GIT-ICS-84/18

SOFTWARE TOOLS SUBSYSTEM
USER'S GUIDE

4th Edition

September, 1984

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

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
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

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 "*S*oftware *T*ools") 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 "*l*ist *f*iles") 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.

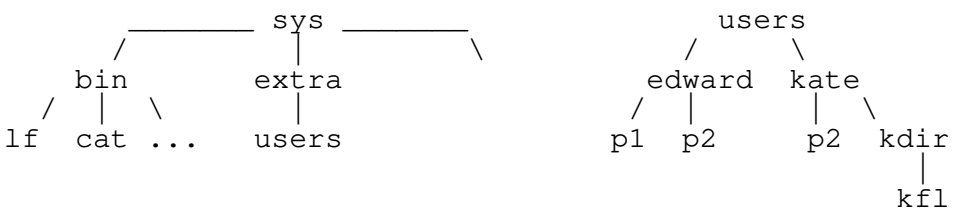
(2) List the names of all files in the current directory, including all information that is known about each file.

(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' ("*mak*e *dir*ectory) 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.

TABLE OF CONTENTS

User's Guide to the Primos File System

Perry B. Flinn
Jefferey S. Lee

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

September, 1984

Foreword

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*, the *Reference Guide*, *File Management System* (Prime publication number FDR3110), and the *Prime User's Guide* (Prime publication number DOC4130).

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.

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:

$ & - * . / _

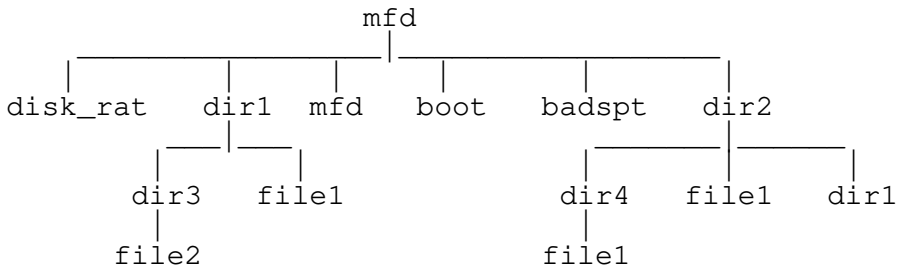
The first character in the entryname must not be a digit. Also, no distinction is made between upper- and lower-case letters. Thus "file_name" and "FILE_NAME" are the same.

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 *m*aster *f*ile *d*irectory. 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 r*ecord *a*vailability *t*able) 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 set to whatever the system administrator decided when he created your account. 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

In versions of Primos before Revision 19, to guard your files from unwanted perusal or alteration, the file system included a basic access control mechanism that provided two levels of protection to each file. As part of this mechanism, each directory had 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 was created its owner password was blank and its non-owner password was zero; these were the default values. But if the passwords had other than default values, then before you could successfully attach to the directory, you had to prove your worthiness to do so by citing one of them. If you cited the owner password, then you were attached to the directory with "owner status;" if it's the non-owner password that you cited, then you were attached with "non-owner status." If you failed to cite either password, then unless one of them had a default value your attempt would be in vain. Just what status you attained when attaching to a directory bears upon the kinds of things you could do to the files it contains.

For the purposes of password 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.

Revision 19 of Primos introduced Access Control Lists (ACL's). Unlike the password protection previously described, ACL's allow specific permissions on files to be granted on a per-user basis, instead of a broad class of permissions being granted to anyone who happens to know, or guess, the password. They also allow better control over permissions given to users. Previously, in order to allow a user to create files in a directory, he was implicitly given the right to delete any other files in that directory, also. With ACL's, this is no longer the case.

An ACL consists of a list of up to 32 identifiers and privileges associated with each of the identifiers. An identifier can be a user's login name or it can be a group identifier associated with several users. If a user's name and associated group are both in an ACL, the user's login name takes precedence. The seven different privileges associated with ACL's are:

add This privilege is associated with a directory and allows the user to create a new file within that

directory. Once the file is created, the user has full read/write access to the file until the file is closed, at which point other privileges determine the accessibility of the file.

delete This privilege is associated with a directory and allows the user to delete an existing file within that directory.

list This privilege is associated with a directory and allows the user to list the contents of the directory (like with 'lf').

protect This privilege is associated with a directory and allows the user to set ACL protection for objects in the directory.

read This privilege is associated with a file and allows the user to open a file for reading or to execute a file. The user must first be able to attach to the directory before he can read the file, which implies *use* privilege (see below).

use This privilege is associated with a directory and allows the user to attach to the directory (like with 'cd'). In order to access a file or a directory, the user must have *use* privilege on all intervening directories between the MFD and the desired file or directory.

write This privilege is associated with a file and allows the user to open a file in write mode or to truncate a file.

Associated with the ACL is its type. There are five different types of ACL's. The first type is the *specific ACL*. This gives protection on one specific file object and is associated with only that object. If the object is deleted then the specific ACL goes away, also.

The second type of ACL is the *default specific ACL* where a specific ACL is set on an ancestor directory of the current object. If the object is not protected by a specific ACL or an access category (the next type), then it is given the same protection as the ancestor directory.

The third type of ACL is the *access category* ("acat"). An access category, unlike the two previous types, may protect many objects at one time with the same protections. An acat appears in the file system as a file that cannot be read or written, and its name must end in ".acat". It is a separate type of file system object (just as in 'lf -l' listings, DAM files are different from SAM files -- acats are of type ACT). An access category need not protect any object since it exists independant of any other object in the file system. If an access category is deleted, any object that it was protecting becomes default

protected, or becomes protected by the directory that contains it.

The fourth type of ACL is the *default access category*. This is an access category that protects a directory that contains other objects that are then protected by default.

The last type of ACL is the *priority ACL*. This is an ACL that is set on an entire disk partition by the system administrator, normally at boot time. Any rights given by a priority ACL override any rights given by any other ACL's.

In order to allow for a gradual change from the older versions of Primos to Revision 19, it is possible for password directories and ACL's to exist in the same system, although password directories will eventually be unsupported. There is a restriction in that ACL directories may contain both password and ACL directories but password directories may not contain ACL directories. In order for any directory to be an ACL directory on a logical disk, the MFD of that partition has to be ACL protected. Password directories also overcome some of the limitations of ACL's. If an ACL gives someone the privilege of writing a file, then under all circumstances they are allowed to write the file. If the file is in a password directory, though, they may only write the file if they know the password. This means that a password can be nested deep in a program that is used to control their access to a file, even if the person running the program does not know the password.

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 *back*slashes (\). 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

The following discussion is applicable only for password protected directories, since ACL protected items do not need pas-

swords. 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, or if the directories are ACL directories. 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 is constructed from letters, digits and underscores and is always enclosed in equals bars (=). (Templates do *not* have to begin with a letter). 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.

=home=
The current user's initial login directory (set by the system administrator when he created the account). This may vary on a per-user per-project basis. I.e., the system administrator may set it up so that the initial login directory for a given user is different

for different projects.

=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 three-digit number in the range 001-128 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 system 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 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. If you wish to create templates that will take effect immediately, use the 'template' command (do a 'help template' for details).

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
   macros    //smith/misc/macros     #Smith's macros
```

The above example defines a template "macros" referring to the file "//smith/misc/macros." A quick perusal of the contents of "=template=" should clear up any lingering questions you may

have. Just for convenience, all dynamic and system 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 stan-

dard 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. This is the only difference 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.

=bug=
: The directory in which the Subsystem bug reporting mechanism collects bug reports.

=cldata=
: Defines the location of the Primos CLDATA structure, used internally by the Subsystem command interpreter (shell).

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

=crondir=
: The directory where the 'cron' program creates temporary files for phantoms.

=cronfile=
: The file that contains the directive lines for the 'cron' program.

=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.

=ebin=
: A directory of programs called by shell programs in "=bin=".

=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.

=histfile=
The current user's saved command history file.

=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 **include**d by Ratfor and C 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.

=new_words=
If this template exists and describes a legal file name, the 'spell' program will write a copy of unrecognized words to this file.

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

=newcmdnc0=
The Subsystem directory into which newly-compiled Subsystem files that belong in "cmdnc0" are placed during a recompilation of the entire Subsystem.

=newebin=
The Subsystem directory into which newly-compiled commands destined for "=ebin=" 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.

=newsystem=
The Subsystem directory into which newly-compiled Subsystem files that belong in "system" are placed during a recompilation of the entire Subsystem.

=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 three-digit number in the range 001-128 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).

=statistics=
The system template which controls whether or not command statistics are to be kept. (See the "Application Notes" section of the *Command Interpreter User's Guide*.)

=statsdir=
The directory where command statistics are recorded. (See the "Application Notes" section of the *Command Interpreter User's Guide*.)

=syscom=
The directory where the Primos subprogram keys (predefined constants) are stored.

=sysname=
This is the system's Primenet node name, if it is a network system.

=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.

=ttypes=
A file containing a list of terminals supported by your Subsystem and their characteristics.

=ubin=
By convention, the user's private command directory.

=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 aficionados 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). These entrynames and associated values must be separated by slashes or blanks, e.g. "/dev/lps/b/TECH/" or "/dev/lps/b TECH."

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.

TABLE OF CONTENTS

Introduction to the Software Tools Text Editor

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

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:

```
3
```

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 response 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, "^", or a single minus sign, "-", 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
-
The quick brown fox
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 *after* 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: *back*slashes search *back*ward. 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.

In pattern searches, and in other commands which we will get to later, 'ed' allows you to leave off the trailing the delimiter. I.e., instead of typing

/jumps/

you can type

/jumps

to search forward for the first occurrence of the pattern "jumps". Similarly, to search backwards, you may type

\quick

instead of

\quick

This feature can save considerable time and frustration when you are doing some involved editing, and accidentally leave off the trailing delimiter ("/" or "\"). The rest of this guide will continue to use examples with the trailing delimiter, but you do not have to in your actual editing.

'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" patterns.) 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/
```

Some more 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
```

In the replacement part of a substitute command, the character "&", *as the only character in the pattern*, means "the replacement part of the previous substitute command". (This allows an empty replacement pattern as well.) Thus, to step through the buffer, and change selected occurrences of one pattern into another, you might do the following:

```
/pat1/
Line containing pat1.
s/pat1/stuff1/p
Line containing stuff1.
//
Another line with pat1.
//
Yet another line with pat1.
s//&/p
Yet another line with stuff1.
```

You may leave off the trailing delimiter in the substitute command. This will cause 'ed' to print out the changed line. I.e., "s/stuff/junk" is the same as "s/stuff/junk/p".

```
/quick/
The quick brown fox
s/quick/really fast
The really fast brown fox
```

If you wish to delete an occurrence of a pattern, you may leave it off. 'Ed' will delete the pattern, and then print the line. In other words, "s/stuff" is the same as "s/stuff//p".

```
p
The quick brown fox
s/quick
The  brown fox
```

Finally, you may leave off the search pattern and replacement string entirely. If you do, 'ed' will behave as though you had typed "s//&/p", in other words, substitute the previous replacement pattern for the previous search pattern, and print.

```
1,$d
a
line 1
line 2
.
1s/line/this is &/p
this is line 1
2s
this is line 2
```

This can save considerable typing.

Line Changes, Insertions, and Concatenations

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.

The join command "j" can be used to put two or more lines together into a single line. It works like this:

```
starting-line,ending-line j[/string[/]]
```

The defaults for starting-line and ending-line are "^" and "." respectively, that is, "join the line before the current line to the current line". You may specify in "string" what is to replace the newline(s) which currently separate the lines which are to be joined. If you do not specify any string, 'ed' will replace the newline with a single blank. If you do specify a string, you may leave off the trailing delimiter (which can be any character), and 'ed' will print out the resulting joined line. An extended example should make this clear:

```
1,$p
The quick brown fox
    jumps over
   the lazy dog.
2,$s/%  *//
1,$p
The quick brown fox
jumps over
the lazy dog.
1,2j
The quick brown fox jumps over
1,2j/ the back of /p
The quick brown fox jumps over the back of the lazy dog.
```

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.

More Line Number Syntax

So far, the commands that you have seen can be given either no line numbers elements (the command tries to make an intelligent assumption about the line(s) on which to perform an operation), one line number element (the command acts only on that line), or two line numbers separated by a comma (the command acts on the given range of lines). There is one more way to specify line number elements, and that is to separate them by a semicolon. When line number elements are separated by semicolons, each line number element encountered sets the "current line" marker before the next line number element is evaluated. This is especially useful when using patterns as line number elements; some examples will illustrate what we mean.

Suppose that you wanted to print all the lines which lie between two lines, each containing the string "fred". An initial effort might yield the following command line:

```
/fred/,/fred/p
```

This, however, will only print out the first line which contains "fred" after the current line. This is because both patterns will start their search after the current line where the command was executed, instead of the second one starting where the first pattern was found. To correct this, we would issue the fol-

lowing:

```
/fred/;/fred/p
```

When the first occurrence of "fred" is found, the "current line" is set to that line, and the second occurrence of "fred" will be found starting at this new line. This will print the lines between two succeeding occurrences of "fred" from the current line.

As a final example, suppose that we wanted to print the lines between the second and third occurrence of "fred" after the current line; to do this, we would do:

```
/fred/;//;//p
```

The first pattern search would find "fred", the next two null strings will cause the previous pattern ("fred") to be searched for again, each time resetting the "current line" marker. Of course, the command "p" may be replaced by any command you wish.

For both comma-separated and semicolon-separated line number elements, you may specify more than two such elements, as the above example shows; only the last two such elements will be used as the range for the given command. In general, using more than two line number elements separated by commas is not too useful, because the "current line" is not modified for any of the line number expression evaluations. Also, using integer line numbers means that multiple expressions (more than two) are not useful, since the equivalent behavior can be obtained by specifying only the last two line numbers.

Escaping to the Shell

With Version 9 of Software Tools and Revision 19.2 or later of PRIMOS, it is now possible to call the Software Tools Subsystem command interpreter (the shell) from within a program.

'Ed' provides access to this facility with the shell escape "~" command. It works like this:

```
~[<Software Tools Command>]
```

If present, the <Software Tools Command> is passed to the shell to be executed. Otherwise, an interactive shell is created. After either the command or the shell exits, 'ed' prints a "~" to indicate that the shell escape has completed. If the first character of the <Software Tools Command> is a "!", then the "!" is replaced with the text of the previous shell command. An unescaped "%" in the <Software Tools Command> will be replaced with the current saved file name. If the shell command is expanded, 'ed' will echo it first, and then execute it.

This feature is useful when you want to temporarily stop editing and do something else, or find something out, without

having write your file and leave the editor.

```
{editing session}
~lf -l %
lf -l file
sam  a/r  06/17/84  16:25:08        19463  sys  file
~
```

For a deeper discussion of using the shell from within a program, see the help on the 'shell' subroutine. In particular, due to operating system constraints, you *must not* run another instance of the editor from the new shell, or you will end up clobbering your current edit buffer.

WARNING: Until Prime supports EPFs, and the editor is reloaded in EPF format, you *must not* run any external commands (like 'lf') from a shell started from 'ed'. If you do, the new program will load *over* 'ed', and wipe out your current editing session. You can use commands which are internal to the shell (like 'cd'), without any ill effect. This restriction, for various arcane reasons, does *not* apply to the Subsystem screen editor, 'se'.

In essence, this feature is provided in the editor with an eye to the future.

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'.

The screen editor has a built-in "help" facility, which documents all the commands and options. When in doubt, type "help", and the help screens should guide you to further information on what you need to know.

Invoking the Screen Editor

You can invoke the screen editor with either of the following commands:

] **se**

or

] **se myfile**

'Se' will automatically fetch your terminal type from the Subsystem. If you never told the Subsystem your terminal type or set an unknown terminal type with the 'term' command, 'se' will prompt you for another terminal type; if you type a '?', 'se' will give you a list of possible terminal types and prompt you again for yours.

'Se' can also be invoked by the command 'e'. 'E' remembers the name of the last file you edited, so if you don't specify a file, 'e' will enter the last file you edited.

Using 'Se'

'Se' first clears the screen, draws in its margins, and executes the commands in the file "=home=/.serc", if it exists. It then processes the command line, obeying the options given there, 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' automatically displays an error message in the status line. 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 used to modify an entire line rather than just add to the end of the line. Also, if you use "v" over a range of lines and find that you want to terminate the command before all lines have been considered, the control-f key is used instead of a

period.

. If a file name is specified in the "w" command and the file already exists, 'se' will display "file already exists"; entering the command again (by typing a NEWLINE) will cause the file to be overwritten. Given the command "w! <file>", 'se' will never warn about the destruction of an existing file.

Keeping these few differences in mind, you will see that 'se' can perform all of the functions 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
```

Finally, 'se' accepts "#" as a line number element; it always refers to the first line on the screen; like the line number letters, it may be used in any context which requires a line number element or expression.

Case Conversion

When 'se' is displaying upper-case letters for line numbers, it accepts command letters only in lower case. For those who edit predominantly 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.

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-g 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.

control-_ Insert a newline.

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. The most commonly used is the control-v. A newline (or carriage-return) can be used but beware that it deletes all characters over and to the right of the cursor.

control-v Terminate.

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

The .serc File

When 'se' starts up, it tries to open the file "=home=/.serc". If that file exists, 'se' reads it, one line at a time, and executes each line as a command. If a line has "#" as the *first* character on the line, or if the line is empty, the entire line is treated as a comment, otherwise it is executed. Here is a sample ".serc" file:

```
# turn on unix mode, tabs every 8 columns, auto indent
opu
ot+8
oia
```

The ".serc" file is useful for setting up personalized options, without having to type them on the command line every time, and without using a special shell file in your bin. In particular, it is useful for automatically turning on UNIX mode for Software Tools users who are familiar with the UNIX system.

Command line options are processed *after* commands in the ".serc" file, so, in effect, command line options can be used to override the defaults in your ".serc" file.

NOTE: Commands in the ".serc" file do *not* go through that part of 'se' which processes the special control characters (see above), so *do not* use them in your ".serc" file.

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.
g	controls the behavior of the "s" (substitute) command when it is under the control of a "g" (global) command. By default, if a substitute inside a global command fails, 'se' will not continue with the rest of the

lines which might succeed. If "og" is given, then the global substitute will continue, and lines which failed will not be affected. Successive "og" commands will toggle this behavior. An explanatory message is placed in the status line.

h[<baud>] lets the editor know at what baud rate you are receiving characters. Baud rates can range from 50 to 19200; the default is 9600. This option allows the editor to determine how many, if any, delay characters (nulls) will be output when the hardware line insert/delete functions of the terminal are being used (if available). Use of the built-in terminal capabilities to insert/delete lines speeds up editing over slow-speed lines (i.e., dialups). Entering 'oh' without an argument will cause your current baud rate to appear on the status line.

i[a | <indent>] selects indent value for lines inserted with "a", "c" and "i" commands (initially 1). "a" selects auto-indent which sets the indent to the value which equals the indent of the previous line. If <indent> is an integer, then the indent value will be set to that number. If neither "a" nor <indent> are specified, the current value of indent is displayed.

k Indicates whether the current contents of your edit buffer has been saved or not by printing either a "saved" or "not saved" message on your status line.

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 in the status line. <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.

lm[<col>] sets the left margin to <col> which must be a positive integer. This option will shift your entire screen to the left, enabling you to see characters at the end of the line that were previously off the screen; the characters in columns 1 through <col> - 1 will not be visible. You may continue editing in the normal fashion. To reset your screen enter the command 'olm 1'. If <col> is omitted, the current left margin column is displayed in the status line.

m[d] [<user>] displays messages sent to you by other users (via the 'to' command) while you are editing. When a message arrives while you are editing, the word "message" appears on your status line. To send other users messages while inside of the editor, you can insert the text of your message into the edit buffer, and then issue the command "line1,line2om <user>", where "line1" and "line2" are the first and last lines, respectively, of where you appended your message in the edit buffer and "<user>" is the login name or process id of the person to whom you want to send a message. The given lines are sent and deleted from the edit buffer. To prevent the lines from being deleted after they are sent, use the command line "line1,line2omd <user>"

p[s | u] converts to or from UNIX (tm) compatibility mode. The "op" command, by itself, will toggle between normal (Software Tools mode) and UNIX mode. The command "opu" will force 'se' to use UNIX mode, while the command "ops" will force 'se' to use Software Tools mode.

When in UNIX mode, 'se' uses the following for its patterns and commands:

?pattern[?] searches backwards for a pattern.

^ matches the beginning of a line.

. matches any character.

^ is used to negate character classes.

% used by itself in the replacement part of a substitute command represents the replacement part of the previous substitute command.

\(<regular expression>\) tags pieces of a pattern.

\<digit> represents the text matched by the tagged subpattern specified by <digit>.

\ is the escape character, instead of @.

t copies lines.

y transliterates lines.

~ does the global exclude on markname (see the "!" command, in the help on 'ed').

![<Software Tools Command>] will create a new instance of the Software Tools shell, or execute <Software Tools Command> if it is present (see the "~" command, in the help on 'ed').

All other characters and commands are the same for both

UNIX and normal (Software Tools) mode. The help command will always call up documentation appropriate to the current mode. UNIX mode is indicated by the message "UNIX" in the status line.

UNIX mode is available *only* in 'se'. This extension is not available in 'ed'.

s[pma | ftn | f77 | s | f] sets other options for case, tabs, etc., for one of the three programming languages listed. The option "oss" is the same as "ospma" and the option "osf" is the same thing as "osftn" (the corresponding command line options are "-ss" and "-sf"). If no argument is specified the options effected by this command revert to their default value.

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

(Files can be edited with control characters only when you are in overlay mode, which you can enter with the 'v' command. A control-v will exit overlay mode and put you back into command mode. While in command mode you can use these characters to edit your command.)

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 at 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. An error message is generated if the editing buffer contains text that has not been saved. The enter command may be resubmitted after the error message, in which case it will be obeyed. The "enter now" command "e!" may be used to avoid the error message.
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. You may specify in "stuff" what is to replace the newlines that previously separated the lines. The default is a single blank. If you use the default, 'ed' automatically prints out the result. If the "p" option is used, the resulting line (which becomes the new current line) is printed. Thus "j" and "jp" are equivalent to "j/ /p". In general, 'ed' and 'se' will supply trailing delimiters for you. So "j/" is the same as "j//", i.e. replace the newline(s) with nothing (delete them).
.,.	km	marK The specified lines are marked with 'm' which may be any single character other than a newline. If 'm' is not present, the lines are marked with the default name of blank. The current line pointer is never changed.
none	l	Locate "l" will print the first line of the file =installation=. This is so that one can tell what machine he is using from within the editor. This is particularly useful for installations with many machines that can run the editor, where the user can switch back and forth between them, and become confused as to where he is at a given moment.

.,.	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[m]	Name If 'm' 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 of blank. In 'ed', if 'm' 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. In 'ed', options "d", "g", "k", and "p" are available. Options "d", "g", and "k" are the same as in 'se'. In 'ed', option "p" sets the prompt to be used (useful for the user who is disturbed by 'ed's quiet behavior). The prompt can be set by the command "op/string[/]", which sets the prompt to "string". The trailing delimiter is optional. If no string is given, the prompt is set to "* ". An empty string ("op//") restores 'ed's no prompting behavior. Successive "op" commands will toggle prompting mode. In 'se', the "op" command controls what metacharacters are used for pattern matching.
.,.	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. An error message is generated if the editing buffer contains text that has not been saved. The quit command may be resubmitted after the error message, in which case it will be obeyed. The "quit now" command "q!" may be used to avoid the error message.

. 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. In 'ed', if you leave off the trailing slash, the result of the substitute will be printed automatically. Thus "s/junk/stuff" is entirely equivalent to "s/junk/stuff/p". If you type an "s" by itself, without a pattern and replacement string, 'ed' will behave as though you had typed "s//&/p", i.e. substitute the previous replacement pattern for the previous search pattern, and print.

.,. 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. Again, if you leave off the trailing delimiter, 'ed' will print the result of the transliteration. In addition, like the "s" command, both the 'from' and 'to' parts are saved; "t//&/" will perform the same transliteration as the last one, and "t" is the same as "t//&/". The "&" is special if it is the only character in the 'to' part, otherwise it is treated as a literal "&". In Unix mode (for 'se' only), use "%" instead of "&". See *Software Tools* and the help on 'tlit' for some examples of character transliterations.

. 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-v may be used to quit overlay mode. 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. If "+" is given, the portion of the buffer is appended to the file; otherwise the portion of the buffer replaces the file. In 'se' only, if "!" is present, an existing file specified in the command is overwritten without comment. If "filename" is not present, the specified lines will be written to the current file name specified on the status line.

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.

.,. zb<left>[,<right>][<char>] draw Box
In 'se' only, a box is drawn using the given <char> (blank by default, allowing erasure of a previously-drawn box). Line numbers are used to specify top and bottom row positions of the box. <Left> and <right> specify left and right column positions of the box. If second line number is omitted, the box degenerates to

a horizontal line. If right-hand column is omitted, the box degenerates to a vertical line.

.	=[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,$	!mcommand	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 'm'.
1,$	'mcommand	Global on markname Similar to the 'g' prefix except that 'command' is performed for all lines in the range that have the mark name 'm'.
.	:	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.
none	~[<Software Tools Command>]	Escape to the shell If present, the <Software Tools Command> is passed to the shell to be executed. Otherwise, an interactive shell is created. After either the command or the shell exits, 'ed' prints "~" to indicate that the shell escape has completed. For a command, 'se' asks you to type a newline before redrawing the screen, but for an interactive shell, 'se' will redraw the screen immediately. If the first character of the <Software Tools Command> is a "!", then the "!" is replaced with the text of the previous shell command. An unescaped "%" in the <Software Tools Command> will be replaced with the current saved file name. If the shell command is expanded, both 'ed' and 'se' will echo it first, and then execute it. Until EPFs are supported, when using 'ed', do not use the shell to execute

external commands. Internal commands (like 'cd') are OK. This does not apply to 'se'.

For a deeper discussion of using the shell from within a program, see the help on the 'shell' subroutine.

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).
-	number of the previous line in the buffer (same as ^).
#	number of the first line on the screen (only in 'se')
/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. The trailing "/" is optional.
\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. The trailing "\" is optional.
>name	number of the next line having the given markname (search wraps around, like //).
<name	number of the previous line having the given mark-name (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 preceding 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 specified 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. If "&" is the only character in the replacement part, however, then it represents the replacement part used in a previous substitute command.
@<digit>	Appearing in the replacement part of a substitute command, represents the text actually matched by the tagged sub-pattern specified by <digit>.

TABLE OF CONTENTS

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

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

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

September, 1984

Foreword

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 ninth 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

Commands

Input to the command interpreter consists of "commands". Commands, in turn, consist of a "command name", which is the name of an executable file. A command is executed simply by entering its name. For example,

] **help**

is a command that will describe how you can obtain online documentation.

Some commands may have arguments. Arguments are values supplied by you to the command. Arguments can be required or they may be optional in which case the system uses a default. In the above example when 'help' is invoked with no arguments the Subsystem assumes the command 'help help' (i.e. get me on-line documentation for the 'help' command). However, if you wanted on-line documentation for a specific command you would supply the command name as an argument, e.g.

] **help lf**

will describe the command that can be used to list information about files in a directory. Some commands may have options. Options are used to make the same command execute in slightly different ways. Options usually consist of one letter and are preceded by a dash. The command,

] **help -f file**

will list the names of commands and subroutines that may be associated with the keyword "file". The "-f" is an option and "file" is an argument. Commands, arguments and options are separated from each other by blanks.

Here is a final example:

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

'Lf' is used to list the names of your files. Executed without any arguments, 'lf' prints the files in your current directory, but (like 'help') 'lf' may be used with or without arguments and options.

How the Command Interpreter Locates a Command

Recall that you can access files by their entrynames only if they are located in your current directory. Without help from the shell this would also be true for commands. That is, in order to execute 'help' you would need to have a copy of the 'help' command in your current directory or you would have to enter its full pathname so that the shell could locate it in another directory. Obviously, neither alternative is desirable. In reality, the shell uses a "variable" called "_search_rule" to find commands like "help" in other directories. Each user has his own search rule. (Refer to the section in this guide entitled "Shell Control Variables" for more information.) The search rule tells the shell in what locations to look for commands, and if there is more than one location possible, it specifies the order in which the locations will be searched.

Most new users are given the search rule that causes the command interpreter to look for commands in the following five locations in the order shown:

1. The shell's internal library for an internal command (e.g, 'stop', 'set')
2. The user's variables currently stored in memory
3. The user's current directory
4. The Subsystem library containing locally supported external commands, "=lbin=" (e.g. memo, moot)
5. The Subsystem library containing standard external commands, "=bin=" (e.g. 'lf', 'help')

This variable is explained in more detail in the "Application Notes" section of this guide.

Beware that this flexibility can get beginners (and some experienced users) into trouble. With the search rule above, the command interpreter will always look in your current directory for a command before it looks in one of the Subsystem command directories. Therefore, if you create a file having the same name as a command, the shell will try its best to execute the contents of that file.

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
]
```

'Echo' is simply a command that types back its arguments. Obviously this example is not working as it should. The 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.

I/O Redirection

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.

Programs and commands in the Subsystem do not have to be written to read and write to specific files and devices. In fact most of them are written to read from "anything" and write to "anything." Only when the program is executed do you specify what "anything" is, which could be your terminal, a disk file, the line printer, or even another program. "Anything"s are more formally known as "standard input ports" and "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.

I/O Redirection to Disk Files or Devices

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'.

Recall that special pathnames which begin with "/dev" may refer to peripheral devices. For example, by redirecting output to "/dev/lps" you can print a file on the line printer.

```
] cat myfile >/dev/lps
```

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. See the Application Notes for examples.

I/O Redirection to other Commands

The last section discussed I/O redirection --- the process of making standard ports refer to disk files or devices, 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, in a single column. 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.

I/O Redirection for a Group of Commands

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."

I/O Redirection to a Command Argument

As mentioned before, some commands may have arguments. The standard output of a command (or a series of commands) can be used as an argument(s) by using the "function call" mechanism. 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" command:

```
] 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.

When setting a variable to a string containing unprintable characters, you may use a special mnemonic form to prevent having

to type the literal characters. For example

```
set crlf = "<cr><lf>"
```

sets the variable 'crlf' to a literal carriage return followed by a linefeed. There are times when this is not desirable, so to prevent the interpretation of the string, simply escape the start the start on the mnemonic with the Subsystem escape character (an '@'). To set set the variable 'crlf' to the literal string "<cr><lf>" you would type

```
set crlf = "@<cr>@<lf>"
```

The quotes in these two cases are necessary, otherwise the shell would try to interpret the '>' as an I/O redirector. If the string between the "<>" characters is not a legal ASCII mnemonic, no substitution will be made and the string will be passed unchanged.

Interrupts, Quits and Error Handling Mechanisms

Normally, if you interrupt a program, it will terminate and the next thing you will see is the Subsystem's prompt for your next command. However, by defining the shell control variable "_quit_action" in your "=varsdir=/.vars" file, the fault handler will, upon detection of the interrupt, prompt you as to whether to abort the current program, continue, or call Primos. For program errors, the fault handler will always ask whether you want to abort the program, continue, or call Primos (regardless of whether "_quit_action" is defined or not). The Application Notes discuss how to go about creating shell variables (which are kept in "=varsdir=/.vars" for storage between login sessions).

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 precedes 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.

. 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 overridden 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.) In addition to restricting a user to commands in specific directories, the system administrator can also restrict a user from using certain internal commands (and allow use of all other internal commands). This is accomplished by adding "qualifiers" after the internal command flag in the search rule. The qualifiers are characters representing the class of commands to be *excluded* in the search for internal commands to be executed. Qualifiers follow the "^int" flag, separated from it by a slash.

The following table summarizes the qualifiers and which internal commands they exclude :

Qualifier	meaning
a	access to arguments in shell files ('arg', 'args', 'argsto', 'nargs', and 'quote')
b	access to debugging commands ('dump' and 'shtrace')
c	access to flow of control commands ('case', 'elif', 'else', 'esac', 'exit', 'fi', 'goto', 'if', 'label', 'out', 'repeat', 'then', 'until', and 'when')
d	ability to change directories (via 'cd')
h	access to environment information ('date', 'day', 'echo', 'eval', 'installation', 'line', 'login_name', and 'time')
m	access to string manipulation functions ('drop', 'index', 'substr', and 'take')
q	ability to exit the shell (via 'stop')
s	access to variable setting commands ('forget', 'set', and 'sh')
v	access to variable manipulating commands ('declare', 'declared', and 'vars')
x	access to commands which allow execution of Primos commands ('dbg', 'primos', 'vpsd', and 'x')

For instance, if the system administrator wanted to keep someone from executing the Primos Fortran compiler directly, then the following search rule would accomplish this :

```
"^int/qxv,^var,=bin=/&"
```

The "q" qualifier prevents exit from the shell (so that you can't run the Primos Fortran compiler directly), the "x" qualifier prevents you from accessing external commands from within the shell (i.e., via "x ftn prog"), and the "v" qualifier prevents you from using 'declare' to modify or create a search rule (the shell file 'fc', which is the Subsystem interface to the Primos Fortran compiler, declares its own search rule) which contains an unqualified "^int" flag. It should be noted, however, that this is not a fool-proof method of limiting a user's access to com-

mands; a better solution is to write a program which is run at login and which "supervises" the user's session. One way of overcoming such a restriction placed by the system administrator would be to execute a command within a function call, such as the following:

```
[declare _search_rule = "<normal search rule>"; _
        <unrestricted command>]
```

By redefining the search rule, the user is then allowed to execute any desired command, including a new invocation of the command interpreter.

<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,

```
2>>stuff
```

causes the data written to output port 2 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>

| <value> ::= { <printable char> | <unprintable char> }

```
<unprintable char> ::= '<' <ascii mnemonic> '>'

<set command> ::= set [ <variable> ] = [ <value> ]

<declare command> ::= declare { <variable [ = <value> ] }

<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.

A variable may contain any legal ASCII character. To allow the user to enter unprintable characters that might be a problem to Primos or the shell, the commands that manipulate variables allow the use of ASCII mnemonics in the value of a shell variable. The following would set the "_kill_resp" variables to

two ASCII escape characters, a backspace, and the string "*del*":

```
set _kill_resp = "<esc><esc><bs>*del*"
```

To prevent the interpretation of the mnemonics (i.e. to enter a literal "<esc><esc><bs>*del*", in this case) the user simply uses the Subsystem escape character in front of the mnemonics:

```
set _kill_resp = "@<esc>@<esc>@<bs>*del*"
```

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.

History Mechanism

<history_command> ::= <cmd_select> <arg_select> <substitution>

The shell provides a sort of dynamic macro replacement facility for commands that are entered from the terminal. This is called a command history mechanism. It allows the user to recall commands he has previously entered, extract portions of the command, edit the portions he has selected, and either execute what remains or incorporate it into another command, with a minimum of typing.

A history substitution contains three parts; command selection, argument selection, and editing. Command selection chooses what command will be used. Argument selection decides which arguments are to be extracted from the chosen command line, and the editing phase allows the result to be edited to change spelling or substitute a different word for portions of the line. To prevent any history substitution from taking place, the 'hist' command can turn off the history mechanism. It also controls the saving and restoration of the current history environment. For the rest of this discussion, the assumption will be that history is currently enabled.

History substitution is triggered by the '!' character. A history substitution is normally stopped by a blank or tab character, but a trailing '!' will stop the interpretation of any further characters. This is used when concatenating supplementary text to the result of a history substitution. To prevent this and any other interpretation of the special history characters, they may be escaped with the Subsystem escape character, '@'. When a history substitution is discovered, the mechanism modifies the command line, prints the resulting command line on the user's terminal, and then passes the command to the

rest of the shell for execution. History processing occurs before any other evaluation in the shell, such as function calls and iteration. However, the use of '_' to continue an input line is done even before the history mechanism sees what you have typed; if the '_' is the last character in your history command, and the last character on the line, follow it with a terminating '!'.

Command Selection.

<cmd_select> ::= '!' [<str> | '?' <str> '?' | <num>]

The first thing in a history substitution is command selection. This is used to retrieve a given command line for use, or further processing. In a history command selection '!<str>' will find the most recent command line that started with the characters in <str>. '!?<str>?' will find the most recent command line that contained <str> anywhere on the line. It also allows <str> to contain blanks or tabs whereas the first form does not. '!<num>' allows the user to specify the number of a command according to the output of the 'hist' command. As a convenience, '!' by itself will repeat the last command entered.

Argument Selection.

<arg_select> ::= '`' [<num>] ['-' <num>]

The next portion of a history substitution is an optional argument selection. This chooses which portions of the command are to be kept. History arguments are not exactly the same as the arguments the rest of the shell uses, since history expansion occurs before argument collection. Arguments in this context are blank or tab seperated words on the command line. Function calls, iterations, and quotations will be extracted as a single argument, even if they contain blanks or tabs. Arguments are numbered from zero, starting at the leftmost portion of the line. In an argument selection, '`<num>' specifies that only argument <num> is to be extracted and kept for further processing or use, and the rest of the command line is to be dropped. '`<num>-<num>' specifies that arguments from the first <num> to the last <num> are to be kept. In place of any <num>, '$' may be specified to obtain the last argument on the line. The form '`-<num>' is a shorthand for '`1-<num>' and '`<num>-' is a short form for '`<num>-$'.

Substitution.

<substitution> ::= { '^' <str> '^' <str> '^' ['g'] }

The last portion of a history substitution is also optional and is the editing phase. This allows the portions of the command line that remain to actually be modified like the substitution command in 'ed', although much more limited. In the history mechanism, <str> is not a regular expression, as in 'ed', but is taken as a simple string. The regular expression special characters are not recognized in the history mechanism. Each substitu-

tion happens only once on the line unless a 'g' is appended on the substitution, in which case the change occurs globally on the line. Substitutions may be strung together, so that more than one may be performed at a time.

Finally, after all history substitutions have been made, the Shell will echo the new command line to the terminal, and then execute it. See the Application Notes for a discussion of the 'hist' command.

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":

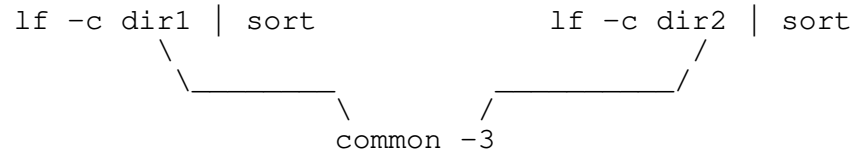
```
declare _quote_opt = YES
lf -c [args 2] | find [arg 1]
```

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 s1 _
       | 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 "s1". 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 "s1" 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 'elif', '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
#       (this is not the current version of 'e' in =bin=)

   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
```

The 'repeat' command is used to set up loops but, unlike the 'goto' command, will also work from the terminal. The following loop will do exactly what the previous command file did, but will also work when entered from a terminal:

```
# not quite as bogus a loop to show computer counting

   declare i = 1

   repeat
      i
      set i = [eval i + 1]
   until [eval i '>' 10]
```

History Examples

Command history provides a quick way of re-executing a command without retyping the entire command line. The following example shows how a user can run the previous command again by only typing a '!':

```
] time
11:59:04
] !
time
11:59:08
```

Another advantage is the ability to fix a mistyped command. For example, to list the contents of the directory "stuff.u" where the ".u" was omitted in the 'lf' command and then correc-

ted.

```
] lf stuff
stuff: not found
] !!.u
lf stuff.u
bogus        gorf         snert
```

Two '!'s are used because text must be entered right next to the history substitution. Any other time, the trailing '!' is not needed.

The 'hist' command, without any arguments, will print a list of the current history and their command numbers.

```
] hist
  1: pmac gorf.s; ld gorf.b -o snert
  2: se gorf.s
  3: pmac gorf.s; ld gorf.b -o gorf
  4: gorf
  5: se gorf.s
```

At this point it is time to execute the 'pmac' and 'ld' statements, again. There are several ways to do this. One is to give the specific command number (as printed by 'hist'):

```
] !3
pmac gorf.s; ld gorf.b -o gorf
```

or let the history do more of the work for us by telling it to look for the command starting with 'pmac':

```
] !pmac
pmac gorf.s; ld gorf.b -o gorf
```

or if that is not the correct command, entering a unique string that appears anywhere on the command line:

```
] !?-o sn
pmac gorf.s; ld gorf.b -o snert
```

Notice that the trailing '?' wasn't needed. This is because it would have occured at the end of the line. None of the delimiting characters need to be entered at the end of the line because the command substitution will place them there for you at the end of a line. Also notice that the shell will *always* echo the command produced by the history mechanism to the terminal, so that you can know for sure exactly what the shell is doing.

Argument selection allows the user to retrieve certain arguments from the selected command line. After a command line is selected (as in the previous examples) then argument selection takes place. For example, given the command line

```
] echo 1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
```

to retrieve only arguments 3 to 7 one can type:

```
] echo 1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
] echo !`3-7
echo 3 4 5 6 7
3 4 5 6 7
```

or to grab the first item on the line,

```
] echo 1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
] echo !`0
echo echo
echo
```

because argument zero (the command name) is the first item on the line.

The history mechanism does not know about command <nodes>. E.g., a '|', and the command name after it, are treated as just plain arguments. Numbering starts at zero, and each successive blank separated "item" is considered another argument. In the case of a function call, iteration, or quoted string, blanks and tabs are insignificant until all the brackets, parentheses, and quotes match up. In this manner, an entire function call, iteration group, or string counts as a single argument, whether or not it contains spaces.

```
] echo (gorf.s snert.r)
gorf.s snert.r
] cat -h !`1
cat -h (gorf.s snert.r)
==================== gorf.s ====================
 SEG
 DYNT BURF$
 END
==================== snert.r ====================
   call print(STDOUT, "burf*n"s)
   stop
   end
```

or for a more complicated example

```
] echo [echo berf] (blert blort) "final word"
berf blert final word
berf blort final word
] echo !`3 !`1 !`2
echo "final word" [echo berf] (blert blort)
final word berf blert
final word berf blort
```

The last portion of a history replacement is substitution. This allows previously selected portions of the command line to be placed through a set of substitutions similar to the 'change'

command or the substitute command in the editor. To change the "blert" in the previous example to "bonzo", you would type

```
] echo [echo berf] (blert blort) "final word"
berf blert final word
berf blort final word
] !^blert^bonzo^
echo [echo berf] (bonzo blort) "final word"
berf bonzo final word
berf blort final word
```

The operations can be combined. For instance to move arguments around, and make substitutions

```
] echo one two three
one two three
] echo !`3 !`1^one^1^ !`2
echo three 1 two
three 1 two
```

There can be more than one substitution per command line, and the given changes can be made globally.

```
] echo aa bb cc dd ee
aa bb cc dd ee
] !^a^z
echo za bb cc dd ee
za bb cc dd ee
] !?aa?^b^y^g
echo aa yy cc dd ee
aa yy cc dd ee
] !?a bb?^a^z^g^b^y^g^ee^ve^^d^w
echo zz yy cc wd ve
zz yy cc wd ve
```

The first substitution simply changes the first "a" to a "z". The second one recalls the most recent command with an "aa" in it and changes the first "b" to a "y". The last one looks for the most recent command that contains an "a bb" string (the first line) and then substitutes a "z" for all occurences of an "a", a "y" for all occurences of a "b", a "ve" for the first "ee", and a "w" for the first "d". Notice that for the last substitution, the trailing '^' was not necessary.

History processing takes place across the entire input line, even inside quoted strings. To get one of the literal history characters (!^`), you *must* escape it with the Subsystem escape character, '@'.

