could be your terminal, a disk file, device etc. "Anythings" are more formally known 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, you can change this through a facility known as "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".

You need to be careful with the files to which you redirect i/o. In the above example, if a file by the name of "total" already exists, its contents will be replaced by the concatenation of "file1", "file2" and "file3". Similarly if you try the command

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

disaster results as it first clobbers "file1", destroying its contents for good.)

```
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.

There are other, more complicated i/o redirectors available to you. See the User's Guide for the Software Tools Subsystem Command Interpreter for a full, in-depth discussion of the facilities the shell provides.

Using Primos from the Subsystem

Unfortunately, a few functions of Primos and its support programs have not been neatly bundled into the Subsystem. The Subsystem commands that address this problem are the topic of this section.

Executing Primos Commands from the Subsystem

The commands 'x' and 'primos' can be used to access Primos programs and commands without having to go through the work of leaving and re-entering the Subsystem.

'X' may be used in either of two ways; the first is

```
x  Primos-command
```

This is the method of choice for executing a single Primos command. You will probably want to put double quotes around the Primos command to keep the Subsystem from becoming annoyed at metacharacters such as ">" and "<" being used in the Primos command.

The second way to use 'x' is to use it without arguments. Here is an example:

```
] x
ok, status net
ok, message -9 now
Hi there.
ok, <control-c>
]
```

This method allows many Primos commands to be executed. In this case, 'x' reads a line at a time and passes it to the Primos command interpreter for execution. If the Primos return code is positive, 'x' continues to the next line; if not, 'x' exits to the Subsystem. 'X' will also return to the Subsystem when it encounters a control-c or a Primos REN. The prompt, "ok,", is in small letters to remind you that it is the command 'x' producing the prompt and not Primos.

The second command, 'primos', invokes a new level of the Primos command interpreter from the Subsystem. (With this command, the Primos command interpreter prints the prompt "OK," and your commands are received directly by it.) You can return to the Subsystem by typing the Primos REN command.

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.

Developing Programs

To acquaint you with the several steps of program development, we present an example in which we develop a simple Ratfor program. We use a Ratfor example here because Ratfor is the most widely used language in the Subsystem --- but for a few lines here and there, the entire Subsystem is written in Ratfor. If you want to learn more about Ratfor programming, you can read the **User's Guide for the Ratfor Preprocessor**. Meanwhile, on with the example . . . .

The Subsystem Text Editor

The first program most users will see when they wish to create another program is 'ed', the Subsystem text editor, or if you have a crt, 'se', the screen editor. A complete description of either is beyond the scope of this tutorial, but a short list of commands (accepted by both the line editor and full screen editor) and their formats, as well as an example using 'ed,' should help you get started. For more 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. Anything previously in the buffer is lost.
(.,.)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.

You should note 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"s, 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)
11                                                      (2)
8                                                       (3)
   call print (STDOUT, "Now: *s*n"s, now                (4)
s/, now/, now)/p                                        (5)
   call print (STDOUT, "Now: *s*n"s, now)               (6)
w                                                       (7)
11                                                      (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,' should be used to compile Subsystem programs, since it selects several useful compiler options and standardizes the compilation process:

```
] fc now.f
0000 ERRORS [<.MAIN.>FTN-REV18.4]
]
```

All of the garbage between the "fc" and the "]" prompt is stuff produced 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.' 'Fc' has left the output of Fortran in the file "now.b". We will use 'ld's "-o" option to select the name of the executable file:

```
] ld now.b -o now
[SEG rev 19.2.GT]
# vl #
$ co ab 4001
$ sy swt$cm 4040 40000
$ sy swt$tp 2030 120000
$ mi
$ s/lo now.b 0 4000 4000
$ s/lo 'lib>vswtlb' 0 4000 4000
$ s/li 0 4000 4000
LOAD COMPLETE
$ ma 6
$ re
# sh
TWO CHARACTER FILE ID: ..
# delete
# q
]
```

Again, all the noise between "ld" and "]" comes from 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.

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), "return" from the main program (Pascal and PL/I G), or call the subroutine "swt". Specifically, the Primos routine "exit" must 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 cannot handle all aspects of the Subsystem files. If, however, programs must use native i/o routines, remember that they must inform their native i/o routines of the Subsystem by calling the proper initialization routines (see Subsystem Interface Subroutines in the table below), or they will not be able to take advantage of standard input, standard output or any of the other i/o related features provided by the Subsystem.

The Subsystem interfaces available for Primos languages and utilities are summarized below:

Language or Utility	Primos Subsystem Interface	Primos Commands Interfaced	Subsystem Interface Subroutines
C	xcc	CC	-
	xccl	CC, SEG	
Cobol	cobc	COBOL	-
	cobcl	COBOL, SEG	
Database	fsubc	FSUBS	-
	fdmlc	FDML	
	fdmlcl	FDML, FTN, SEG	
	csubc	CSUBS	-
	cdmlc	CDML	
	cdmlcl	CDML, COBOL, SEG	
	ddlc	SCHEMA	-
Debugger	dbg	DBG	-
	vpsd	SEG	
Fortran 66	fc	FTN	init$f,
	fcl	FTN, SEG	geta$f
Fortran 77	f77c	F77	init$f,
	f77cl	F77, SEG	geta$f
Loader	ld	SEG	-
Pascal	pc	PASCAL	init$p,
	pcl	PASCAL, SEG	file$p, geta$p
PL/P	plpc	PLP	-
	plpcl	PLP, SEG	
PL/1 G	plgc	PL1G	init$plg,
	plgcl	PL1G, SEG	geta$plg
Prime Assembler	pmac	PMA	-
	pmacl	PMA, SEG	
SPL	splc	SPL	-
	splcl	SPL, SEG	

Use 'help' or refer to the Subsystem Reference Manual for a complete description of Primos/Subsystem interface commands and

Subsystem interface subroutines.

Errors

Although the Software Tools Subsystem provides a very nice program development and applications environment, Murphy's Law indicates that things will still go wrong. "To err is human...", so it is best to anticipate errors, and know what to do when you encounter them. This section indicates some of the more common causes of errors, and what to do when you encounter them. The non-technical user can probably skip this section.

Recovering from Errors

Everyone encounters errors sometimes. Eventually you will divide by zero, or try to execute source code, or do something even worse. Primos will make you pay for little mistakes like this, and typically will kick you out of the Subsystem. Although graceful recovery is sometimes possible, more often than not, it is so tedious that it is easier just to start all over again.

When an error occurs, and after you have satisfied yourself reasonably well as to why, the "cure-all" for Subsystem problems is simply to type:

```
swt
```

Sometimes, this will not work. The stack may screwed up, or something else may have gone terribly wrong. To clear everything completely, and restart the Subsystem, type the following:

```
OK, rls -all
OK, dels all
OK, swt
```

All error messages that cause an exit to Primos (signalled by the "OK," or "ER!" prompts) 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 may be unintelligible to the novice. The rest of this section contains a brief description of some of those messages. You need not read what follows if you don't make programming errors.

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.

```
POINTER FAULT
```

usually indicates that a subprogram was called that was not 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 referencing in a subprogram an argument that was not supplied in the subprogram call; e.g., 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.

```
ACCESS VIOLATION
ILLEGAL INSTRUCTION AT <address>
ILLEGAL SEGNO
PROGRAM HALT AT <address>
```

all can result from a subscript exceeding its bounds. Because the program may have destroyed part of itself, the memory addresses sometimes given may well be meaningless.

To find errors such as these, time can often be saved by using a program trace. In addition to the manual insertion of 'print' statements in the source program, both 'rp' and 'fc' have options to produce a program trace. The "-t" option will cause 'rp' to insert code to trace the entry and exit of subprograms. (One should note that only subprograms preprocessed with the "-t" option will be traced.) 'Fc' will emit code to produce a Fortran statement-label and assignment trace when called with the "-t" option. Although this trace will contain the statement labels generated by 'rp', the intermediate Fortran code may be listed and the execution path followed.

See the subsection on debugging in the Application Notes section of the User's Guide for the Ratfor Preprocessor for more suggestions on finding and eliminating errors in your ratfor programs.

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 that 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 ninth 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 path-names 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.

Version 8:

- Features: Additional I/O speed, reduced working set, support for PL/I G, Pascal, Fortran 77, DBG, improved error handling, terminal type handling, virtual terminal handler.

- Language: Ratfor, Assembler (PMA), and some PL/I.

Version 9:

- Features: Increased security for shared segments, improved shell, extended text editors and formatter, access to new Primos file system features, some support for Prime's C compiler, a high precision mathematics library, and an improved stacc.

- 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.