Finally, the 'hist' command is available to control the use of the history mechanism. 'Hist on' turns on history processing. By default, it is off. 'Hist off' turns history processing off. 'Hist save <file>' will save the current list of remembered commands into <file>, or into =histfile= if <file> is not specified. 'Hist restore <file>' will retrieve a saved history session from <file>, or from =histfile= if <file> is not specified. It is

recommended that you put a 'hist restore' into your '_hello' variable or the file it executes (if you want to save your shell sessions across logins). If history processing is not turned on when you do a 'hist restore', the shell will automatically turn it on for you, and then restore your saved command history. If history is turned on, whenever you issue a 'stop' command (like =bin=/bye does), the shell will automatically do a 'hist save' for you. This will also happen if you type an EOF at the shell (usually control-c), unless you also have "_nottyeof" set (see below).

Shell Control Variables

Many special shell variables are used to control the operation of the command interpreter. You can define or change any shell variable with 'set' and can delete it with 'forget'. The current value of a shell variable can be examined by entering its name. The values of all your shell variables can be examined with the 'vars' command. Certain shell variables are read into the SWT common block at Subsystem initialization to control the terminal input routines. If these variables are changed, the shell will modify the Subsystem common to reflect the change immediately. The variables that could accept control characters as values may be entered using the ASCII mnemonics supported by the shell variable commands (see the heading "variables" in the reference section of this manual). 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. This variable is only checked on entrance to the Subsystem, so if this is changed, the user should exit the Subsystem (say with 'stop') and then reenter (using the 'swt' command). The default value is "=bin=/sh".
_eof	This variable may be set to a single character which will be used to signal the end of file from a terminal. The Subsystem input routines will recognize an instance of this character anywhere on the input line and send the appropriate signal to the input routine. The default value is the ASCII character ETX (control-c).
_erase	This variable may be set to a single character to be used as the "erase," or character delete, control character for Subsystem terminal input processing.

_escape	This variable may be set to a single character to be used as the "escape" control character for Subsystem terminal input processing. Note that this will *not* not change the standard Subsystem escape character, it remains an '@'. (See the help on 'tcook$' for the gory details.)
_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 to be used as the "kill," or line delete, control character for Subsystem terminal input processing.
_kill_resp	This variable may be set to any string which will appear on the user's terminal when the kill character is entered. If this variable is not present "\\" is the kill response.
_mail_check	This variable determines how often mail is checked during the login session. If not declared, the user is not notified of incoming mail while he is logged in. If the variable is set to an integer value, the shell will check for changes in his mailbox status after that many seconds has elapsed, just before his prompt string is printed. The user is notified by the message, "You have new mail". If the variable is declared but not set, or set to an illegal value, the default is to check every 60 seconds.
_newline	This variable may be set to a single character which will be interpreted as the end-of-line. Whenever this character is encountered, a carriage return and linefeed will be echoed to the terminal. If it is not set, then the ASCII character LF is the default.
_nottyeof	An EOF character typed at command level 1 will normally terminate the Subsystem and place the user face to face with the Primos operating system. Most commands accept input from the terminal if an alternate file is not specified and if the user's keyboard happens to bounce, the user is bounced into Primos. If this variable is declared, an EOF typed at command level 1 will not terminate the shell but will type the message "use 'stop' to exit the subsystem" and return to command level.

_pause_gossip	This variable controls the paging of gossip messages. If this variable is set, the gossip will pause at the last page, otherwise it simply returns to command level without allowing any paging commands.
_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 (]).
_prt_dest	This variable contains the location where all files spooled by this user are to be printed. If this variable is not present, files will be printed at the system-defined default printer.
_prt_form	This variable contains the form to be used for files spooled by this user (e.g. "narrow"). If this variable is not present, files will be printed on the system-defined default form.
_quit_action	If this variable is present, whenever the fault handler detects a break, it will prompt you as to whether you want to continue, terminate the program or call Primos. Otherwise, a break will return you to the Subsystem.
_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 to be used as the "retype" control character for Subsystem terminal input processing.
_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.

_vth_gossip This causes any gossip that is received to be paged using the screen oriented paging mechanism.

Shell Command Statistics

If the public or private template "=statistics=" is defined with the value "yes", the shell will record every command issued by the user in the directory defined by the system template "=statsdir=". If you set your private template "=statistics=" to "yes" then your commands will be recorded in the directory defined by your "=statsdir=" template. The files in the directory "=statsdir=" are named "sh<pid>"; command statistics for a given process are stored in the file with the corresponding process id. Here is an example of the file:

```
122680 171812 16 system  1 F //bin/x
122680 171816 16 system  1 F //bin/lf
122680 171822 16 system  1 F //bin/template
(date) (time)    (user)  | | (command)
                 (pid) (level) (F - command found)
```

The date begins in the first column. The (level) is the depth of nesting of shell files at which the command is requested; 1 is the terminal level.

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 occurrence 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. 'Vars' can also save and restore the user's variables from arbitrary files. 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.

Program Interface

The shell provides a set of routines which allows the user of the standard shared libraries to create shell variables, retrieve their values, and change them as well. You may also execute shell commands from within a program. This facility is not available when using the non-shared libraries, and even using the shared libraries it is somewhat restrictive until Prime supports EPF runfiles. Further information on these routines can be found in the Software Tools Subsystem Reference Manual.

shell
: 'Shell' is the subroutine which starts another level of the SWT shell. It is used to execute commands read from an open input file. It is analagous to the 'sh' command.

subsys
: 'Subsys' is used to execute a single command from within a program. It combines all the operations needed to execute a string with 'shell' without the user having to perform the operations. It is a convenience for the user.

svdel
: 'Svdel' accepts the name of a shell variable and

deletes it at the current shell level. It takes care of updating the SWT common block in the case of a special shell variable (see "Shell Control Variables", above). It is analagous to the command 'forget'.

svdump
'Svdump' prints a representation of the internal shell variable common block. It scans all levels of the variables dumping the chains and the hash tables. It is analagous to the 'dump sv' command.

svget
'Svget' simply retrieves the value of a given shell variable. Since "executing" a variable from the command level prints the value of the variable, the action of 'svget' is closest to the execution of a variable.

svlevl
'Svlevl' returns the current lexic level of the shell. This is useful in cooporation with 'svscan' (described below) to retrieve the value of all currently declared variables. This routine has no command equivalent.

svmake
'Svmake' creates a given shell variable at the current lexic level of the shell. It returns the lexic level of the shell. If the variable already exists at the current level, then 'svmake' will have no effect. Any special variables (see "Shell Control Variables", above) that are changed will cause a change in the SWT common block to reflect the value of the variable. 'Svmake' is analagous to the 'declare' command.

svput
'Svput' sets the value of a given shell variable in the most recent lexic level that it appears. If the variable does not exist in any scope of the shell, it is created in the current level. 'Svput' also makes modifications to the SWT common block if any special variables are changed. 'Svput' is analagous to the 'set' command.

svrest
'Svrest' reads a file written by 'svsave' (see below) and attempts to merge those variables with those at the current lexic level. 'Svrest' is analagous to the 'vars -r' command.

svsave
'Svsave' attempts to save the shell variables at lexic level number 1 (the top level) in the given file. 'Svsave' is analagous to the 'vars -s' com-

mand.

svscan
'Svscan' provides a way for the user to obtain the value of all shell variables at any or all lexic levels. It operates in a method similar to 'tscan$'. There is no command associated with 'svscan'.

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.

TABLE OF CONTENTS

User's Guide for the Ratfor Preprocessor

Second Edition

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

July, 1984

Foreword

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 described 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, 0
"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 possibility of mistyping or forgetting 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)
```

Fortran Statements in Ratfor Programs

Ratfor provides the escape statement to allow Fortran statements to be passed directly to the output without the usual processing, such as case mapping and automatic continuation. The escape statement has three forms, summarized below. In the first form listed below, the first non-blank character of the Fortran statement is output in column seven. In the second form, the first non-blank character of the Fortran statement is output in column seven, but column six contains a "$" to continue a previous Fortran statement to that stream. In the third form, the Fortran statement is output starting in column one, so that the user has full control of the placement of items on the line. The following is a summary of this description:

Escape Statement Format	*Output Column*
%<stream><Fortran statement>	7
%<stream>&<Fortran statement>	6
%<stream>%<Fortran statement>	1

"Stream" can take on the following values:

1	declaration
2	data
3	code

If no stream is specified (i.e. %%<Fortran statement>), the Fortran statement is sent to the code stream.

Escaped statements *must* occur inside a program unit, i.e., between a **function** or **subroutine** statement, and its corresponding **end** statement. Otherwise 'rp' gets confused about where the escaped statements should go, since it won't have any streams open. If you have a large amount of self contained FORTRAN that you want 'rp' to include in its output, you can accomplish this in two steps. First, put '%1%' at the beginning of each line, and then put the FORTRAN at the *beginning* of your ratfor source file.

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 parameters 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 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. While this isn't the greatest syntax

in the world, it is easy to implement local variables in this fashion.

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 specification 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 occurring **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 expression 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, but before the terminating **end** statement (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 possibly 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-circuited 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>)

Escape Statements

Escape statements can be used to output Fortran statements that will not be touched by the Ratfor preprocessor. The escape statement has three possible forms. In the first form listed below, the first non-blank character of the Fortran statement is output in column seven. In the second form, the first non-blank character of the Fortran statement is output in column seven, but column six contains a "$" to continue a previous Fortran

statement to that stream. In the third form, the Fortran statement is output starting in column one, so that the user has full control of the placement of items on the line. The following is a summary of this description:

Escape Statement Format	*Output Column*
%<stream><Fortran statement>	7
%<stream>&<Fortran statement>	6
%<stream>%<Fortran statement>	1

"Stream" can take on the following values:

1	declaration
2	data
3	code

If no stream value is given, it is assumed to be the code stream. Escaped statements have to come between a **function** or **subroutine** statement and the corresponding **end** statement.

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 <identifier>, and the subsequent occurrence 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 occurrences 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 occurrences 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> associated with **ifany** is executed; if none are found true, the <statement> associated 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 the 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 translate 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> [<rp options>]
```

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".

Next 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.

Finally, there are preprocessor options which may be specified to change the output in some way or affect preprocessor operation. For a complete list of available options and a more detailed description of the command line syntax, see Appendix F.

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:

-v 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.)

-t Insert code to produce a statement-level trace during execution.

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 p1.f -l


# Compile "p1.f" to produce the binary "bin" and
#    the listing on "list"

  fc p1.f -b bin -l list


# Compile "p3.f", produce a pseudo-assembly code
#    listing and default to 32-bit integers

  fc -v -i p3.f -l
```

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 "vshlib",
#    and produce the output file "nprog"

  ld nprog.b -l vshlib -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.
Abort (a), Continue (c) or Call Primos (p)? p
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:
...
```

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 - 0033	Operating System
2030	Software Tools Shell
2031	Software Tools Screen Editor
2035	Software Tools Library
2050	Fortran Library
4000 - 4037	User Program
4040	Software Tools Common
4041	Software Tools Stack
6001	Fortran Library
6002	Primos Ring 3 Stack

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 4041.

   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 4041) 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 as 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 course, 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.

Portability

If your intent is to produce portable Fortran code, the Ratfor preprocessor, 'rp' can be invoked with the following four options:

-h Produce Hollerith-format string constants rather than quoted string constants. This option useful in producing character strings in the proper format needed by your Fortran compiler.

-v Output "standard" Fortran. This option causes 'rp' to generate only standard Fortran constructs (as far as we know). This option does not detect non-standard Fortran usage in Ratfor source code; it only prevents 'rp' from generating non-standard constructs in implementing its data and control structures.

-x Translate character codes. 'Rp' uses the character correspondences in a translation file to convert characters into integers when it builds Fortran "data" statements containing EOS-terminated or PL/I strings. If the option is not specified, 'rp' converts the characters using the native Prime character set.

-y Do not output "call swt". This option keeps 'rp' from generating a "call swt" in place of all "stop" statements, which are required for Fortran programs to run under the Subsystem.

The following option for 'fc' may also help:

-i Consider all integers to be "long" (32-bit) rather than short.

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 sub-program, 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 cant open file message
   subroutine cant (str)
   character str (ARB)

   call putlin (str, ERROUT)
   call error (": can't 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 indentation 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"s)
|   ---}
|   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.

Termination

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 array, 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"s)
```

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) == '1'c) {
   # handle '1' 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 str1 (MAXLINE), str2 (MAXLINE)
...
call scopy (str1, 1, str2, 1)
```

'Scopy' is also capable of appending one string to another, as in the following example:

```
character str1 (MAXLINE), str2 (MAXLINE)
...
call scopy (str1, 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 independence 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. Use "*1s" to read in a single character. (Admittedly, this is an inconsistency; there really should be a "*c" format.)

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 immediately 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"s, 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"s)
```

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 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
   }
```

Parscl. In many programs, argument syntax is quite complex. 'Parscl' exists for the benefit of both programmers and users: it makes coding argument parsing simple and it helps keep argument conventions uniform. Of course, to do this, it must automatically enforce certain argument conventions. 'Parscl' and its accompanying macros expect to recognize arguments of a single letter without regard to case. Rather than a lengthy explanation, let's look at an example: For its arguments, a program requires a page length (which should default to 66 if not present), a title (which may also not be present), a flag to tell whether to format for for a printer or a terminal, and a list of file names to process. In this case, a reasonable option syntax is

```
prog [-l <page length>] [-t [<title>]] [-p] {<file name>}
```

We have used single letter flags to avoid the need for always specifying arguments. Now, in terms of 'parscl', what we have is an "required integer", an "optional string", and a "flag". This means that "-l" cannot be specified without a <page length>, but "-t" can be specified without a <title> (in this case, of course, we would use an empty title). Be sure to note that a "required" argument means that if the letter is specified, it must be followed by a value. It does **not** mean that the letter argument must always be present. In other circumstances, we can also have "optional integer" and "required string" arguments.

To use 'parscl' in our program, we must first include the argument macros and declare the argument data area:

```
include ARGUMENT_DEFS
ARG_DECL
```

Then, near the beginning of the main program, we use a macro call to call 'parscl' that contains the syntax of the command line and a "usage" message to be displayed if the command line is incorrect. For our example, we can use

```
PARSE_COMMAND_LINE ("l<req int> t<opt str> p<flag>"s,
   "prog [-l <page len>] [-t [<title]] [-p] {<file}>"s)
```

For "optional integer" and "required string" arguments, the argument types are "<opt int>" and "<req str>", respectively.

If the command line is parsed successfully, 'parscl' returns and the program continues; otherwise, 'parscl' prints the "usage" message with a call to 'error'. Once 'parscl' has returned, we can set the default values, test for the presence or absence of arguments, and obtain values of arguments. First we usually set default values:

```
ARG_DEFAULT_INT (l, 66)
if (ARG_PRESENT (t))
   ARG_DEFAULT_STR (t, ""s)
else
   ARG_DEFAULT_STR (t, "Listing from prog"s)
```

Remember, default values are set **after** the call to 'parscl'!

In the preceding example, we set the value of the argument for "l" to 66. This is simple enough. But for the "t" argument, we really have three different cases: the argument was specified with a string, the argument was specified without a string (meaning that we must use an empty title), or the argument was not specified at all (meaning that we use some other default). In the first case, neither call to ARG_DEFAULT_STR will do anything, since the string was specified by the user; in the second case, ARG_PRESENT (t) will be ".true." setting the default to the empty string (since the "t" argument was specified, even though it was without a string); and in the third case ARG_PRESENT (t) will be ".false.", setting the default to "Listing from prog".

Now that we have finished setting defaults, we can obtain the values of arguments with more macros: the call ARG_VALUE (l) will return the page length value: either the value specified by the user or the value 66 that we set as the default. ARG_TEXT (t) references an EOS-terminated string containing the title: either the value specified the user, an empty string, or "Listing from prog". Use of the values in our example might look like this:

```
page_len = ARG_VALUE (l)
call ctoc (ARG_TEXT (t), title, MAXTITLE)
if (ARG_PRESENT (p))
   ### do printer formatting
else
   ### do terminal formatting
```

And now, here's how all of the argument parsing will look:

```
include ARGUMENT_DEFS
ARG_DECL

PARSE_COMMAND_LINE ("l<req int> t<opt str> p<flag>"s,
   "prog [-l <page len>] [-t [<title]] [-p] {<file}>"s)

ARG_DEFAULT_INT (l, 66)
if (ARG_PRESENT (t))
   ARG_DEFAULT_STR (t, ""s)
else
   ARG_DEFAULT_STR (t, "Listing from prog"s)

page_len = ARG_VALUE (l)
call ctoc (ARG_TEXT (t), title, MAXTITLE)
if (ARG_PRESENT (p))
   ### do printer formatting
else
   ### do terminal formatting
```

Now, what about the file name arguments we were supposed to parse. Where did they go? 'Parscl' deletes arguments that it processes; it also ignores any arguments not starting with a

hyphen (that do not appear after an letter-argument looking for a string). So the file name arguments are still there, ready to be fetched by 'getarg', with none of the "-t <title>" stuff left to confuse the logic of the rest of the program.

Now, how about some example commands to call this program:

```
prog -p
      (page_len = 66, title = "Listing from prog",
         formatted for printer)

prog -l34 -t new title
      (page_len = 34, title = "new",
         file name = "title",
         formatted for terminal)

prog file1 file2 -p -t -l70
      (page_len = 70, title = "",
         file names = file1 file2,
         formatted for printer)

prog filea -t"my new title" -l 60
      (page_len = 60, title = "my new title",
         file name = filea, formatted for printer)

prog -x filea
      (the "usage" message is printed)

prog fileb -l
      (the "usage" message is printed)
```

As you can see, 'parscl' allows you to specify arguments in many different ways. For more information on 'parscl', see its entry in the Reference Manual.

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 sub-programs 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 identifier 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 associated 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 "firstentry" 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 circumstances. 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 stdump (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

_^H**A**_^H**p**_^H**p**_^H**e**_^H**n**_^H**d**_^H**i**_^H**x**_^H**e**_^H**s**

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:

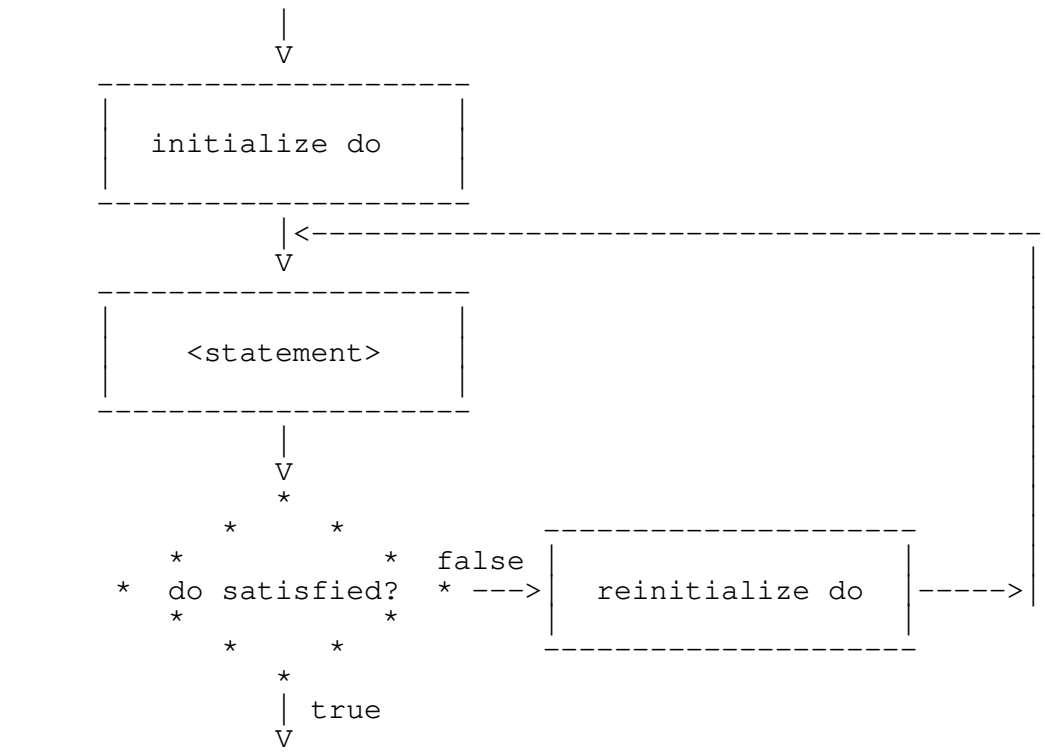
```
for (i = length (str); i > 0; i = i - 1)
   if (str (i) ~= ' 'c)
      break


      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:

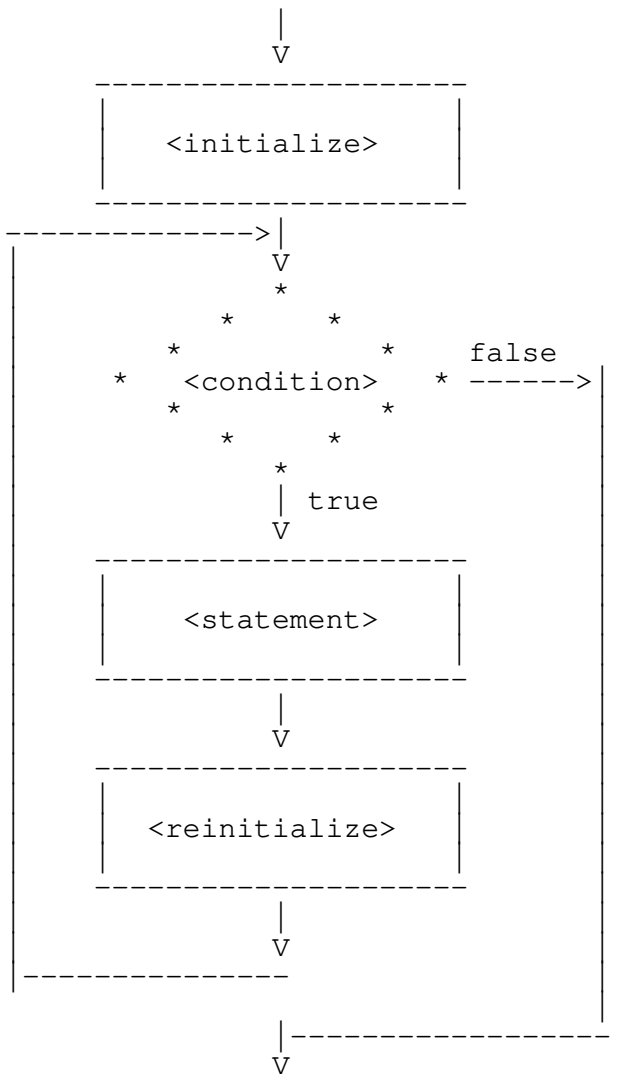
```
do i = 1, 10
   array (i) = 0


      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)
   }
```

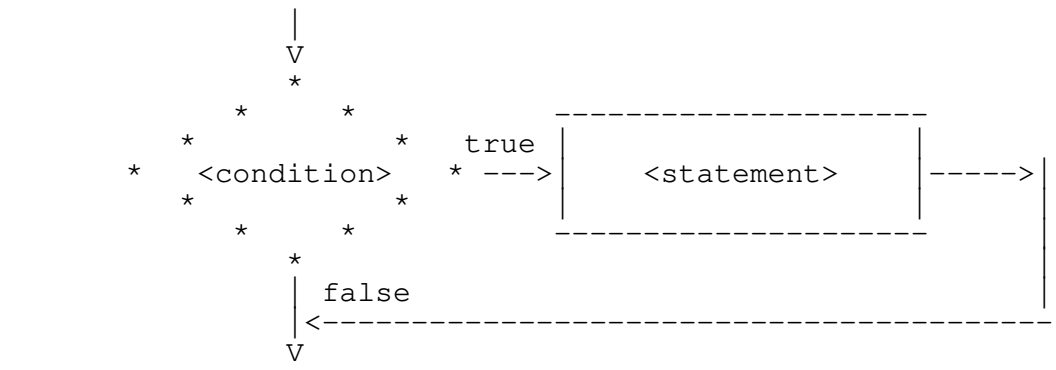
```
      i=limit-1
      goto 10002
10000 i=i-1
10002 if((i.le.0))goto 10001
        array1(i)=array1(i+1)
        array2(i)=array2(i+1)
      goto 10000
10001 continue
```

If

Syntax:

```
if (<condition>)
   <statement>
```

Function:



Example:

```
if (a == b) {
   c = 1
   d = 1
   }
```

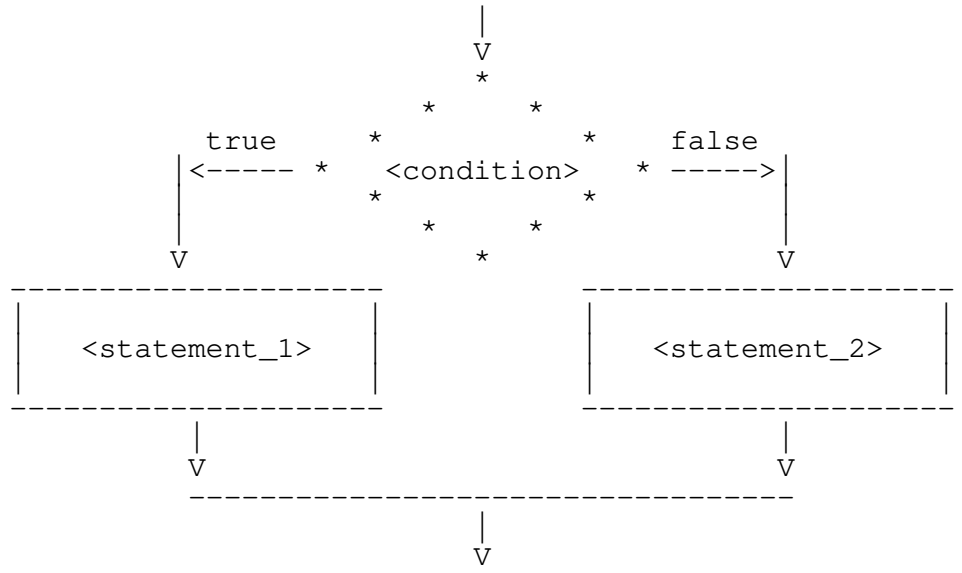
```
      if((a.ne.b))goto 10000
        c=1
        d=1
10000 continue
```

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)
   }
```

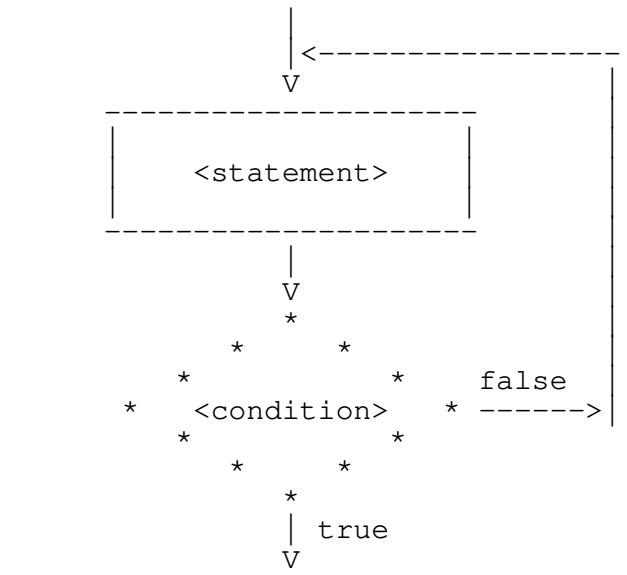
```
      i=1
      goto 10002
10000 i=i+1
10002 if((i.gt.50))goto 10001
        j=1
        goto 10005
10003   j=j+1
10005   if((str(j,i).eq.-2))goto 10004
          if((str(j,i).ne.160))goto 10006
            goto 10000
10006   goto 10003
10004   call putlin(str(1,i),-11)
      goto 10000
10001 continue
```

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)
```

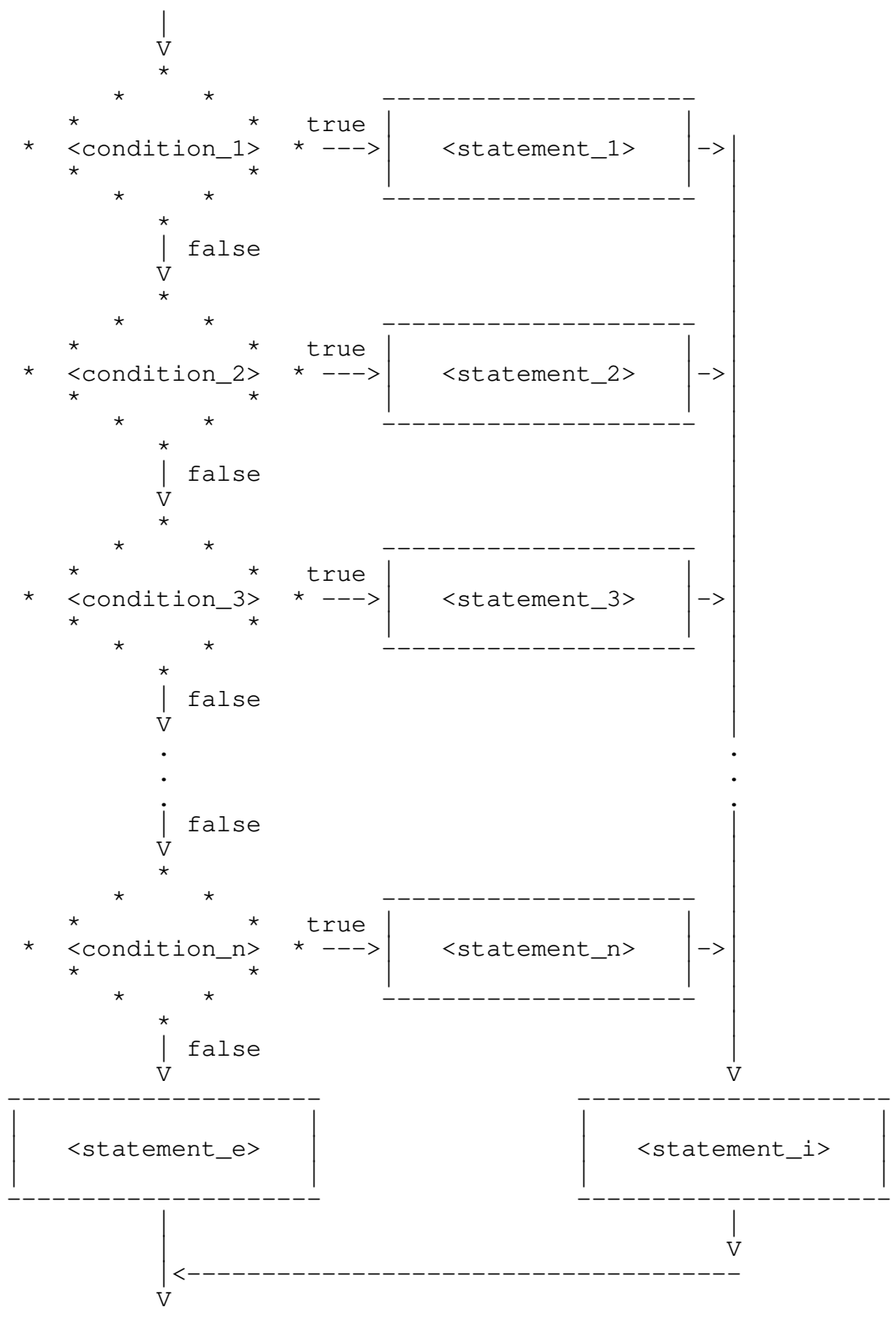
```
integer function fcn (x)
...
fcn=a+12
return
```

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:


<condition_1>
true
<statement_1>
false
<condition_2>
true
<statement_2>
false
<condition_3>
true
<statement_3>
false
<condition_n>
true
<statement_n>
false
<statement_e>
<statement_i>

Example:

```
select
   when (i == 1)
      call add_record
   when (i == 2)
      call delete_record
else
   call code_error
```

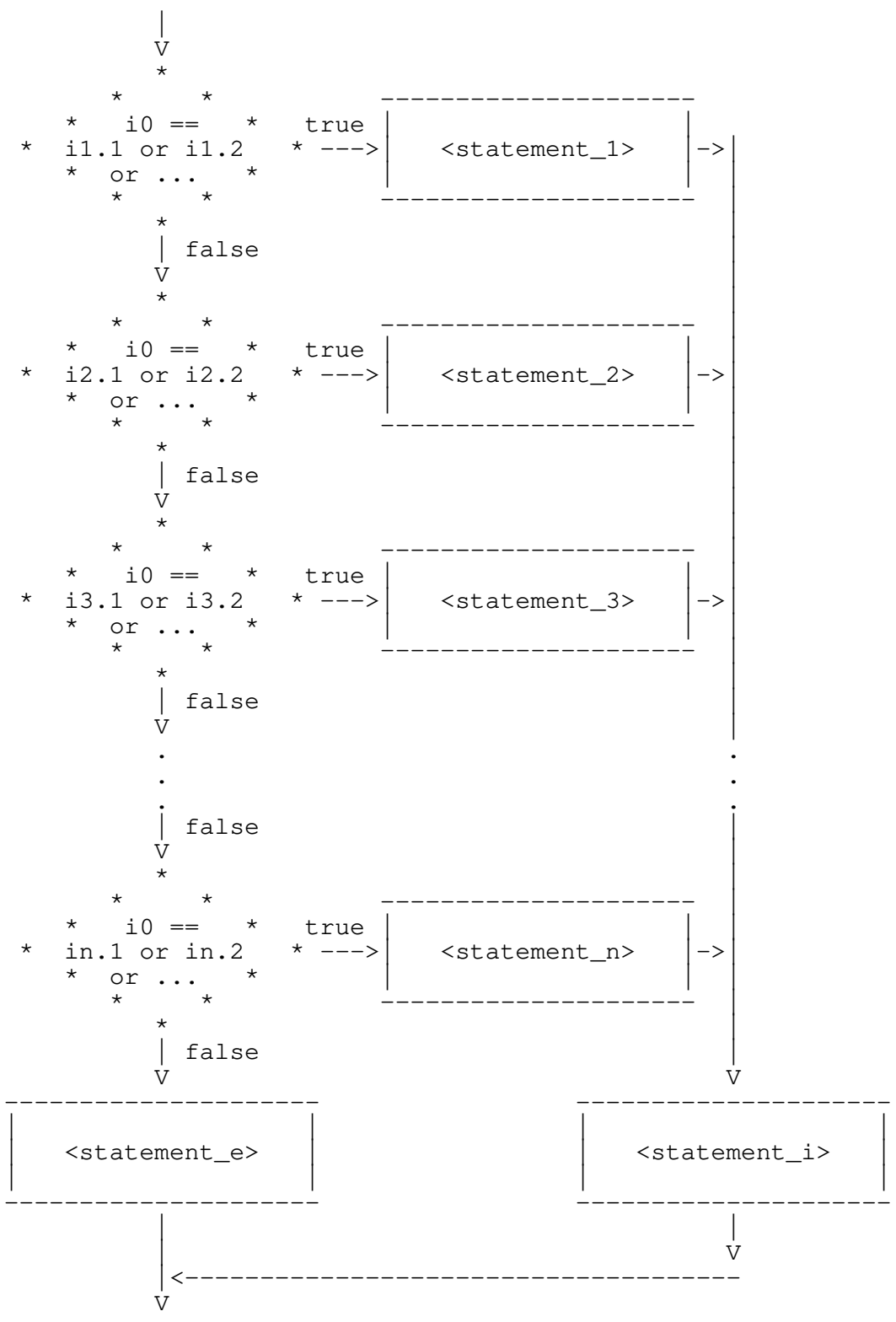
```
      goto 10001
10002   call addre0
      goto 10000
10003   call delet0
      goto 10000
10001 if((i.eq.1))goto 10002
      if((i.eq.2))goto 10003
        call codee0
10000 continue
```

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:

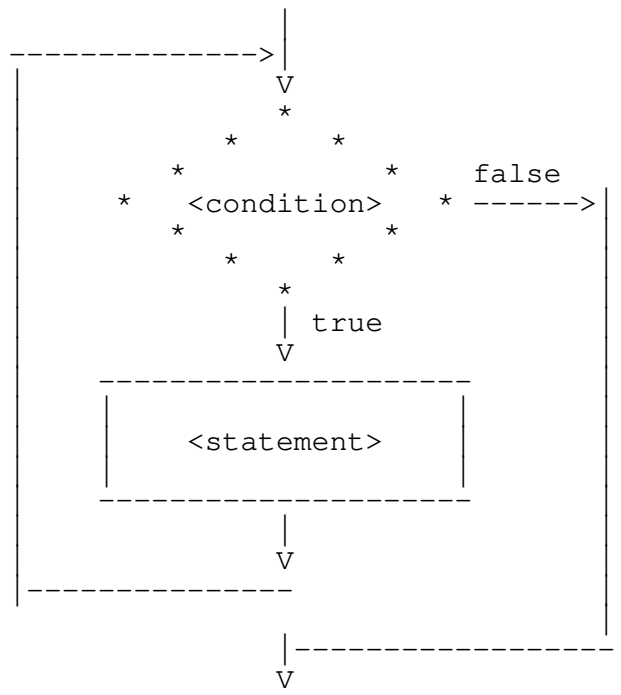
```
          select (i)
             when (4, 6, 3003)
                call add_record
             when (2, 12, 5000)
                call delete_record
          else
             call code_error


                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 must be linked with one of two slightly different procedures: If the object file can be a segment directory (this is usually not a problem), you can have the object file placed in a segment directory. Just add the "-d" option to the 'ld' command line. Assuming your binary file is named "prog.b", you can use the command

```
ld -d prog.b
```

If you would rather the object program be stored in a regular file, you can use a slightly different procedure. With this procedure, the program is 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 must be placed in a segment directory.

This 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 Subsystem.

64V addressing mode

- Segments '4040 and '4041 may not be disturbed.
- When a Subsystem program is executed, the stack is already constructed in segment '4041. However, the executing program may rebuild it if desired.
- Programs that use native i/o routines must inform their native i/o routines of the Subsystem (if they wish to take advantage of Subsystem i/o) by calling the proper initialization routines, i.e. 'init$f' for Fortran 66 and Fortran 77, 'init$p' for Pascal and 'init$plg' for PL/I G.
- 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 or a SEG-compatible segment directory.
- If the program is in a P300 format runfile, it 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, unless it is loaded from a SEG-compatible segment directory.
- 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. Programs loaded from SEG-compatible segment directories may be as large as the operating system permits, as long as they do not modify segments '4040 and '4041.

32I addressing mode

- Programs in 32I mode may be executed under the Subsystem subject to the same constraints as 64V mode programs.

Appendix D -- The Subsystem Definitions

The file "=incl=/swt_def.r.i" contains Ratfor **define** statements for all the symbolic constants 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 horizontal 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>.

Language 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 expression 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 constants 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).
PG_END	Make 'page' return after the last page of input.
PG_VTH	Make 'page' use the VTH routines when writing to the terminal.
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
```

Appendix F -- Command Line Syntax

'Rp' provides a rich set of processing options to allow the user much flexibility and control over the code which is produced. The command line syntax is as follows:

```
rp [-{a | b | c | d | f | g | h | l | m | p | s | t | v | y}]
   [-o <output_file>] {<input_file>} [-x <translation file>]
```

The following is a full description of each option:

a Abort all active shell programs if any errors were encountered during preprocessing. This option is useful in shell programs like 'rfl' that wish to inhibit compilation and loading if preprocessing failed. By default, this option is not selected; that is, errors in preprocessing do not terminate active shell programs.

b Do not map long indentifiers or identifiers containing upper case letters into unique six character Fortran identifiers. This option is useful if your Fortran compiler will accept names longer than six characters.

c Include statement-count profiling code in the generated Fortran. When this option is selected, calls to the library routines 'c$init', 'c$incr', and 'c$end' will be placed (unobtrusively) in the output code. When the preprocessed program is run, it will generate a file named "_st_count" containing execution frequencies for each line of source code. The utility program 'st_profile' may then be used to combine source code and statement counts to form a readable report.

d Inhibit generation of the long-name dictionary. Normally, a dictionary listing all long names used in the Ratfor program along with their equivalent short forms is placed at the end of the generated Fortran as a series of comment statements. This option prevents its generation.

f Suppress automatic inclusion of standard definitions file. Macro definitions for the manifest constants used throughout the Subsystem reside in the file "=incl=/swt_def.r.i". 'Rp' will process these definitions automatically, unless the "-f" option is specified.

g Make a second pass over the code and remove GOTOs to GOTOs generated in Ratfor control structures. Use of this option lengthens preprocessing time significantly, but can result (sometimes) in a 2-5% speedup of the object program.

h Produce Hollerith-format string constants rather than

quoted string constants. This option useful in producing character strings in the proper format needed by your Fortran compiler.

l Include Ratfor line numbers in the sequence number field of the Fortran output. This may be useful in tracking down the Ratfor statement that caused a Fortran syntax error. By default, no sequence field is generated.

m Map all identifiers to lower case. When this option is selected, 'rp' considers the upper case letters equivalent to the corresponding lower case letters, except inside quoted strings.

p Emit subroutine profiling code. When this option is selected, 'rp' places calls to the library routines 't$entr', 't$exit', and 't$clup' in the Fortran output, and creates a text file named "timer_dictionary" containing the names of all subprograms seen by the preprocessor. When the profiled program is run, a file named "_profile" is created that contains timing measurements for each subprogram. The utility program 'profile' may then be used to print a report summarizing the number of times each subprogram was called and the total time spent in each.

s Short-circuit all logical conditions. The order of evaluation of logical operands in Fortran is unspecified; that is, in the expression "a&b" there is no guarantee that "a" will be evaluated before "b". Occasionally this creates inconveniences; one would like to say something like "if(i>1&array(i)~=0)...". 'Rp' supplies the short-circuit logical operators "&&" and "||" (read "andif" and "orif") for these occasions. Both operators evaluate their left operands; if the value of the logical expression is predictable solely on the basis of the value of the left operand, then the right operand remains unevaluated and the correct expression value is yielded. Otherwise the right operand is evaluated and the proper expression value is determined. The "-s" option may be used to automatically convert all "logical and" operators in a program to "andifs," and all "logical or" operators to "orifs." In addition to improving program portability, this option may also reduce execution time. By default, however, this option is not in effect.

t Trace subprograms. When a program preprocessed with the "-t" option is run, an indented trace of the subprograms encountered will be printed on ERROUT. This trace output is generated by calls to the library routine 't$trac' that are inserted automatically by 'rp'.

v Output "standard" Fortran. This option causes 'rp' to

generate only standard Fortran constructs (as far as we know). This option does not detect non-standard Fortran usage in Ratfor source code; it only prevents 'rp' from generating non-standard constructs in implementing its data and control structures. Programs preprocessed with this option are slightly larger and slower; the intermediate Fortran and binary files are approximately 10% larger.

x Translate character codes. 'Rp' uses the character correspondences in the <translation file> to convert characters into integers when it builds Fortran DATA statements containing EOS-terminated or PL/I strings. If the option is not specified, 'rp' converts the characters using the native Prime character set. The format of the translation file is documented below.

y Do not output "call swt". This option keeps 'rp' from generating "call swt" in place of all "stop" statements.

The remainder of the command line is used to specify the names of the Ratfor input file(s) and the Fortran output file. If the "-o" option, followed by a filename, is selected, then the named file is used for Fortran output. Any remaining filenames are considered Ratfor source files. If no other file names are specified, standard input is read. If the "-o" option is not specified, then the output filename is constructed from the first input filename by changing a ".r" suffix (if present) to ".f". If the ".r" suffix is not present, the output filename is the input filename followed by the suffix ".f".

The format of the translation file used with the "-x" option is as follows. Each line contains descriptions of two characters: the Prime native character to be replaced, and the character value to replace it. These descriptions may be any one of the following: a single non-blank Prime ASCII character, a number in a format acceptable to 'gctoi' (must be more than one digit), or an ASCII mnemonic acceptable to 'mntoc'. In addition, the character to be replaced may also be the mnemonic "EOS" to indicate that the value of the end-of-string indicator is to be changed. For example, here is a portion of the table for converting the EBCDIC character set:

```
A 16rc1
B 16rc2
...
Z 16re9
0 16rf0
...
9 16rf9
SP 16r40
```

TABLE OF CONTENTS

Ratfor Language Guide

Ratfor Language Reference

Ratfor Programming Under the Subsystem

Appendixes

Software Tools Text Formatter
User's Guide

Terrell L. Countryman
Perry B. Flinn
Jeanette T. Myers
Arnold D. Robbins
Peter N. Wan

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

July, 1984

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.

It is possible skip an arbitrary number of pages in a controlled fashion. To do this, use the 'page-skip' command

.ps <max> <modulus>

<Max> is the maximum number of pages plus one that 'fmt' will skip. <Modulus> is the number which 'fmt' uses modulo the next output page number to count skipping pages. It works as follows: 'Fmt' sees the **.ps** command. It computes the page number of the current page plus one, and then takes the remainder of that number divided by the <modulus>, and saves it. 'Fmt' skips pages, adding one to this saved value. As long as this value is *less than* <max>, it continues to skip pages. For instance, if the current page is 15, and you issue a

.ps 3 5

command, 'fmt' would compute ((15 + 1) mod 5), yielding (16 mod 5), which is one (16 divided by 5 is 3, with 1 left over). It will then skip two pages, since it started with one, then skipped one, which is two. This is still less than three, so it skips one more page, yielding three, which is not less than three, so it stops. It is really quite simple. For instance, to skip to the next even page, use

.ps 2 2

and to skip to the next odd page, use

.ps 1 2

This feature is particularly useful for writing macros which aid with large documents. For example, it may be necessary that a chapter always start on an odd numbered page. So the 'begin chapter' macro would have a '**.ps** 1 2' as one of its lines. (See later for more details on how to write macros.)

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.
.ps N M	N=M=0	N=M=0	yes	Skip pages while (page number mod M) is less than N.
.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 causes 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.

'Eo' and 'oo' commands allow you to specify different page offsets for even- and odd-numbered pages respectively. Like 'po', they are initialized to zero and revert to that value when no parameter is specified. For instance,

.eo *+*N

will change the even-numbered page offset by N (or to N if no sign is specified).

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
.eo +N	N=0	N=0	yes	Set even page offset.
.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.
.oo +N	N=0	N=0	yes	Set odd page 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/Output 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.

Output File Control

The output of the formatter is always written on STDOUT unless you divert it with the divert output stream command, 'dv'. 'Dv' can be used to divert fmt's output to a named file:

.dv <file>

All output is written in <file> until a 'dv' command with no parameter is specified. 'Dv' can also be used to divert output to a temporary file that can be later read with the 'so' command. This is useful for generating tables of contents for documents (see the "Application Notes" section). The command

.dv N

diverts output to stream 'N' and can be read at any time and repeatedly by the command

.so N

Output will be diverted until the the 'dv' command is seen again without parameters. (N can be an integer between 1 and 100; the upper limit may be somewhat less for you --- it depends on the number of file units that you can have open and the number of file units that you actually have open at the time the command is executed).

The basic difference between the two variants of 'dv' is that 'dv <file>' opens <file> for WRITE access; <file> cannot be used as an input file. 'Dv N' opens a temporary file for READ/WRITE access; therefore, 'so N' causes the temporary file to be rewound and read. If the command 'dv N' occurs a second, third, fourth etc. time, diverted lines are appended to the end

of that same temporary file.

One final important comment is necessary. We were hesitant to even tell you about 'dv' because of its rather nasty habit of executing commands instead of diverting them. Since it is the only way for you to generate automatic table of contents we decided to document it. Just keep in mind that when you want to divert commands, precede them by a character other than your control character; you can later designate that character as your control character before you read the stream.

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:

```
01:22:49
```

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]
```

Similarly, a right bracket may appear literally inside a function call when preceded by an escape character:

```
[bf [item 1@]]
```

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 200 accumulators (numbered 1-200) 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., 11/27/84
day	Current day of the week; e.g., Tuesday
ldate	Current date: e.g., November 27, 1984
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, e.g. 'sprint')
sup	Output the arguments as a superscript (requires post-processor)
time	Current time of day; e.g., 01:22:54
ul	Underline the arguments on output
letter	Convert a number to its lower case equivalent
LETTER	Convert a number to its upper case equivalent
vertspace	Change the vertical spacing on a NEC Spinwriter (requires spinwriter)
even	Test if number is even
odd	Test if number is odd
cap	Capitalize Text
small	Map all characters of text to lower case
plus	Add two numbers
minus	Subtract two numbers
header	Return the page header
evenheader	Return the even page header
oddheader	Return the odd page header
footer	Return the page footer
evenfooter	Return the even page footer
oddfooter	Return the odd page footer
cmp	Perform string comparison
icmp	Perform integer comparison
bottom	Return the number of the last printed line
top	Return the number of the first printed line

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
lt	Length of titles
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
ns	True or false if no-space is in effect
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
* ALPHA	upper-case Greek alpha
beta	lower-case Greek beta
* BETA	upper-case Greek beta
* chi	lower-case Greek chi
* CHI	upper-case Greek chi
delta	lower-case Greek delta
* DELTA	upper-case Greek delta
epsilon	lower-case Greek epsilon
* EPSILON	upper-case Greek epsilon
eta	lower-case Greek eta
* ETA	upper-case Greek eta
gamma	lower-case Greek gamma
GAMMA	upper-case Greek gamma
infinity	infinity symbol
integral	integration symbol
* INTEGRAL	large integration sign
* iota	lower-case Greek iota
* IOTA	upper-case Greek iota
* kappa	lower-case Greek kappa
* KAPPA	upper-case Greek kappa
lambda	lower-case Greek lambda
LAMBDA	upper-case Greek lambda
mu	lower-case Greek mu
* MU	upper-case Greek mu
nabla	inverted delta (APL del)
not	EBCDIC-style not symbol
* nu	lower-case Greek nu
* NU	upper-case Greek nu
omega	lower-case Greek omega
OMEGA	upper-case Greek omega
* omicron	lower-case Greek omicron
* OMICRON	upper-case Greek omicron
partial	partial differential 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
*	RHO	upper-case Greek rho
	sigma	lower-case Greek sigma
	SIGMA	upper-case Greek sigma
	tau	lower-case Greek tau
*	TAU	upper-case Greek tau
	theta	lower-case Greek theta
	THETA	upper-case Greek theta
*	upsilon	lower-case Greek upsilon
*	UPSILON	upper-case Greek upsilon
	xi	lower-case Greek xi
*	XI	upper-case Greek xi
	zeta	lower-case Greek zeta
*	ZETA	upper-case Greek zeta
*	downarrow	arrow pointing down
*	uparrow	arrow pointing up
*	backslash	back slash symbol
*	tilde	tilde symbol
*	largerbrace	large square right brace
*	largelbrace	large square left brace
*	proportional	proportional symbol
*	apeq	approximately equal to
*	ge	greater than or equal to
*	imp	implies
*	exist	there exists
*	AND	logical and
*	ne	not equal to
*	psset	proper subset
*	sset	subset
*	le	less than or equal to
*	nexist	there does not exist
*	univ	for every
*	OR	logical or
*	iso	congruence
*	lfloor	left floor
*	rfloor	right floor
*	lceil	left ceiling
*	rceil	right ceiling
*	small0	a small 0
*	small1	a small 1
*	small2	a small 2
*	small3	a small 3
*	small4	a small 4
*	small5	a small 5
*	small6	a small 6
*	small7	a small 7
*	small8	a small 8
*	small9	a small 9
*	scolon	semicolon
*	dquote	double quote
*	dollar	dollar sign

The special characters marked with an asterisk (*) are only available on the NEC **Spinwriter,** and so the output of 'fmt' *must*

be post-processed with 'sprint'.

In particular, these characters require that the special Times-Roman/Mathematics type wheel be in the **Spinwriter**. This wheel, in order to accommodate the special characters, lacks certain of the regular ASCII graphics. These are substituted for by special functions. For example, [scolon] is used to produce a semi-colon.

Summary - Input Processing

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.dv [stream]	-	end '.dv'	no	Temporarily divert the output stream to a "filename" or to a temporary file designated by an integer "N" (to be later read by a ".so N" command) until a 'dv' command with no arguments is seen.
.nx file	-	next arg	no	Move on to the next input file.
.so <stream>	-	ignored	no	Temporarily alter the input source. "Stream can be a "-" to indicate standard input, a "filename," or an integer "N" corresponding to a temporary file created by a previous '.dv N' command.

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.

Appending To A Macro

It is possible to add text to the body of a previously defined macro, using the 'append macro' command:

.am xx

where xx is a previously defined macro. It is an error to append to a macro which has not been previously defined. The additional text of the macro is terminated with a '**.en** xx' command, just like the initial definition of the macro. The rules for the additional text of the macro are the same as for the initial text, i.e. any function calls or special characters must be escaped with an "@" sign to prevent their immediate evaluation.

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.
.am xx	-	ignored	no	Add additional text to the body of a previously defined macro.

Conditional Line Processing

Introduction

This sections discusses the features of 'fmt' which provide you with considerable control and flexibility over the formatting of your documents.

The .if command

'Fmt' allows you to test a condition and if that condition is true, it will execute a command. Optionally, you may specify a command to be executed if the condition is not true (an 'else' part). This is done using the 'if' command:

.if cond delim true_part [delim else_part]

This evaluates a condition ('cond') which, if it is true, will cause 'true_part' to be executed, just as if 'true_part' had been on a line by itself. If the condition is false, and the

'else_part' is present, then 'else_part' will be executed as if it had been on a line by itself. The 'delim' is any single non-blank character. For instance, the command

.if [odd [pn]] / .er odd page@n / .er even page@n

will write either 'odd page' or 'even page' to the terminal, depending on whether or not the current page is odd (the [odd] function will be discussed shortly).

The 'cond' can be negated by putting a '~' in front of it. Note that 'fmt' only checks for a single '~' to see if the condition is to be inverted. 'Fmt' is not a true programming language! It is probably almost always better to rewrite your condition than to use a '~' to negate it. The functions discussed below, and the ability to specify an 'else' part, provide ample flexibility to do whatever needs to be done.

A **.if** command with no arguments has no effect on the formatted output. The **.if** command may or may not cause a break, depending on the contents of the 'if' and 'else' parts.

Conditional Functions

'Fmt' provides four function calls which return either true or false (1 or 0) depending on the truth values of the conditions specified in their arguments. The four functions are as follows:

odd — Return true (false) if its integer argument is odd (even).

even — Return true (false) if its integer argument is even (odd).

cmp — Does a string comparison on its arguments, returning true if the specified relation is true, false otherwise. The form of this call is described below.

icmp — Does an integer comparison on its arguments, returning true if the specified relation is true, false otherwise.

The two comparison functions are called with three arguments, the first operand, a relational operator, and the second operand. The relational operators are:

<=	Less than or equal to.
=<	Less than or equal to.
<	Less than.
==	Equal to.
=	Equal to.
~=	Not equal to.

```
     <>        Not equal to.
     ><        Not equal to.
     >=        Greater than or equal to.
     =>        Greater than or equal to.
     >         Greater than.
```

A missing or incorrect operator is an error, and will cause 'fmt' to exit. As an example, to determine where you are, you could do the following:

```
          This must be
          .if [cmp [day] = Tuesday] / Belgium. /  Somewhere.
```

would cause the output to be "This must be Belgium." if it were Tuesday. Otherwise your text would simply wonder where it is.

Summary - Conditional Line Processing

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.if <args>	-	ignored	maybe	Conditional execution of an input line.

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 right 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.

Tables of Contents

Table of contents can be automatically generated by writing the contents to a temporary file, then at the end reading that file to produce the table of contents. In the examples above we could divert subheadings and headings to a temporary file; e.g., add the following to the 'sh' and 'mh' macros. (These examples are similar to what is used to produce the table of contents of this guide; for pedagogical reasons we have simplified it a little).

```
.# generate a table of contents entry for a heading
.dv 5
.cc #
#sp
#ne 8
[bf [1]] @[tc]@[tcpn]
#cc .
.dv

.# table of contents entry for sub-heading
.dv 5
.cc #
#ne 4
   [1] @[tc]@[tcpn]
#br
#cc .
.dv
```

Each time a heading is printed a line is written to temporary file "5" containing the heading, boldfaced, followed by a blank, a tab and finally the current page number right justified in four columns. Each time a subheading is printed a line is written containing three blanks, the subheading, a blank, a tab and finally the current page number. Note that we precede diverted commands by a different control character because 'dv' will execute commands instead of diverting them.

The very last command of the document would be a generate table of contents macro, e.g.,

```
.# TC --- generate table of contents
.de TC
.cc #
#bp
#fo ..- @[rn @[pn]] -..
#ce "TABLE OF CONTENTS"
#rm -6
#ta @[rm]
#rm +6
#rc .
#ns
#so 5
#cc .
.en TC
```

This macro will set the control character to correspond to the control characters written to output stream "5," advance to the top of the next page, center the heading "TABLE OF CONTENTS", set the footer to print the page number in small roman numerals (the page number must be set prior to calling 'TC'), set the tab column to 6 columns to the left of the right margin (this generates 2 blanks followed by the page number which is right justified in four columns), sets the replacement tab character to "." and reads the contents of temporary file "5".

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).

Subsystem Macro Packages

Introduction

The previous section discussed how you might go about writing macros which do all kinds of nifty things, including building a table of contents. Fortunately, you do not have to write your own macro packages, since the Subsystem comes with several already written.

The two major packages are the User Guide Macros, and the Report macros. The Report macros are an older set of macros; their use is discouraged in favor of the User Guide Macros, which can actually be easily adapted for almost any kind of paper you may have to write. Users who wish to use the Report macros may print them off to see what they do and how they work. They are in =fmac=/report and =fmac=/ds_report for single- and double-spaced reports, respectively.

There are also macros for formatting Master's and Ph.D. theses. These are contained in =fmac=/gt_thesis. They are meant to be used by themselves, without any of the =fmac=/ev?* files (discussed below). The macros are documented in the file itself; see there for details on using them. You will probably want to change them to have your school's name, instead of Georgia Tech.

Accessing The User Guide Macros

To use the User Guide Macros in your paper, you may name them on the command line, or more conveniently, use one of the lines

```
.so =fmac=/ugh
```

- or -

```
.so =fmac=/ugnh
```

as the first line in your 'fmt' input file. The first command provides you with a report that uses plain headings (like the ones in this guide), while the second provides you with numbered

headings (useful for technical reports). In either case, the macros are used in an identical fashion. You should not need to change the text of your document in order to get either numbered or plain headings; you just need to switch macro packages.

Each of these files sets up the macros for headings, and then does a

.so =fmac=/ugm

to include the rest of the User Guide macros.

Using The User Guide Macros

The User Guide macros will automatically produce a title page and table of contents. The macros and their functions are:

.TP	Start the Title Page.
.AU	List the name(s) of the author(s).
.PD [<date>]	Give the publication date.
.CH [<heading text>]	Chapter heading.
.MH [<heading text>]	Major heading (within a chapter).
.SH [<heading text>]	Sub-heading (within a major heading).
.PH [<heading text>]	Paragraph heading (within a sub-heading).
.pp	Start a new paragraph (do not use after .PH).
.bq [<length>]	Begin an indented quote.
.eq	End an indented quote.
.be [<length>]	Begin an example.
.ee	End an example.
.ep	Skip to an even page.
.op	Skip to an odd page.
.HI	Produce a hanging Indent. Used for lists like this one.

.TC	Generate the table of contents (reset the page number with a **.bp** *n*, first).

So, a full paper might look something like this:

```
.TP
On The Preservation Of The Arithmetic IF
.AU
Arnold D. Robbins
Eugene H. Spafford
.PD "[ldate]"
.op
.HE "Saving The Arithmetic IF"
.#  The .HE macro will be explained shortly
.fo ''- # -''
.CH "Chapter 1"
...
.MH "Major 2"
...
.SH "Sub 3"
...
.PH "Par 4"
...
.bp 3
.TC
```

The title page produced would look just like the title page of this guide. You may want to change the **.PD** macro in =fmac=/ugm to have the name and address of your school or business, instead of Georgia Tech.

The heading macros each use two additional macros; one to help generate the table of contents, and one to actually produce the heading. For instance, **.CH** calls **.Ch** to produce the table of contents entry, and **.ch** to produce the chapter heading. The other header macros are implemented in a similar fashion. It is occasionally useful to access these macros directly; for instance in order to produce a foreword to a document, without having the foreword show up in the table of contents.

You should use all the .?H macros when writing your papers, i.e., the **.CH** macro, as well as the **.MH** and **.SH** macros. If you do not use the **.CH** macro, and you wish to use the numbered headings macros, your major sections will be sections 1, 2, 3, ... of Chapter 0, not Chapter 1, so bear this in mind.

It is *never* necessary to use a **.pp** macro after any of the heading macros, since they all do a **.pp** for you. In particular, the **.PH** heading macro should not be followed by a **.pp**; while after the other macros a **.pp** will only cause an extra line to be skipped.

The **.be** and **.bq** macros each take an optional argument, which is the the length of the example or quote. For a small quote or

example, you probably do not need to provide the length.

Since your entire document has to be formatted before the table of contents can be produced, the **.TC** macro should come at the end of your paper. You need to do a **.bp** *n* to the proper page for the table of contents (usually *n* = 3). The macros use diversion stream number five for the table of contents, so you should not use stream five for doing any of your own diversions.

The Printing Environment And The .HE Macro

The User Guide macros are designed so that a paper which uses them may be formatted on a variety of output devices, without changing the text of the paper. This is done by defining the printing environment in a macro; specifically the **.EV** macro. This macro takes care of setting the margin values, the page and margin offsets, the even and odd offsets, and the page length, among other things.

There are different environment files for different output devices. The files and the environments they are designed for are:

=fmac=/evd	Format output for the Diablo.
=fmac=/evp	Format output for the line printer.
=fmac=/evl	Format output for the Georgia Tech Xerox 9700 laser printer (See the help on 'lz'). These macros are for the User Guides.
=fmac=/evl2	Format output for the Georgia Tech Xerox 9700 laser printer. These macros are for the Reference Manual.
=fmac=/evt	Format output for "typesetting" on the Spinwriter. The output produced is designed to be photo-reduced to 8 1/2" by 11".

Unless you are positive that you will always use a particular output device, these files should *not* be included in your 'fmt' input file. Instead, they should be named on the command line. The **.TP** macro automatically calls the **.EV** macro to reset the environment.

The ev? files also define the **.HE** macro, which is used for designating the page headings. For single sided output, **.HE** is:

```
.de HE <left> <center> <right>
@[cc]he `[1]`[2]`[3]`
.en HE
```

while for double sided output (like the printed user guides), **.HE** is:

```
.de HE <left> <center> <right>
@[cc]eh `[1]`[2]`[3]`
@[cc]oh `[3]`[2]`[1]`
.en HE
```

The **.HE** macro should be placed right after the **.bp 1** command for the first page of your document, and before the first **.CH** command.

There is no special macro for footers. They are left to the **.fo** command. The usual choice is:

```
.fo ''- # -''
```

which places the page number at the bottom of the page.

There are environment files for the Report macros as well. The files are =fmac=/envd and =fmac=/envp for the Diablo and line printer, respectively.

Conclusion

The macros available to you with the Subsystem should satisfy most of your documentation needs, particularly with the variety of output devices that are supported. They can also be easily changed to suit your requirements, since the source files for the macro packages are included with the Subsystem.

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.
.am xx	–	–	no	Add additional text to the body of a previously defined macro.
.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.
.dv <stream>	–	end '.dv'	no	Temporarily divert the output stream to a "filename" or to a temporary file designated by an integer "N" (to be later read by a ".so N" command) until a 'dv' command with no arguments is seen.
.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.
.eo +N	N=0	N=0	yes	Set even page offset.
.er text	–	ignored	no	Write a message to the terminal.

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.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.
.if <args>	–	ignored	maybe	Conditional execution of an input line.
.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.
.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.
.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.

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.ns	on	-	no	Turn on 'no-space' mode.
* .nx file	-	next arg	no	Move on to the next input file.
.of /l/c/r/	blank	blank	no	Set odd-numbered page footing.
.oh /l/c/r/	blank	blank	no	Set odd-numbered page heading.
.oo +N	N=0	N=0	yes	Set odd page offset.
.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.
.ps N M	N=M=0	N=M=0	yes	Skip pages while (page number mod M) is less than N.
.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 <stream>	-	ignored	no	Temporarily alter the input source. "Stream can be a "-" to indicate standard input, a "filename," or an integer "N" corresponding to a temporary file created by a previous '.dv N' command.
.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.

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.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.

TABLE OF CONTENTS

Software Tools Subsystem
Manager's Guide

T. Allen Akin
Terrell L. Countryman
Daniel H. Forsyth, Jr.
Jefferey S. Lee
Jeanette T. Myers
Arnold D. Robbins
Peter N. Wan

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

September, 1984

Overview

You are reading the **Software Tools Subsystem Manager's Guide**. The machine-readable text of this Guide is supplied with the Subsystem in the file "=doc=/fguide/mgr" (already formatted) and in the directory "=doc=/guide/mgr" (unformatted).

Purpose

This Guide addresses the needs of the Subsystem Manager: that individual or small group of individuals that is responsible for the installation, maintenance and daily operation of the Subsystem.

Bringing up a large software system is a complicated process that involves several bootstrapping steps. In this case, the Subsystem Manager will be responsible for at least three: the installation of the Subsystem, initial distribution of documentation, and creation of user accounts.

Once the system is running and a sizable user community develops, the responsibilities of the Manager will change. In particular, he will control the generation and distribution of new copies of documentation, maintain lists of active users, update the description of the local hardware configuration, and possibly modify the Subsystem itself by changing either the system code or documentation.

It is the intent of this Guide to provide the Manager with the information necessary to carry out these duties and with procedures for their performance that are recommended by the Subsystem's designers.

For further information on the philosophy and use of the Subsystem, the reader is referred to the *Software Tools Subsystem Reference Manual*, the *Software Tools Subsystem User's Guide*, and, of course, to Kernighan and Plauger's *Software Tools*.

Summary of Contents

This Guide is divided into five sections as outlined in the following five paragraphs.

The **Subsystem Configuration** section deals with the directory structure of the Subsystem. It describes the standard configuration supplied on the Release Tape as well as options for changing directory names, locating directories on specific logical disks, and storing certain portions of the Subsystem off-line.

The **Installation Procedure** covers the initial installation of the Subsystem. It catalogs the contents of the Subsystem Installation Package and Release Tape, provides instructions for loading the tape, and details the steps the Manager must take in

order to make the Subsystem operational.

The **Conversion Procedure** section describes the steps necessary to update your current Subsystem to this release. If you are a new customer, then you need only peruse this section for various caveats, but not necessarily act upon the information contained therein.

The section on **Documentation Structure** describes the nature, coverage and physical location of all Subsystem documentation. Its two sub-sections correspond to the two major Subsystem documents: the *Software Tools Subsystem Reference Manual* and the *Software Tools Subsystem User's Guide*.

The day-to-day activities of the Subsystem Manager are covered in **Subsystem Management**. Such concerns as maintaining user accounts and hardware configuration files, adding local software tools and documentation, and operating Subsystem user services are treated.

Subsystem Configuration

The Subsystem is a complex piece of software that resides in several disk directories. This section discusses the standard directory structure and the means provided for changing it, some alternative directory structures, and naming and structural conventions that must be followed if changes are made.

Standard Directory Structure

The following chart outlines the structure of the major Subsystem directories as supplied on the Release Tape:

aux
    primes (file of prime numbers less than 1,000,000)
    spelling
        {dictionaries of English words, place names, etc.}

bin
    {standard Subsystem commands, supported by GT}

lbin
    {locally-supported commands}

temp
    {scratch files created by Subsystem programs}

```
| vars
      {subdirectory for each user}
          .mail      (old mail storage)
          .vars      (shell variable storage)
          .template  (user template definitions)
|         .hist      (user shell session storage)

  extra
      bin
          {programs called by shell files in 'bin'}

      bug
          {bug reports gathered by 'bug'}

|     cron
|         {example files for use by the 'cron' program}
|
      clist (list of command names used by 'guess')

      gossip
          {file for each user, for messages sent by 'to'}

*     installation (installation name)

      mail
          {file for each user, for messages sent by 'mail'}

      memo
          {file for each user, for memoranda from 'memo'}

      moot.u
          {files used by 'moot'}

      news
*         articles
              {files containing one news article each}
          index       (index to all past news articles)
          subscribers (list of news service subscribers)

          delivery
              {file for each user, for undelivered news}

      phones (list of phone numbers used by 'phone')

      terms (list of terminals attached to the system)

      users (list of authorized Subsystem users)

      fmacro
          {miscellaneous text formatter macro files}

      incl
|         {macro definitions for Ratfor, C, and PMA}

      numsg    (propaganda message sent to new users)
```

```
    template (pathname template definition file)

    vth
        {files describing terminal hardware characteristics}

    ttypes (list of Subsystem-supported terminals and their
        characteristics)

src

    lcl
        lib
            {subdirectories for locally-supported libraries}
        spc
            {subdirectories for local programs with non-
             standard compilation requirements}
        std.r
            {source files for local Ratfor programs}
        std.sh
            {source files for local shell programs}
        std.stacc
            {source files for local Ratfor/'stacc' programs}

    lib
        c$main
                {files to build the C startoff routine ---
                only for sites which license the C compiler}
        cio
            {source files for C I/O library --- only for
            sites which license the C compiler}
        edt
            {source files for line editor routines}
        math
            {source files for 'vswtmath' library}
        sh
            src
                {source files and directories for 'vshlib'}
            {miscellaneous files for 'vshlib'}
        swt
            obj
                {object code files for 'vswtlb' routines}
            src.a
                (archive containing source code for 'vswtlb' routines)
            {miscellaneous files for library 'vswtlb'}
        vcg
            {source files for the vcg support library ---
            only for sites which license the C compiler}
        vcg_main
            {source for main routine for vcg for individuals
            that have written their own front ends for vcg ---
            only for sites which license the C compiler}

    spc
        {subdirectories for programs with nonstandard
         compilation requirements}
```

```
    std.r
        {source files for standard Ratfor programs}

    std.sh
        {source files for standard shell programs}

    std.stacc
        {source files for standard Ratfor/'stacc' programs}

    ext.c
        {source files for C programs in "=ebin=" --- only
        for sites which license the C compiler}

    ext.r
        {source files for Ratfor programs in "=ebin="}

    ext.sh
        {source files for shell programs in "=ebin="}

    misc
        {Subsystem support and maintenance routines}

doc
    build
        guide   (format a new version of the User's Guide)
        man     (format a new version of the Reference Manual)
        rebuild (format a new Reference Manual entry)

    fguide
        {files containing formatted portions of the
         User's Guide}

    fman
        s1
            {formatted standard command documentation}
        s2
            {formatted standard library documentation}
        s3
            {formatted local command documentation}
        s4
            {formatted local library documentation}
        s5
            {formatted low-level command documentation}
        s6
            {formatted low-level library documentation}
        {miscellaneous formatted portions of Reference Manual}

    guide
        {subdirectories containing portions of the User's
         Guide, unformatted}

    hist
        history (history of changes to the Subsystem)
```

man
s1
{unformatted standard command documentation}
s2
{unformatted standard library documentation}
s3
{unformatted local command documentation}
s4
{unformatted local library documentation}
s5
{unformatted low-level command documentation}
s6
{unformatted low-level library documentation}
{miscellaneous unformatted portions of Reference Manual}

print
guide (print a copy of the User's Guide)
man (print a copy of the Reference Manual)

se_h
{files containing on-line help information for 'se'}

misc
{Documentation support and maintenance routines}

Top-Level Directories. The top-level directories 'aux', 'bin', 'doc', 'extra', 'lbin', 'src', 'temp' and 'vars' are dedicated to the Subsystem. In addition, use of the Subsystem requires that several files be added to the Primos directories 'cmdnc0', 'lib', and 'system'. (A list of these files may be found in the section on Installation Procedures in this Guide.)

Previously, the 'cmdnc0' directory used for Subsystem files had to be the directory to which the console was attached after a cold start. Under the new revision (Revision 19) of the operating system, the 'cmdnc0' directory used for Subsystem support commands must be the 'cmdnc0' located on the lowest numbered logical disk partition, to ensure that users can enter the Subsystem properly. The 'lib' directory used for Subsystem libraries must also be on the lowest-numbered logical disk, so the libraries will be locatable by the loader. The 'system' directory used for Subsystem shared segment files should be the directory in which all standard Primos shared code resides, so that the shared Subsystem programs may be installed during a cold-start.

Directory Security and Placement on Disk. The subsystem is supplied with ACL protections but if the tape is restored onto a password partition, the password protections will override the ACL protections. Although the Subsystem will operate properly regardless of the placement of its top-level directories, substantial reduction in overhead may be had by following these recommendations. The following discussion normally describes the necessary protections for password directories and then follows that with the protections needed in the case of ACL directories.

The directories 'bin' and 'lbin' are accessed very frequently (about once per command) and so should be located on a low-numbered logical disk. The files in these directories must be readable by non-owners but should be protected against alteration by ordinary users. This can be accomplished by placing an owner password on the directories or by making the directories ACL protected directories with permissions of "list", "use", and "read" for everyone. (But see the *User's Guide for the Software Tools Subsystem Command Interpreter*, for information on "search rules.")

Scratch files created by Subsystem programs reside in the directory 'temp'. The concept of a "temporary file directory" is necessary to allow editing of files on read-only disks or in directories in which the user has non-owner status. Depending on the application at hand, files in 'temp' may grow to excessive size, so it should be placed on a logical disk with plenty of available storage space. It is not accessed frequently, so its placement is otherwise unconstrained. 'Temp' must be public; that is, it must have either a blank owner password or, if it is an ACL protected directory, permissions of "list", "use", "add", "delete", "read", and "write" for everyone.

Subdirectories of the directory 'vars' are used for storage of personal profile information. 'Vars' is accessed infrequently, and is typically small; it may be placed on any convenient logical disk. 'Vars' itself should have an owner password to preserve the privacy of individual users; it may not have a non-owner password. If 'vars' is made an ACL protected directory, it should have permissions of "list" and "use" for everyone. The Subsystem manager must create in 'vars' a subdirectory for each Subsystem user, named by that user's login name. Each of these subdirectories should be protected by an owner password of the user's own choosing and should have no non-owner password. If they are ACL protected, they should be given ACL protection for the owner of "list", "use", "add", "delete", "read", and "write" and "list" and "use" permissions for everyone else. If the directory is ACL protected, the Subsystem will not request a password to allow entry. If it is password protected, the 'swt' command prompts for the owner password and records it internally before entering the Subsystem; this saved password is then used in all future references to the directory by the Subsystem. If the user wishes to change the directory's password, he must do so outside of the Subsystem, so that the Subsystem will be able to exit normally.

Miscellaneous information that pertains to the Subsystem resides in the directory 'extra'. 'Extra' is relatively small and is frequently referenced (to check for messages sent from user to user via the 'to' command), so it should be placed on a low-numbered logical disk. All contents of 'extra' and its subdirectories should be readable by non-owners and free of non-owner passwords. If it is ACL protected, each file should be protected so that everyone can read it and each directory should be protected with "list", "use", and "read" protections for everyone. The subdirectories 'mail', 'gossip', and 'memo' must

be public so that anyone can create a file in them ("list", "use", "add", "delete", "read", and "write"). The files in the subdirectory 'news' should normally be writeable by anyone and its subdirectories public; however, the Subsystem Manager may see fit to restrict subscriptions to the news service by removing non-owner write access to the 'subscribers' file, and publishing of news articles by removing non-owner write access to the 'index' file.

'Src' contains all Subsystem source code. It is extremely large and very infrequently used. It should be placed on a high-numbered logical disk, and, at the discretion of the Subsystem Manager, be protected to prevent unauthorized access.

'Aux' contains several large auxiliary files, particularly the dictionary of English words and the list of prime numbers less than one million. It should be placed on a high-numbered logical disk. Files in 'aux' and its subdirectories should be readable by non-owners, and there should be no non-owner passwords. An owner password may be employed at the discretion of the Manager to enforce security. The ACL protections would be "list", "use", and "read" permissions for everyone. If you uncomment the template =new_word=, and leave it as =aux=/spelling/new_words, then this file needs to writeable by everyone. (Permissions of a/rw, or "read" and "write" ACL permissions.)

'Doc' contains the formatted and unformatted versions of both the Reference Manual and the User's Guide. It should be placed on a high-numbered logical disk. Generally, its contents should be readable by non-owners. It may be owner password protected at the discretion of the Manager, but should not have a non-owner password (ACL permissions of "list", "use", and "read" for everyone). The same applies to all of its subdirectories.

Alternative Directory Structures

For various reasons (lack of disk space or naming conflicts, for example) the Subsystem manager may need to restructure the Subsystem or even remove portions of it entirely. This section describes the actions necessary to reconfigure the Subsystem directory structure to meet local needs.

Templates and Top-Level Directories. File names (alias "pathnames") in the Subsystem feature a number of extensions beyond the capabilities of Primos treenames. (For a full discussion of pathnames, please see the *User's Guide to the Primos File System* in the *Software Tools Subsystem User's Guide*.) The extension that bears on Subsystem directory structure is a simple macro substitution facility that goes by the name of "templates." When an identifier enclosed in equals bars (=) appears in a pathname, it is automatically replaced by some appropriate substitution text. In particular, such "templates" have been provided for the names of all Subsystem top-level directories, and all Subsystem code follows the convention that top-level directories

are always named by a pathname containing the appropriate template.

Reconfiguration of the Subsystem's directory structure may be accomplished simply by changing the substitution text that replaces the top-level directory templates. The templates and their substitution text may be found in the file 'template' in the directory 'extra' (on the Release Tape). (This file may be edited with either the Primos editor or one of the Subsystem editors.) For example, suppose that the directory 'doc' conflicts with a local directory of the same name. Edit the template definition file, and change the following line

```
doc         //doc
```

to

```
doc         //tools_doc
```

then change the name of 'doc' to 'tools_doc'. The reconfiguration is complete.

It should be noted that if the name or location of the directory 'extra' or of the template definition file itself is changed, the 'initswt' program run at cold start time must be given a command line argument that specifies the new location of the template file. See the section on Subsystem Installation for further details.

As supplied, the template definitions for all top-level Subsystem directories use the omitted-packname option of the pathname syntax. This means that any time one of these directories is referenced, an ascending search of the MFDs on all logical disks is made until the directory is found. If circumstances prevent placement of the frequently-referenced Subsystem directories on low-numbered disks, it is still possible to avoid the overhead of long directory searches by changing the template definitions to include explicit packnames or logical disk numbers. If this is done, however, the Subsystem must be reinitialized any time one of its directories is moved to another pack.

Off-Line Storage. Certain portions of the Subsystem are not required for everyday usage, and may be removed in order to conserve disk space. The following paragraphs list the directories that may be stored off-line.

The source code directory 'src' is extremely large and may be useless on a production system. It may be stored on tape with impunity (although doing so will cause the 'locate' and 'source' commands to cease functioning).

The on-line documentation supplied with the Subsystem has been found extremely useful in the past, both to new users learning the system and to expert users needing a refresher course on the usage of particular commands. However, none of it is

essential to the operation of the Subsystem; the entire directory 'doc' may be stored off-line. As a less drastic measure, the unformatted versions of the Reference Manual and User's Guide that reside in the subdirectories 'man' and 'guide' may be stored off-line, while everything else remains on disk. (This allows the 'help' and 'guide' commands and the 'h' command of 'se' to function properly.)

If the dictionary of English words and the list of prime numbers are not frequently used, the directory 'aux' may be stored off-line. This affects the commands 'spell', 'speling' and 'rsa', the template =new_words= (if it is defined), and the local math library routine 'prime'.

Installation Procedure

This section covers the procedures necessary for installation of the Subsystem. It lists the contents of the Installation Package and the Release Tape, and provides instructions for loading the tape and initializing the Subsystem. Before reading this section, a thorough study of the **Subsystem Configuration** section of this Guide is recommended.

Subsystem Installation Package

For new customers, the Subsystem Installation Package as sent from Georgia Tech contains the following items:

1 Release Tape
1 Copy of the Subsystem Manager's Guide
1 Copy of the Reference Manual
1 Copy of the User's Guide

Old customers who are updating to Release 9 will only receive the Release Tape, the Conversion Guide, and the Manager's Guide.

Release Tape Contents

The Subsystem Release Tape contains all files and directories necessary for proper operation of the Subsystem. It is in standard MAGSAV/MAGRST format and contains four "logical tapes." Each logical tape contains a number of separate directories that normally would reside on the same logical disk.

Logical Tape 1. The first logical tape contains the following three directories:

cmdnc0 lib system

and these directories contain the following files:

	cmdnc0>swt	used to enter the Subsystem
	cmdnc0>swtseg	latest revision of SEG, modified slightly for the Subsystem. Its output is completely compatible with standard SEG.
	cmdnc0>snplnk	snaps dynamic pointer links (see below)
	lib>vswtlb	shared Subsystem I/O and utility library
	lib>nvswtlb	unshared version of vswtlb
*	lib>p4clib	bootstrap Pascal compiler run-time-support library
	lib>vedtlb	line editor library
	lib>vswtmath	high precision mathematical function library
	lib>shortlb	shortcall routines for FORTRAN
	lib>vlslb	linked string library
	lib>vrnglb	ring support library
	lib>vshlib	shared Shell utility library
	system>cron.comi	example startup file for 'cron'
	system>ring.comi	example startup file for 'ring'
	system>sh2030	shared portion of the command interpreter
	system>st2030	shared data area for templates
	system>se2031	shared portion of the screen editor
	system>sw2035	shared portion of the Subsystem library
	system>sh4000	used to install the command interpreter
	system>sw4000	used to install the Subsystem library
*	system>initswt	used to initialize pathname templates

These files must be placed in the appropriate Primos directories at your installation. They should be placed in 'cmdnc0', 'lib', and 'system' on the lowest-numbered logical disk containing those directories.

Logical Tape 2. The second logical tape contains the following directories:

bin lbin extra

'Bin' is the standard Subsystem command directory. It contains the executable versions of all Georgia Tech-supported Subsystem commands.

'Lbin' is a command directory for locally-written tools. Commands in 'lbin' are normally useful at only one installation, or have not been found valuable enough to merit full support.

'Extra' contains miscellaneous information used by various parts of the Subsystem. In particular, it houses the mail, news and memo delivery directories, which tend to grow steadily in size over a period of time.

Logical Tape 3. The third logical tape contains the following directories:

vars temp

These directories should be placed on a disk partition with a large amount of free space, since files in 'temp' may become arbitrarily large.

'Vars' is used to store personal profile information for all Subsystem users.

'Temp' is a special directory dedicated to containing scratch files.

Logical Tape 4. The fourth logical tape contains the following directories:

doc src aux

These directories are all very large and infrequently accessed. They do not normally vary much in size.

'Doc' contains formatted and unformatted copies of all Subsystem documentation.

'Src' contains all releasable Subsystem source code.

'Aux' contains miscellaneous auxiliary files, such as the dictionary of English words and the list of prime numbers.

Loading the Tape

To load the release tape, follow the instructions below:

1. Assign a tape drive:

```
ASSIGN MT0
```

2. Mount the release tape on the assigned drive.

3. Attach to the master file directory on the logical disk containing 'cmdnc0', 'lib', and 'system' (usually disk 0):

```
ATTACH MFD <owner-password> <disk-number>
```

or if the tape is being restored to an ACL or priority ACL protected partition, type

```
ATTACH MFD <disk-number>
```

4. Load the contents of the first logical tape with MAGRST:

```
MAGRST
Tape Unit (9 Trk): 0
```

```
          Enter logical tape number: 1
          <tape label information>
          Ready to Restore:  yes
```

(This will load the files in 'cmdnc0', 'lib', and 'system'.)

5. Attach to the master file directory on the logical disk selected for the 'bin', 'lbin' and 'extra' directories.

6. Load the contents of the next logical tape (i.e., reply "0" to the "Enter logical tape number:" prompt) with MAGRST. (This will load the directories 'bin', 'lbin' and 'extra'.)

7. Attach to the master file directory on the logical disk selected for the 'temp' and 'vars' directories. It should have ample free space.

8. Load the contents of the next logical tape with MAGRST. (This will load directories 'vars' and 'temp'.)

9. Attach to the master file directory on a logical disk with a great deal of free space.

10. Load the contents of the next logical tape with MAGRST. (This will load directories 'aux', 'doc', and 'src'.)

This completes the loading of the Subsystem from tape.

Reconfiguration of Primos for the Subsystem

Primos Revisions 18.0 and above have now used all normally available private memory segments. In order to bring up the Subsystem, it is necessary to increase the NUSEG parameter in the Primos configuration file to at least 43 (octal), up from the default of 40 (octal), to provide private segments for the Subsystem that do not conflict with standard Prime programs. It also implies that you cannot bring up the Subsystem without rebooting your system, unless you already have the NUSEG parameter set high enough.

Initialization of Shared Segments

Several important portions of the Subsystem reside in shared memory segments. Once the release tape is loaded, these segments must be initialized.

One of the enhancements provided with Version 9 is increased security of shared segments. The SNPLNK ("Snap Link") program shown in the commands below runs through a given segment and "snaps" the dynamic subroutine linkages. In other words, all pointers which are set up as dynamic links are turned into real pointers. This is usually done when a program runs, by the Ring 3 pointer fault handler. By snapping all the links at one time, these segments can then be shared as read only. This will

prevent an errant program from scrambling the shared libraries.

Type the following commands on your system console:

```
OPR 1
SHARE SYSTEM>SW2035 2035 700
SHARE SYSTEM>SH2030 2030 700
SHARE SYSTEM>ST2030 2030 700
SHARE SYSTEM>SE2031 2031 700
R SYSTEM>SW4000
R SYSTEM>SH4000
R SYSTEM>INITSWT

SNPLNK 1/2030; SHARE 2030 600
SNPLNK 1/2031; SHARE 2031 600
SNPLNK 1/2035; SHARE 2035 600
OPR 0
```

Ideally, the preceding commands would be placed in your cold-start procedure file CMDNC0>C_PRMO or CMDNC0>PRIMOS.COMI, so they will be performed automatically after every cold-start. Note: if you have changed the name or location of the template definition file or the 'extra' directory, you must specify the new name of the template file on the invocation of 'initswt'. For example, if you have changed the name of the 'extra' directory to 'etc', use the following command instead of "r system>initswt":

```
R SYSTEM>INITSWT ETC>TEMPLATE
```

For installations that had a previous release of the Subsystem, this completes the installation procedure. The Subsystem should now be ready to go. Otherwise, new Subsystem managers should read the next subsection, which describes the remaining steps.

Initial Log-in by SYSTEM

If this is your first Subsystem release, several further steps are necessary to complete installation. As delivered, the Subsystem has only one active user account: that for the login name SYSTEM, which is assumed to be used only by system administrative personnel. Once the Subsystem is loaded and initialized, the Subsystem Manager should log in as user SYSTEM and verify that the Subsystem is working.

Login as user SYSTEM and type the following command:

```
swt
```

If the 'vars' directory was restored on a password partition, 'swt' will prompt for the owner password of SYSTEM's profile directory. The Subsystem is delivered with a null password for SYSTEM, so just strike the RETURN key. The shell (Subsystem command interpreter) will then be executed. If the 'vars' directory was restored on an ACL partition, the 'swt' will not prompt for any password, but will immediately execute the shell. Before it

will accept any commands, the shell will prompt you with "Enter terminal type: ". You should respond with the mnemonic for your terminal type; if you do not know the correct mnemonic, respond with a "?" and the shell will provide a list of acceptable responses. After you have entered an acceptable mnemonic or a RETURN (if you do not wish a terminal type associated with your login session), the shell will be ready to accept commands. You should see a "]" prompt, indicating that the Subsystem is up and running.

Modify the file "=installation=" to contain the name of your installation. (The easiest way to do this from the Subsystem is to type a command similar to the following:

```
echo "Georgia Tech System B" >=installation=
```

Simply replace "Georgia Tech System B" with the name of your installation.)

Before the Subsystem can be released for general use, profile storage directories must be created for all potential users, and their names must be entered in the "=userlist=" file. In addition, descriptions of all terminals attached to the computer must be entered in the "=termlist=" and "=ttypes=" files. For information on these tasks, see the **Subsystem Management** section of this Guide.

Resolving Shared Segment Conflicts

If the segment numbers used by the shared Subsystem programs and libraries conflict with those used by other programs at your installation, you can change the Subsystem segment numbers; however, you must first install the Subsystem as supplied. Also note that you must change the SHARE commands used in your 'c_prmo' or 'primos.comi' cold start command file to reflect the changed segment numbers.

The Subsystem makes use of three shared segments: 2030 for the Shell and system template table, 2031 for the screen editor 'se', and 2035 for the shared library.

The directory for building the Shell is "=src=/lib/sh". In this directory there is a file named "segment" which contains the segment number to be used for the shared portion of the object code. First, change the contents of this file to the desired segment number; then simply execute the Shell program 'build'. This will produce three object codes files, 'sh', the interlude program which should be placed in "=bin=" as 'sh' and "=system=" as 'sh4000', "sh", the shared code which should be copied to "=system=" for automatic sharing by your 'c_prmo' or 'primos.comi' cold start procedure, and 'vshlib', which should be copied to "=lib=". This copying can be done by executing the Shell program 'install'. For example, the following would fix the shell to run in segment 2037:

```
cd =src=/lib/sh
echo "2037" >segment
build
install
```

Also in segment 2030 is the shared portion of 'swt', the Subsystem initialization program. To change its segment, attach (using 'cd') to the directory "=src=/spc/swt.u", change the contents of the file "segment" to the desired segment number, and execute the shell program 'build', just like changing the Shell's segments. Then execute 'install' to copy the shared portion in the file "st" into "=system=" and copy 'swt' into "=cmdnc0=". (Please note that if you change the code for 'swt', it must generate no sector-zero links. If it does, you will wipe out the shell when sharing it!)

Segment 2030 is also used for the storage of system templates. If you must change the location of their storage area, you must alter the loader interface program 'ld' in order to specify a new absolute address for the storage area, then rebuild the Shell (as outlined above), the libraries (as outlined below), the program 'initswt', and any local program that uses the unshared version of the Subsystem library ('nvswtlb'). For further information on the implications of moving the template storage area, please contact Georgia Tech.

The screen editor normally resides in segment 2031. To move it, attach to the directory "=src=/spc/se.u", change the contents of the file "segment" to the desired segment number, then execute the Shell program 'build'. This will yield two object code files: 'se', the interface program that should be placed in "=bin=", and "se", the shared portion that should be placed in "=system=" to be shared in at cold-start time. This copying can be done by executing the Shell program 'install'.

The shared libraries normally reside in segment 2035. To move them, attach to the directory "=src=/lib/swt", change the contents of the file "segment" to the desired segment number, then execute the Shell program 'build'. The object code files, 'vswtlb' and 'nvswtlb' should be copied to the "=lib=" directory; the shared code file "sw" should be copied to "=system="; and the file 'inst' should be copied to "=system=" and renamed "sw4000". This copying can be done by executing the Shell program 'install'.

Segments Used

The following table lists the segments in virtual memory, and how they are used by the operating system and various Subsystem programs.

Segment	*Use*
0000 - 0401	Operating System

2030	Software Tools Shell
2031	Software Tools Screen Editor
2035	Software Tools Library
2050	Fortran Library
4000 - 4037	User Program
4040	Software Tools Common
4041	Software Tools Stack
4042	Software Tools Common
6000	Operating System Data
6001	Fortran Library
6002	Primos Ring 3 Stack
6003	Operating System Stack

Some user programs use certain predefined user segments for their own use, so you have to be careful where you load your programs. For instance, if a program uses segment 4006, and you run it from the shell, from within the screen editor, you will destroy the screen editor's common blocks. If any of the programs or routines that use these predefined segments are not in use, they are available for user programs. For example, If the screen editor is not in use, segments 4006 and 4007 can be used with no problem, and if the Primos routine P$ALC is not being used (it is used by C, Pascal, and PL/I programs) then segments 4010 through 4027 become available.

Segment	*Use*
4000 - 4005	User Program
4006 - 4007	Screen Editor Per User Common
4010 - 4027	Primos Dynamic Memory -- P$ALC routine
4036 - 4037	SEG Symbol Tables

Changes for Primos Rev. 19.4

When Rev. 19.4 of Primos is released, you will need to make two changes to allow you to run the Subsystem under it. First, change the definition of =cldata= to be "6002 12", instead of "6002 6". There is a commented out template definition for this in the =template= file. All you have to do is remove the leading comment symbol, for this definition, and comment out the old one. Secondly, in the file =src=/spc/swt.u/init_s.s, change the definition of CLDATA$SM_FAULT_ERR from "XB%+90" to "XB%+91". This is because several Primos internal data structures change their location at Rev. 19.4.

Conversion Procedure

This section contains pointers for re-installing the Subsystem on a system running an older version of the Subsystem.

User Impact

Before trying to use a newer Subsystem release, first study the Conversion Guide included with the new release to determine what, if any, impact will be felt by your user community. If you need complete information on the changes made to programs, you can load the documentation directories "//doc/fman" and "//doc/fguide" from the release tape.

Usually, a Subsystem release is largely compatible with the release it replaces. Those incompatibilities that do exist are noted in the Conversion Guide. If incompatible changes have been made to a command so that you determine it unreasonable to force your user community to convert to the new command, you have several alternatives: (1) you can write shell programs to cover the incompatibilities, (2) in many cases, you can install the old command from the previous release (you may have to recompile it, though), or (3) you can not install the new Subsystem release. If you find it necessary to take the third alternative, please contact us so that we can try to find a better solution to your dilemma.

Compatibility is a different story when you have locally modified versions of Subsystem programs or you have locally written programs that take advantage of "secret knowledge" of the Subsystem's internals. In this situation, you must examine the newly released programs that interface with your local software to determine the changes necessary to interface with the new release. If you have difficulty in this area, please contact us and perhaps we can suggest possible solutions.

Installing the New Subsystem

Once you have determined the suitability of the new Subsystem release and have mapped out a conversion plan, you are ready to test and install the new release. Unfortunately, two different versions of the Subsystem cannot run simultaneously because the Primos shared library mechanism has no provision for duplicate shared entry points. Therefore, to test and install the new Subsystem, you must bring up the new Subsystem only when no other users are running with the old Subsystem.

Before loading the new release, you must first save the old Subsystem files and directories. You can then immediately restore the old Subsystem in the case that the new one malfunctions. If you have about 16 million bytes of disk available, you can just change the directory names:

```
CNAME BIN OLD_BIN
CNAME LBIN OLD_LBIN
CNAME DOC OLD_DOC
CNAME SRC OLD_SRC
CNAME EXTRA OLD_EXTRA
CNAME AUX OLD_AUX
```

Otherwise, you must copy the directories to tape (or removable disk) and delete the original versions. If you copy the files to tape, copy 'extra' and change its name; do not remove it from disk -- you will need it later. Do not change the directories 'vars' and 'temp'; these can be used as they are. Be sure to save the Subsystem files in 'cmdnc0', 'lib', and 'system'.

Then, load the release tape just as explained in the **Installation Procedure** section, leaving out the load of 'vars' and 'temp'. After you place the new Subsystem files in 'system', 'cmdnc0', and 'lib', re-boot your system. (Unless you are familiar with the shared library re-installation procedures, a re-boot is the only safe way to re-install a shared library.) The new Subsystem should now be available for use.

What To Do About Pre-8.1 Programs

Any calls to the subroutine 'init' should be removed from your programs. You should then recompile them, making sure that nothing depends on the value of EOS being less than 0. In fact, no program should depend on any properties of EOS. The Version 8 Compatiblity libraries are no longer supported. You may, at your own risk, continue using the V8-compatible libraries supplied with Release 8.1.

Modifications to Subsystem Files

Once you have installed the new release, you must move your local modifications into 'extra'. First, compare the files "=extra=/template" and "//old_extra/template" (using 'diff', if you like); add any local templates to "=extra=/template". Then update the templates: Exit the Subsystem and type

```
OPR 1
SHARE 2030 700
R SYSTEM>INITSWT
SHARE 2030
OPR 0
```

Copy the following files from 'old_extra' to 'extra':

```
installation
phones
terms
users
```

The 'users' file has changed format (see the section on adding and deleting users under **Subsystem Management**) so you might want to copy it the following way

```
=ebin=/cvusr old_extra/users extra/users
```

=ebin=/cvusr is a shell file that takes the pathnames of the old userlist and the new userlist as arguments, and expands the login names to the 32 character length needed by the 'whois' program. This conversion only needs to be done once, when the new Subsystem is first installed and brought up. New customers do not have to do this, of course.

Delete the following directories in 'extra' and copy them from 'old_extra' using 'cp':

```
gossip
mail
memo
moot.u
news
```

You may want to examine the articles in 'news' before replacing them with your local articles.

You must examine the following files and directories in 'extra' to determine if local changes need to be made:

```
bug
fmacro
incl
numsg
ttypes
vth
```

The shell uses a slightly different shell variable save file format to allow special characters to be encoded as mnemonics and prevent the file from getting scrambled if NEWLINE characters are accidently entered in a variable's value. A program, 'csv', is supplied to make this change simpler for each of the users. 'Csv' takes a list of user names as standard input, opens up the file "=vars=/<user-name>/.vars" and changes the appropriate shell variable values. This will only need to be done when the system is first brought up. If the directory "=vars=" contains only user variable directories then a simple command to perform the conversion is

```
lf -c =vars= | =ebin=/csv
```

or another way, if it contains other files, is to list the names into a file, edit the file, and then redirect the input into 'csv'. For example:

```
lf -c =vars= >user_names
se user_names               # make any changes here
user_names> =ebin=/csv
del user_names              # don't need it any more
```

Now, you may perform any tests that you like. We suggest that at minimum, you try the screen editor, a few shell files, and any other commands that are frequently used on your system.

Documentation Structure

Given a question about the Subsystem, where does one go in order to find the answer? This section attempts to address this problem, at least to the extent of enabling the Subsystem Manager to identify the document required and produce a printed copy of it if needed.

Software Tools Subsystem documentation is divided into two parts: the Reference Manual and the User's Guide. The Reference Manual is mostly technical information: usage summaries, listings of differences from standard *Software Tools* programs, etc. The User's Guide is mostly "soft" information: tutorials, applications notes, and the like. Each has its place, and must be accessed in its own way.

Reference Manual

The Reference Manual is normally the first port of call for anyone seeking an answer to a specific question. The Manual is composed of a number of entries, one for each command or subprogram in the Subsystem, divided into six sections: standard (i.e., supported by Georgia Tech) commands, standard subprograms, local (i.e., supported by the local installation, or not at all) commands, local subprograms, low-level commands, and low-level subprograms. The Manual is indexed by a simple list of entries and by an automatically generated key-word-in-context index.

A copy of the Reference Manual, formatted for 8.5" by 11" paper at 10 characters per inch horizontal, 6 lines per inch vertical, may be spooled for printing with the command

```
=doc=/print/man
```

Frequent use of this command is not recommended because of the
* manual's sheer size.

Individual Manual entries may be printed with the 'help' command. Simply typing

```
help <command-name>...     or
help <subprogram-name>...
```

will cause the selected Manual entries to be printed on the user's terminal, with a pause between each screenful. The user may also type

```
help -p <command-name> | os >/dev/lps/f
```

to obtain a printed copy of an entry exactly as it appears in the Manual.

'Help' may also be used to read the Manual's index. If "-i" is used in place of a command name, then the Manual's index will be printed. If "-f <pattern>" is used in place of a command name, then only those index entries matching the given pattern will be printed. For example, all commands and subprograms whose Manual entry heading lines contain the word "string" could be identified by typing

```
help -f string
```

This may be useful for people just learning to use the Subsystem, who may know several names for a function they wish to perform, but not the exact command or subprogram they need.

An unformatted copy of the Reference Manual resides in the directory "=doc=/man". A formatted version resides in "=doc=/fman". Should it ever be necessary to rebuild the formatted version, simply type

```
=doc=/build/man
```

User's Guide

The Guide is recommended for anyone just learning to use the Subsystem, as well as for those requiring a deeper knowledge of the workings of some of the more complex tools. It is actually a collection of separate papers, arranged roughly in the order required by a novice user. It is not intended as a quick reference; the Reference Manual performs that function.

Most papers in the Guide contain a tutorial section and an applications notes section, for beginners and experienced users, respectively. Some papers also contain a formal definition section; these would be of use only to those requiring a very complete understanding of the workings of the Subsystem.

The entire User's Guide, formatted for 8.5" by 11" paper, 10 characters per inch horizontal, 6 lines per inch vertical, may be spooled for printing by executing the following command:

```
=doc=/print/guide
```

The Guide is small enough that this operation is not overly expensive in terms of time or paper.

The 'guide' command may be used to print the individual papers that comprise the Guide. For example,

```
guide ed
```

would print the *Introduction to the Software Tools Text Editor* on the user's terminal, pausing after each screenful. To obtain a printed copy instead, one should type

```
guide -p ed | os >/dev/lps/f
```

It is generally good policy to have a number of copies of the *Software Tools Subsystem Tutorial* and the *Introduction to the Software Tools Text Editor* on hand for distribution to new users.

An unformatted copy of the User's Guide resides in the directory "=doc=/guide". Should it ever be necessary to rebuild the formatted copy in "=doc=/fguide", it can be done by typing

```
=doc=/build/guide
```

Subsystem Management

This section outlines the day-to-day responsibilities of the Subsystem Manager: adding and removing user accounts, keeping track of local hardware configuration changes, maintaining local tools, etc.

Adding and Deleting Users

Adding and deleting Subsystem user accounts boils down to the maintenance of one file and one directory.

The list of authorized users is addressed by the template "=userlist=". On a standard Subsystem, it resides in the file //extra/users. There is one line in the user list corresponding to each authorized Subsystem user. Users may appear in any order in the user list, although conventionally it is kept sorted alphabetically. The format of a line in the user list is as follows:

Columns	Information
1-32	User's login name, in upper case, left-justified, blank-padded
33	Blank
34-80	User's name and commentary information

Example:

```
12345678901234567890123456789012345678901...
BURDELL                          George P. Burdell (Development)
```

Whenever someone is added to the Subsystem user community, an appropriate entry must be made in "=userlist=". Whenever a user is no longer authorized to use the system, his entry must be removed.

Each Subsystem user possesses a "profile" directory, which must be created for him by the Subsystem Manager. When adding an account, create the profile directory with the command

```
mkdir =vars=/<login-name> -o <password>
```

if it is a password directory or just

```
mkdir =vars=/<login-name>
sacl =vars=/<login-name> <login-name>=adlurw $rest=lu
```

if it is an ACL protected directory. "<Login-name>" represents the login name of the new user. "<Password>" represents a standard file system owner password, which must be cited by the user when he enters the Subsystem if his variables directory is password protected. Example:

```
mkdir =vars=/gpb -o sesame
```

When a user is removed from the system, his profile directory must be deleted. This can be done most conveniently with the 'del' command:

```
del -sd =vars=/<login-name>:<password>
```

For example,

```
del -sd =vars=/gpb:sesame
```

In addition to the above measures for removing a user's account, the Subsystem Manager should check for any undelivered mail, gossip messages, or news articles. See the **Operation of Communications Systems** subsection below.

Specifying Local Hardware Configuration

The screen editor 'se' and a number of other programs that employ the virtual terminal handler library need to know the type and make of the terminals attached to each AMLC line on the system. This information is contained in the terminal list, which resides in the file "=termlist=" (nominally "//extra/terms"). Each line of the terminal list has the following format:

Columns	Information
1-3	Octal AMLC line number (000-177)
4	Blank
5-9	Three digit decimal user number of associated process, in parentheses
10	Blank
11-16	Terminal type (as recognized by the Subsystem); blank if unknown
17	Blank
18-80	Comments (usually physical location of terminal, etc.)

For example:

```
12345678901234567890...
007 (009) b200   George Burdell's office (Room 1729)
```

The contents of the "=termlist=" file must be kept up-to-date, or the 'e', 'whereis', 'se', and 'term_type' commands will cease to operate properly.

Adding Terminal Types

To extend the terminal type knowledge of the Subsystem, there are three places where changes need to be made in Subsystem files. These locations are the "=ttypes=" file, the "=vth=" directory, and various files under the screen editor directory, "=src=/spc/se.u". In all cases, the mnemonic for the new terminal may not be longer than six characters.

The "=ttypes=" file contains the terminal's general attributes. Its format consists of a mnemonic for the terminal name, the full terminal name, and then a series of flags. These flags currently indicate whether the terminal type is supported by 'se', whether the terminal type is supported by the VTH package, and whether the terminal type represents an upper-case only terminal.

The file "=src=/spc/se.u/how_to_add_terminal_types" gives the details on adding new terminal types to the screen editor. Basically, new code must be added for the cursor movement routines and the editor must be recompiled and installed so that it incorporates the new code. Also, the screen editor's 'usage' routine, as well as the Reference Manual entry, should be updated to include the mnemonic of the new terminal.

To add a new terminal type to the VTH package, an initialization file must be created in the "=vth=" directory, with the same name as the mnemonic that you have chosen for that terminal type. To find out what the format and contents of this file should be, please refer to the other files in that directory for examples.

* **Adding Local Tools and Library Routines**

When the Subsystem is used for program development, there is a tendency for an installation to collect a set of locally-useful tools. Here are a few suggestions on how to incorporate these local tools into the Subsystem environment.

Of course the primary repository for local commands is the directory 'lbin'. The Georgia Tech 'lbin' is supplied on the release tape as an example of a local command library, and because some of the commands contained therein may be of general use. To place a local tool in 'lbin', simply copy its object code into the directory and use the 'chat' command to make sure it is readable by all users, or if the 'lbin' is ACL protected then the command will automatically be readable.

Since the command search rule employed by the command interpreter may be changed by the user, any number of local command directories may be created. For example, a directory named 'games' might be created for the purpose of keeping employees out of the local arcade at lunchtime. A search rule including this directory might be

```
^int,^var,&,=lbin=/&,=bin=/&,//games/&
```

Any of the programs in 'games' may then be invoked without need for using their full pathnames.

Of course, local subprogram libraries may also be created at will. As with standard Primos, the only steps necessary to make such libraries accessible are to place them in 'lib' and make them readable by all users.

Adding Local Documentation

If local tools and libraries are added to the Subsystem (as outlined above), there will be a need for some means of documenting them. Sections three and four of the Reference Manual are provided for this purpose.

Section three of the Manual deals with local commands. To document such a command, one places a standard documentation file in "=doc=/man/s3" and uses "=doc=/build/rebuild" to place a formatted copy in "=doc=/fman/s3". The documentation may then be extracted by the 'help' command, or printed with the entire manual by executing "=doc=/print/man".

A standard documentation file for a command has several distinguishing characteristics. First, the documentation for a tool named "xxx" resides in a file named "xxx.d". Second, the structure of the file's contents is determined by a number of standard text formatter macro commands. Each macro begins a separate subject in a manual entry. They must appear in the following order: ".hd" (heading), ".ds" (description), ".es" (examples), ".fl" (files used), ".me" (messages issued), ".bu" (bugs and deficiencies), and ".sa" (see also). If a section is empty, it should be omitted entirely.

Once a documentation file has been entered, it must be formatted and the formatted copy placed in a separate directory. The shell program "=doc=/build/rebuild" has been provided for this purpose. An example:

```
=doc=/build/rebuild s3 xxx
```

This would format the file "=doc=/man/s3/xxx.d" and place the result in "=doc=/fman/s3/xxx.d", where it may be accessed by 'help' and 'usage'.

Section four of the Manual deals with local library subprograms. The documentation procedure for subprograms is similar to that for major tools; an unformatted copy of the documentation is placed in "=doc=/man/s4", and a formatted copy in "=doc=/fman/s4". Again, this makes the documentation available through 'help'.

Formatter macros for library routine documentation differ somewhat from those for command documentation. In order, they are: ".hd" (header), ".fs" (function of subprogram), ".im" (implementation sketch), ".am" (arguments modified by subprogram), ".ca" (other subprograms called by this subprogram), ".bu" (bugs or deficiencies), and ".sa" (see also). Again, if a section is empty, it should be omitted entirely.

The rebuild procedure for subprogram documentation is very similar to that for command documentation; simply use "s4" in place of "s3" in the "rebuild" command.

Operation of the 'Cron' Program

One of the new features of Version 9 of Software Tools is the 'cron' program, which allows the system administrator to arrange for the computer to automatically do tasks of a periodic nature. The manual entry for 'cron' (help cron) will give you the information you need in setting up 'cron'. As distributed, "=cronfile=", contains an example entry and a brief description of what cronfile entries should look like, and "=system=/cron.comi" contains an example startup file to initialize 'cron'. 'Cron' executes as an ordinary Software Tools user so it must have an entry in "=varsdir=" and "=userlist=" (see **Adding and Deleting Users** in this section). The 'cron' user must have all permission to the directory "=crondir=". As supplied, 'cron' expects to be run as the system administrator with an ACL protecting "=crondir=" ("=crondir=" is protected with SYSTEM having $all access and everyone else has "list" and "use" privileges).

The supplied Primos routine SPH should be used to start 'cron'. The following command should be entered at the console or placed in the Primos cold start command file ('c_prmo' or 'primos.comi'):

```
SPH SYSTEM>CRON.COMI -U <user-name> -P <project> -V 1 -G <groups>
```

where <user-name> is the name under which 'cron' should run, <project> is 'cron's login project, and <groups> is the list of file system groups with which the 'cron' user should be associated. For example, the following will startup 'cron' with all the default attributes and no groups:

```
SPH SYSTEM>CRON.COMI -U SYSTEM -P DEFAULT -V 1 -G
```

Operation of Communications Systems

The Subsystem provides three different means of passing information from user to user: the postal service, the grapevine, and the news service.

Postal Service. Most electronic communication between Subsystem users is accomplished through the mail system. The 'mail' command stores arbitrary messages in the directory "=mail=" (nominally "//extra/mail"), from where they may be retrieved by the addressee at a later time. All letters are postmarked with the time of mailing and the login name of the sender. Whenever a user enters the Subsystem via 'swt', he is informed if there is any undelivered mail addressed to him.

Grapevine. 'Mail' is not real-time; a letter sent is generally not received until the next time a user enters the Subsystem, or if the user has set his mail notification in the Shell, the next time the Shell notifies him. The "grapevine" managed by the 'to' command alleviates some of this problem. Messages sent from user to user via 'to' are stored in the directory "=gossip=" (nominally "//extra/gossip"), which is searched by the command interpreter before executing each terminal-level command. Thus, the delay between sending a message with 'to' and its receipt by the addressee is no longer than the longest time he spends executing a command. Since users typically spend a great deal of time text editing, the screen editor has also been given the ability to display a message sent by 'to'. See the "om" command in 'se' for details.

News Service. Occasionally an item of general interest or special importance must be delivered to the user community at large. The Subsystem news service provides a means of delivering these articles, while making an archival copy of them and placing their headlines in an index file for later reference.

Since the disk space required to store undelivered news articles may be prohibitively expensive, users who wish to receive news must "subscribe" to the service with the 'subscribe' command. This need only be done once in an account's lifetime. The list of subscribers may be found in the file "=news=/subscribers".

News articles are published with the 'publish' command. 'Publish' takes one file name argument. The named file is copied into the news boxes ("=news=/delivery/<user>") of all subscribers, an archival copy is made in "=news=/articles", and the first line is entered into the index file "=news=/index" for later reference.

News articles that were published incorrectly or are outdated may be removed with the 'retract' command. 'Retract' can remove one or more articles at one time by specifying the article numbers as arguments. A notice of retraction for each article removed is placed in the news boxes of all subscribers who have seen the retracted article; subscribers who have not seen a retracted article are not notified of the retraction. Since removal of outdated news articles is not of great importance, such articles may be retracted quietly by using the "-q" option.

When a subscriber enters the Subsystem, he is informed if there is any news he has not yet seen. He may then retrieve the article with the 'news' command. At any time, he may also use the 'news' command to review the index or any archived articles.

Modifying the Dictionary of English Words

The dictionary of words supplied with the Subsystem is still rather incomplete, and may require additions from time to time.

The template =new_words= is commented out in the system template file. If you make it an active template (by removing the comment symbol), the 'spell' program will write into =new_words= any words it finds which are not in the dictionary. (=new_words= is defined to be =aux=/spelling/new_words.) You may then wish to periodically clean up the file as follows:

```
cd =aux=/spelling
sort new_words | uniq >new_words
```

which will sort the file and remove duplicate entries. You can then go through the file with a dictionary, and remove words which are misspelled.

To add new words to the dictionary, the following procedure is recommended.

Obtain a list of words to be added, or use =new_words= as described above (or both). Obtain from each word as many derivative words as possible, by changing prefixes and suffixes, forming compounds, etc. Check each of these for correct spelling.

Attach to the directory "=aux=/spelling/new" and split the new words into the word files there according to the following scheme:

dictionary	ordinary English words
gazetteer	names and trademarks
abbreviations	abbreviations, acronyms
glossary	computer science terms

Words may appear in more than one file; for example, "assemble" may appear both in the dictionary and in the glossary.

When the new words have been split to their respective files, append these files onto the files of the same name in "=aux=/spelling." You may then empty out the files in "=aux=/spelling/new" by 'echo'ing into them. Go back to "=aux=/spelling", and execute the shell program 'build', which combines the files to form the file 'words', which is used by the spelling check program. You may also run the program 'info' in "=aux=/spelling" for more information.

TABLE OF CONTENTS

Software Tools Subsystem

Version 7.1 to Version 8 Conversion Guide

Terrell L. Countryman
Jeanette T. Myers
Peter N. Wan

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

April, 1982

Version 8 Conversion Guide

Introduction

Version 8 of the Software Tools Subsystem differs from Version 7.1 in a number of ways, most of which will not impact the average user. Most changes are extensions or internal performance improvements, and affect one or two commands rather than the entire Subsystem. This conversion guide is divided into three sections: **Global Changes** discusses the alterations that affect large portions of the user interface; **Status of V7.1 Commands** and **Status of V7.1 Subroutines** describe additions, deletions, and modifications made to individual commands and subroutines.

Global Changes

Terminal Type Handling

Various programs and library routines now support the tailoring of output for specific terminals. This entailed changes in the library, the macro definitions file, the Subsystem common blocks, 'swt', 'se', 'term', and 'term_type'. You will be affected by these changes if you have added terminal types to 'se' or if you have used the preliminary version of the Virtual Terminal Handler (VTH) library on the Version 7.1 release. Since a prompt for terminal type may now occur upon Subsystem entry, you may have to add terminal types to the "=ttypes=" file or educate your users about terminal types.

If you want to extend the terminal type knowledge of the Subsystem, you must add new terminal types and information concerning them to the file "=ttypes=", add a new initialization file in the directory "=vth=" for each new terminal type, and modify and recompile 'se' with new code to handle the new terminal.

Templates

Templates are no longer mapped to a single case. Unless you have some single-case terminals or regularly program in upper-case only, this change is unlikely to affect you.

If a template must contain imbedded equals signs, use two consecutive equals signs to pass one through the template expander; in an earlier version of the template processor, the "at sign" was used to "escape" the equals sign.

Speed

Version 8 of the Subsystem uses considerably less CPU time for I/O than Version 7.1. Unfortunately, if you are moving up to

Revision 18 of Primos, you may not notice much difference; Revision 18 can be that much slower than Revision 17.

Memory Segments

Prime has now used all available private memory segments. At Version 8 of the Subsystem, it has become necessary to increase the NUSEG parameter in the Primos configuration file to at least 42 (octal), to provide private segments for the Subsystem that do not conflict with standard Prime programs. This implies that programs using secret knowledge of the Subsystem's common blocks must be relinked. It also implies that you cannot bring up Version 8 without rebooting your system.

Listed below are the segments required for the Subsystem:

	Version 8	*Version 7.1*
SWT Shell and template area	2030	2030
SWT Screen Editor	2031	2031
SWT Library	2035	2035
SWT Common	4040	4036
SWT Stack	4041	4035

Process Ids to Three Digits

To accomodate the increase in the number of processes in Primos 18, process ids will be three digits instead of two. The "=termlist=" file has changed slightly in format to accomodate the increased id lengths and AMLC numbers.

Cldata Template

The location of "cldata", a Primos command interpreter data structure referenced by the Subsystem shell, is now a template. The Version 8 release has "cldata" defined to be segment 6002, word 6, which applies to you if you are running Primos 18.3 or above. If you are running Primos 18.2 or lower, there is a commented template in the "Configuration Options" section of "=template=" that you need to use.

Exception Handling

The new version of the shell now allows you to intercept exceptional conditions, such as pointer faults, arithmetic value errors, interrupts, etc. Quits (via control-p or "break") abort the current program and return to command level in the shell, rather than leaving the you stranded in Primos, as was the case in previous versions of the shell. If you have a shell variable named "_quit_action" (the value is not important), then when the quit occurs, the shell will prompt you as to whether to abort the

current program, continue, or call Primos. If Primos is called, the current program may be continued by typing START, or the Subsystem may be re-entered by typing REN.

DBG Support

There is now some support for invoking the symbolic debugger from the Subsystem. Please see the Reference Manual entry on 'dbg' for more information.

New Shell Control Variables

The variable "_kill_resp" can be used to set the character string that will be printed on your screen whenever you type a "kill" (the default is the string "\\"); "_prt_form" can be used to specify your usual printing form (the default is to use the default installation printer form); "_prt_dest" can be used to specify your favorite line printer (the default is to use the default installation printer); and, "_quit_action" can be used to take advantage of the new quit handling capabilities of the shell (by default, this variable is not declared and you are automatically returned to the command level of the shell, with no choice of error handling for quits).

Note that when a variable is set using either 'declare' or 'set', the value does not take effect until you exit the Subsystem and re-enter.

Deleted Macro Definition

One macro definition has been removed from the standard macros file. ESCCHAR is no longer defined; use ESCAPE instead.

Change in Value of EOS

After Version 8 of the Subsystem, the value of the end-of-string character (EOS) will be changed. The current value of EOS, as defined in the Subsystem definition file, is -2. It will be changed to the value 0 to maintain consistency with the way the C library handles the end-of-string. If you have any programs which depend on the value of EOS being of a certain magnitude (i.e., being negative), you should recode them to avoid depending on that assumption. This change will require the recompilation of all local Subsystem programs.

New Reference Manual Sections

Two new sections have been created to contain low level commands (Section 5) and low level subroutines (Section 6). You should not invoke these commands and routines under normal circumstances; they are usually support routines for user-callable

Subsystem commands and routines.

The following commands were moved from sections 1 and 3 to section 5:

bmerge	**bnames**	**bs**
bs1	**bugfm**	**bugn**
guess	**mkcl**	

The following routines were moved from sections 2 and 4 to section 6:

at$	**bponu$**	**c$end**
c$incr	**c$init**	**call$$**
chunk$	**cof$**	**cpfil$**
cpseg$	**dgetl$**	**dmark$**
dmpcm$	**dmpfd$**	**dopen$**
dputl$	**dread$**	**dsdbiu**
dseek$	**dwrit$**	**findf$**
finfo$	**first$**	**flush$**
gcdir$	**gcifu$**	**getfd$**
gfnam$	**icomn$**	**iofl$**
ioinit	**ldseg$**	**ldtmp$**
lopen$	**lutemp**	**mkdir$**
mkfd$	**mkpa$**	**mktr$**
reonu$	**rmfil$**	**rmseg$**
rtn$$	**sprot$**	**st$lu**
t$clup	**t$entr**	**t$exit**
t$time	**t$trac**	**tcook$**
tgetl$	**tmark$**	**tputl$**
tread$	**tscan$**	**tseek$**
ttyp$f	**ttyp$l**	**ttyp$q**
ttyp$r	**ttyp$v**	**twrit$**
upkfn$	**vt$alc**	**vt$clr**
vt$db	**vt$db1**	**vt$db2**
vt$db3	**vt$def**	**vt$del**
vt$dsw	**vt$err**	**vt$get**
vt$gsq	**vt$idf**	**vt$ier**
vt$ndf	**vt$out**	**vt$pos**
vt$put	**vt$rdf**	**zmem$**

New Subsystem Libraries

Vthlib has been totally rewritten, and is now supported as part of the Subsystem. It is much faster than the earlier version, and offers more features. In general, the rewritten routines are not compatible with those of the earlier release of VTH. Programs which used the earlier version of VTH will have to have the VTH calls recoded to use the new routine names (and new calling formats).

New Subsystem Templates

The templates "=newcmdnc0=" and "=newsystem=" specify where newly compiled Subsystem files that belong in "cmdnc0" and "system" are placed during a recompilation of the Subsystem. The "=ttypes=" file contains a list of terminals supported by your Subsystem and their characteristics. The "=cldata=" template, mentioned above, indicates where the Primos "cldata" data structure is located. The "=sysname=" template is used to indicate the Primenet node name, if the system is a network system.

Obsolete templates have been removed from the Version 8 release template file.

Status of Version 7.1 Commands

This section summarizes the user-visible changes that have been made to Subsystem commands for Version 8. It is divided into several subsections: obsolete commands, superseded commands, modified commands, enhanced commands, and unchanged commands. The final subsection is a summary of commands that are new for the Version 8 release.

Obsolete Commands

The commands in this subsection were part of the Version 7.1 Subsystem, but are not included in the Version 8 release. Most of them were used only by certain shell programs and have outlived their usefulness. In other cases, the commands were relics of past Subsystems, and either were no longer useful, or no longer worked.

No commands are obsoleted at Version 8.

Superseded Commands

The commands in this subsection are not part of the Version 8 Subsystem; their functionality has been subsumed by other commands. Each entry describes the command and options you can use to get the same results.

ar
: Has been completely rewritten. The old 'ar' is now 'old_ar'. All existing archives should be processed with 'old_ar', and then reprocessed with 'ar', since support of 'old_ar' will be dropped in a future release.

dumpls
: Use the new command 'dump' instead. The command line would be "dump ls".

dumpsv
Use the new command 'dump' instead. The command line would be "dump sv".

Modified Commands

The commands listed in this subsection have been modified for the Version 8 release and are no longer completely compatible with their Version 7.1 counterparts. Each entry gives a brief description of the changes, but before using any of these commands, please check the corresponding Reference Manual entry to be sure of the command's exact behavior.

dprint
The argument syntax has been changed slightly. The length option is now "-l <length>", rather than "-<length>".

'Dprint' can now handle the generation of multiple copies.

mon
Accepts four commands while running; three are used to reformat the screen and the fourth command redraws the screen.

Now uses VTH to do output, so it will work on terminals besides Beehives (any terminal supported by VTH).

pg
The control-c response causes 'pg' to ignore any remaining file names that were command line arguments and exit to the command interpreter.

Default prompt now shows the file being displayed as well as the page number.

The "-s" option has been added; it allows you to specify the screen size as an argument. The old syntax of "-<screen size>" was ambiguous if the <screen size> was 1, 2, or 3 (it was too close to the Subsystem convention of referencing the standard input/output ports with a "-<port number>").

rtime
By default, the output of the command being measured is diverted to /dev/tty; you can specify diversion to /dev/null if no output is desire.

se

Your terminal type can now be obtained by a call to the Subsystem. (This usually eliminates the need to know terminal type mnemonics, or at least the need to retype them as long as the Subsystem knows your terminal type --- if the Subsystem does not know your terminal type, it will prompt you for it when you invoke 'se').

Long input lines are now scrolled horizontally, allowing the cursor to remain visible at all times.

New options include "oh[<baud>]", to tell 'se' your baud rate; "olm[<column>]", to set the left margin of text to be displayed in the window (permitting handling of very wide files); and, "os[s | f | f77]", to set several programming language related parameters at once ("oss" for PMA, "osf" for FTN, "osf77" for F77).

Documented options include "ok", to indicate whether or not the current edit buffer has been saved; and, "om" to display a message (sent via 'to').

New commands include "e!", "w!" and "q!" which can be used to enter, write or quit without having 'se' tell you if you are about to destroy the contents of your edit buffer or the contents of an existing file. These replace the old forms "ea", "wa", and "qa", respectively.

New terminal types supported by 'se' are the hz1510, ts1, tvi, and z19.

'Se' now takes advantage of terminal hardware line insert/delete functions (if they are available for the given terminal) to speed up processing over slower transmission lines (i.e., dialups). The "-h" command line option and the "oh" 'se' command set/query the baud rate you are running at; 'se' decides how many nulls to put out, and whether to use the hardware line insert/delete functions or not, based on a combination of the baud rate and terminal type. If you want to add terminal types to 'se' for locally available terminals, the file "=src=/spc/se.u/how_to_add_terminal_types" explains how to do it (it also includes information on where to add the code necessary to handle the hardware line insert/delete functions). We at Georgia Tech would be interested to know about terminals that you add to both 'se' and the VTH package, so that we can include them in future releases of the Subsystem.

sh
The shell now handles breaks and control-p. If you declare the variable "_quit_action", you receive a prompt after a break and are allowed to continue the program, terminate the program and return to the Subsystem, or terminate the program and drop out to Primos (for debugging). If "_quit_action" is not declared, interrupted programs simply return control to the shell.

term
Support for the Subsystem terminal type management routines has been added.

New "-newline" and "-eof" options to specify newline and end-of-file characters; new "-vth" and "-se" options to specify whether or not the terminal type is supported by the Virtual Terminal Handler (VTH) and 'se'.

Removed "-tab" and "-enb".

term_type
Now uses the Subsystem terminal type management facilities. Options have been added to query the values of particular terminal attributes, as well.

x
'X' now calls the Primos command interpreter directly, via the Primos routine CP$. (This reduces execution time and the amount of garbage displayed on your terminal.)

'X' can execute Primos commands interactively, regaining control when you generate an end-of-file.

Enhanced Commands

Commands in this subsection have been functionally enhanced for the Version 8 release, but remain compatible with their Version 7.1 counterparts.

change
Accepts a string as an argument that is to be searched for a pattern.

declare
Will not modify a shell variable that has already been declared at the current level.

declared
No longer decides whether a shell variable is "declared" if an illegal lexical level offset is supplied.

define
Finally does what the documentation says it will do.

diff
A "-b" option has been added to allow direct binary comparison of files. (Note that files that compare "equal" under the usual text comparison may not be equal under the binary comparison, because of blank compression.)

e
Takes 'se' options as arguments and uses new terminal type handling.

f77c
New "-w" option to generate floating round instructions which improves the accuracy of single precision floating point calculations.

fc
New "-w" option to generate floating round instructions which improves the accuracy of single precision floating point calculations.

file
No longer returns "-1" when it could not perform a test on a file; it returns a zero, which is in accordance to the documentation. An error message is written to ERROUT for this case.

fmt
New ".eo" and ".oo" commands to specify different page offsets for even- and odd-numbered pages.

Documented the ".dv" (divert stream) command which, when used in conjunction with ".so", can produce an automatic table of contents.

guess
Requires an argument that is the command name to be used and optionally accepts a "tolerence" level.

guide
Version 8 Conversion Guide now available.

hd
New "-n" and "-u" options to display "normalized" (440 words/record) or "unnormalized" (1024 words per record) record counts, respectively.

help
Documented the "-u" option to print usage for command(s).

'Help' now uses 'page' for paging so you can use all the responses acceptable to 'page'.

ld

The templates "=cm_loc=" and "=tp_loc=" are checked to allow overriding the default segment numbers for Subsystem common blocks and template storage areas. This is useful if you are modifying the Subsystem and must run a production copy and a development copy side-by-side.

lf

New "-q" option to print nonowner password.

mail

Mail is saved in the file defined by the template "=mailfile=". The Subsystem default is "=varsdir=/.mail".

Checks for valid addressee name(s) before reading the letter to be sent.

mkclist

New "-s" argument to create the system defined command list ("=ubin=" is ignored).

moot

An "index" command has been added to summarize the entries that have been made in the current conference.

mt

'Mt' has been heavily modified to fix all known bugs.

New "-v" option to cause 'mt' to print the number of blocks read or written.

plgc

New "-w" option to generate floating round instructions which improves the accuracy of single precision floating point calculations.

print

New "-i" option to indent listing, "-j" option to cause 'print' to put a FORMFEED character at the end of a page instead of generating the number a blank lines required to get to the top of the next page, and a "-l" option to indicate the number of lines per page.

publish

A warning is now issued if you interrupt a 'publish' (interrupting a 'publish' has possible harmful side effects).

retract

A "-q" option has been added to allow retraction of a news article without printing a retraction notice.

rp

Declarations are now handled as a separate data stream.

You can now put statements in your Ratfor program that will not be touched by 'rp' and you can indicate if those statements are to be routed to the "declaration", "data", or "code" stream.

A "-g" option has been added to invoke an algorithm that tries to eliminate chains of GOTO statements. (When this was applied to the preprocessor itself, a 10% speedup resulted.)

New "-x" option and a accompanying translation table can be used for user definable character code translation.

A new "-y" option is available that causes 'rp' to not place "call init" and "call swt" statements in the Fortran code.

Several internal speed-up improvements have been made.

String tables now allow multiple slashes, causing marginal index entries to be duplicated. The maximum string table size has been increased.

The standard Ratfor macro definitions file now includes "SET_OF_GRAPHICS" and "SET_OF_SPECIAL_CHAR" for use in "when" clauses in Ratfor.

sema

Now handles named and negative (Primos system) semaphores.

stacc

A "null token" construct (epsilon) has been added to cause a match without scanning any input.

A "quick select" construct has been added to permit fast selection between a number of alternatives beginning with distinct terminal symbols.

General processing speed has been improved by eliminating the use of temporary files.

'Stacc' can now generate code in the C programming language, as well as Ratfor. The Reference Manual entry has been corrected so that it no longer indicates that SSPL, Pascal, and PLP are supported (they are not).

tail

Now correctly accepts a filename as an argument, even if it is the only argument (before, 'tail' would try to convert the lone file name as the number of lines

parameter, get an error in the conversion, and end up reading the default number of lines from standard input).

tlit

Running time has been improved drastically.

For compatibility with 'take', 'drop', and other command line utility functions, 'tlit' can also accept strings as arguments to be transliterated.

to

The header line format has been changed, to provide more information.

translang

Added the "nor" operator, which was inadvertently left out of the lexical analyzer.

who

Changed to call the new Primos GMETR$ to access the system data bases.

An "r" flag is appended onto the pid if the user is a remote user.

Unchanged Commands

This subsection lists the commands that have no user-visible changes made for Version 8.

alarm	**arg**	**args**
argsto	**as11**	**as6800**
as8080	**banner**	**basys**
batch	**block**	**bs**
bug	**bye**	**cal**
case	**cat**	**cd**
chat	**chown**	**clear**
clock	**cmp**	**cn**
col	**common**	**como**
copy	**copyout**	**cp**
crypt	**ctime**	**cto**
date	**day**	**declared**
del	**detab**	**dmach**
dnum	**drop**	**echo**
ek	**else**	**entab**
error	**esac**	**eval**
exit	**f77cl**	**fcl**
fdmp	**fi**	**field**
files	**find**	**fixp**
fmt	**focld**	**forget**
fos	**fsize**	**goto**
history	**hp**	**if**
imi	**include**	**index**
installation	**intel**	**iota**
join	**kill**	**kwic**
lam	**ld**	**length**
lex	**lib**	**line**
link	**lk**	**locate**
log	**login_name**	**lps**
macro	**memo**	**mkclist**
mkdir	**mklib**	**mktree**
mkusr	**mot**	**mv**
nargs	**news**	**opt6800**
opt8080	**os**	**out**
p4c	**p4cl**	**passwd**
pause	**pc**	**pcl**
ph	**phist**	**phone**
plgcl	**pmac**	**pmacl**
pr	**print**	**profile**
publish	**pwd**	**pword**
quote	**rcl**	**rf**
rfl	**rmusr**	**rnd**
rot	**rsa**	**save**
scroll	**sep**	**set**
sh	**show**	**shtrace**
size	**slice**	**sort**
source	**sp**	**speling**
sspl	**ssr**	**st_profile**
stats	**stop**	**subscribe**
substr	**symbols**	**systat**
take	**tc**	**tee**

template	**then**	**time**
ts	**uniq**	**unoct**
unrot	**us**	**usage**
vars	**when**	**whereis**
whois	**xref**	

New Commands

This subsection lists commands that are new for Version 8.

basename
Select various portions of a pathname.

bmerge
Merge object code files into one file for building a library.

bnames
Print entry point names in object files.

bs1
Identical to 'bs' except that it reduces search time, with the possible result of having a less intelligent guess.

bugfm
Format a bug report created with the 'bug' command.

bugn
Process the highest bug number.

cc
Compiles a C program with the Subsystem C compiler.

ccl
Compiles and loads a C program.

cdmlc
Compiles a Prime DBMS Cobol Data Manipulation Language program.

cdmlcl
Compiles and loads a DBMS Cobol Data Manipulation Language program.

cobc
Compiles a Cobol program.

cobcl
Compiles and loads a Cobol program.

csubc
Compiles a Primos DBMS Cobol subschema.

dbg
Interface to Primos source level debugger.

ddlc
Compiles a Prime DBMS schema.

des
An implementation of the National Bureau of Standards Data Encryption System.

dump
Debugging aid which dumps the shell's various internal data bases in a semi-readable format. This command supersedes the Version 7.1 commands 'dumpls' and 'dumpsv'.

fdmlc
Compiles a program written in the Prime DBMS Fortran Data Manipulation Language.

fdmlcl
Compiles and loads a Prime DBMS Fortran Data Manipulation Language program.

fsubc
Compiles a Primos DBMS Fortran subschema.

last
Allows you to look at the last few lines of a file.

Can also count the number of lines in a file very quickly.

mkcl
Make a command list in compressed binary format for use with the 'guess' command.

old_ar
Subsystem archiver from Version 7.1; included to allow you to retrieve your files and convert to the new archiver, 'ar'.

plpc
Compiles a PL/P program.

plpcl
Compiles and loads a PL/P program.

primos
Allows the use of the Primos command interpreter from the Subsystem. This command is somewhat different from the 'x' command, in that 'primos' causes a new level of the Primos command interpreter to be initiated.

radix
Convert numbers from one radix representation to another.

raid
Examine bug report submitted with 'bug' command (this is intended for the use of the Subsystem manager).

The bug reports can also be optionally be printed so that a hardcopy may be obtained.

rdcat
Relational database command which concatenates two identical relations.

rdextr
Relational database command which extracts relation data from a given relation.

rdjoin
Relational database command which joins two relations.

rdmake
Relational database command which constructs a relation from a data file.

rdprint
Relational database command to print a relation or a relation descriptor.

rdproj
Relational database command to project a relation.

rdsel
Relational database command to select tuples of a relation.

rdsort
Relational database command to sort a relation.

rduniq
Relational database command to remove duplicate tuples from a relation.

sol
Game of solitaire. A good demonstration of the new Virtual Terminal Handler (VTH) package.

spell
Faster than 'speling'.

Has a "verbose" output format to aid in locating misspelled words.

Is more intelligent about not reporting formatter commands as misspelled words.

tip
Check if terminal input is pending.

vpsd
Interface to invoke the Primos V-mode Symbolic Debugger on Subsystem programs.

Status of Version 7.1 Subroutines

This section summarizes the user-visible changes to the Subsystem library routines. It is divided into several subsections: obsolete routines, superseded routines, modified routines, enhanced routines, unchanged routines and new routines.

Obsolete Routines

The routines listed here were only used by other library routines. Since their services are no longer required, they have been deleted.

cmdf$$
Obsoleted because of a smarter shell.

rtr6800
The SSPL run-time support library for the M6800 microprocessor has been removed.

vt$bc
Obsoleted by the new VTH routines.

vt$cc
Obsoleted by the new VTH routines.

vt$ld
Obsoleted by the new VTH routines.

vt$ll
Obsoleted by the new VTH routines.

vt$mv
Obsoleted by the new VTH routines.

vt$pk
Obsoleted by the new VTH routines.

vt$rc
Obsoleted by the new VTH routines.

vt$upk
Obsoleted by the new VTH routines.

vtceol
Obsoleted by the new VTH routines.

vtceos
Obsoleted by the new VTH routines.

vtenc
Obsoleted by the new VTH routines.

vtinl
Obsoleted by the new VTH routines.

vtins
Obsoleted by the new VTH routines.

vtmvdn
Obsoleted by the new VTH routines.

vtmvlf
Obsoleted by the new VTH routines.

vtmvrt
Obsoleted by the new VTH routines.

vtmvup
Obsoleted by the new VTH routines.

vtpos
Obsoleted by the new VTH routines.

Superseded Routines

The following routines have been subsumed by other more powerful routines. Each entry names the Version 8 routine that performs the same function.

inloc$
Use 'decode'.

itoc0
Use 'gitoc' or 'encode'.

itoc8
Use 'gitoc' or 'encode'.

prot$
Use 'sprot$'. The name was changed to avoid a conflict with the Primos routine of the same name.

Modified Routines

The routines listed in this subsection have been modified so that they are no longer compatible with their Version 7.1 counterparts. Although each entry briefly describes the changes

that have been made, you should examine the corresponding Reference Manual entries to determine the exact behavior of the routines.

cof$
Requires a "state" argument.

enter
'Enter' is now a function that returns a pointer to the dynamic storage area containing text of next symbol.

expand
If a template must contain uninterpreted "="s, do not precede it by a "@" but by another "=".

iofl$
Requires a "state" argument.

sys$$
New argument to specify file unit from which the Primos command takes its input.

tscan$
The 'path' argument is changed by this routine, but was not documented to say so. The documentation has been changed.

vt$db
Has been rewritten for new VTH library.

vt$del
Has been rewritten for new VTH library.

vt$out
Has been rewritten for new VTH library.

vtclr
Has been rewritten for new VTH library.

vtinit
Has been rewritten for new VTH library.

vtputl
Has been rewritten for new VTH library.

vtterm
Has been rewritten for new VTH library.

vtupd
Has been rewritten for new VTH library.

Enhanced Routines

The routines listed in this subsection have additional functionality in the Version 8 release, but remain compatible with

their Version 7.1 counterparts.

call$$
Accepts an optional argument for the creation of an on-unit.

date
There are now system defines for the request keys (so that actual numbers for the type of request need not be supplied).

New values returned are minutes, seconds and milliseconds past midnight.

dopen$
Now takes an argument to determine the number of retries on encountering a "file in use" situation.

getto
MFD passwords are now consistently assumed to be "XXXXXX". Mixed-case passwords have caused several problems; the real source of the difficulty is a change Prime made to TA$ that renders it incompatible with earlier revisions of Primos.

lopen$
Will put in the values for the user's shell variables "_prt_form" and "_prt_dest", if available, in the spooler entry.

open
Now takes a fourth argument to determine the number of retries on encountering a "file in use" situation.

Unchanged Routines

No user-visible changes have been made to the routines listed in this subsection.

addset	**amatch**	**atoc**
c$end	**c$incr**	**cant**
catsub	**chkarg**	**chkinp**
close	**cpfil$**	**cpseg$**
create	**ctoa**	**ctoc**
ctod	**ctoi**	**ctol**
ctomn	**ctop**	**ctor**
ctov	**decode**	**delarg**
delete	**dgetl$**	**dmark$**
dodash	**dputl$**	**dread$**
dsdbiu	**dsdump**	**dseek$**
dsfree	**dsget**	**dsinit**
dtoc	**dwrit$**	**edit**
encode	**enter**	**equal**
error	**esc**	**exec**
execn	**fcopy**	**filcpy**
filset	**filtst**	**findf$**
finfo$	**flush$**	**follow**
gcd	**gcdir$**	**gctoi**
gctol	**getarg**	**getccl**
getch	**getkwd**	**getlin**
getto	**getvdn**	**gfnarg**
gitoc	**gklarg**	**gltoc**
gtemp	**gvlarg**	**icomn$**
index	**init**	**input**
invmod	**ioinit**	**isatty**
itoc	**jdate**	**ldseg$**
ldtmp$	**length**	**locate**
lookup	**lsallo**	**lscmpk**
lscomp	**lscopy**	**lscut**
lsdel	**lsdrop**	**lsdump**
lsextr	**lsfree**	**lsgetc**
lsgetf	**lsinit**	**lsins**
lsjoin	**lslen**	**lsmake**
lspos	**lsputc**	**lsputf**
lssubs	**lstake**	**ltoc**
lutemp	**makpat**	**maksub**
mapdn	**mapfd**	**mapstr**
mapsu	**mapup**	**markf**
match	**mkdir$**	**mkfd$**
mkpa$	**mktabl**	**mktemp**
mktr$	**mntoc**	**move$**
omatch	**open**	**page**
parsdt	**parstm**	**patsiz**
prime	**print**	**ptoc**
putch	**putdec**	**putlin**
putlit	**pwrmod**	**readf**
remark	**remove**	**reonu$**
rewind	**rmfil$**	**rmseg$**
rmtabl	**rmtemp**	**rtn$$**
rtoc	**scopy**	**sctabl**

sdrop	**seekf**	**set_copy**
set_create	**set_delete**	**set_element**
set_equal	**set_init**	**set_insert**
set_intersect	**set_remove**	**set_subset**
set_subtract	**set_union**	**seterr**
st$lu	**stake**	**stclos**
strbsr	**strcmp**	**strim**
strlsr	**substr**	**swt**
t$clup	**t$entr**	**t$exit**
t$time	**t$trac**	**tgetl$**
tmark$	**tputl$**	**tquit$**
tread$	**trunc**	**tseek$**
twrit$	**type**	**upkfn$**
vfyusr	**vtoc**	**wind**
wkday	**writef**	**zmem$**

New Routines

The routines listed in this section are new for the Version 8 release.

at$
Subsystem interlude to Primos ATCH$$.

bponu$
On-unit handler for "BAD_PASSWORD$" condition.

c$init
Initializes a Ratfor program in preparation for a statement count run.

chkstr
Check a string for printable characters.

dmpcm$
Dump the contents of Subsystem common blocks in a printable format.

dmpfd$
Dump information about a file descriptor.

file$p
When called from a Pascal program, allows the program to use the I/O redirection and piping features of the Subsystem.

first$
This routine sees if it has been called before; it is used by the Subsystem for initialization purposes.

gcifu$
Get the file unit which is providing command input to the shell.

geta$f
Allows Fortran programs access to the arguments from the Subsystem command line.

geta$p
Allows Pascal programs to access the arguments from the Subsystem command line.

geta$plg
Allows PL/I (subset G) programs to access the arguments from the Subsystem command line.

getfd$
Look for an empty file descriptor.

getwrd
Retrieve the next word from a buffer.

gfnam$
Get pathname of an open file.

gtattr
Returns a user's terminal attributes.

gttype
Returns the user's terminal type name.

init$f
Allows the Fortran programmer to take advantage of Subsystem I/O (especially the standard input and output ports).

init$p
Allows the Pascal programmer to take advantage of Subsystem I/O (especially the standard input and output ports).

init$plg
Allows the PL/I (subset G) programmer to take advantage of Subsystem I/O (especially the standard input and output ports).

isadsk
Test to see if a file is a disk file.

sprot$
Set the protection attributes for a file. This routine used to be named 'prot$', but had to be renamed because of a name conflict with a Primos routine.

tcook$
Read a line from the terminal and handle operations of processing escape sequences, case and character set mapping, line kills, etc. ("cooking" the line).

ttyp$f
Obtain the user's terminal type from the "=termlist=" file, if available.

ttyp$l
List the available terminal types (as defined in the "=ttypes=" file).

ttyp$q
Query for the terminal type from the user.

ttyp$r
Return the user's terminal type from the Subsystem common area, if available.

ttyp$v
Set the terminal's attributes in the Subsystem common areas.

vt$alc
Allocate another VTH definition table for the keyboard macros.

vt$clr
Send the clear screen sequence.

vt$db1
VTH debugging routine which prints mnemonics for the unprintable characters to be output.

vt$db2
VTH debugging routine to dump the terminal input tables.

vt$db3
VTH debugging routine to dump the macro definition table.

vt$def
Allows the user to define a keyboard macro.

vt$dsw
Perform a garbage collection on the VTH definition tables.

vt$err
Print a VTH error message.

vt$get
VTH input routine.

vt$gsq
VTH input routine to receive a delimited sequence of characters.

vt$idf
VTH input processor which invokes user-defined keyboard macros.

vt$ier
Report an error in a VTH initialization file.

vt$ndf
Remove a VTH macro definition.

vt$pos
VTH positioning routine which moves the terminal cursor by means of absolute positioning sequences.

vt$put
Copy a string into a VTH screen buffer.

vt$rdf
Remove a VTH keyboard macro from the definition table.

vtenb
Enable input on a particular screen line.

vtgetl
Retrieve a line from the VTH screen buffer.

vtinfo
Return information contained in the VTH common block.

vtmove
Position the cursor to a given row and column.

vtmsg
Display a message in the VTH status line.

vtopt
Set optional parameters for the VTH screen.

vtpad
Pad the rest of a field with blanks.

vtprt
Output formatted information to the screen buffers.

vtread
Read characters from the terminal into the screen buffers.

vtstop
Reset a terminal's attributes before terminating a program.

TABLE OF CONTENTS

Software Tools Subsystem

Version 8 to Version 8.1 Conversion Guide

Terrell L. Countryman
Peter N. Wan

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

March, 1983

Introduction

Version 8.1 of the Subsystem represents a big change over Version 8, as well as being the last release of the Subsystem targeted for a Primos 18 system. The next release, Version 9, will be targeted for the new operating system, Primos 19. Although this release will run under Primos 19 (if you have gotten it already), you will not have access to some of the newer features, such as Access Control Lists (ACLs) and disk quotas. Estimates for the release date of Version 9 are for around the end of the third quarter of 1983 or at the beginning of the last quarter; the reason for this delay is that as of the writing of this paragraph, we have not yet received our release copy of Primos 19.1. We ask your indulgence in this matter; we are endeavoring to obtain a copy as soon as possible.

This conversion guide is divided into three sections: **Global Changes** discusses the alterations that affect large portions of the user interface; **Status of V8 Commands** and **Status of V8 Subroutines** describe additions, deletions, and modifications made to individual commands and subroutines.

Global Changes

Change in Value of EOS

As described in the *V7.1 to V8 Conversion Guide*, the value of EOS (end of string) has changed from the value of -2 to 0. This change should not affect the operation of your programs, unless they make (unwise) assumptions as to the value or magnitude of this constant. The purpose of this change is to better support the C language (available as a separate package to Subsystem customers) and to slightly improve run-time performance of the Subsystem in general (it is faster to compare against 0 than -2).

Although a change of this magnitude normally requires the recompilation of all code (yours and ours), we have come up with a scheme whereby we build *two* Subsystem libraries: one handles EOS being -2, and the other one handles EOS being 0. The library that will be used for a particular object program is determined by whether the program calls 'init' or not -- programs which contain "call init" are assumed to be "old" and get an EOS value of -2. We can get by with this because 'init' hasn't been needed for years; not calling it has caused no ill effects for several releases, although its call was automatically included by 'rp'. This means as long as existing object programs behave (the only ones in doubt are non-Ratfor programs) by calling 'init', use the shared library, and don't muddle with the Subsystem common blocks, they will work perfectly under Version 8.1. Of course, Ratfor programs compiled under Version 8.1 will no longer call 'init' and will receive EOS as 0. (It is still possible to

recompile EOS=-2 programs under Version 8.1, but it will not be as convenient.) Locally-written routines (that do not access Subsystem common blocks) can be incorporated into both versions of the library automatically by putting them in the proper source directory and rebuilding the libraries.

This horrendous kluge will not be in effect for more than a couple of releases; good taste prevents us from allowing such an abomination to live any longer than necessary. We are doing it in the first place only to allow users at both your and our sites time to gradually rebuild programs (and because there were threats against our personal safety if we forced recompilation again). We do expect you to recompile all your local programs in the months following the installation of Version 8.1.

In recompiling your code, you should look for several things which could cause problems at execution time with the new library. First, make sure that your code does not depend on the value or magnitude of EOS, except to note that its value is different from the characters returned by 'getlin'. Next, if any of your main programs are introduced by the 'subroutine' keyword, you should recompile them immediately, since they are definitely not going to work with this version of the Subsystem. Third, make sure that none of your code (Ratfor or otherwise) contains an explicit call to the 'init' routine. This routine is no longer needed, and will cause the wrong value of EOS to be used while the compatible library is in use. Finally, if you use the unshared version of the Subsystem library ("nvswtlb") in the loads of your programs, they must be recompiled also. Except for the exceptions noted above, you may recompile your programs at your leisure; but be sure to do it soon, since the compatibility library will disappear eventually.

However, if you must recompile a program which has not been purged of its EOS value dependencies (and therefore must run with the value of EOS used at Version 8), you can do so by first making sure that the program (we are assuming Ratfor here) *calls 'init'*. Then, compile and load it via the following:

```
rp =src=/lib/swt/v8def.r.i <program>.r _
     -x =src=/lib/swt/v8rptab -o <program>.f
fc <program>.f
ld <program>.b -l v8vswtlb -o <program>
```

In the 'rp' call, the "v8def.r.i" file changes EOS references in your source and the file "v8rptab" changes the strings generated by 'rp' to have the correct terminator. The library call to the "v8vswtlb" library in the 'ld' line will cause the compatability library to be loaded, which does expect EOS to have the value used in Version 8.

Macro Definition Changes

The Subsystem definition files have been changed to clean up some old definitions and add some new ones. The old values INH

and ENB, which are used with the Primos 'break$' routine, have been changed to DISABLE and ENABLE, respectively. The values of PRIMOS_KEYS and PRIMOS_ERRD have been changed to contain the current names of those respective files. The names MAXUSERNAME and MAXPACKEDUSERNAME have been added to help interface with the 'date' routine; these values should also be used when dealing with login names (warning: Primos 19 will allow much longer names, so using these constants will ease your transition to the new operating system for programs which process login names).

New Subsystem Libraries

There have been many additions to the Subsystem libraries for this release. Two new libraries, "v8vswtlb" and "nv8vswtlb", have been added to provide compatability for programs which were compiled with Version 8 and must have EOS at the old value. There is a new library, "shortlb", which contains short-callable routines to provide Ratfor/Fortran programmers with operations that before this time were available only to assembly language programs. The Subsystem math library, "vswtml", contains new routines which provide double precision functionality. Finally, the support library for the Portable Pascal compiler has been renamed to "p4clib", to lessen confusion with the Prime Pascal library.

Deleted Subsystem Libraries

The pattern-matching library, "vpatlb", has been merged with the standard Subsystem library, and is therefore no longer needed. All programs that used to be loaded with this library can be loaded with the standard Subsystem library (automatically included by 'ld'). The old version of the Portable Pascal compiler library, "pasclib", has been removed (as noted above); Subsystem managers should make sure that this library is removed from =lib= to avoid having users access an older copy of the routines formerly in this library.

New Subsystem Template

The Subsystem template file has been enhanced by the addition of the template "=phonelist=". The 'phone' program was changed to use this template so that the user may set a private value for this template and use personal phone number lists.

Command Interpreter Enhancements

Terminal configuration (suppressed output and duplex) are restored properly after a command aborts and between execution of commands on a command line.

There are two new variables, "_eof" and "_newline", which have been documented.

There is now documentation in the *User's Guide to the Software Tools Command Interpreter* about restrictions that the Subsystem administrator can impose on Subsystem users in terms of which commands may be executed.

Update to SWTSEG

The Subsystem segmented loader has been updated to Primos version 18.3. This will solve most problems with loading the output of the current compilers; the temporary solution for program loading as described in the newsletter is no longer needed.

Status of Version 8 Commands

This section summarizes the user-visible changes that have been made to Subsystem commands for Version 8.1. It is divided into several subsections: obsolete commands, superseded commands, modified commands, enhanced commands, and unchanged commands. The final subsection is a summary of commands that are new for the Version 8.1 release.

Obsolete Commands

The commands in this subsection were part of the Version 8 Subsystem, but are not included in the Version 8.1 release. Most of them were used only by certain shell programs and have outlived their usefulness. In other cases, the commands were relics of past Subsystems, and either were no longer useful, or no longer worked.

lex
: The lexical analyzer for the SSPL compiler has been removed because support of the compiler no longer exists.

opt6800
: The Motorola 6800 code generator for the SSPL compiler has been removed because there is no longer any support for the compiler.

opt8080
: The Intel 8080 code generator for the SSPL compiler has been removed because there is no longer any support for the compiler.

sspl
: Support for the Small Systems Programming Language compiler (SSPL) has been removed from the Subsystem, because it enjoyed very limited use.

Superseded Commands

The commands in this subsection are not part of the Version 8.1 Subsystem; their functionality has been subsumed by other commands. Each entry describes the command and options you can use to get the same results.

No commands are superseded at Version 8.1.

Modified Commands

The commands listed in this subsection have been modified for the Version 8.1 release and are no longer completely compatible with their Version 8 counterparts. Each entry gives a brief description of the changes, but before using any of these commands, please check the corresponding Reference Manual entry to be sure of the command's exact behavior.

No commands are modified at Version 8.1.

Enhanced Commands

Commands in this subsection have been functionally enhanced for the Version 8.1 release, but remain compatible with their Version 8 counterparts.

bmerge
Updated to handle new object code format.

bnames
Updated to handle new object code format.

copyout
Updated to use new spooler library.

define
Enhanced to allow dollar signs in identifiers (to be compatible with 'rp').

dmach
Installed in the correct location (it is supposed to be in "=lbin=").

f77c
Now handles the "-u" option to list undefined variables and routines (its default behavior), and allows new levels of optimization.

fc
Added "-k" option to list compilation statistics.

fsize
Gives the number of records in a file system object as the default, and has "-w" option to list sizes in words

(like 'lf').

hd
Gives record size for unnormalized records, searches all possible disks instead of stopping at the first one that it could not size, and has new verbose option "-v".

include
Continues to process input despite errors in opening included files, and handles more deeply nested include calls.

ld
Added "-b" option to handle the C language library, added "-f" option to provide full map, and updated "-u" option to issue "ma 6" instead of "ma 3" to increase load speed.

lps
Updated to use the newer spool library, accepts more than one disk pack specification to indicate spool directories to be searched, prefixes the currently printing spooler entry with an asterisk, modified the "-q" option to provide more verbose information, queue entry lists are now prefixed by a label indicating on which disk partition the queue was found, and the "-c" option no longer allows cancellation of print files on remote spool queues.

macro
Now accepts the "-e" option to allow the escaping of characters.

mon
Accepts new commands "?", "x", and "q".

os
Includes speed enhancements and accepts "-x" option to reverse the order in which it outputs the overstrikes (needed for Printronix printers).

pc
Extended the "-f" option to handle the new map options, meaning of the "-q" option changed so that the meaning of the levels is now reversed.

pg
Calls the extended 'page' subroutine to allow search by pattern, etc. See the Reference Manual entry for both the 'pg' command and the 'page' subroutine for more information.

phone
Changed to use the new template "=phonelist=", to allow private phone lists to be used.

plgc
Extended the "-f" option in the same manner as 'pc', added the "-p" option to control short-call routine generation, and added the "-s" option to control copying of constant subroutine parameters.

plpc
Added "-q" option to control listing of warning messages.

pr
Now kicks the spooler after the file has been spooled.

radix
Prints on standard output instead of the error output, as stated in the Reference Manual entry.

rp
No longer generates calls to the 'init' routine and transliterates single character constants correctly. New 'b' option to prevent mapping of long identifiers or identifiers which contain upper case letters, and new 'h' option to force the output of Hollerith constants rather than quoted string constants.

se
Handles more terminal types, handles more and longer lines, and fixed errors caused by an uninitialized variable. Documentation has been added for "oss" and "osf" options, "&" pattern element, ";" and "#" line number elements, and extended message command.

sp
Now kicks the spooler after the file has been spooled.

who
Added the "-q" option to suppress printing of header lines.

Unchanged Commands

This subsection lists the commands that have no user-visible changes made for Version 8.1.

alarm	**ar**	**arg**	**args**
argsto	**as11**	**as6800**	**as8080**
banner	**basename**	**basys**	**batch**
block	**bs**	**bs1**	**bug**
bugfm	**bugn**	**bye**	**cal**
case	**cat**	**cd**	**cdmlc**
cdmlcl	**change**	**chat**	**chown**
clear	**clock**	**cmp**	**cn**
cobc	**cobcl**	**col**	**common**
como	**copy**	**cp**	**crypt**
csubc	**ctime**	**cto**	**date**
day	**dbg**	**ddlc**	**declare**
declared	**del**	**des**	**detab**
diff	**dmach**	**dnum**	**dprint**
drop	**dump**	**e**	**echo**
ed	**ek**	**else**	**entab**
error	**esac**	**eval**	**exit**
f77cl	**fcl**	**fdmlc**	**fdmlcl**
fdmp	**fi**	**field**	**file**
files	**find**	**fixp**	**fmt**
focld	**forget**	**fos**	**fsubc**
goto	**guess**	**guide**	**help**
history	**hp**	**if**	**imi**
index	**installation**	**intel**	**iota**
join	**kill**	**kwic**	**lam**
last	**length**	**lf**	**lib**
line	**link**	**lk**	**locate**
log	**login_name**	**mail**	**memo**
mkcl	**mkclist**	**mkdir**	**mklib**
mktree	**mkusr**	**moot**	**mot**
mt	**mv**	**nargs**	**news**
old_ar	**out**	**p4c**	**p4cl**
passwd	**pause**	**pcl**	**ph**
phist	**plgcl**	**plpcl**	**pmac**
pmacl	**primos**	**print**	**profile**
publish	**pwd**	**pword**	**quote**
raid	**rcl**	**rdcat**	**rdextr**
rdjoin	**rdmake**	**rdprint**	**rdproj**
rdsel	**rdsort**	**rduniq**	**retract**
rf	**rfl**	**rmusr**	**rnd**
rot	**rsa**	**rtime**	**save**
scroll	**sema**	**sep**	**set**
sh	**show**	**shtrace**	**size**
slice	**sol**	**sort**	**source**
speling	**spell**	**ssr**	**st_profile**
stacc	**stats**	**stop**	**subscribe**
substr	**symbols**	**systat**	**tail**
take	**tc**	**tee**	**template**
term			

New Commands

This subsection lists commands that are new for Version 8.1.

brefs
Provide a list of caller-callee pairs for an object file.

cc
Compiles a C program with the Subsystem C compiler. **This program is only available to customers who have also licensed the C language compiler package.**

ccl
Compiles and loads a C program. **This program is only available to customers who have also licensed the C language compiler package.**

isph
Allows shell files to determine whether they are running in a phantom environment. This is useful for scripts which might attempt to write to the terminal unless their output is redirected.

lorder
Provides the ordering of a library necessary for a one-pass load.

splc
Compiles an SPL program.

splcl
Compiles and loads an SPL program.

sprint
Filters formatter output for a NEC Spinwriter and provides similar functionality to 'dprint.'

tsort
Performs topological sort of caller-callee pairs for ordering library routines.

ucc
Compiles and loads a C program, ala Unix(tm). **This program is only available to customers who have also licensed the C language compiler package.**

vcg
Generates V-mode object code for Prime 50-Series computers. Allows the ambitious installation to write "front-ends" for local implementations of compilers. **This program is only available to customers who have also licensed the C language compiler package.**

vcgdump
Displays the input files for 'vcg' in a semi-readable format; useful for debugging compiler "front-ends." **This program is only available to customers who have also licensed the C language compiler package.**

yesno
Provides selective input filtering.

Status of Version 8 Subroutines

This section summarizes the user-visible changes to the Subsystem library routines. It is divided into several subsections: obsolete routines, superseded routines, modified routines, enhanced routines, unchanged routines and new routines.

Obsolete Routines

The routines listed here were only used by other library routines. Since their services are no longer required, they have been deleted.

No routines were obsoleted at Version 8.1.

Superseded Routines

The following routines have been subsumed by other more powerful routines. Each entry names the Version 8.1 routine that performs the same function.

at$
Use 'at$swt'. Its name was changed to avoid naming conflicts with a Primos 19 routine.

Modified Routines

The routines listed in this subsection have been modified so that they are no longer compatible with their Version 8 counterparts. Although each entry briefly describes the changes that have been made, you should examine the corresponding Reference Manual entries to determine the exact behavior of the routines.

file$p
Updated for Prime Pascal version 18.3/18.4 release, which is incompatible with previous releases.

init
Modified to allow use of the compatibility library (funeral notices will soon appear in a Subsystem new-

sletter near you).

init$p
Updated for Prime Pascal version 18.3/18.4 release, which is incompatible with previous releases.

Enhanced Routines

The routines listed in this subsection have additional functionality in the Version 8.1 release, but remain compatible with their Version 8 counterparts.

addset
Cleaned up code and is now part of the standard Subsystem library.

amatch
Cleaned up code and is now part of the standard Subsystem library.

call$$
Modified to handle the "output suppressed" bits.

cant
Changed its error message to the one specified in the Reference Manual.

catsub
Cleaned up code and is now part of the standard Subsystem library.

dmpcm$
Also prints the current EOS value, which is currently stored in the common block.

dodash
Cleaned up code and is now part of the standard Subsystem library.

esc
Cleaned up code and is now part of the standard Subsystem library.

filset
Cleaned up code and is now part of the standard Subsystem library.

getccl
Cleaned up code and is now part of the standard Subsystem library.

locate
Cleaned up code and is now part of the standard Subsystem library.

makpat
Cleaned up code and is now part of the standard Subsystem library.

maksub
Cleaned up code and is now part of the standard Subsystem library.

match
Cleaned up code and is now part of the standard Subsystem library.

omatch
Cleaned up code and is now part of the standard Subsystem library.

page
Handles the page count correctly, allows pattern searching, and has been modified to provide better performance.

patsiz
Cleaned up code and is now part of the standard Subsystem library.

stclos
Cleaned up code and is now part of the standard Subsystem library.

vfyusr
Checks the length of its argument, and immediately returns if the argument string is too long to be a legal login name.

vt$def
Uses Primos C1IN instead of T1IN for faster response.

vt$get
Uses Primos C1IN instead of T1IN for faster response.

vt$gsq
Uses Primos C1IN instead of T1IN for faster response.

vt$ndf
Uses Primos C1IN instead of T1IN for faster response.

vt$pos
Supports positioning for Hewlett-Packard terminals.

Unchanged Routines

No user-visible changes have been made to the routines listed in this subsection.

atoc	**bponu$**	**c$end**	**c$incr**
c$init	**chkarg**	**chkinp**	**chkstr**
chunk$	**close**	**cof$**	**cpfil$**
cpseg$	**create**	**ctoa**	**ctoc**
ctod	**ctoi**	**ctol**	**ctomn**
ctop	**ctor**	**ctov**	**date**
decode	**delarg**	**delete**	**dgetl$**
dmark$	**dmpfd$**	**dopen$**	**dputl$**
dread$	**dsdbiu**	**dsdump**	**dseek$**
dsfree	**dsget**	**dsinit**	**dtoc**
dwrit$	**edit**	**encode**	**enter**
equal	**error**	**exec**	**execn**
expand	**fcopy**	**filcpy**	**filtst**
findf$	**finfo$**	**first$**	**flush$**
follow	**gcd**	**gcdir$**	**gcifu$**
gctoi	**gctol**	**geta$f**	**geta$p**
geta$plg	**getarg**	**getch**	**getfd$**
getkwd	**getlin**	**getto**	**getvdn**
getwrd	**gfnam$**	**gfnarg**	**gitoc**
gklarg	**gltoc**	**gtattr**	**gtemp**
gttype	**gvlarg**	**icomn$**	**index**
init$f	**init$plg**	**input**	**invmod**
iofl$	**ioinit**	**isadsk**	**isatty**
itoc	**jdate**	**ldseg$**	**ldtmp$**
length	**lookup**	**lopen$**	**lsallo**
lscmpk	**lscomp**	**lscopy**	**lscut**
lsdel	**lsdrop**	**lsdump**	**lsextr**
lsfree	**lsgetc**	**lsgetf**	**lsinit**
lsins	**lsjoin**	**lslen**	**lsmake**
lspos	**lsputc**	**lsputf**	**lssubs**
lstake	**ltoc**	**lutemp**	**mapdn**
mapfd	**mapstr**	**mapsu**	**mapup**
markf	**mkdir$**	**mkfd$**	**mkpa$**
mktabl	**mktemp**	**mktr$**	**mntoc**
move$	**open**	**parscl**	**parsdt**
parstm	**prime**	**print**	**ptoc**
putch	**putdec**	**putlin**	**putlit**
pwrmod	**readf**	**remark**	**remove**
reonu$	**rewind**	**rmfil$**	**rmseg$**
rmtabl	**rmtemp**	**rtn$$**	**rtoc**
scopy	**sctabl**	**sdrop**	**seekf**
set_copy	**set_create**	**set_delete**	**set_element**
set_equal	**set_init**	**set_insert**	**set_intersect**
set_remove	**set_subset**	**set_subtract**	**set_union**
seterr	**sprot$**	**st$lu**	**stake**
strbsr	**strcmp**	**strim**	**strlsr**
substr	**swt**	**sys$$**	**t$clup**
t$entr	**t$exit**	**t$time**	**t$trac**
tcook$	**tgetl$**	**tmark$**	**tputl$**
tquit$	**tread$**	**trunc**	**tscan$**
tseek$	**ttyp$f**	**ttyp$l**	**ttyp$q**

ttyp$r	**ttyp$v**	**twrit$**	**type**
upkfn$	**vt$alc**	**vt$clr**	**vt$db**
vt$db1	**vt$db2**	**vt$db3**	**vt$del**
vt$dsw	**vt$err**	**vt$idf**	**vt$ier**
vt$out	**vt$put**	**vt$rdf**	**vtclr**
vtenb	**vtgetl**	**vtinfo**	**vtinit**
vtmove	**vtmsg**	**vtoc**	**vtopt**
vtpad	**vtprt**	**vtputl**	**vtread**
vtstop	**vtterm**	**vtupd**	**wind**
wkday	**writef**	**zmem$**	

New Routines

The routines listed in this section are new for the Version 8.1 release.

abq$xs
Adds an entry to the bottom of a queue.

at$swt
Provides interlude to Primos ATCH$$ (formerly 'at$').

atq$xs
Adds an entry to the top of a queue.

dacos
Returns the double precision inverse cosine value of its argument.

dasin
Returns the double precision inverse sine value of its argument.

dbexp
Returns the double precision exponentiation of its argument to the base of the natural logarithms.

dbsqrt
Returns the double precision square root of its argument.

dflot
Returns the double precision float of its long integer argument.

drand
Returns a double precision random number.

get$xs
Returns a character from an array by using efficient indexing and byte-swapping operations.

gky$xs
Returns the current CPU keys.

isnull
Test to see if a given file is the null device.

mkq$xs
Initializes a hardware-defined queue.

pek$xs
Returns the value in a given memory location (performs a peek operation).

pok$xs
Changes the value in a given memory location (performs a poke operation).

put$xs
Put a character into an array by using efficient indexing and byte-swapping operations.

rbq$xs
Returns the value removed from the bottom of a queue.

rdy$xs
Returns the character that was typed at a terminal, if any.

rtq$xs
Returns the value removed from the top of a queue.

s1c$xs
Implements an atomic set-and-test operation.

s2c$xs
Implements an atomic set-and-test operation on a double-word.

sky$xs
Changes the value of the CPU keys.

stk$xs
Sets and reads the value of the stack extension pointer.

tsq$xs
Returns the number of entries in a queue.

TABLE OF CONTENTS

A Report on the Accuracy of PR1ME Computers' Floating Point Software and Hardware

- and -

The SWT Math Library User's Guide

Technical Report GIT-ICS-83/09

Eugene H. Spafford

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

April 24, 1983

Introduction

Users of Prime computers have been aware for some time of a number of shortcomings in the floating point arithmetic firmware. In addition, there have been a number of inaccuracies found in the standard math libraries which have gone uncorrected for years ({1}, {2}). Unlike other major computer firms, Prime has not published any documents dealing with the algorithms or error analysis of their math routines.

In the winter of 1982 I undertook the coding of a new math library to support the Georgia Tech SWT Pascal compiler, and the Georgia Tech C compiler. The results of tests on that library and the standard Prime libraries have revealed a number of interesting facts. Additionally, further experimentation with the floating point mechanisms has revealed some bugs in the way arithmetic is performed, in some cases.

First, this guide describes the architecture of the floating point mechanism, including some error analysis and description of quirks in the hardware. This includes a description of incompatibilities between the 400/550 cpu and the 750/850 cpu floating point register structure. Next is a description of the SWT Math library. Last is a discussion of some preliminary error analysis of the SWT library and the Prime standard library functions. The appendices contain information on auxiliary programs supplied with the library which will aid users in writing their own routines, and checking existing routines and floating point firmware.

Acknowledgements

I would like to thank Roy Mongiovi for his help in debugging some of the SWT Math routines, and Peter Wan for his help in preparing this guide. I would also like to thank Ann Vitale, Ron Kurtzer, and especially Emory Stevens of the Atlanta Prime office for their co-operation and aid in the testing of these routines.

Research contributing to the development of this report was conducted while the author was receiving a National Science Foundation Graduate Fellowship, support which is gratefully acknowledged.

The Hardware

Internal Representation of Floating Point Values

There are two basic forms of floating point representation on the Prime: single precision and double precision. Both forms are stored in memory and the registers in about the same manner. It should be noted, however, that the storage format **in memory** and the storage format **in the registers** are different from each other. Also, the representation of values is different on 750/850 models than on the others.

Note that both forms of floating point values are available in three of the four Prime addressing modes: R, V and I. For purposes of this discussion, assume that all references are being made to the V mode instructions and registers unless noted otherwise. Also note that when I refer to the 400/550 machines, this also includes the 550-II.

The reader might be interested in perusing {12} through {15} for information about the proposed IEEE 754 standard on floating point representation. These articles also contain information about internal representation and accuracy of results. As a matter of interest, Prime Computer, Inc. had two voting representatives on the committee.

Storage Formats

A floating point value consists of three parts: a sign, a normalized mantissa, and an exponent. The mantissa is a two's complement value with an implied leading binary point (radix point). A normalized mantissa always represents a value in the interval [0.5, 1) unless it represents zero. The sign bit is set to indicate a negative value, reset to indicate a positive value. The sign bit is always in the most significant bit position (bit one). Following the sign bit is the mantissa.

A single precision value consists of the sign bit, 23 mantissa bits, and 8 exponent bits. The sign bit is bit one, the mantissa is bits 2 to 24, and the exponent is bits 25 to 32. The exponent is stored in excess-128 representation. That is, the value stored in the 8 bits of the exponent, if viewed as a two's complement value, is always 128 greater than the value it represents. Thus,

0 0 0 0 0 0 0 0	represents -128
1 1 1 1 1 1 1 1	represents 127
1 0 0 0 0 0 0 0	represents 0

```
1 0 0 0 0 1 0 0     represents    4

0 1 1 1 1 1 0 0     represents   -4
```

This implies that the largest possible exponent is +127, and the smallest possible exponent is -128. The exponent is taken to the base 2. (You may wish to refer to a reference such as {3} or {4} for more information about value representations.)

A double precision value consists of the sign bit, a 47 bit mantissa, and 16 exponent bits. The sign bit is bit one, the mantissa is bits 2 to 48, and the exponent is bits 49 to 64. The exponent is stored as a 128-biased value. This is similar to excess-128 except that the most significant bit of the exponent is taken as a sign bit. Thus,

```
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0    represents      0

0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0    represents      4

0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0    represents     -4

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    represents -32896

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1    represents  32639
```

As you can see from the examples, the range of the exponent is larger in the negative direction than in the positive. This means that it is possible to have values in the register whose multiplicative inverses cannot be represented.

Normalization

Every arithmetic operation on a floating point value causes the mantissa to be normalized. **On the Primes** normalization means that the mantissa is shifted towards the sign bit until the bit next to the sign bit is different from the sign bit. The exponent is decreased by the same amount as the number of places shifted. Normalization does **not** always mean shifting until a "1" is present in the second bit.

Let us examine an example. Suppose we have just completed a single precision add, and the result is either 5 1/2 or -5 1/2 as follows:

```
0 00010110000000000000000 10000110     5.5
1 11101010000000000000000 10000110    -5.5
```

Neither of these values is normalized. The mantissa is shifted left until its first bit is different from the sign bit. Note that it takes exactly 3 such shifts for each value:

```
0 10110000000000000000 10000011     5.5
1 01010000000000000000 10000011    -5.5
```

Both of these values are now normalized. The value of each is unchanged. There is no assumed first bit as on some machines (such as certain PDP machines).

Normalization helps maintain accuracy of results between computations. Additionally, comparisons between floating point numbers is made much easier -- a zero can always be recognized by examining the first word of the value only, and comparison between two floating point numbers can sometimes be done by a simple compare of the exponents and mantissa sign. It also helps to ensure that only one of the two values needs to be adjusted prior to some arithmetic operations (such as add).

A special case is when the sign bit is one (a negative value) and every bit of the mantissa is zero. This is *not* equal to zero, but rather is equal to -0.5 (assuming the exponent represents zero, of course).

It should be noted that load and store operations do not cause the register contents to be normalized. There is also no "normalize" instruction which will allow the user to normalize the bit pattern in the register.

Floating skip operations (eg, FSGT, FSZE) and comparison operations (eg, FCS and DFCS) will not work correctly unless the values involved are normalized.

Representation in the Registers

The single precision floating point register has more range than can be accommodated in the memory format. The single precision floating point register overlaps the double precision register and uses the extra bits available in the double floating register as guard bits. The register is organized as follows on 400/550 cpus:

```
S MMMMMMMMMMMMMMMMMMMMMMM GGGGGGGG HHHHHHHH EEEEEEEE 0000000000000000
1      2..24                25..32   33..40   41..48      49..64
```

Where:

- S is sign of the mantissa
- M is the mantissa (2's complement)
- G is mantissa extension (guard bits)
- H is exponent extension (guard bits)
- E is exponent (128-biased)
- 0 extra bits -- must be zero

On 750 and 850 cpus (with hardware floating point) the organization is:

```
S MMMMMMMMMMMMMMMMMMMMMMM GGGGGGGGGGGGGGGGGGGGGGG HHHHHHHH EEEEEEEE
1      2..24                    25..48              49..56   57..64
```

Where:

- S is sign of the mantissa
- M is the mantissa (2's complement)
- G is mantissa extension (guard bits)
- H is exponent extension (guard bits)
- E is exponent (excess 128)
- 0 extra bits -- must be zero

The guard bits are always zeroed whenever a floating load operation is done (FLD). The high-order guard bit may be used to round the least significant bit of the regular mantissa just before storage by using the FRN instruction. This increases accuracy somewhat at the cost of increased execution time. See the section on "Firmware Accuracy" for more details.

Double precision floating point values are similar in nature to single precision and are organized as follows on 400/550 machines:

```
S MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM EEEEEEEEEEEEEEEE MMMMMMMMMMMMMMMM
1             2..32                    33..48           49..64
```

Where:

- S is sign of the mantissa
- M is the mantissa (2's complement)
- E is exponent (excess 128, two's complement)

On 750 and 850 machines, the double precision register is organized as:

```
S MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM EEEEEEEEEEEEEEEE
1                    2..48                          49..64
```

Where:

- S is sign of the mantissa
- M is the mantissa (2's complement)
- E is exponent (128 biased)

Access Methods

Besides the standard load and store instructions, it is possible to access portions of the floating point registers with integer operations. These accesses are done either through the use of P300 address traps, or through the LDLR/STLR instructions.

If short memory references are made to locations 4, 5, and 6, the instructions actually are accessing the first two words of the mantissa and the exponent, respectively. In single precision references, the reference to the exponent fetches both the exponent and exponent guard bits. In double precision, the reference to location 6 fetches the complete exponent. Thus, the PMA sequence:

```
LDA   ='40000
STA#  4
CRA
STA#  5
LDA   =128
STA#  6
```

results in the value 0.5 being in the single precision floating register (note that this sequence also loads all the guard bits correctly on a 400/550).

It is also possible to access the floating point register via the LDLR and STLR instructions. In V mode, the first two words (bits 1 to 32) of the mantissa can be loaded into the L register by loading from register file location '12. The third word of the mantissa and the exponent can be obtained by loading from location '13. **The organization of the register file on 750/850 machines and 400/550 machines means that the L register contents after a "LDLR '13" will be different on these machines.** On 400/550 machines, the A register will contain the exponent and the B register will contain the third word of the mantissa. On a 750/850 these will be reversed. The program in Appendix I can be used to discover which case is present on your machine. When dealing with the two floating accumulators in I mode addressing, a "LDLR '11" will have the same problem.

Additionally, the floating accumulator shares the same register file location as the second field address and length registers (in the V mode register file). In the I mode registers, the first floating accumulator shares the same location as the first field address register, and the second floating accumulator shares the same location as the second field address register. Thus, various character manipulation instructions including decimal (character) arithmetic instructions may change the floating accumulators as a side effect.

Ranges

The effective range for single precision floating point values is approximately 1.701412 * (10 ** 38) to -1.701412 * (10 **38). The smallest, non-zero magnitude that can be represented is approximately 1.469368 * (10 ** -39). **This is the range for single precision storage in memory.** The guard bits in the register give extended range to values held in the register.

Effective range for double precision floating point values is approximately 2.079833 * (10 ** 9825) to -2.079833 * (10 ** 9825). The smallest, non-zero magnitude that can be represented is approximately 1.03808 * (10 ** -9903).

Available Operations

The following lists describe the instructions available on Prime 50 series machines to manipulate floating point values in 64V mode. This material has been extracted from the paper *64V Mode Instruction Summary and Addressing Formats*, by T. Allen Akin, Perry Flinn, and Eugene Spafford, Georgia Tech 1981. The abbreviation FAC refers to the floating accumulator, meaning the combination (overlapped) register. The instructions will be presented first grouped by function, then alphabetically. In the following instruction set summary, instruction formats are abbreviated as follows:

branch	branch
gen	generic
mr	memory reference

The descriptions of restricted instructions are preceded by an asterisk (*). Note that these instructions are *not* restricted unless segmented memory is turned on (bit 14 in current modals) and only if a reference is made outside of the range '0 to '17 (zero to 15, decimal).

In the descriptions of effects on the C-bit, L-bit, and condition codes, the following abbreviations are used:

C-bit:

-	unchanged
V	arithmetic overflow indication
X	indeterminate

L-bit:

-	unchanged
X	indeterminate

Condition Codes (CC):

-	unchanged
S	properly set to reflect value of result, may be used for condition code branches
X	indeterminate

Branch

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
BFEQ	branch	-	-	S	branch if FAC = 0
BFGE	branch	-	-	S	branch if FAC >= 0
BFGT	branch	-	-	S	branch if FAC > 0
BFLE	branch	-	-	S	branch if FAC <= 0
BFLT	branch	-	-	S	branch if FAC < 0
BFNE	branch	-	-	S	branch if FAC <> 0

Floating Point Arithmetic

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
FAD	mr	V	X	X	add memory to single precision FAC
FCM	gen	V	X	X	complement single precision FAC arithmetically
FDBL	gen	–	–	–	convert single precision floating to double precision
FDV	mr	V	X	X	divide memory into single precision FAC
FLTA	gen	V	X	X	convert 16 bit integer to single precision float
FLTL	gen	V	X	X	convert 32 bit integer to single precision float
FMP	mr	V	X	X	multiply single precision FAC by memory
FRN	gen	V	X	X	floating round double to single
FSB	mr	V	X	X	subtract memory from single precision FAC

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
FCS	mr	X	X	X	compare single precision FAC to memory and skip
FLD	mr	–	–	–	load single precision FAC from memory
FLX	mr	–	–	–	load double word index
FST	mr	V	X	–	store single precision FAC into memory
INTA	gen	V	X	X	convert single precision FAC to 16 bit integer
INTL	gen	V	X	X	convert single precision FAC to 32 bit integer

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
DFAD	mr	V	X	X	add memory to double precision FAC
DFCM	gen	V	X	X	complement double precision FAC arithmetically
DFDV	mr	V	X	X	divide memory into double precision FAC
DFMP	mr	V	X	X	multiply double precision FAC by memory
DFSB	mr	V	X	X	subtract memory from double precision FAC
FDBL	gen	–	–	–	convert single precision floating to double precision
FRN	gen	V	X	X	floating round double to single

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
DFCS	mr	X	X	X	compare double precision FAC with memory and skip
DFLD	mr	–	–	–	load double precision FAC
DFLX	mr	–	–	–	load quadruple word index
DFST	mr	–	–	–	store double precision FAC

Logicize

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
LFEQ	gen	–	–	S	set A to 1 if FAC = 0; else reset A to 0
LFGE	gen	–	–	S	set A to 1 if FAC >= 0; else reset A to 0
LFGT	gen	–	–	S	set A to 1 if FAC > 0; else reset A to 0
LFLE	gen	–	–	S	set A to 1 if FAC <= 0; else reset A to 0
LFLT	gen	–	–	S	set A to 1 if FAC < 0; else reset A to 0
LFNE	gen	–	–	S	set A to 1 if FAC <> 0; else reset A to 0

Skip

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
FSGT	gen	–	–	–	skip if FAC > 0
FSLE	gen	–	–	–	skip if FAC <= 0
FSMI	gen	–	–	–	skip if FAC < 0
FSNZ	gen	–	–	–	skip if FAC <> 0
FSPL	gen	–	–	–	skip if FAC >= 0
FSZE	gen	–	–	–	skip if FAC = 0

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
DFCS	mr	X	X	X	compare double precision FAC with memory and skip
FCS	mr	X	X	X	compare single precision FAC to memory and skip

Data Movement

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
DFLD	mr	–	–	–	load double precision FAC
DFLX	mr	–	–	–	load quadruple word index
DFST	mr	–	–	–	store double precision FAC
FLD	mr	–	–	–	load single precision FAC from memory
FLX	mr	–	–	–	load double word index
FST	mr	V	X	–	store single precision FAC into memory
LDLR	mr	–	–	–	*load L from register file
STLR	mr	–	–	–	*store L into register file

Address Manipulation

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
DFLX	mr	–	–	–	load quadruple word index
FLX	mr	–	–	–	load double word index

Type Conversion

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
FDBL	gen	-	-	-	convert single precision floating to double precision
FLTA	gen	V	X	X	convert 16 bit integer to single precision float
FLTL	gen	V	X	X	convert 32 bit integer to single precision float
FRN	gen	V	X	X	floating round double to single
INTA	gen	V	X	X	convert single precision FAC to 16 bit integer
INTL	gen	V	X	X	convert single precision FAC to 32 bit integer

Instructions Grouped Alphabetically

Mnemonic	*Format*	*C*	*L*	*CC*	*Description*
BFEQ	branch	-	-	S	branch if FAC = 0
BFGE	branch	-	-	S	branch if FAC >= 0
BFGT	branch	-	-	S	branch if FAC > 0
BFLE	branch	-	-	S	branch if FAC <= 0
BFLT	branch	-	-	S	branch if FAC < 0
BFNE	branch	-	-	S	branch if FAC <> 0
DFAD	mr	V	X	X	add memory to double precision FAC
DFCM	gen	V	X	X	complement double precision FAC arithmetically
DFCS	mr	X	X	X	compare double precision FAC with memory and skip
DFDV	mr	V	X	X	divide memory into double precision FAC
DFLD	mr	-	-	-	load double precision FAC
DFLX	mr	-	-	-	load quadruple word index
DFMP	mr	V	X	X	multiply double precision FAC by memory
DFSB	mr	V	X	X	subtract memory from double precision FAC
DFST	mr	-	-	-	store double precision FAC
FAD	mr	V	X	X	add memory to single precision FAC
FCM	gen	V	X	X	complement single precision FAC arithmetically
FCS	mr	X	X	X	compare single precision FAC to memory and skip
FDBL	gen	-	-	-	convert single precision floating to double precision
FDV	mr	V	X	X	divide memory into single precision FAC
FLD	mr	-	-	-	load single precision FAC from memory
FLTA	gen	V	X	X	convert 16 bit integer to single precision float
FLTL	gen	V	X	X	convert 32 bit integer to single precision float
FLX	mr	-	-	-	load double word index
FMP	mr	V	X	X	multiply single precision FAC by memory
FRN	gen	V	X	X	floating round double to single
FSB	mr	V	X	X	subtract memory from single precision FAC
FSGT	gen	-	-	-	skip if FAC > 0

FSLE	gen	-	-	-	skip if FAC <= 0
FSMI	gen	-	-	-	skip if FAC < 0
FSNZ	gen	-	-	-	skip if FAC <> 0
FSPL	gen	-	-	-	skip if FAC >= 0
FST	mr	V	X	-	store single precision FAC into memory
FSZE	gen	-	-	-	skip if FAC = 0
INTA	gen	V	X	X	convert single precision FAC to 16 bit integer
INTL	gen	V	X	X	convert single precision FAC to 32 bit integer
LDLR	mr	-	-	-	*load L from register file
LFEQ	gen	-	-	S	set A to 1 if FAC = 0; else reset A to 0
LFGE	gen	-	-	S	set A to 1 if FAC >= 0; else reset A to 0
LFGT	gen	-	-	S	set A to 1 if FAC > 0; else reset A to 0
LFLE	gen	-	-	S	set A to 1 if FAC <= 0; else reset A to 0
LFLT	gen	-	-	S	set A to 1 if FAC < 0; else reset A to 0
LFNE	gen	-	-	S	set A to 1 if FAC <> 0; else reset A to 0
STLR	mr	-	-	-	*store L into register file

Error Handling

There are basically four floating point errors determined by the floating point firmware: store exception, overflow, underflow, and divide by zero. The action on these errors is determined by the state of bit 7 in the current cpu keys. If bit 7 is set, a floating point fault simply sets the C bit and no other action is taken. If bit 7 is reset, then an arithmetic fault is signalled and the standard fault handler invoked. In Primos, this usually entails signalling the ERROR condition.

A store exception is triggered when an attempt is made to FST (single precision floating store) a value which is too big or too small (negative) to be accomodated in the two word memory format used by single precision values. This can happen because the value in the floating point register may have been loaded or generated using double precision operations. Alternatively, the value in the register could have been generated by single precision operations, but the value is larger than the memory format can accomodate due to the extra capacity provided by the guard bits. A double precision store cannot cause a store exception.

Overflow and underflow operations are the result of arithmetic operations (add, subtract, multiply, or divide) whose result is too big or too small to fit (normalized) in the register. Thus, the exponent of the result must be bigger than 32639 for overflow (base 2 exponent), or less than -32896 for

underflow (see the next section).

A divide by zero fault is exactly what the name implies -- an attempt to divide by a floating point value, single or double precision, whose value is identical to zero.

Another type of fault, not strictly a floating point fault, is triggered when an attempt is made to convert a floating value to an integer, and the floating value is too big or too small to be held in the corresponding integer register.

It is possible for user programs to set bit 7 in the keys to ignore these fault conditions, but in doing so the user should realize that results could be invalid without any indication of error. Explicit checks should be made of the C bit after any operation which might cause an error. By default, the standard compilers and the PMA assembler generate entry control blocks (ECBs) for procedures with bit 7 reset to zero.

Firmware Accuracy

In this document, the word "firmware" refers to the microcode or hardware which performs the floating point arithmetic. 750 and 850 cpus have floating point operations implemented in hardware, while the other models have these operations implemented in microcode. Programs and subroutines depend on the accuracy of these operations, so it is crucial that these operations be implemented correctly.

Problems in Multiplication

There appears to be a bug in the double precision floating point multiplication at a few points near the maximum value. If a value whose base 2 exponent is 32639 (maximum possible) is multiplied by a value greater than 0.5, an overflow fault is triggered. Thus, it is possible to multiply a value in the register by something less than 1.0 and get an overflow! In some cases, attempting to multiply smaller values to yield a value theoretically in range also results in an overflow. We have not attempted exhaustive testing to determine limits where this occurs since the likelihood of encountering such an error is small. However, the problem is there, and the user is advised to be careful when writing tests which need to deal with values at the upper limit of register capacity.

A much more serious flaw is to be found in the DFMP instruction on 400/550 machines. The double floating multiply instruction appears to *always* return a result whose two least significant bits of the mantissa are zero. That is, every multiplication potentially loses 2 out of 47 bits of precision! It is possible to multiply a value by 1.0 and not obtain a result equal to the original value. Such errors can, of course, cascade

and result in severe accuracy problems in chains of calculations. The hardware on 750/850 machines appears to be free of this defect. Appendix II contains a program to test your machine and illustrate this problem.

Oddly enough, division on the 400/550 machines does not appear to truncate any bits of precision, and according to published timing figures {5} the DFDV instruction is just as fast (slow) as the DFMP instruction. Thus, it might be advisable to recode critical calculations on these machines to be composed of divisions rather than multiplications, whenever possible.

Loss of Precision in Type Conversion

When converting from integers to floating point there are basically two machine instructions: FLTA and FLTL. The FLTA instruction converts a 16 bit integer into a single precision floating point value (24 bit mantissa). The FLTL instruction converts a 32 bit integer into a single precision floating point value. Note that such a conversion potentially drops 8 bits of precision. There is adequate storage in the double precision floating point register to convert without a loss of precision, but there is no instruction to convert from long integer to double precision real. Rather, the conversion must be done by a series of instructions; see the code for the SWT 'dble$m' routine.

Problems in the Other Operations

We have not observed any loss of precision in the addition, subtraction or division of double precision quantities. We have also not been able to detect any precision losses in any of the single precision operations. However, this does not indicate the absence of errors, rather it just indicates that we have not extensively tested for such errors and none have appeared in any of our other tests.

Floating Round

Studies performed at The Flinders University of South Australia on a 750 have indicated that some calculations performed in single precision floating point may benefit from the fact that the register contains extra precision, but that the results may be somewhat uneven depending on how the code is organized {6}. Their studies have also indicated that use of the FRN (floating round) instruction before each store greatly enhances the accuracy of some calculations in single precision:

> "In fact for the single precision problem a simple and almost complete cure for the problem has been demonstrated, and that is for the compiler to force a round before every store (i.e. emit an FRN instruction before each FST instruction emitted)." {6}

Their studies also indicated that the double precision arithmetic failed to do correct rounding. In fact, double precision operations truncate their results rather than rounding (see the next section). This leads to slightly skewed results which are especially noticeable in problems requiring very precise results:

> "... Consequently the Prime-750 exhibits a far larger error than the VAX-11/780 when we use the sum of squares measure. This error has been detected by our users in other calculations and programs and is particularly critical when nearly unstable matrix problems are investigated.... The consequences of such inaccuracy in a research-oriented application area could be critical." {6}

That conclusion was made for a 750 with hardware floating point operations. It can certainly be concluded that a 400 or 550 is not at all appropriate for double precision calculations requiring any high degree of accuracy.

Precision

The various models of Prime computer perform floating point operations to slightly different precisions. To quote from section 6.2.1 of {9}:

> "In double and single precision add, subtract, and multiply operations, the *750 and 850* truncate results to 48 sign and magnitude bits. Single precision divide operations on these processors produce 32 sign and magnitude bits of rounded result....
>
> Double precision operations on the 500-II (and 650) are identical to those performed on the 750 and 850. Single precision divide is also identical to 750/850 single precision divide. Single precision add, subtract, and multiply operations truncate results to 32 sign and magnitude bits.
>
> For all other 50 Series systems, double precision add and subtract operations truncate results to 48 sign and magnitude bits; multiply and divide operations truncate to 47 sign and magnitude bits. All single precision operations on these processors truncate results to 32 sign and magnitude bits."

These statements tend to raise serious doubts about the accuracy of similar programs run on different model machines due to precision changes. It also would indicate that some program behavior might change when run on a different model cpu.

The SWT Math Library

In General

The Software Tools Subsystem (SWT) is a major software package developed at Georgia Tech for Prime 50-series machines. It includes an advanced command interpreter with command pipes and i/o re-direction, a full screen editor with advanced regular pattern matching and replacement, and a large library of utility routines. One of the libraries which is to be included in further releases of the Subsystem is the SWT Math Library.

The SWT Math Library contains thoroughly tested routines to calculate various useful functions, including standard trigonometric functions. All of the routines share a number of common features which will be described in the next section. The individual routines will be described in the sections following.

Source

Most of the routines were obtained from the book *Software Manual for the Elementary Functions* by William Cody, Jr. and William Waite {7}. The random number generator was written utilizing material from {8}, and a few routines such as 'dble$m' and 'dint$m' were developed by the author. Testing of the routines is described in the next chapter.

Implementation

All of the SWT Math routines have been coded in Prime assembly language. Although this may make the code somewhat harder to read, it helps to enhance the accuracy and efficiency of the routines. A number of actions, such as direct manipulation of the exponent in the register file, are not available in higher level languages and this was a major factor in the decision to use assembler.

One factor which helps to increase the accuracy of the routines is the manner in which constants for the routines were obtained. Almost all of the constants used in the SWT Math Library are given as hexadecimal data constants in the assembly language programs. These values were derived from the constants given in {7} and the program in Appendix III. The program in Appendix III was run on a Cyber 760 which has over 90 bits of precision in the mantissa of double precision floating point values. The program calculates the proper rounded representation of the given input constants and returns the appropriate hex values.

It is interesting to note that some of the standard Prime library routines were also derived from {7} but many of the constants are given in the source code as decimal values. Tests by the author indicate that the PMA assembler does not always translate double precision decimal values into the correct bit pattern, thus inducing error.

Timing

One factor that is of interest to users of any math package is that of the efficiency of the code. Unfortunately, it is not possible to make a direct comparison of the speed of routines in the SWT Math Library to that of equivalent routines in the standard Prime libraries. The Prime native compilers are able to generate special "shortcalls" to known library subprograms which enhance their apparent speed. The SWT Math library routines are all done as regular procedure calls and will thus appear much slower if compared directly. The only statement that can be made about the efficiency of these routines is that they were coded in PMA by someone expert in that language, and they have been optimized as much as possible without sacrificing accuracy.

Naming and Function

All of the functions in the SWT Math Library return double precision values. Most of the functions have two entry points for every calculation -- one for a single precision argument and one for a double precision argument. The routines which take single precision arguments do argument verification and will not return a value which is out of range for a single precision floating point value. Thus, the value returned by those functions can be considered to be single precision. Since the single and double precision registers overlap, it is trivial to use these functions as either single or double precision.

In general, routines whose names begin with the letter 'd' are intended to take double precision arguments. Specific considerations are given in the sections below.

Errors

In the standard Prime library routines, calling a function with an improper value (such as trying to take the square root of a negative value) results in a signal to the condition ERROR. This signal cannot be returned from and thus execution of the program is terminated. Furthermore, the nature of the error and the routine involved is not well specified. In the Fortran 66 library the cause of the error is better identified but the general result is the same.

In the SWT Math Library whenever an error condition is encountered, the condition SWT_MATH_ERROR$ is signalled. The "ms" structure indicated by the call to SIGNL$ is the stack frame

of the routine calling the math routine, and the "info" structure is composed of the faulty argument (4 words), default return value (4 words), and a pointer (2 words) to a message describing the error. The user may specify an on-unit which can examine and change the default return value. The signal can be returned from and thus execution may continue.

The routine 'err$m' is a default on-unit handler which can be used to print the name of the faulting routine and the value of the faulting argument. This guide is not intended to present the information necessary to understand the Prime on-unit mechanism, so the interested reader is directed to the code for 'err$m' and to {10}.

Each routine sets the 'owner' pointer at offset 18 within the stack frame, and each routine has its ECB labelled according to standard conventions. Thus, the Primos DMSTK command will print the names of activations of SWT Math Library routines, as will programs such as DBG.

To the best of my knowledge, no error can occur during the execution of any of the SWT Math routines which does not signal the condition SWT_MATH_ERROR$. Thus, unlike many of the Prime routines, the user will not encounter errors such as 'SIZE' or 'OVERFLOW' during execution of these routines (see the section on Tests for more specific details).

The Routines

ACOS$M and DACS$M

These two functions calculate the inverse cosine of an angle. The argument to the functions is the cosine of the angle, and the function returns the measure of the angle, in radians. The 'dacs$m' function expects a double precision argument, and the 'acos$m' function expects a single precision argument. Arguments to the functions must be in the closed interval [-1.0, 1.0] or else the condition SWT_MATH_ERROR$ is signalled. In the case of an error, the default return value is zero.

The functions are implemented as rational minimax approximations on a modified argument value.

ASIN$M and DASN$M

These two functions calculate the inverse sine of an angle. The argument to the functions is the sine of the angle, and the function returns the measure of the angle, in radians. The 'dasn$m' function expects a double precision argument, and the 'asin$m' function expects a single precision argument. Arguments to the functions must be in the closed interval [-1.0, 1.0] or else the condition SWT_MATH_ERROR$ is signalled. If an error is

signalled, the default function value is zero.

The functions are implemented as rational minimax approximations on a modified argument value.

ATAN$M and DATN$M

These two functions calculate the inverse tangent of an angle. The argument to the functions is the tangent of the angle, and the function returns the measure of the angle, in radians. The 'datn$m' function expects a double precision argument, and the 'atan$m' function expects a single precision argument. The functions will not signal any errors based on input values.

The functions are implemented as a rational approximation on a modified argument value. Note that there is no equivalent to the ATAN2 function which is available in some implementations of Fortran; if users wish such a function, they may construct it from this function.

COS$M and DCOS$M

These two functions return the cosine of the angle whose measure (in radians) is given by the argument. The 'dcos$m' routine expects a double precision argument, and the 'cos$m' routine expects a single precision argument. If the absolute value of the angle plus one-half pi is greater than 26353588.0 then the condition SWT_MATH_ERROR$ is signalled. If an error is signalled, the default function return is zero.

The functions are implemented as minimax polynomial approximations.

COSH$M and DCSH$M

These two routines calculate the hyperbolic cosine of their arguments, defined as cosh(x) = [exp(x) + exp(-x)]/2. The function 'dcsh$m' expects a double precision value as argument, and the 'cosh$m' function expects a single precision argument. The condition SWT_MATH_ERROR$ is signalled if the absolute value of the argument is greater than 22623.630826296. In the single precision case, arguments which produce a value too large for single precision storage will also signal the error condition. If an error is signalled, the default function value is zero.

COT$M and DCOT$M

These two functions calculate the cotangent of the angle whose measure is given (in radians) as the argument to the functions. The 'dcot$m' function expects a double precision argument, and the 'cot$m' routine expects a single precision

argument. The arguments must have an absolute value greater than 7.064835966E-9865 and less than 13176794.0 or else the condition SWT_MATH_ERROR$ will be signalled. If an error is signalled, the default function return is zero.

The functions are calculated based on a minimax polynomial approximation over a reduced argument.

DBLE$M

The 'dble$m' function implements something akin to the Fortran 66 'dble' function, or the Fortran 77 'dreal' function. It takes as an argument a 32 bit integer and returns a double precision floating point number of the same value. This function should always be used when converting 32 bit integers to double precision real numbers because the code generated by some of the compilers will (potentially) lose up to 8 bits of mantissa precision (see the discussion in the previous chapter).

The 'dble$m' function has no single precision counterpart in this library. The routine, as defined, does not recognize or signal any error conditions. It is written so as to work on both 550 and 750 style machines, despite the internal difference in register structure.

The algorithm involved was derived from known register structure by the author.

DINT$M

The 'dint$m' function implements the Fortran 'dint' function. That is, it takes one double precision value and resets bits in the mantissa to remove any fractional part of the value. The return value is a double precision real. This routine also has a shortcall (JSXB) entrance labelled 'dint$p' which is used in some of the other math routines; users should not attempt to use this shortcall entrance unless they are aware of its structure.

The 'dint$m' of 1.5 is 1.0, the 'dint$m' of -1.5 is -1.0, and the 'dint$m' of anything less than 1.0 and greater than -1.0 is equal to zero.

The dint$m function has no single precision counterpart in this library. The routine, as defined, does not recognize or signal any error conditions. It is written so as to work of both 550 and 750 style machines, despite the internal difference in register structure.

The algorithm involved was developed by the author based on the known register structure.

ERR$M

The 'err$m' procedure is provided as a default handler for the SWT_MATH_ERROR$ condition. It takes a single argument, a 2 word pointer as defined by the condition mechanism, and prints information about the routine and values which signalled the fault. All output from the 'err$m' routine is sent to SWT ERROUT. Included in the output is the name of the faulting routine, the location from which the faulting routine was called, the value of the argument involved, and the default return value to be used.

The following code illustrates how to set up this default handler for use in Fortran 66 programs:

```
EXTERNAL ERR$M

CALL MKON$F ('SWT_MATH_ERROR$', 15, ERR$M)
```

The following code illustrates how to set up this default handler for use in Fortran 77 programs:

```
EXTERNAL ERR$M, MKON$P

CALL MKON$P ('SWT_MATH_ERROR$', 15, ERR$M)
```

The user may wish to copy and modify the source code for the 'err$m' procedure so as to provide a more specific form of error handling. If this is done, it would probably be a good idea to rename the user's version to something other than 'err$m.'

EXP$M and DEXP$M

These two functions implement the inverse of the 'ln$m' and 'dln$m' functions. That is, they raise the constant **e** to the power of the argument. The 'dexp$m' function takes a double precision argument, and the 'exp$m' function takes a single precision argument. Arguments to the 'exp$m' routine must be in the closed interval [-89.415985, 88.029678] and arguments to the 'dexp$m' routine must be in the closed interval [-22802.46279888, 22623.630826296], or else the SWT_MATH_ERROR$ condition will be signalled. If an error is signalled, the default function return value is zero.

It should be noted that the functions could simply return zero for sufficiently small arguments rather than signalling an error since the actual function value would be indistinguishable from zero to the precision of the machine. However, there is no mapping to zero in the actual function, and that is why the function signals an error in this case.

The routines are implemented as a functional approximation performed on a reduction of the argument.

LN$M and DLN$M

These two functions implement the natural logarithm (base **e**) function. The 'ln$m' function works for single precision arguments, and the 'dln$m' function works for double precision arguments. Arguments less than or equal to zero will signal the SWT_MATH_ERROR$ condition; the default return is the log of the absolute value of the argument, or zero in the case of a zero argument.

The algorithm involved uses a minimax rational approximation on a reduction of the argument. All positive inputs will return a valid result.

LOG$M and DLOG$M

These two functions implement the common logarithm (base 10) function. The 'log$m' function works for single precision arguments, and the 'dlog$m' function works for double precision arguments. Arguments less than or equal to zero will signal the SWT_MATH_ERROR$ condition; the default return is the log of the absolute value of the argument, or zero in the case of a zero argument.

The algorithm involved uses a minimax rational approximation on a reduction of the argument. All positive inputs will return a valid result.

POWR$M

The 'powr$m' function raises a double precision real value to a double precision real power. The function return is also double precision; there is no single precision equivalent. The algorithm is taken from {7}.

The function is coded so as to adhere to ANSI Fortran standards which do not allow raising negative values to a floating point power, and which do not allow zero to be raised to a zero or negative power. Other inputs may trigger an error if the result of the calculation would result in overflow.

The function implements the following equivalent operation in Fortran:

```
DOUBLE PRECISION A, B, C
A = B ** C
```

as

```
DOUBLE PRECISION A, B, C
DOUBLE PRECISION POWR$M
EXTERNAL POWR$M
A = POWR$M (B, C)
```

There are four cases where this function may signal SWT_MATH_ERROR$. If an attempt is made to raise a negative value to a non-zero power, then the default return value will be the absolute value of that quantity raised to the given power. If an attempt is made to raise zero to a zero or negative power, the default return is zero. If the result would overflow then the default return value is the largest double precision quantity that can be represented. If the result would cause underflow, the default return is the smallest positive value which can be represented on the machine.

SEED$M and RAND$M

The 'seed$m' procedure is used to reset the pseudo-random number generator to a known state. It is called with any 4 byte value which is not equal to 32 bits of zero. The seed can therefore be 4 characters, a long pointer, a long integer, or a real number. If the input is identical to zero then the SWT_MATH_ERROR$ condition is signalled. 'Seed$m' does not return a value.

The 'rand$m' function returns a double precision floating value in the open interval (0.0, 1.0). The argument to the function is set to a 32 bit integer in the range (0, 2**31 - 1). The generator is a linear congruential generator derived from information presented in {8}. The values returned seem to be very well distributed, both from the standpoint of spectral tests and lattice tests.

The 'rand$m' routine does not detect or signal any errors. The first time the 'rand$m' function is called, if the generator has not been initialized with the 'seed$m' procedure, a seed is derived based on the current time of day and cpu utilization.

SIN$M and DSIN$M

These two functions return the sine of the angle whose measure (in radians) is given by the argument. The 'dsin$m' routine expects a double precision argument, and the 'sin$m' routine expects a single precision argument. If the absolute value of the angle is greater than 26353588.0 then the condition SWT_MATH_ERROR$ is signalled. If an error is signalled, the default return value will be zero.

The functions are implemented as minimax polynomial approximations. Note that for angles sufficiently small the value of the sine function is equal to the measure of the angle.

SINH$M and DSNH$M

These two routines calculate the hyperbolic sine of their arguments, defined as sinh(x) = [exp(x) - exp(-x)]/2. The function 'dsnh$m' expects a double precision value as argument, and the 'sinh$m' function expects a single precision argument. The condition SWT_MATH_ERROR$ is signalled if the absolute value of the argument is greater than 22623.630826296. If an error is signalled, the default return value will be zero.

SQRT$M and DSQT$M

These two functions calculate the square root of a floating point value. The 'sqrt$m' function calculates the root of a single precision value, and the 'dsqt$m' routine works for double precision arguments. Attempts to take the square root of negative values will result in an error (signal to SWT_MATH_ERROR$). The default return in this case will be the square root of the absolute value of the argument. All other arguments are in range and return valid results.

The algorithm involved is based on Newton's approximation method with an initial multiplicative approximation. The argument is scaled to within the range [0.5, 2.0) and then the algorithm is iterated to a solution.

TAN$M and DTAN$M

These two functions calculate the tangent of the angle whose measure is given (in radians) as the argument to the functions. The 'dtan$m' function expects a double precision argument, and the 'tan$m' routine expects a single precision argument. The arguments must have an absolute value of less than 13176794.0 or else the condition SWT_MATH_ERROR$ will be signalled. If an error is signalled, the default return value will be zero.

The functions are calculated based on a minimax polynomial approximation over a reduced argument.

TANH$M and DTNH$M

These two routines calculate the hyperbolic tangent of their arguments, defined as tanh(x) = 2/[exp(2x) + 1]. The function 'dtnh$m' expects a double precision value as argument, and the 'tanh$m' function expects a single precision argument. The functions never signal an error and return valid results for all inputs.

_^H**T**_^H**e**_^H**s**_^H**t**_^H**i**_^H**n**_^H**g**

In General

It is important to test the standard mathematical functions which may be used in critical calculations. Not only will the tests measure the accuracy of the routines involved for use in later error estimations, but the testing helps provide information about the allowed domain and range of the functions. Many computer systems have quirks that require special case code for values near the extremes of precision {11}.

The Source of the Tests

The tests were taken from {7}. The tests were altered somewhat to help automate a test suite and also to provide a slightly more consistent form of output for comparison purposes. All of the tests use a set of common routines for non-test calculations and invocation. Where appropriate, the tests have been coded in both Fortran 66 (FTN) and Fortran 77 (F77) so as to test 3 libraries: the SWT Math library, the standard FTN library used by Fortran 66 and Ratfor programs, and the new standard library used in Fortran 77, Pascal, and PL/I programs.

The source code for the tests and support routines is located in the directory along with the source to the SWT Math library. There is a separate set of tests for single precision and double precision. These have been provided in case you wish to verify your own software, or re-run the tests on your own machine. Instructions on how to build and run a test are given in Appendix IV.

The Test Results

There are a number of error measures that could be used to describe these library routines. (For an involved discussion of some of the issues involved, see {7}) The tests which will be described below were taken from {7} and involve a number of checks and comparisons. Each test involves some random accuracy checks in various argument domains. These checks are made against known identities or calculations; for instance, the square root function is checked by comparing a random X against the square root of X*X.

Each accuracy test was performed for 5000 random arguments in each domain. The results of each test are given below, listed as the number of exact matches against the expected value, the number of times less than, and the number of times greater than. Also given are the MRE (maximum relative error) and the point at which that error occurred, and the RMS (root mean square) error

over all the tests in that domain. For those unfamiliar with these measures, the MRE can be thought of as a "worst case" error, and the RMS can be viewed as an "average case" measure of error.

The tests are given single precision first, then double precision. The tests with an asterisk ("*") after the CPU model are double precision test results.

Most of the routines have also been tested with special arguments at the limits of the argument domain or machine precision to help validate the entire range of possible input values. You will note that a few of the Prime standard library routines fail or return incorrect values at the extreme points of the domain. Other special tests are performed and described with each routine, as appropriate. The results given for some of these tests are worst-case results and not average-case; the average case performance was often much better with special arguments.

Finally, each routine was tested with values that would trigger an error (if appropriate). Again, some of the Prime library routines performed badly -- some of them returned incorrect values and never triggered an error.

A Special Note on 550 Results

Each test was run on a 550 model cpu at Georgia Tech and on a 750 model cpu at the Atlanta office of Prime Computer, Inc. The results for the 550 are intended for comparison purposes and *should not* be taken as a strict measure of accuracy. This is due to the problem with truncation of bits in double precision multiplies discussed in the last chapter. The vast differences in accuracy results between the 550 and 750 may be a measure of improvement in the library routines due to increased accuracy, or they may be an artifact caused by a change in the values calculated by the test programs themselves. The figures given should still allow some comparison between the Prime libraries and the SWT Math library, however.

Other Points of Interest

All of the tests invoke a special subroutine named 'machar' which determines machine characteristics to be used in the tests. The double precision version of this routine cannot be run unmodified on Prime machines due to their odd exponent structure. The double precision routine was modified by the author to return the results as defined by {7}. To recap the few most important points: a single precision value has 23 bits of mantissa and 8 bits of exponent, and rounds results. A double precision value has 47 bits of mantissa and 16 bits of exponent, and multiplication truncates results.

Since single precision arithmetic can include extra bits of accuracy if intermediate results are kept in the extended register, the test routines have been modified in places to force storage (and thus, truncation) of intermediate results. All of the single precision tests were compiled with the -FRN option set on. All of the tests were compiled with minimal optimizations enabled and full debugging. The debug option defeats register tracking optimizations and forces numerous stores. As an aside, this is often why erroneous numerical results disappear when a module is compiled with the debug option -- often to the amazement and indignation of the user.

The random number generator was not extensively tested since it was coded based on published, previous tests {8}. It should be noted, however, that a number of distribution and spectral tests were done locally to ensure that the implementation was not suspect. For comparison purposes, it should be noted that the multiplier used in the Prime APPLIB random number generator (16807) has been shown to be poor in performance on both spectral and lattice tests {8}. The Fortran intrinsic random number generators ('rnd' and 'irnd') behave very poorly in simple spectral tests. They are implemented as 16 bit generators rather than as 32 bit generators.

Use of These Results

It should be noted that these results are general in nature and should not be taken as a complete measure of accuracy on Prime computers. The author has not had extensive training in numerical analysis. A few of the tests did not appear to work correctly, and I found what I believe to be at least one genuine bug in the logic of one of the published test programs. The unusual and inconsistent register structure also leads to problems in running the tests.

It should also be noted that the Primos 18.4 version of the libraries was used in these tests. Future releases of these libraries may demonstrate better performance.

These tests are to be used for general comparison purposes of the Software Tools Math Library routines and the standard Prime libraries. There appear to be a number of accuracy problems in the Prime library routines and floating point firmware and hopefully some of these problems have been indicated in the following tests. Any user wishing to use the Primes or the SWT Math library for any critical applications should make their own tests before placing any great confidence in the results.

Inverse Sine and Cosine

There are no inverse sine or cosine functions in the Fortran 66 library, so these tests are for the other two libraries only.

Test 1
ASIN(X) vs. Taylor Series in (-0.125, 0.125)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	1031	3227	742	1.50 of 23	0.0110	0.00 of 23
550	SWT	1	4998	1	0.63 of 23	-0.0808	0.00 of 23
750	F77	1041	3234	725	1.50 of 23	0.0110	0.00 of 23
750	SWT	1	4998	1	0.63 of 23	-0.0808	0.00 of 23
550*	F77	3	66	4931	3.55 of 47	0.802E-2	2.15 of 47
550*	SWT	0	2348	2652	2.00 of 47	-0.1247	0.05 of 47
750*	F77	309	1563	3128	2.58 of 47	-0.0157	0.72 of 47
750*	SWT	0	2347	2653	2.00 of 47	-0.1247	0.05 of 47

Test 2
ACOS(X) vs. Taylor Series in (-0.125, 0.125)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	0	3320	1680	0.47 of 23	0.1249	0.00 of 23
550	SWT	0	4904	96	0.47 of 23	0.1249	0.00 of 23
750	F77	0	3319	1681	0.47 of 23	0.1249	0.00 of 23
750	SWT	0	4904	96	0.47 of 23	0.1249	0.00 of 23
550*	F77	0	816	4184	1.29 of 47	-0.0606	0.23 of 47
550*	SWT	0	1796	3204	0.47 of 47	0.1250	0.01 of 47
750*	F77	0	681	4319	1.27 of 47	-0.0874	0.25 of 47
750*	SWT	0	1795	3205	0.47 of 47	0.1250	0.01 of 47

Test 3
ASIN(X) vs. Taylor Series in (0.75, 1.00)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	76	1313	3611	2.00 of 23	0.84175	0.58 of 23
550	SWT	237	4123	640	1.00 of 23	0.8416	0.00 of 23
750	F77	76	1318	3606	2.00 of 23	0.84175	0.58 of 23
750	SWT	237	4123	640	1.00 of 23	0.8416	0.00 of 23
550*	F77	0	6	4994	6.95 of 47	1.0000	2.36 of 47
550*	SWT	0	446	4554	1.24 of 47	0.7502	0.70 of 47
750*	F77	125	1413	3462	4.88 of 47	1.0000	0.86 of 47
750*	SWT	0	595	4405	1.24 of 47	0.7500	0.66 of 47

Test 4
ACOS(X) vs. Taylor Series in (0.75, 1.00)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	3210	1261	529	2.95 of 23	0.9746	1.20 of 23
550	SWT	593	3785	622	1.00 of 23	0.8773	0.00 of 23
750	F77	3193	1270	537	2.92 of 23	0.9805	1.19 of 23
750	SWT	593	3785	622	1.00 of 23	0.8773	0.00 of 23
550*	F77	4955	41	4	14.43 of 47	1.0000	8.50 of 47
550*	SWT	2656	2344	0	2.00 of 47	0.8773	0.15 of 47
750*	F77	2560	1267	1173	12.47 of 47	1.0000	6.52 of 47
750*	SWT	2377	2623	0	1.99 of 47	0.8762	0.07 of 47

Test 5
ACOS(X) vs. Taylor Series in (-1.0,-0.75)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	0	2287	2713	0.73 of 23	-0.7504	0.15 of 23
550	SWT	0	4571	429	0.73 of 23	-0.7504	0.00 of 23
750	F77	0	2286	2714	0.73 of 23	-0.7504	0.15 of 23
750	SWT	0	4572	428	0.73 of 23	-0.7504	0.00 of 23
550*	F77	0	12	4988	5.35 of 47	-1.0000	1.46 of 47
550*	SWT	0	547	4453	0.73 of 47	-0.7500	0.51 of 47
750*	F77	15	930	4055	2.68 of 47	-1.0000	0.56 of 47
750*	SWT	0	608	4392	0.73 of 47	-0.7500	0.50 of 47

Examining the test results shows that the standard Prime library routines are not as accurate as one might wish, especially in test 4. According to {7}, the MRE error should not exceed 1.5 on any of the tests, and the RMS error should be no more than 0.75 in all tests. With the exception of the MRE in the double precision test 2, the SWT Math library performs within these limits; even the error in test 2 is acceptable when the RMS error for the same test is noted.

The tests of special arguments and error returns showed no problems or unexpected results.

Inverse Tangent

Test 1
ATAN(X) vs. Taylor Series in (-0.0625, 0.0625)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	527	4211	262	1.00 of 23	-0.0039	0.00 of 23
550	F77	527	4211	262	1.00 of 23	-0.0039	0.00 of 23
550	SWT	0	4999	1	0.32 of 23	0.0500	0.00 of 23
750	FTN	529	4213	258	1.00 of 23	-0.0039	0.00 of 23
750	F77	529	4213	258	1.00 of 23	-0.0039	0.00 of 23
750	SWT	0	4999	1	0.32 of 23	0.0500	0.00 of 23
550*	FTN	0	0	5000	3.32 of 47	0.0314	2.12 of 47
550*	F77	0	47	4953	3.20 of 47	-0.0043	1.95 of 47
550*	SWT	0	2508	2492	1.59 of 47	0.0313	0.00 of 47
750*	FTN	0	3	4997	2.00 of 47	-0.0156	1.11 of 47
750*	F77	0	697	4303	2.00 of 47	-0.0156	0.80 of 47
750*	SWT	0	2530	2470	1.59 of 47	0.0313	0.00 of 47

Test 2
ATAN(X) vs. ATAN(1/16)+ATAN((X-1/16)/(1+X/16)) in (0.0625, 0.2679)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	538	2636	1826	2.34 of 23	0.2007	0.21 of 23
550	F77	664	2482	1854	2.34 of 23	0.2007	0.35 of 23
550	SWT	425	3530	1045	1.40 of 23	0.1917	0.00 of 23
750	FTN	543	2626	1831	2.34 of 23	0.2007	0.21 of 23
750	F77	665	2475	1860	2.34 of 23	0.2007	0.35 of 23
750	SWT	423	3530	1047	1.40 of 23	0.1917	0.00 of 23
550*	FTN	372	1454	3174	2.99 of 47	0.0631	1.27 of 47
550*	F77	1774	723	2503	3.28 of 47	0.2081	1.68 of 47
550*	SWT	947	3933	120	1.02 of 47	0.2523	0.00 of 47
750*	FTN	63	2245	2692	1.70 of 47	0.0773	0.15 of 47
750*	F77	1773	1656	1571	2.64 of 47	0.2033	0.94 of 47
750*	SWT	192	4021	787	1.03 of 47	0.2503	0.00 of 47

Test 3
ATAN(X)*2 vs. ATAN(2X/(1-X*X)) in (0.2679, 0.4142)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	1868	2729	403	1.91 of 23	0.2717	0.05 of 23
550	F77	1531	2931	538	1.91 of 23	0.2717	0.02 of 23
550	SWT	882	3678	440	0.93 of 23	0.2680	0.00 of 23
750	FTN	1862	2734	404	1.91 of 23	0.2717	0.05 of 23
750	F77	1526	2933	541	1.91 of 23	0.2717	0.01 of 23
750	SWT	878	3679	443	0.93 of 23	0.2680	0.00 of 23
550*	FTN	158	175	4667	4.81 of 47	0.2731	3.40 of 47
550*	F77	1597	1506	1897	2.93 of 47	0.2693	1.05 of 47
550*	SWT	142	567	4291	3.30 of 47	0.3155	1.83 of 47
750*	FTN	119	137	4744	4.77 of 47	0.2817	3.43 of 47
750*	F77	3576	1015	409	2.76 of 47	0.3050	1.23 of 47
750*	SWT	146	1017	3837	2.80 of 47	0.2952	1.22 of 47

Test 4
ATAN(X)*2 vs. ATAN(2X/(1-X*X)) in (0.4142, 1.0)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	1943	3010	47	2.00 of 23	0.5483	0.07 of 23
550	F77	1970	2986	44	2.00 of 23	0.5479	0.12 of 23
550	SWT	453	4386	161	1.00 of 23	0.5465	0.00 of 23
750	FTN	1941	3012	47	2.00 of 23	0.5483	0.08 of 23
750	F77	1968	2988	44	2.00 of 23	0.5479	0.13 of 23
750	SWT	452	4387	161	1.00 of 23	0.5465	0.00 of 23
550*	FTN	188	576	4236	4.12 of 47	0.4254	2.35 of 47
550*	F77	939	1521	2540	2.76 of 47	0.6689	0.99 of 47
550*	SWT	20	906	4074	3.34 of 47	0.4166	1.94 of 47
750*	FTN	2	48	4950	4.32 of 47	0.4246	2.78 of 47
750*	F77	1913	1042	2045	2.35 of 47	0.6693	0.93 of 47
750*	SWT	872	1859	2269	2.35 of 47	0.4145	0.64 of 47

Examining the test results leads to some interesting conclusions. The SWT Math Library routines are definitely better than either Prime library version, especially in test 2. The margin of error suggested in {7} is met only by the SWT routines.

In the testing of special arguments, the Prime FTN library ATAN had problems with the identities ATAN(-x) = - ATAN(x) and ATAN(x) = x for small x. Errors in both cases were about 10E-7 of the magnitude of x in both single and double precision.

Exponential

In the following tests, the double precision tests did not run to completion when testing the FTN library due to problems in the EXP function. Due to incorrect coding of the function, a floating to fixed conversion raised a SIZE error when taking the exponential of a value which was theoretically in range. Thus, only the results for the first test are available for the FTN exponential function in double precision.

Test 1
EXP(X-0.0625) vs. EXP(X)/EXP(0.0625) in (-0.2841, 0.3466)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	624	3537	839	1.09 of 23	0.9069E-3	0.00 of 23
550	F77	1217	2173	1610	2.41 of 23	0.1815	0.39 of 23
550	SWT	553	3704	743	1.00 of 23	0.0629	0.00 of 23
750	FTN	636	3525	839	1.09 of 23	0.9069E-3	0.00 of 23
750	F77	1218	2172	1610	2.41 of 23	0.1815	0.39 of 23
750	SWT	553	3704	743	1.00 of 23	0.0629	0.00 of 23
550*	FTN	1555	906	2539	4.53 of 47	0.0150	2.51 of 47
550*	F77	1619	711	2670	3.74 of 47	0.2457	1.96 of 47
550*	SWT	325	1759	2916	2.48 of 47	-0.2730	0.72 of 47
750*	FTN	479	1762	2759	4.17 of 47	0.6161E-2	2.23 of 47
750*	F77	1007	1597	2396	2.37 of 47	0.0293	0.78 of 47
750*	SWT	227	2056	2717	2.08 of 47	-0.2777	0.42 of 47

Test 2
EXP(X-2.8125) vs. EXP(X)/EXP(2.8125) in (-0.2277E+5, -0.3466E+1)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	2838	23	2139	6.42 of 23	-0.4405E+2	4.94 of 23
550	F77	1201	2287	1512	2.01 of 23	-0.6125E+2	0.32 of 23
550	SWT	499	3745	756	1.02 of 23	-0.1799E+2	0.00 of 23
750	FTN	2838	23	2139	6.42 of 23	-0.4405E+2	4.94 of 23
750	F77	1201	2285	1514	2.01 of 23	-0.6125E+2	0.32 of 23
750	SWT	499	3745	756	1.02 of 23	-0.1799E+2	0.00 of 23
550*	F77	2638	426	1936	47.00 of 47	-0.2268E+5	43.85 of 47
550*	SWT	1034	205	3761	13.95 of 47	-0.2264E+5	11.75 of 47
750*	F77	2036	1426	1538	47.00 of 47	-0.2089E+2	43.85 of 47
750*	SWT	441	424	4135	13.95 of 47	-0.2264E+5	11.75 of 47

Test 3
EXP(X-2.8125) vs. EXP(X)/EXP(2.8125) in (6.931, 87.92)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	2993	15	1992	5.69 of 23	0.8669E+2	5.05 of 23
550	F77	1204	2311	1485	1.93 of 23	0.5069E+2	0.30 of 23
550	SWT	489	3704	807	1.00 of 23	0.4371E+2	0.00 of 23
750	FTN	2993	15	1992	5.69 of 23	0.8669E+2	5.05 of 23
750	F77	1204	2311	1485	1.93 of 23	0.5069E+2	0.30 of 23
750	SWT	489	3704	807	1.00 of 23	0.4371E+2	0.00 of 23
550*	F77	2676	444	1880	6.12 of 47	0.1571E+5	4.28 of 47
550*	SWT	3082	899	1019	4.28 of 47	0.1592E+5	2.07 of 47
750*	F77	2078	1400	1522	5.08 of 47	0.1584E+5	2.65 of 47
750*	SWT	1065	2205	1730	3.47 of 47	0.2018E+5	1.29 of 47

The results of test 2 are a bit surprising. After careful checking of the code and the test, it seems likely that there is a problem in the test since the routines from both libraries appear so bad. The MRE values appear to be close to the limit of what the routines can compute without underflow. Performing a check on the MRE error in each case reveals that there is no measurable error in the exponential function at this point in regard to the logarithm function. That is, the values of the exponential functions at the MRE point, when used as arguments to the SWT logarithm function (which is known to be fairly accurate; see below), produce the exact same value as the MRE point. This leads to the conclusion that the testing procedure is somehow faulty due to the unusual register structure of the Primes. It can be concluded that (in this domain) the functions are probably correct, but the measure of error cannot be determined by this test.

The results of the other tests indicate major differences in accuracy amongst the routines. The SWT routine seems much better in most cases.

The tests of special arguments revealed a number of interesting items. For instance, the single precision F77 EXP routine does not signal an error when given arguments very much out of range. Instead, it returns either zero (in the case of underflow) or a very large value (in the case of overflow). Also, all of the functions have some amount of error in the identity EXP(X)*EXP(-X) = 1.0, with single precision values of X near 1.0 producing errors of approximately 10E-6, and double precision values near 1.0 producing errors of near 10E-12.

Logarithms

Test 1
ALOG(X) vs. Taylor Series of ALOG(1+Y) in (1-.7813E-2, 1+.7813E-2)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	2246	234	2520	2.57 of 23	0.9961	1.35 of 23
550	F77	1392	2607	1001	1.89 of 23	1.0021	0.07 of 23
550	SWT	1	4996	3	0.59 of 23	0.9948	0.00 of 23
750	FTN	2251	229	2520	2.57 of 23	0.9961	1.36 of 23
750	F77	1389	2603	1008	1.89 of 23	1.0021	0.07 of 23
750	SWT	1	4996	3	0.59 of 23	0.9948	0.00 of 23
550*	FTN	2449	315	2236	25.55 of 47	1.000	19.52 of 47
550*	F77	2038	996	1966	2.98 of 47	1.000	1.42 of 47
550*	SWT	1013	2493	1494	2.13 of 47	1.0000	0.19 of 47
750*	FTN	1314	1603	2083	25.55 of 47	1.000	19.52 of 47
750*	F77	1206	2507	1287	1.94 of 47	1.000	0.04 of 47
750*	SWT	1206	2507	1287	1.94 of 47	1.000	0.04 of 47

Test 2
ALOG(X) vs. ALOG(17X/16)-ALOG(17/16) in (0.7071, 0.9375)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	0	1930	3070	2.01 of 23	0.8300	0.41 of 23
550	F77	0	2753	2247	1.97 of 23	0.8253	0.07 of 23
550	SWT	0	3628	1372	1.00 of 23	0.7788	0.00 of 23
750	FTN	0	1936	3064	2.01 of 23	0.8300	0.41 of 23
750	F77	0	2760	2240	1.97 of 23	0.8253	0.07 of 23
750	SWT	0	3628	1372	1.00 of 23	0.7788	0.00 of 23
550*	FTN	0	54	4946	4.24 of 47	0.9299	2.49 of 47
550*	F77	0	132	4868	3.00 of 47	0.7323	1.28 of 47
550*	SWT	0	2053	2947	2.28 of 47	0.7347	0.51 of 47
750*	FTN	0	1022	3978	4.26 of 47	0.9367	2.47 of 47
750*	F77	0	2067	2933	1.99 of 47	0.7779	0.46 of 47
750*	SWT	0	2067	2933	1.99 of 47	0.7779	0.46 of 47

Test 3
ALOG10(X) vs. ALOG10(11X/10)-ALOG(11/10) in (0.3162, 0.900)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	0	1870	3130	2.58 of 23	0.8659	0.90 of 23
550	F77	0	1986	3014	2.72 of 23	0.7045	0.74 of 23
550	SWT	0	2462	2538	2.06 of 23	0.8708	0.35 of 23
750	FTN	0	1867	3133	2.58 of 23	0.8659	0.90 of 23
750	F77	0	1983	3017	2.72 of 23	0.7045	0.75 of 23
750	SWT	0	2462	2538	2.06 of 23	0.8708	0.35 of 23
550*	FTN	0	924	4076	4.44 of 47	0.8936	2.18 of 47
550*	F77	0	1029	3971	3.58 of 47	0.8974	1.66 of 47
550*	SWT	0	1244	3756	3.73 of 47	0.8974	1.71 of 47
750*	FTN	0	1383	3617	4.40 of 47	0.8963	2.27 of 47
750*	F77	0	1684	3316	3.37 of 47	0.8946	1.34 of 47
750*	SWT	0	1499	3501	3.37 of 47	0.8943	1.29 of 47

Test 4
ALOG(X*X) vs. 2*ALOG(X) in (16.00, 240.0)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	2490	2510	0	1.00 of 23	0.5473E+2	0.15 of 23
550	F77	2499	2501	0	1.00 of 23	0.5473E+2	0.15 of 23
550	SWT	127	4873	0	0.99 of 23	0.5575E+2	0.00 of 23
750	FTN	2499	2501	0	1.00 of 23	0.5473E+2	0.15 of 23
750	F77	2491	2509	0	1.00 of 23	0.5473E+2	0.15 of 23
750	SWT	127	4873	0	0.99 of 23	0.5575E+2	0.00 of 23
550*	FTN	2911	2089	0	2.36 of 47	0.2263E+2	0.98 of 47
550*	F77	1195	3805	0	3.00 of 47	0.5491E+2	1.58 of 47
550*	SWT	1437	3563	0	1.53 of 47	0.1604E+2	0.00 of 47
750*	FTN	1548	3452	0	1.44 of 47	0.1909E+2	0.00 of 47
750*	F77	1537	3463	0	1.44 of 47	0.1909E+2	0.00 of 47
750*	SWT	333	4667	0	1.06 of 47	0.4591E+2	0.00 of 47

These tests indicate that both the SWT Math library and the F77 library implementations of the logarithm functions are within acceptable error bounds (as defined in {7}), with the SWT version being somewhat better. The Fortran 66 version obviously has some points at which it behaves very poorly (see test 1). The similarity between the results for the SWT and F77 versions as shown in tests 1 and 2 can probably be explained by the fact that the same algorithm was used in each.

The SWT MRE errors in the double precision part of tests 1 and 2 are a bit large, but the corresponding error in the RMS indicates that the error is not systematic in nature. The error is of no major significance, although it could possibly be less.

The SWT routine performed very well in tests of the identity ALOG(X) = -ALOG(1/X), returning exactly the same values in every

test. The FTN and F77 routines returned occasional matches, but were often in error by amounts close to 10E-6 (single precision) and 10E-12 (double precision).

What is most interesting is to note that both the F77 and FTN double precision routines are seriously flawed for very small arguments. Due to a rather obvious coding error, any double precision value whose exponent is less than -32640 will have its logarithm calculated as a large positive number -- just as if the sign of the exponent was reversed!! It would appear as if these routines were never tested at any values near the limits of their domains. The SWT routine does not suffer from this problem.

The POWR$M Function

The SWT 'powr$m' function was tested against the intrinsic "**" operation in these tests. That is, when testing the FTN and F77 libraries, the operation "x ** y" was used and the compilers were allowed to generate the calls to the appropriate library routines.

Although there is no single precision version of the SWT 'powr$m' function, it was tested within the range for single precision values and compared against the Prime power operation. Due to recurring problems in the division of very small values, and the multiplication of very large values caused by the faults in the hardware, tests 1 and 2 were the only double precision tests run to completion.

Test 1
X ** 1.0 vs. X in (0.50, 1.00)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	2399	2583	18	0.99 of 23	0.5022	0.00 of 23
550	F77	2886	1935	179	1.50 of 23	0.7072	0.37 of 23
550	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
750	FTN	2394	2586	20	0.99 of 23	0.5022	0.00 of 23
750	F77	2888	1932	180	1.50 of 23	0.7072	0.37 of 23
750	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
550*	FTN	5000	0	0	4.76 of 47	0.8469	4.12 of 47
550*	F77	4996	4	0	4.19 of 47	0.6025	3.06 of 47
550*	SWT	4997	3	0	1.94 of 47	0.5222	0.69 of 47
750*	FTN	4920	80	0	4.28 of 47	0.7735	3.66 of 47
750*	F77	4437	563	0	2.62 of 47	0.6511	1.38 of 47
750*	SWT	4837	163	0	1.06 of 47	0.9578	0.50 of 47

Test 2
(X*X)**1.5 vs. (X*X)*X in (0.50, 1.00)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	3330	1394	276	2.90 of 23	0.5118	0.98 of 23
550	F77	2047	2082	871	1.87 of 23	0.5147	0.35 of 23
550	SWT	332	4359	319	1.00 of 23	0.6300	0.00 of 23
750	FTN	3305	1410	285	2.94 of 23	0.5068	0.98 of 23
750	F77	2086	2062	852	1.98 of 23	0.9135	0.37 of 23
750	SWT	317	4349	334	0.99 of 23	0.7954	0.00 of 23
550*	FTN	4939	55	6	5.12 of 47	0.5712	4.03 of 47
550*	F77	4869	131	0	4.94 of 47	0.5466	3.30 of 47
550*	SWT	1172	2051	1777	2.57 of 47	0.5012	0.66 of 47
750*	FTN	4172	649	179	4.23 of 47	0.7358	3.47 of 47
750*	F77	3782	1010	208	2.62 of 47	0.6875	1.20 of 47
750*	SWT	2432	2566	2	1.06 of 47	0.7833	0.07 of 47

Test 3
(X*X)**1.5 vs. (X*X)*X in (1.00, 0.5541E+13)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	5000	0	0	8.51 of 23	0.1129E+13	7.80 of 23
550	F77	4950	0	50	17.93 of 23	0.5487E+13	13.83 of 23
550	SWT	315	4362	323	1.00 of 23	0.6928E+12	0.00 of 23
750	FTN	5000	0	0	8.53 of 23	0.2676E+12	7.80 of 23
750	F77	4950	0	50	17.93 of 23	0.5487E+13	13.83 of 23
750	SWT	337	4313	350	0.99 of 23	0.4407E+13	0.00 of 23

In test 4, the point given at which the MRE was recorded is the value of X. The Y value is available on request.

Test 4
X**Y vs. (X*X)**(Y/2), X in (0.01, 10.0), Y in (-19.42, 19.42)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	1006	3052	942	6.49 of 23	9.8692	4.20 of 23
550	F77	2266	518	2216	5.46 of 23	0.0120	3.43 of 23
550	SWT	1700	1604	1696	3.25 of 23	2.0541	1.28 of 23
750	FTN	958	3104	938	6.49 of 23	7.7463	4.20 of 23
750	F77	2251	514	2235	5.46 of 23	0.0120	3.47 of 23
750	SWT	1644	1591	1765	3.20 of 23	2.8599	1.30 of 23

It seems fairly obvious from the above test results that the Prime library routines are rather sadly lacking in precision. Test 3 alone shows a RME loss of nearly 18 out of 23 bits. Conclusions about tests 3 and 4 can possibly (with a cautionary warning!) be extrapolated to the double precision cases, at least for the SWT routine, since the routine is the same for both precisions. The tests of special arguments indicate that the 'powr$m' routine does behave well in the double precision case. Running small portions of the test to avoid some of the firmware arithmetic problems tends to support these conclusions.

Sine and Cosine

Test 1
SIN(X) vs. 3*SIN(X/3)-4*SIN(X/3)**3 in (0.0, 1.571)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	3047	32	1921	1.41 of 23	0.8183	0.00 of 23
550	F77	466	2204	2330	1.49 of 23	0.7910	0.00 of 23
550	SWT	498	3979	523	1.00 of 23	0.5243	0.00 of 23
750	FTN	3095	0	1905	1.41 of 23	0.8183	0.00 of 23
750	F77	466	2202	2332	1.61 of 23	0.3330	0.00 of 23
750	SWT	498	3979	523	1.00 of 23	0.5243	0.00 of 23
550*	FTN	31	233	4736	4.50 of 47	0.7888	3.17 of 47
550*	F77	3204	1276	520	4.07 of 47	0.3021	2.09 of 47
550*	SWT	2880	1207	913	3.41 of 47	0.1898	1.51 of 47
750*	FTN	130	679	4191	2.81 of 47	0.1495	1.44 of 47
750*	F77	863	1773	2364	2.30 of 47	0.6557	0.46 of 47
750*	SWT	494	2459	2047	2.04 of 47	1.3513	0.21 of 47

Test 2
SIN(X) vs. 3*SIN(X/3)-4*SIN(X/3)**3 in (18.85, 20.42)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	3546	12	1442	18.00 of 23	18.850	11.87 of 23
550	F77	431	2267	2302	1.73 of 23	19.001	0.00 of 23
550	SWT	510	3939	551	1.00 of 23	19.103	0.00 of 23
750	FTN	3560	0	1440	18.00 of 23	18.850	11.87 of 23
750	F77	431	2267	2302	1.73 of 23	19.001	0.00 of 23
750	SWT	511	3938	551	1.00 of 23	19.103	0.00 of 23
550*	FTN	394	118	4488	18.98 of 47	18.850	12.87 of 47
550*	F77	1800	660	2540	19.33 of 47	18.850	13.20 of 47
550*	SWT	1776	491	2733	19.33 of 47	18.850	13.20 of 47
750*	FTN	1852	160	2988	18.98 of 47	18.850	12.84 of 47
750*	F77	891	1803	2306	6.93 of 47	18.850	1.14 of 47
750*	SWT	699	2463	1838	2.02 of 47	20.242	0.14 of 47

Test 3
COS(X) vs. 4*COS(X/3)**3-3*COS(X/3) in (21.99, 23.56)

CPU	*Library*	5000 *gt*	Comparisons *eq*	*lt*	Maximum Rel. Error *Bitloss*	*At*	Root Mean Sq. *Bitloss*
550	FTN	1911	13	3076	11.83 of 23	23.555	7.14 of 23
550	F77	1850	24	3126	1.36 of 23	23.150	0.00 of 23
550	SWT	2470	33	2497	0.70 of 23	23.529	0.00 of 23
750	FTN	1923	0	3077	11.83 of 23	23.555	7.14 of 23
750	F77	1845	0	3155	1.37 of 23	23.150	0.00 of 23
750	SWT	2471	0	2529	0.70 of 23	23.530	0.00 of 23
550*	FTN	1470	658	2872	17.42 of 47	23.562	11.44 of 47
550*	F77	4978	20	2	17.77 of 47	23.562	11.70 of 47
550*	SWT	4657	291	52	17.77 of 47	23.562	11.70 of 47
750*	FTN	855	564	3581	15.33 of 47	23.561	9.78 of 47
750*	F77	4490	464	46	2.85 of 47	23.353	1.46 of 47
750*	SWT	1334	2614	1052	1.63 of 47	22.237	0.00 of 47

This is another test which illustrates how the multiplication bug in the 550 firmware can affect critical results. Observe the differences in double precision results in test 2 and 3. It is also fairly obvious that the FTN library sine and cosine functions have severe accuracy problems. The SWT library routines perform well within error limits {7} and are much better than the F77 routines.

When testing special arguments it is found that both the F77 and FTN routines have difficulty with the identities SIN(-X)=-SIN(X) and COS(-X)=COS(X). The ratio of the calculated difference to X is about 10E-8 for single precision, and 10E-12 to 10E-28 for double precision (the F77 library is more accurate). The SWT Library routines calculate no differences in these identities.

When special values are tested for error checking, it is discovered that the Prime routines trigger a SIZE error in float to fixed conversion when presented with a large argument rather than checking for (and reporting) the actual problem of an error of excessive magnitude. The SWT routine properly traps the error.

Hyperbolic Sine and Cosine

There are no hyperbolic sine (sinh) or hyperbolic cosine (cosh) routines in the FTN library, so the tests below are only for the F77 and SWT libraries.

Test 1
SINH(X) vs. Taylor Series in (0.00, 0.50)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	1	2418	2581	1.38 of 23	0.1908	0.17 of 23
550	SWT	8	4965	27	1.00 of 23	0.4827	0.00 of 23
750	F77	1	2430	2569	1.38 of 23	0.1908	0.17 of 23
750	SWT	7	4966	27	1.00 of 23	0.4827	0.00 of 23
550*	F77	87	754	4159	3.00 of 47	0.0156	1.69 of 47
550*	SWT	360	2597	2043	1.99 of 47	0.4855	0.06 of 47
750*	F77	343	2643	2033	1.99 of 47	0.4833	0.06 of 47
750*	SWT	370	2643	1987	2.00 of 47	0.4822	0.06 of 47

Test 2
COSH(X) vs. Taylor Series in (0.00, 0.50)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	0	3547	1453	1.00 of 23	0.0348	0.03 of 23
550	SWT	6	4905	89	0.98 of 23	0.1599	0.00 of 23
750	F77	1	3548	1451	1.00 of 23	0.3937E-2	0.03 of 23
750	SWT	6	4905	89	0.98 of 23	0.1599	0.00 of 23
550*	F77	0	143	4857	4.15 of 47	0.4906	2.74 of 47
550*	SWT	0	1442	3558	3.41 of 47	0.4997	1.28 of 47
750*	F77	0	2098	2902	3.41 of 47	0.4955	1.30 of 47
750*	SWT	0	2809	2191	3.15 of 47	0.4919	1.04 of 47

Test 3
SINH(X) vs. C*(SINH(X+1)+SINH(X-1)) in (3.00, LOG(XMAX))

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	2051	1879	1070	16.53 of 23	87.017	10.43 of 23
550	SWT	1618	3303	79	1.24 of 23	43.500	0.00 of 23
750	F77	2051	1881	1068	16.53 of 23	87.017	10.43 of 23
750	SWT	1618	3303	79	1.24 of 23	43.500	0.00 of 23
550*	F77	3615	388	997	5.92 of 47	0.2124E+5	4.19 of 47
550*	SWT	4579	262	159	4.45 of 47	0.2067E+5	3.17 of 47
750*	F77	3937	337	726	4.85 of 47	0.1669E+5	2.73 of 47
750*	SWT	4498	303	199	3.03 of 47	0.1304E+5	1.49 of 47

In test 4, the double precision COSH routine in the F77 library generated numerous errors for large values that should have been in range. These errors aborted the test and therefore there are no results for the double precision F77 COSH.

Test 4
COSH(X) vs. C*(COSH(X+1)+COSH(X-1)) in (3.00, LOG(XMAX))

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	2051	1903	1046	17.75 of 23	87.000	11.74 of 23
550	SWT	1558	3341	101	1.00 of 23	49.214	0.00 of 23
750	F77	2051	1904	1045	17.75 of 23	87.000	11.74 of 23
750	SWT	1558	3341	101	1.00 of 23	49.214	0.00 of 23
550*	SWT	4566	263	171	4.53 of 47	0.1794E+5	3.16 of 47
750*	SWT	4522	284	194	3.06 of 47	0.1281E+5	1.49 of 47

The results of tests 3 and 4 show that the F77 routines are rather inaccurate at the extremes of range. The RME measures for the SWT routines are a bit large, but the corresponding RMS error is small. According to the figures given in {7}, the SWT routines perform within the range of acceptable error.

As with many of the other tests, fundamental identities involving negated arguments were not calculated quite correctly in the Prime routines. Another interesting(?) result occurred when the F77 SINH routine was called with a very large positive value. The SINH routine did not signal an error, but rather returned the maximum floating point value -- an incorrect result.

Square Root

The square root function is one of the easiest to code and the accuracy of such a routine should be very, very good if done correctly. Newton's method converges quickly and requires only a few iterations on a reduced argument to reach a solution. Due to the nature of the square root function and its use, the random arguments are logarithmically distributed over the sample interval; all the other tests use a uniform distribution.

Test 1
SQRT(X*X) vs. X in (0.7071, 1.00)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	0	5000	0	0.00 of 23	-----	0.00 of 23
550	F77	1	4999	0	0.42 of 23	0.7500	0.00 of 23
550	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
750	FTN	0	5000	0	0.00 of 23	-----	0.00 of 23
750	F77	1	4999	0	0.42 of 23	0.7500	0.00 of 23
750	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
550*	FTN	0	0	5000	2.50 of 47	0.7095	1.33 of 47
550*	F77	0	1	4999	2.08 of 47	0.7074	1.13 of 47
550*	SWT	0	0	5000	2.49 of 47	0.7114	1.31 of 47
750*	FTN	0	2403	2597	0.50 of 47	0.7072	0.00 of 47
750*	F77	0	4481	519	0.50 of 47	0.7072	0.00 of 47
750*	SWT	0	2493	2507	0.50 of 47	0.7072	0.00 of 47

Test 2
SQRT(X*X) vs. X in (1.00, 1.414)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	0	5000	0	0.00 of 23	-----	0.00 of 23
550	F77	77	4923	0	1.00 of 23	1.0004	0.00 of 23
550	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
750	FTN	0	5000	0	0.00 of 23	-----	0.00 of 23
750	F77	80	4920	0	1.00 of 23	1.0004	0.00 of 23
750	SWT	0	5000	0	0.00 of 23	-----	0.00 of 23
550*	FTN	0	0	5000	3.00 of 47	1.0003	2.00 of 47
550*	F77	0	6	4994	3.00 of 47	1.0003	1.97 of 47
550*	SWT	0	0	5000	3.00 of 47	1.0003	2.00 of 47
750*	FTN	0	3384	1616	1.00 of 47	1.0001	0.00 of 47
750*	F77	1	3766	1233	1.00 of 47	1.0001	0.00 of 47
750*	SWT	0	3387	1613	1.00 of 47	1.0001	0.00 of 47

All of the routines perform well in these tests, and all have results within acceptable margins of error. Test 2 readily illustrates how results can change due to the double precision multiply bug on 550 machines. Nothing in these tests would particularly recommend one routine against any other, although the SWT and FTN routines appear to be marginally more accurate

than the F77 version.

Tests of special arguments, however, reveal some difficulties. The FTN and F77 double precision functions generate overflow faults when presented with a large enough argument. There is no valid mathematical reason for this to occur. Additionally, the Prime double precision functions calculated incorrect square roots for selected small values near the limits of storage precision. The SWT library routine behaved correctly for all special arguments.

Tangent and Cotangent

There is no tangent routine in the standard FTN library, so the results of the tests below apply to only the F77 and SWT libraries.

Test 1
TAN(X) vs. 2*TAN(X/2)/(1-TAN(X/2)**2) in (0.00, 0.7854)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	1978	2361	661	1.99 of 23	0.2458	0.22 of 23
550	SWT	2054	2518	428	1.97 of 23	0.1273	0.11 of 23
750	F77	1968	2369	663	1.99 of 23	0.2458	0.21 of 23
750	SWT	2044	2525	431	1.79 of 23	0.5237	0.11 of 23
550*	F77	191	1085	3724	3.43 of 47	0.7483	1.93 of 47
550*	SWT	190	996	3814	3.62 of 47	0.7734	2.03 of 47
750*	F77	439	2565	1996	2.79 of 47	0.2815	0.99 of 47
750*	SWT	542	2384	2074	2.87 of 47	0.2678	1.11 of 47

Test 2
TAN(X) vs. 2*TAN(X/2)/(1-TAN(X/2)**2) in (2.749, 3.534)

		5000 Comparisons			Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	2318	2018	664	2.21 of 23	2.9813	0.48 of 23
550	SWT	991	3178	831	1.17 of 23	3.0306	0.00 of 23
750	F77	2340	2009	651	2.09 of 23	2.8026	0.49 of 23
750	SWT	987	3176	837	1.17 of 23	3.0306	0.00 of 23
550*	F77	3715	815	470	3.88 of 47	3.1342	2.09 of 47
550*	SWT	3827	868	305	3.87 of 47	3.1116	2.00 of 47
750*	F77	2197	1870	933	2.79 of 47	2.9978	1.07 of 47
750*	SWT	2131	2213	656	2.60 of 47	3.4601	0.83 of 47

Test 3
TAN(X) vs. 2*TAN(X/2)/(1-TAN(X/2)**2) in (18.85, 19.63)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	1933	2374	693	1.93 of 23	19.332	0.20 of 23
550	SWT	2074	2467	459	1.94 of 23	19.104	0.14 of 23
750	F77	1934	2374	692	1.96 of 23	19.102	0.20 of 23
750	SWT	2071	2470	459	1.94 of 23	19.104	0.14 of 23
550*	F77	193	1136	3671	3.55 of 47	19.448	1.93 of 47
550*	SWT	178	1076	3746	3.59 of 47	19.541	2.03 of 47
750*	F77	399	2583	2018	2.94 of 47	19.104	0.99 of 47
750*	SWT	499	2403	2098	2.92 of 47	18.981	1.10 of 47

Test 4
COT(X) vs. (COT(X/2)**2-1)/(2*COT(X/2)) in (18.85, 19.63)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	F77	2602	16	2382	2.16 of 23	19.377	0.18 of 23
550	SWT	2311	32	2657	1.36 of 23	19.086	0.00 of 23
750	F77	2593	8	2399	2.16 of 23	19.377	0.18 of 23
750	SWT	2307	13	2680	1.35 of 23	19.086	0.00 of 23
550*	F77	261	818	3921	3.91 of 47	18.857	2.20 of 47
550*	SWT	335	772	3893	3.79 of 47	19.455	1.95 of 47
750*	F77	973	1843	2184	3.00 of 47	19.439	1.13 of 47
750*	SWT	989	1794	2217	2.53 of 47	19.616	0.73 of 47

These tests show that both implementations are correct to within a reasonable error bound. Tests on special arguments revealed that the double precision F77 tangent routine signals an error for a large input value that should be well within the range that can be dealt with.

Hyperbolic Tangent

There does not appear to be a double precision hyperbolic tangent routine in the FTN library, although there is a single precision version. The following test results reflect that fact.

Test 1
TANH(X) vs. (TANH(X-1/8)*TANH(1/8))/(1+TANH(X-1/8)*TANH(1/8)) in (0.125, 0.5493)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	1396	1788	1816	2.99 of 23	0.1268	0.72 of 23
550	F77	1860	1253	1887	3.71 of 23	0.1347	1.41 of 23
550	SWT	1203	2833	964	1.77 of 23	0.1479	0.00 of 23
750	FTN	1401	1782	1817	2.99 of 23	0.1268	0.73 of 23
750	F77	1863	1248	1889	3.71 of 23	0.1347	1.41 of 23
750	SWT	1200	2832	968	1.77 of 23	0.1479	0.00 of 23
550*	F77	2731	230	2039	6.64 of 47	0.1315	4.08 of 47
550*	SWT	4624	348	28	3.55 of 47	0.1288	1.98 of 47
750*	F77	2380	605	2015	4.83 of 47	0.1328	2.58 of 47
750*	SWT	3966	957	77	2.99 of 47	0.1270	1.33 of 47

Test 2
TANH(X) vs. (TANH(X-1/8)*TANH(1/8))/(1+TANH(X-1/8)*TANH(1/8)) in (0.6743, 17.33)

		5000	Comparisons		Maximum Rel. Error		Root Mean Sq.
CPU	*Library*	*gt*	*eq*	*lt*	*Bitloss*	*At*	*Bitloss*
550	FTN	1103	2707	1190	1.69 of 23	0.7217	0.00 of 23
550	F77	1288	2316	1396	1.91 of 23	1.0990	0.00 of 23
550	SWT	1204	2324	1472	1.73 of 23	0.6974	0.00 of 23
750	FTN	1100	2704	1196	1.69 of 23	0.7217	0.00 of 23
750	F77	1281	2324	1395	1.91 of 23	1.0990	0.00 of 23
750	SWT	1198	2328	1474	1.73 of 23	0.6974	0.00 of 23
550*	F77	1846	2234	920	3.34 of 47	0.8543	0.86 of 47
550*	SWT	2676	2258	66	2.11 of 47	1.6430	0.62 of 47
750*	F77	974	3464	562	2.26 of 47	0.7330	0.14 of 47
750*	SWT	1185	3442	373	1.76 of 47	1.3987	0.14 of 47

The above tests show that any of the three routines is acceptable for use in single precision, but the error in the double precision F77 routine in test 1 is rather large. The SWT routine is once again the best.

Tests of special arguments indicate a definite problem in the Prime single precision library routines when calculating various identity operations such as TANH(-X) = -TANH(X). The difference in calculated values is about 10E-6; the SWT routine calculates no differences.

Conclusions

It appears as if the standard libraries under Primos have been implemented without anything other than a cursory check of accuracy. A number of the library routines return incorrect results that are mathematically absurd. Other routines trigger errors on values which should be well within range.

Although the single precision arithmetic is acceptable for most calculations, the double precision floating point arithmetic on Prime 400/550 machines (and possibly on the new 2250, as well) is seriously flawed. Critical calculations should not be performed on any of these machines since the error induced by certain unstable operations can completely ruin the accuracy of the results. Bizarre behavior of programs which work on other machines may also be noted due to some of the odd quirks in the floating point structure. Users should run their own tests to determine if their applications will be affected adversely by these problems.

An increase in accuracy may very well be obtained in some programs by recoding the standard functions. It has been shown that the SWT Math Library significantly outperforms the standard Prime libraries in virtually every instance; it is possible that the encoding of different algorithms might also result in increased precision.

This paper has also presented differences in the architecture of the Prime 400/550 computer and the 750 which violate the claim of strict upward compatibility of software. Programs which directly access the register structure or make specific assumptions about precision should be coded with these differences in mind.

References

{1} Dr. John Spitzer; private communication to Academic Computing Center, State University College at Brockport, NY, and reported to Prime Computer; 1978

{2} Mark P. C. Legg; copies of TAR reports to Prime computer dated 1980 to 1982 from The Flinders University of South Australia; private communication; 1982

{3} Harold Stone, editor; *Introduction to Computer Architecture, 2nd. Edition*; Science Research Associates, Inc; 1980

{4} Andrew Tanenbaum; *Structured Computer Organization*; Prentice-Hall; 1976

{5} M. Sporer; Prime PE-T 416, "P400 Instruction Times"; Prime Computer, Inc.; 1978

{6} Mark P. C. Legg; untitled report contained in private communication; Computer Centre of The Flinders University of South Australia, Bedford Park; 1982

{7} William J. Cody, Jr. and William Waite; *Software Manual for the Elementary Functions*; Prentice-Hall; 1980

{8} George S. Fishman and Louis R. Moore; *A statistical Evaluation of Multiplicative Congruential Random Number Generators with Modulus 2 ^ 31 - 1*; Journal of the American Statistical Association, March 1982, Volume 77 # 377

{9} Martha August; Prime PE-T 1025, "50 Series General Architecture"; Prime Computer, Inc.; 1982

{10} Anne P. Ladd; *Subroutines Reference Guide*; Doc 3621-190, Revision 19.0; Prime Computer, Inc.; 1982

{11} Ivars Peterson; "Can You Count on Your Computer"; Science News, Vol. 122 #5; Jul 31, 1982

{12} W. J. Cody; "Analysis of Proposals for the Floating-Point Standard"; IEEE Computer, Volume 14 #3, March 1981

{13} David Hough; "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic"; IEEE Computer, Volume 14 #3, March 1981

{14} Jerome T. Coonen; "Underflow and the Denormalized Numbers"; IEEE Computer, Volume 14 #3, March 1981

{15} A Proposed Standard for Binary Floating-Point Arithmetic;
IEEE Computer, Volume 14 #3, March 1981

Appendix I

Where is the Exponent?

The following program is written in PMA (Prime Assembly Language) and is intended to indicate where the exponent is stored in the live register set of your machine. It can be entered and run without the Software Tools Subsystem being present on your system.

```
*  EXPTEST --- SEE WHERE DOUBLE FLOATING EXPONENT IS LOCATED
*
*     Eugene Spafford
*     School of Information and Computer Science
*     Georgia Institute of Technology
*     Atlanta, GA 30332
*
*
*  To assemble, load and run this test, copy lines 16 to 31
*  into a file named "exptest.cpl" and remove the "*" from
*  the first column of each line.  Then type (in Primos):
*
*       cpl  exptest
*
*
* /* exptest.cpl --- assemble, load and run the exponent test
*
* pma exptest.pma -l yes -b yes
*
* &data seg
* vload exptest.seg
* load exptest.bin
* library
* map 6
* save
* quit
* &end
*
* seg exptest.seg
*
* &stop
*
*
            SEG
            SUBR     MAIN

            LINK
MAIN        ECB      START
            PROC
```

```
START       EQU     *
            DFLD    =2.5D0
            LDLR    PB% + '13
            BNE     HIGH_HALF

            CALL    IOA$
            AP      LOWM,S
            AP      =99,SL
            PRTN

HIGH_HALF   EQU     *
            CALL    IOA$
            AP      HIGHM,S
            AP      =99,SL
            PRTN


LOWM        BCI     'Exponent is in the low half (2nd 16 bits).%.'
HIGHM       BCI     'Exponent is in the high half (1st 16 bits).%.'

            END     MAIN
            SEG
            DYNT    IOA$
            END
```

Appendix II

A Program to Detect Bit Loss in Multiplication

The following program is written in Prime Fortran 66 (FTN) and is intended to indicate whether multiplication on your machine truncates or deletes bits in the mantissa of products of double precision floating point quantities. It can be entered and run without the Software Tools Subsystem being present on your system.

```
C     CHECK_DFMP --- SEE IF DOUBLE PRECISION MULTIPLY DROPS BITS
C
C     Eugene Spafford
C     School of Information and Computer Science
C     Georgia Institute of Technology
C     Atlanta, GA 30332
C
C
C     To compile, load and run this test, copy lines 16 to 31
C     into a file named "check_dfmp.cpl" and remove the "C" from
C     the first column of each line.  Then type (in Primos):
C       cpl  check_dfmp
C
C
C
C /* check_dfmp.cpl --- compile, load and run the test to check DFMP
C
C ftn check_dfmp.ftn -l yes -b yes -64v -dynm -dclvar -prod
C
C &data seg
C vload check_dfmp.seg
C load check_dfmp.bin
C library
C map 6
C save
C quit
C &end
C
C seg check_dfmp.seg
C
C &stop
C
C
C
      INTEGER BITCNT
C
      DOUBLE PRECISION DA,DB,DC
      INTEGER IDB(4),IDC(4)
      EQUIVALENCE (IDB,DB),(IDC,DC)
```

```
C
      INTEGER LOOP,COMPAR,LOSS,IX
      DOUBLE PRECISION DCON(3)
      DATA DCON /1.0D0,16.0D0,0.125D0/
C
C
      IDB(1) = :77777
      IDB(2) = :177777
      IDB(4) = 128
C
      DO 30 IX = 1,3
        DA = DCON(IX)
        IDB(3) = 0
        DO 20 LOOP = 1,16
          IDB(3) = IDB(3)*2+1
          DC = DA*DB
          DO 10 COMPAR = 1,3
            IF (IDC(4-COMPAR) .EQ. IDB(4-COMPAR)) GO TO 10
            PRINT 70, DA
            PRINT 90, COMPAR,IDC(4-COMPAR),IDB(4-COMPAR)
            LOSS = BITCNT(IDC(4-COMPAR),IDB(4-COMPAR),COMPAR)
            PRINT 100, LOSS
            GO TO 20
   10     CONTINUE
   20   CONTINUE
   30 CONTINUE
C
C
      DO 60 IX = 1,3
        DA = DCON(IX)
        IDB(3) = 0
        DO 50 LOOP = 1,16
          IDB(3) = IDB(3)*2+1
          DC = DB/DA
          DO 40 COMPAR = 1,3
            IF (IDC(4-COMPAR) .EQ. IDB(4-COMPAR)) GO TO 40
            PRINT 80, DA
            PRINT 90, COMPAR,IDC(4-COMPAR),IDB(4-COMPAR)
            LOSS = BITCNT(IDC(4-COMPAR),IDB(4-COMPAR),COMPAR)
            PRINT 100, LOSS
            GO TO 50
   40     CONTINUE
   50   CONTINUE
   60 CONTINUE
C
C
      CALL EXIT
C
C
   70 FORMAT ('Loss of precision multiplying by ',F10.6)
   80 FORMAT ('Loss of precision dividing by ',F10.6)
   90 FORMAT ('Word ',I2,' is ',I6,' and should be ',I6)
  100 FORMAT ('Result is loss of ',I3,' bits out of 47.'//)
      END
C
C
```

```
C     BITCNT --- FIGURE LOSS OF BITS
C
      INTEGER FUNCTION BITCNT(I,J,COMPAR)
C
      INTEGER I,J,COMPAR
C
      INTEGER COUNT,AND,MASK,RS
C
C
      MASK = :100000
      DO 20 COUNT = 1,16
        IF (AND(MASK,I) .EQ. AND(MASK,J)) GO TO 10
        BITCNT = (COMPAR-1)*16+17-COUNT
        RETURN
   10   CONTINUE
        MASK = RS(MASK,1)
   20 CONTINUE
C
      BITCNT = 0
      RETURN
      END
```

Appendix III

A Program to Calculate Prime Hexadecimal Constants

The following program is written in Fortran 77 and can be used to generate Prime PMA-style hexadecimal constants from decimal inputs. The version included here was run on a Cyber 760 under NOS 2.0 to generate the constants used in the SWT Math Library. To be used effectively, if you use this program you should run it on a machine with more precision than the Primes provide.

```
C     MAKE_CONSTANT --- MAKE THE HEX CONSTANTS FOR THE LIBRARY
C
C     The following PROGRAM line is for FTN5 on the Cyber 760
C
      PROGRAM MAKCON (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
C
      DOUBLE PRECISION INP,HALF,TWO,ZERO,ONE
      LOGICAL BITS(0:47)
      INTEGER I,ISIGN,EXP,J
      PARAMETER (ZERO=0.0D0,TWO=2.0D0,HALF=0.5D0,ONE=1.0D0)
      EXTERNAL PUTHEX,PUTHX2
      INTRINSIC DINT
      DOUBLE PRECISION DINT
C
C
   10 CONTINUE
      READ (5,*,END=70) INP
      IF (INP .NE. ZERO) THEN
          ISIGN = 1
          IF (INP .LT. ZERO) THEN
              ISIGN = -1
              INP = -INP
          ENDIF
C
C     START WITH 128 BIAS EXPONENT
          EXP = 128
   20     CONTINUE
          IF (INP .LT. HALF) THEN
              INP = INP*TWO
              EXP = EXP-1
              GO TO 20
C
          ELSEIF (INP .GE. ONE) THEN
              INP = INP/TWO
              EXP = EXP+1
              GO TO 20
          ENDIF
C
```

```
      ELSE
          ISIGN = 1
          EXP = 0
      ENDIF
C
      DO 30 I = 1,47
        IF (DINT(INP*TWO) .GT. ZERO) THEN
            BITS(I) = .TRUE.
            INP = INP*TWO-ONE
C
        ELSE
            BITS(I) = .FALSE.
            INP = INP*TWO
        ENDIF
   30 CONTINUE
C
      IF (INP .GE. HALF) THEN
          I = 47
   40     CONTINUE
          BITS(I) = .NOT.BITS(I)
          I = I-1
          IF ( .NOT. BITS(I+1) .AND.
     &         I .GT. 0) THEN
               GO TO 40
          ELSE IF ( .NOT. BITS(I+1)) THEN
               BITS(1) = .TRUE.
               EXP = EXP+1
          ENDIF
      ENDIF
C
C     NOW GENERATE THE 2'S COMPLEMENT IF NEGATIVE
      IF (ISIGN .LT. 0) THEN
          I = 47
   50     CONTINUE
          I = I-1
          IF ( .NOT. BITS(I+1) .AND.
     &         I .GT. 0) GO TO 50
          DO 60 J = 1,I
            BITS(J) = .NOT.BITS(J)
   60     CONTINUE
          BITS(0) = .TRUE.
C
      ELSE
          BITS(0) = .FALSE.
      ENDIF
C
      CALL PUTHEX (BITS(0))
      CALL PUTHEX (BITS(16))
      CALL PUTHEX (BITS(32))
      CALL PUTHX2 (EXP)
      GO TO 10
C
C
   70 CONTINUE
      STOP
      END
```

```
C     PUTHEX --- PUT OUT A HEXADECIMAL VALUE
C
      SUBROUTINE PUTHEX (BITARR)
C
      LOGICAL BITARR(16)
C
      INTEGER I,J,VAL
      CHARACTER*16 DIGITS
      CHARACTER*4 NUM
      DATA DIGITS /'0123456789ABCDEF'/
C
C
      DO 20 I = 1,4
        VAL = 0
        DO 10 J = 1,4
          VAL = VAL*2
          IF (BITARR((I-1)*4+J)) THEN
              VAL = VAL+1
          ENDIF
   10   CONTINUE
        VAL = VAL+1
        NUM(I:I) = DIGITS(VAL:VAL)
   20 CONTINUE
C
      WRITE (6,30) NUM
      RETURN
C
   30 FORMAT (A4)
      END
C     PUTHX2 --- PUT OUT A HEXADECIMAL VALUE
C
      SUBROUTINE PUTHX2 (EXP)
C
      INTEGER EXP
C
      INTEGER DIG,VAL,POWER2(4),LOOP
      LOGICAL ISNEG
      CHARACTER*17 DIGITS
      CHARACTER*4 NUM
      DATA DIGITS /'0123456789ABCDEF0'/
      DATA POWER2 /4096,256,16,1/
C
C
      VAL = EXP
      IF (EXP .LT. 0) THEN
          VAL = -EXP
          ISNEG = .TRUE.
          VAL = VAL-1
C
      ELSE
          ISNEG = .FALSE.
      ENDIF
C
      DO 10 LOOP = 1,4
        DIG = VAL/POWER2(LOOP)
        VAL = VAL-DIG*POWER2(LOOP)
```

```
         IF (ISNEG) DIG = 15-DIG
         DIG = DIG+1
         NUM(LOOP:LOOP) = DIGITS(DIG:DIG)
   10 CONTINUE
C
      WRITE (6,20) NUM
      RETURN
C
   20 FORMAT (A4/)
      END
```

Appendix IV

Building The SWT Math Library Tests

In General

The tests provided along with the SWT Math library may be recompiled and run on your machine to test your own routines or verify the results presented in this report. The tests are written in Fortran 77 and Fortran 66 with command files in Prime CPL. Consult your system administrator to find where the tests have been stored on disk; the default location is with the source code to the SWT Math Library routines. The single precision tests are in a separate directory from the double precision tests, but the directions given below apply to both sets of tests.

You must have the Software Tools Subsystem and the F77 compiler to run the tests! You can recode the routines written in F77 to either Ratfor or FTN, but be aware of the library that is used when you choose this option!

Make sure that the SWT Math Library has been built and installed in a directory where you can access it. Set a SWT template in your account named "=mathlib=" and equal to the SWT pathname to the library. The format to do this is:

```
template -a mathlib //<some path name here>/mathlib
```

Building the Support Routines

Attach to the directory containing the tests you wish to build and run. Modify the "subs.f77" file, if necessary, to change the library routines to be tested. The routines in the "subs.f77" file which begin with the letter Z are the routines to modify to invoke the correct library functions.

Next, you need to build the support routines. To do this, simply run the SWT shell file "make_support". This will cause the files "main.b" and "sublib" to be created in your account.

Next, edit the file "run_test.cpl" so that any necessary local libraries get loaded along with the tests. Also include any special commands that you might wish to execute as part of the tests.

Running a Test

If you execute the SWT shell file "make" with the name of a test to run (asin, atan, exp, log, power, sqrt, sin, sinh, tan,

or tanh) the SWT shell files and associated Prime CPL files will compile and load the appropriate test programs, execute them with output captured to comoutput files, and then produce a file with labelled results and a report generated by CMPF. The file created will be named after the test executed, with the string ".comparison" added to the end of the name. For example, if you executed the command

```
make power
```

the file "power.comparison" would be created in your account.

If you wish to make further modifications to the test software, examine the SWT shell files and CPL files to determine what needs to be modified.

ADDENDUM

Arnold D. Robbins

August, 1984

Introduction

For Release 9 of the Software Tools Subsystem, in order that there should only be one math library, the old, locally supported, math library, "vswtml", has been merged with the new library described in this report, "vswtmath". This addendum describes these routines.

Deleted Functions

The functions *dacos*, *dasin*, *dbexp*, *dbsqrt*, *dflot*, and *drand*, have all been deleted from "vswtml", since there are new routines to take their places.

Remaining Routines

The following pages contain the Software Tools Reference Manual entries for the remaining routines which have been added to "vswtmath" from "vswtml".

Note that although the original "vswtmath" routines are listed in Section 2 of the SWT Reference Manual, these routines are listed in Section 4, even though they are all in one library.

gcd (4) --- determine greatest common divisor of two integers 07/20/84

Calling Information

```
long_int function gcd (x0, x1)
long_int x0, x1
```

Library: vswtmath (Subsystem mathematical library)

Function

'Gcd' determines the greatest common divisor of the two long integers specified as arguments. The function return is the GCD (always positive).

Implementation

'Gcd' is a straightforward implementation of Euclid's algorithm.

Bugs

Behavior with nonpositive arguments may be considered irrational by some.

See Also

invmod (4)

invmod (4) --- find inverse of an integer modulo another integer 07/20/84

Calling Information

```
long_int function invmod (x1, x0)
long_int x1, x0
```

Library: vswtmath (Subsystem mathematical library)

Function

'Invmod' is used to find the inverse of 'x1' in the ring of integers modulo 'x0'. The function return is the inverse if it could be found, or ERR if 'x1' and 'x0' are not relatively prime.

Implementation

'Invmod' uses a variant of Euclid's greatest common divisor algorithm.

Bugs

Rational behavior for nonpositive arguments has not been established.

Locally supported.

See Also

gcd (4)

Calling Information

```
long_int function prime (i)
long_int i
```

Library: vswtmath (Subsystem mathematical library)

Function

'Prime' is used to retrieve a specified prime number. The argument is the ordinal of the prime number desired. The function return is the specified prime. For example, if 'i' is 1, the function return is 2; if 'i' is 3, the function return is 5, etc.

'Prime' uses the table of prime numbers in the file "=aux=/primes". This file contains the prime numbers up to one million in long-integer binary format. If "=aux=/primes" is unreadable or if 'i' is less than one or greater than 78498, the function return is zero.

Implementation

The file "=aux=/primes" is opened for reading. The read/write pointer for the file is then moved to the desired location and the prime number read. The file is then closed.

Calls

open, close, mapfd, Primos prwf$$

Bugs

Should probably raise cain if the prime numbers file is not available, rather than meekly returning zero.

Locally supported.

pwrmod (4) --- calculate an exponential modulo a given modulus 07/20/84

Calling Information

```
long_int function pwrmod (p, e, n)
long_int p, e, n
```

Library: vswtmath (Subsystem mathematical library)

Function

'Pwrmod' is used to perform an integer exponentiation in the ring of integers modulo a given modulus. The argument 'p' is the base of the expression, 'e' is the exponent, and 'n' the modulus. The function return is p**E (mod n).

Implementation

'Pwrmod' examines the exponent a bit a time, squaring the intermediate result accumulated so far and multiplying it by the base whenever the selected bit is a 1. Each operation is performed modulo 'n', so that intermediate results don't become excessively large.

See Also

invmod (4)

Calling Information

```
subroutine set_copy (source, destination)
pointer source, destination
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_copy' duplicates one set in another. For proper operation, the source set should be larger than or equivalent in size to the destination set. The source set is not altered by the copy operation.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

'Set_copy' uses the size field encoded in the first word of each set to determine the number of words in the bit vector to be copied. A simple loop implements the copy.

Bugs

Should handle sets of different sizes properly.

See Also

other set operations ('set_?*') (4)

set_create (4) --- generate a new, initially empty set 07/20/84

Calling Information

```
pointer function set_create (set, size)
pointer set
integer size
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_create' is used to create a Pascal-style bit vector representation for a set of integers from 1 to 'size'. The function return and the variable 'set' are set to the address in dynamic storage of the newly-created set.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

'Set_create' calls 'dsget' to obtain a contiguous array of 16-bit words that is large enough to represent a bit vector with 'size' elements. The first word of this array is set to 'size' for use by other set manipulation routines. A call to 'set_init' then insures that the new set is empty.

Arguments Modified

set

Calls

dsget, set_init

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_delete (element, set)
integer element
pointer set
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_delete' is used to remove a given element from a set. The first argument is the element (an integer between one and the maximum set size, inclusive), and the second is the set from which it is to be removed.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

The element selected is compared to the size field of the set; if invalid, 'set_delete' prints an error message and terminates the program. Otherwise, the position of the element in the bit vector is calculated, and the bit is reset by straightforward logical operations.

Calls

error

See Also

other set operations ('set_?*') (4)

Calling Information

```
integer function set_element (element, set)
integer element
pointer set
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_element' returns 1 if 'element' is a member of the set 'set', 0 otherwise. The argument 'element' must be an integer from 1 to the maximum size of the set, inclusive. The argument 'set' must have been created beforehand with 'set_create'.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

If 'element' is not in the range of allowable set elements for the given set, the program is terminated by a call to 'error'. Otherwise, the location of the element in the bit vector is calculated, and the function returns the value of the bit at that position.

Calls

error

See Also

other set routines ('set_?*') (4)

Calling Information

```
logical function set_equal (set1, set2)
pointer set1, set2
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_equal' determines if two sets contain the same members. The sets need not be of equal length.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

'Set_equal' makes two calls on 'set_subset'. The function return is true if 'set1' is a subset of 'set2' and 'set2' is a subset of 'set1', false otherwise.

Calls

set_subset

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_init (set)
pointer set
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_init' initializes a set created by 'set_create'. An initialized set is empty, i.e. contains no members.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

'Set_init' simply clears all elements of the bit vector portion of the data structure addressed by its first argument.

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_insert (element, set)
integer element
pointer set
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_insert' is the primary means of placing a given element in a set. 'Element' must be an integer between one and the maximum size of the set, inclusive; 'set' must be a pointer to a set data structure created by 'set_create'. If it is within range, the given element is marked "present" in the bit vector associated with the set.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

If the element is out of range, a call to 'error' is made to inform the user and terminate the program. Otherwise, the location of the element in the bit vector is determined and a few logical operations are employed to set the selected bit.

Calls

error

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_intersect (set1, set2, destination)
pointer set1, set2, destination
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_intersect' determines the intersection of the sets given as its first two arguments and places that intersection in the set specified by the third. For proper operation, all three sets should be equal in size.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

Does a word-by-word logical 'and' of the bit vectors for the first two sets, placing the result in the third.

Bugs

Should be fixed to work with sets of differing lengths.

See Also

other set routines ('set_?*') (4)

set_remove (4) --- remove a set that is no longer needed 07/20/84

Calling Information

```
subroutine set_remove (set)
pointer set
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_remove' reclaims the dynamic storage space used by a set data structure. It is the inverse of 'set_create'. To prevent dynamic storage space from becoming irretrievably lost, sets should always be removed by a call to 'set_remove' when they are no longer needed.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

Calls 'dsfree' to throw away the storage space used by the internal data structure.

Calls

dsfree

See Also

other set routines ('set_?*') (4), dsinit (2), dsget (2), dsfree (2)

Calling Information

```
logical function set_subset (set1, set2)
pointer set1, set2
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_subset' returns the logical value '.true.' if and only if its first argument points to a set that is a subset of or equal to the set pointed to by its second argument. The sets need not be of equal length.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

If one set is larger than the other, it is checked to make sure that none of the higher-order elements is present. The subset condition is then true if and only if every element of 'set1' is also an element of 'set2', a statement which can be checked a word at a time with the proper logical operations.

Calls

set_element

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_subtract (set1, set2, destination)
pointer set1, set2, destination
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_subtract' performs the set subtraction operation, i.e. places in the set 'destination' those elements of 'set1' that are not in 'set2'. For proper operation, all three sets should be the same size.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

Since sets are represented as bit vectors, the subtraction operation is performed by logically 'and'ing the elements of the first set with the negation of the elements of the second set.

Bugs

Should work with sets of differing sizes.

See Also

other set routines ('set_?*') (4)

Calling Information

```
subroutine set_union (set1, set2, destination)
pointer set1, set2, destination
```

Library: vswtmath (Subsystem mathematical library)

Function

'Set_union' computes the union of 'set1' and 'set2', placing the result in 'destination'. For proper operation, all three sets should be the same size.

All set manipulation routines make use of dynamic storage, which must be initialized before use. See 'dsinit' for further information.

Note that all set manipulation routines have long names. To avoid unique name conflicts with other routines, any Ratfor program using the set routines should include the following statement:

```
include "=src=/lib/math/swtmlb_link.r.i"
```

Implementation

The set union is computed by logically 'or'ing the bit vectors associated with 'set1' and 'set2'.

Bugs

Should work with sets of differing sizes.

See Also

other set routines ('set_?*') (4)

TABLE OF CONTENTS

The Hardware

The SWT Math Library

Testing

Appendix I

Appendix II

Appendix III

Appendix IV

ADDENDUM

Ring

The Software Tools Subsystem Network Utility
Version 1.0

Roy J. Mongiovi

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

April, 1983

Ring

Introduction

Ring is a distributed request server for the Software Tools Subsystem which uses PRIMENET to communicate between nodes in a distributed ring. It performs simple system functions such as keeping the time of day synchronized on all the machines in the ring, as well as accepting user requests for services. It validates all requests it receives, which ensures that a devious user cannot create his own Ring server and transmit invalid requests to the other Ring processes.

One copy of the Ring process executes on each of the systems in the ring. Each process establishes two virtual circuits (a transmit and a receive circuit) with the next and previous systems, where next and previous are defined by the system names in lexically sorted order. As systems are brought up and down, the ring dynamically reconstructs itself to maintain that ordering. A user who wishes to make a request of the ring connects to the Ring process on his own system and transmits his request. That Ring process reformats the request and transmits it around the ring where it is eventually seen and acted upon by the Ring process to which it was addressed.

Validation

There are two distinct types of connection request validation performed by Ring. The first is the validation of virtual circuits connecting each of the Ring processes in the ring, and the second is the validation of a virtual circuit connection from a user to the Ring process. These two types of validation are distinguished by the fact that ring connections are normally between two systems, while user connections are restricted to the same system (that is, a user is not allowed to connect to a Ring process on another system).

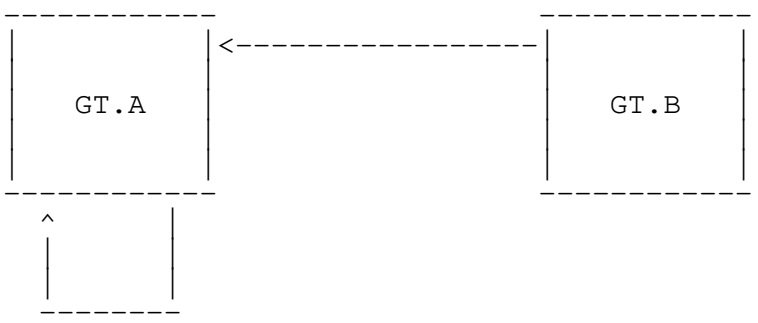
Validation is made difficult by the fact that it is impossible to determine the user name (or any other information) of the process on the other end of a virtual circuit. Information may be returned only for virtual circuits on the current system, and even then only for known virtual circuits. As we shall see, it is possible to find the user name of the process on the other end of a circuit given certain restrictions. In fact, the entire purpose of user validation is to determine the user name and process id of the process on the other end of a virtual circuit.

Ring Connections

When a Ring process attempts to break into a previously existing ring (i.e. when a system has been down and is being brought up), and when a system that was in the ring has gone down, the new connections must be validated before they are accepted as coming from a Ring process. It would be very simple if a user name (such as SYSTEM) could be checked, but as has already been mentioned it is impossible to determine the user name on the other end of a virtual circuit that is on another system. The only piece of information that can be used for validation that is assured by the PRIMENET routines is the fact that a port can be assigned by only one process. Using this fact, together with the assumptions that the Ring process will be started at boot time, will immediately assign its ports, and will never relinquish those ports as long as the system is up, it is possible to validate ring connections. Note that this assumes that Ring will never fail on a hardware/software error, a rather stringent requirement. Should Ring ever fail and unassign the validation port while the system is up, it would be possible for another user process to assign that port and become the Ring process for that system.

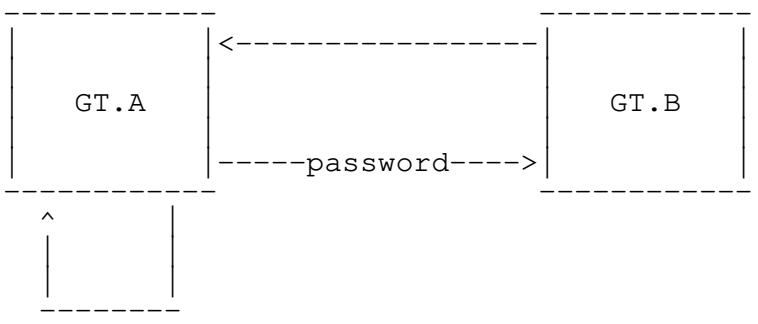
When a Ring process begins execution, the first thing it does is assign three ports: a ring port, a validation port, and a user port. These ports are never unassigned. It then determines all system names, sorts them, and begins attempting to connect to an already existing ring starting with the next system (in the sorted list). Should it be the first Ring process, it will eventually connect to itself and establish the initial degenerate ring. Validation of that connection proceeds as follows:

When a Ring process detects a connection request to its ring port, it accepts it provisionally and then attempts to validate it.



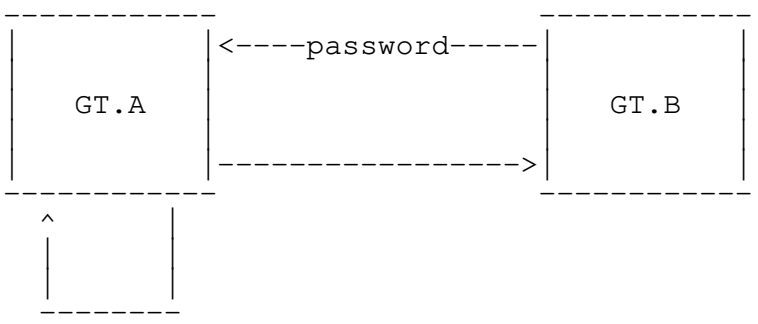
1. The new Ring process makes a connection request.

The Ring process makes a connection request to the validation port on the system from which the ring connection was received. When that connection is accepted, it generates a random number password and transmits it to the validation circuit.



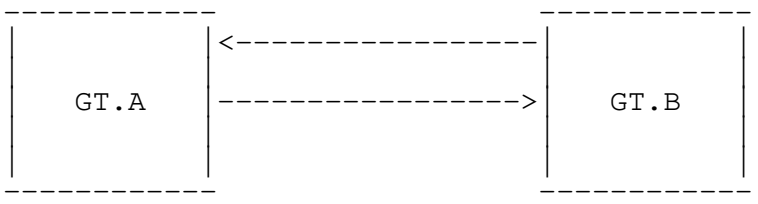
2. The validation password is transmitted.

If the ring connection is indeed valid, then the validation connection is to the same process that issued the ring connection. The password is then received and retransmitted to the ring circuit.



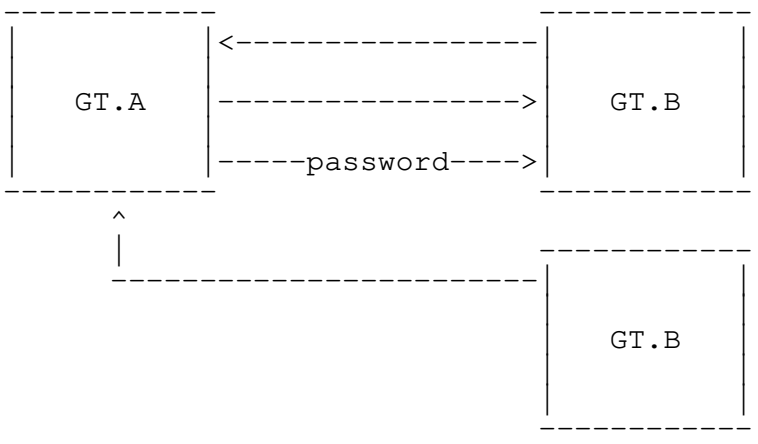
3. The response password is retransmitted.

The Ring process that is validating the connection receives that password on the circuit that is being validated, compares it with the password that was transmitted, and validates the circuit.



4. The new ring connections are established.

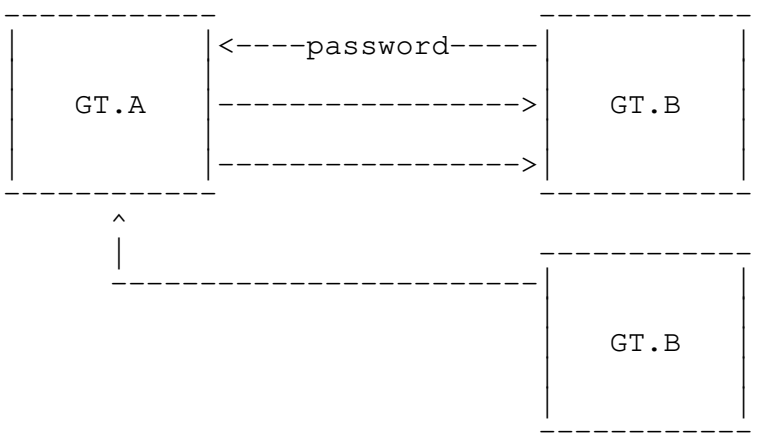
If the ring connection is from a pretender, then the validation connection is to the actual Ring process on that system, the pretender cannot receive the password, and the ring connection is not validated.



5. The false Ring process cannot receive the password.

When the actual Ring process receives the password, it transmits it through the already validated ring circuits, and when the

validating process receives it from that circuit (and not the circuit being validated) it knows that the connection attempt is not valid and clears the connection.



6. The password is received from the existing ring.

User Connections

When a user connection is received, the Ring process must determine the user name and process id of the process making the connection request in order to ensure the validity of any requests that the process may make. It is not good enough to have the user process transmit this information since that process could easily fabricate it. The ability to identify the user process hinges on the following ideas: it is possible to determine the virtual circuit numbers of all allocated virtual circuits open on a system, user connections must be from the same system as the Ring process that they are connected to, and user connections are accepted and identified one at a time.

To identify a user connection, the Ring process obtains a list of all open virtual circuits on the current system. This list is scanned to find all circuits that are to the user port, which have been accepted, and which are not the process id of the Ring process. The list of existing user connections is then scanned, and the corresponding entries in the list of virtual circuits are marked as known. Since user connections are accepted one at a time, there will be exactly one virtual circuit that was not marked as known, and that is the virtual circuit corresponding to the newly accepted user connection. The user name of that process is determined using a system call, and the connection is added to the list of known virtual circuits.

Ring Requests

All operations performed by Ring are initiated by request packets which are passed around the ring connections. Each packet has the same size and consists of two parts: a fixed identification header, and a variable argument array. The header consists of a flag that indicates whether the packet is a request or a response, source and destination addresses, a count of the number of Ring processes that have seen the packet, a process id and unique identifier to indicate what process created the packet, and the Ring request command/status words. The format of the variable argument array depends of the value of the command word in the packet header.

Ring requests are passed around the ring, from receive connection to transmit connection, until they are received by the system to which they are addressed or the number of Ring processes that have seen them is greater than the number of systems in the ring. A packet destination with all bits set (-1) is received by all Ring processes in the ring. When the request packet is performed or destroyed, it is transformed into a response packet which is transmitted to the system that created the request.

Internal Requests

When a new ring is established, as well as when an existing ring is changed because one or more systems have come up or gone down, a special request packet is transmitted around the ring. This packet, the INITIALIZE request, has two purposes. First, it is used to count the number of Ring processes that are actually in the ring. PRIMENET provides a status call which returns the number of systems configured in the network, but they may not all be running Ring. As the INITIALIZE packet goes around the ring, each Ring process increments a counter in the packet. When the request arrives back at the Ring process that created it, an INITIALIZE response packet is created which contains the number of systems that saw the original request. This response packet is then used by each Ring process to set the actual number of systems in the ring. The second purpose of the INITIALIZE request is to determine who is to set the time of day on all systems initially. Normally, the time of day is set by the first (in lexically sorted order) system that is running Ring. However, should that system be the one that caused the ring to change (i.e. it just entered the ring), it is assumed not to know the correct time, and the next system which was in the ring previously should set the time. As the INITIALIZE response is transmitted around the ring, a state variable is transmitted along with it. This variable starts as 0, when the system that is supposed to set the time of day sees the packet, it sets the state to 1 if it just entered the ring and does not know the time of day, and 2 if it does know the time of day. If the state is 1, then the next system that does know the time of day sets the state to 2 and then sets the time of day on all systems.

Each hour on the hour, the Ring process that is first in lexically sorted order transmits the current time of day to all other systems in the ring. Although this is not necessary for orderly system operation, it does make sense for each processor in a distributed system to have the same time of day.

User Requests

Currently, four kinds of user requests are implemented by Ring: a BROADCAST request which allows a PRIMOS message to be sent on all systems in the ring, an EXECUTE request which starts up a SWT phantom on a particular system in the ring, a TERMINATE request which allows one or all of the Ring processes to be stopped and re-executed (so that a new version of the Ring process may be brought up), and a SETTIME request that allows the time to be reset on all systems in the ring.

To make a user request, a user process first connects to the user port of the Ring process which is executing on its system. When the connection has been accepted, the user transmits the request and begins waiting for a response. When the Ring process has received the request and checked its validity, it transmits a status code to indicate that the operation has been initiated or that an error has been encountered back to the user process. The user process receives this status code, and if it indicates that the request has been initiated begins waiting for a completion response. When the Ring request has been completed (successfully or not), the Ring process will transmit a final status code to the user process. The user process then examines the returned status and clears the connection.

BROADCAST. The BROADCAST user request consists of three parts: the BROADCAST request word, a three word user name of the user who is to receive the message (zero if all users), and a Software Tools string which is to be broadcast.

EXECUTE. The EXECUTE user request also consists of three parts: the EXECUTE request word, a three word system name of the system on which the phantom is to be executed (zero if all systems), and a Software Tools string which is the command line to be executed.

TERMINATE. The TERMINATE user request consists of two parts: the TERMINATE request word, and a three word system name of the system which is to be terminated (zero if all systems). Because it is impossible to determine when a transmitted message has been received, the TERMINATE request actually occurs in two stages. After the user's TERMINATE request has been processed and the status response has been transmitted, an internal request (SHUTDOWN) is transmitted around the ring. It is this request which actually causes the selected Ring process(es) to terminate, thus allowing time for the user process to receive its status.

SETTIME. The SETTIME user request consists of two parts: the SETTIME request word, and a five word block which contains

the month, day, year, hour, and minute to which the current time is to be set.

Future Requests

Ring is intended to handle simple requests by itself. A simple request is defined as one which would require no more than one request and response packet to perform. In the future, it is envisioned that complex requests such as remote execution of commands and remote file handling will be performed by a helper phantom which the Ring process will create and which will then be connected directly to the requesting user. Ring can also be used to moderate interprocess communication by allocating ports and controlling access to those ports. This will allow two or more user processes to communicate without requiring fixed port numbers which may be used by other user processes with which communication is not desired.

The major drawback with this scheme of creating helper phantoms is the relatively large amount of time required to create a phantom. In fact, when PRIME itself decided to replace the old FAM (the File Access Manager) with a new version which uses SLAVE$ helper phantoms, it was necessary to special-case the SLAVE$ phantoms so that they would start up more quickly.

PRIMENET Problems

During the development of Ring, only one significant error was found, and that was in the PRIMENET documentation. However, quite a bit of code in Ring is devoted to determining information that should most likely be available directly from the PRIMENET subroutines. Several enhancements to the existing routines come easily to mind.

Errors

The only problem with PRIMENET that may be classified as an error is in the documentation for the message transmission subroutine X$TRAN. The following information about the return status codes (taken directly from the PRIMENET manual) is not correct:

> The codes that may be returned in status by a call to X$TRAN appear below:
>
> XS$CMP The transmit is complete. The message has been copied out of the sender's buffer and transmission is initiated. (A transmit status of complete means only that PRIMENET

will attempt to deliver the message. Applications requiring assured delivery must implement their own end to end acknowledgement.)

XS$IP The transmit is in progress. *status* will be further updated by the completion or failure of the operation.

XS$BVC The calling process does not control the virtual circuit specified in *vcid*.

XS$MEM Temporary PRIMENET congestion prevents the acceptance of the request at this time.

XS$MAX The maximum number of transmits simultaneously in progress over a single virtual circuit has been exceeded. This request to initiate another transmission is denied.

XS$RST The virtual circuit has been reset. The status of this operation is unknown and no further attempts will be made to complete it.

XS$CLR The virtual circuit has been cleared. See the virtual circuit status array for the clearing cause.

XS$ILL The transmit operation is illegal because a circuit connection request or a clear request is pending. This is the result of attempting transmission over an "almost-open" or "almost-closed" circuit.

The description of status codes XS$CMP, XS$MEM, and XS$MAX seems to indicate that once a transmit operation is in progress it must either complete or return an error code. In fact, this is not the case. If too many transmit requests have been issued on a virtual circuit, the status code remains XS$IP until enough receives have been performed to allow the transmit to take place. In its example programs, the PRIMENET manual gives a subroutine which is called after a transmit to wait until the transmit status is not "in progress". In ratfor, this subroutine is essentially:

```
subroutine complete(status)
integer status
```

```
while (status == XS$IP)
   call x$wait(1)
return
end
```

The real difficulty with the documentation is with an application like Ring, when only one system is in the ring. In this case the ring is a loop back to that one system, and the Ring process is talking to itself. If the wait loop given above is used in this case, the Ring process will never receive any of the transmissions that have been made, and space will never become available for the new transmit. In other words, the status will stay XS$IP forever.

Enhancements

X$GVVC. The PRIMENET subroutine call X$GVVC may be used to pass control of a virtual circuit to another process. This would be very useful to Ring when a complex user request requires that a helper process be phantomed, except for the fact that it can only be used to pass a connection to another process on the same system. To be truly useful, it must be possible to pass a connection to any system.

X$STAT. The X$STAT PRIMENET subroutine can be used to determine virtual circuit information about circuits only on the current system. It would be extremely useful if it could return information about circuits on any system. Then it could return the system name and virtual circuit id of the other end of a connection, and it would be possible to find the user name of the owner of the other end of a virtual circuit easily.

X$TRAN. The X$TRAN subroutine call is documented as not informing the transmitting process that the reception has been completed. This is extremely annoying because it means that it is impossible to transmit a response code to a user process, wait until that process has received the code, and then clear the virtual circuit. Saying that "applications requiring assured delivery must implement their own end-to-end acknowledgement" is certainly the easy way out, but it leaves much to be desired. More importantly, it assumes that the processes on both ends of a circuit are intelligent enough to perform an end-to-end acknowledgement. Ring cannot assume that the user process is going to acknowledge that it has received the response since the user program is not under its control. Neither can Ring allow a user connection to remain long past the completion of the user request if no acknowledgement takes place. Ring solves the problem by keeping the time of day when the last activity on a circuit took place, and clearing a circuit when it has been inactive for a sufficiently long period of time.

Bibliography

PRIMENET Guide, DOC3710-190, Second Edition, by Peter A. Neilson, Prime Computer, Incorporated, 500 Old Connecticut Path, Framingham, Massachusetts 01701.

Software Tools Subsystem User's Guide, April 1982, by T. Allen Akin, Terrell L. Countryman, Perry B. Flinn, Daniel H. Forsyth, Jr., Jeanette T. Myers, and Peter N. Wan, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332.

Appendix

The following is a trace of Ring operating on two systems. The text which is **boldfaced** is commentary, not part of the trace itself.

System GT.A

System GT.B

Ring is brought up on GT.A

Wednesday, April 6, 1983 3:53 PM

Attempting connection to GT.B
Attempting connection to GT.C
Attempting connection to GT.D
Attempting connection to GT.E
Attempting connection to GT.A
Connection received from GT.A
Connection received from GT.A
Validated transmission to GT.A
Validated reception from GT.A
Degenerate ring initialized

The ring is initialized

Ring is brought up on GT.B

Wednesday, April 6, 1983 3:54 PM

Attempting connection to GT.C
Attempting connection to GT.D
Attempting connection to GT.E
Attempting connection to GT.A

GT.A receives a connection

Connection received from GT.B

GT.B receives the validation connection request

Connection received from GT.A
Validated transmission to GT.A

New connection validated

New connection validated
Previous connection cleared

Validated reception from GT.B
Attempting connection to GT.B

GT.B receives a connection

Connection received from GT.A

GT.A receives a validation connection request

Connection received from GT.B
Validated transmission to GT.B

New connection validated

New connection validated
INITIALIZE request created

Validated reception from GT.A
Transmitted INITIALIZE request

INITIALIZE request received

Created INITIALIZE response

Initial time set

Transmitted SYNCHRONIZE request at 15:55 on 04/06/83

Synchronized at 15:55 on 04/06/83

New ring is initialized

User issues a BROADCAST

Connection received from GT.B
Connection received from ROY (29)
User request made for ROY (29)

Roy is not logged on

*** Unknown addressee.
Message broadcast to user ROY

this is a test.
Message broadcast to user ROY

User issues EXECUTE on ALL

```
                                        Connection received from GT.B
                                        Connection received from ROY (29)
                                        User request made for ROY (29)

Phantom (58) created for user ROY

                                        Phantom (63) created for user ROY
```

Time is set on the hour

```
Transmitted SYNCHRONIZE request at 16:00 on 04/06/83

                                        Synchronized at 16:00 on 04/06/83
```

User issues EXECUTE on GT.A

```
                                        Connection received from GT.B
                                        Connection received from ROY (29)
                                        User request made for ROY (29)

Phantom (59) created for user ROY
```

4 users issue BROADCASTs

```
                                        Connection received from GT.B
                                        Connection received from ROY (59)
                                        Connection received from GT.B
                                        Connection received from ROY (56)
                                        Connection received from GT.B
                                        Connection received from ROY (63)
                                        User request made for ROY (59)

*** Unknown addressee.
Message broadcast to user ROY

                                        message 4
                                        Message broadcast to user ROY
                                        User request made for ROY (63)

*** Unknown addressee.
Message broadcast to user ROY

                                        message 2
                                        Message broadcast to user ROY
                                        User request made for ROY (56)

*** Unknown addressee.
Message broadcast to user ROY

                                        message 3
                                        Message broadcast to user ROY
                                        Connection received from GT.B
```

Connection received from ROY (61)
User request made for ROY (61)

*** Unknown addressee.
Message broadcast to user ROY

message 1
Message broadcast to user ROY

User issues TERMINATE

Connection received from GT.B
Connection received from ROY (29)
User request made for ROY (29)

TERMINATE request received

User receives the response

TERMINATE request received
SHUTDOWN request transmitted

Ring SHUTDOWN initiated
Shutdown complete

Ring SHUTDOWN initiated
Shutdown complete

TABLE OF CONTENTS

Ring

User's Guide for the Georgia Tech C Compiler

Second Edition

Daniel H. Forsyth, Jr.
Edward J. Hunt
Jeanette T. Myers
Arnold D. Robbins

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

October, 1984

Foreword

The Georgia Tech C compiler and run time support library provide a C programming environment on Prime computer systems. The Georgia Tech C Compiler runs under and **requires** the Georgia Tech Software Tools Subsystem, Version 9 or later. Both run on PRIME 400, 500 and 50-series computers.

This guide documents the second version of the Georgia Tech C compiler and run time library which is released with Version 9 of the Software Tools Subsystem. The eight chapters of this guide

1) explain the use of the compiler,
2) describe the machine-dependent features of the implementation,
3) describe the compile time environment provided,
4) detail the behavior of the run time package,
5) enumerate problems of conversion from other systems,
6) document known compiler bugs and shortcomings,
7) provide some technical information on the implementation and performane of the C compiler, and
8) outline actions necessary to manage the C system.

A complete description of C can be found in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, 1978). Further information on individual commands in the C system can be obtained from the *Software Tools Subsystem Reference Manual*, accessible both on paper and through the Subsystem 'help' command. The C run time library is only documented here. There are no 'help' entries for the individual subroutines.

Wherever a routine or facility has been changed from the first release of the C compiler, it will be explicitly noted as such.

It is to be noted that wherever it appears in this document, the term "Unix" is a trademark of AT&T Bell Laboratories, Inc.

Getting Started

Prerequisites

We assume that you are already familiar with the Subsystem; that you can create, delete, edit and list files; redirect input and output; obtain on-line documentation, etc. We also assume that you are familiar with the C programming language. If you are not, you should examine the *Software Tools Subsystem User's Guide* and *The C Programming Language* by Kernighan and Ritchie before continuing in this guide.

Throughout this guide, we boldface user input in our examples, as is the convention in the *Software Tools Subsystem User's Guide*.

Calling the C Compiler

There are several commands that call the C compiler:

```
cc      - compile a C program
ccl     - compile and load a C program
ucc     - "Unix-like" C compile and load
compile - general purpose compiler interlude
```

We follow with a brief description of each. For detailed information and examples refer to the Reference Manual entries for each command, e.g.:

```
] help cc
```

Cc --- compile a C program

'Cc' behaves much like the other Subsystem compiler interfaces. It is a program that takes a file whose name ends in ".c" and calls the programs necessary to convert it into a relocatable object program in a file whose name ends in ".b". 'Cc' calls two major programs: the compiler front end 'c1', and the code generator 'vcg'. If you have no compile-time errors in your program, you will not see *any* messages at all from either program. 'Cc' automatically "includes" the file "=cdefs=" (which is "=incl=/swt_def.c.i") containing macros and external data declarations for the C Standard I/O Library and for interfacing with the Subsystem.

Ccl --- compile and load a C program

'Ccl' compiles a ".c" file in the same way as 'cc'; it then calls 'ld', the Subsystem loader interface, to produce an executable program in a file with no suffix. Unfortunately, the Prime loader is somewhat noisy, so you receive a good bit of output during the execution of 'ld'.

Ucc --- compile and load a C program

'Ucc' is a "Unix-style" C compiler and loader. It is **not**, however, exactly like Unix's 'cc' or any other known Unix program! 'Ucc' recognizes file naming conventions for Subsystem supported languages and will use the appropriate preprocessor and/or compiler to process non-C files. Consequently, it can be used to compile and load several files of different languages into an executable program, as long as the main program is written in C. 'Ucc' now depends on the new 'compile' program to do most of its work. It is just smart enough to arrange to call 'compile' properly; it no longer knows about all the details of the C compiler, or how to go about compiling other languages.

Compile --- general purpose compile and load

'Compile' is a general purpose compiler interlude. It knows about the Subsystem naming convention for the more popular languages available under SWT and Primos. It will arrange to call the proper compiler for each source file, based on the suffix. You may tell it to pass options on to each different compiler or preprocessor, and also to tell it what is the "main" language, in order for it to load any necessary libraries and/or start-off routines. 'Ucc' now just rearranges its arguments, and calls 'compile'.

C Program Development --- An Example

For this example, the file "inout.c" contains the following C program:

```
main ()         /* copy input to output until EOF */
{
        int c;

        while ((c = getchar()) != EOF)
                putchar (c);
}
```

We can compile and load (i.e., link-edit) "inout" with the command

```
] ccl inout.c
```

Consistent with Subsystem convention, 'ccl' places the executable version of "inout.c" in a file named "inout". You can execute "inout" as follows:

```
] inout
a
a
echo me if you dare
echo me if you dare
<control-c>
]
```

Features of Georgia Tech C

Standard Implemented

The Georgia Tech C compiler is based on the specifications contained in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978.

Additional Features

The Georgia Tech C compiler provides the following extensions to C:

1. Unions may be initialized. The first type entry in the union will be used to determine the format of the data. For example, "union {int a; double b;} x = 1;" would initialize "x" as an int, not a double.

2. Except for external names, all characters in all names are significant. External names are up to 8 characters in length, with no case significance. To allow access to Primos system calls, the dollar sign ("$") is also a legal character in identifiers. The external names in the object code produced by the compiler can be up to 32 characters long; it is the SEG loader that restricts their lengths to 8 characters. The 'bind' EPF loader *does* pay attention to the full 32 character names.

3. The late Unix Version 7 enhancements, structure assignment and "enum" types, are implemented (but not thoroughly tested).

4. C functions can call Fortran, PL/1, etc. routines, and vice versa. C uses the same calling sequence as all other Prime supported languages. SHORTCALL procedure calls (using the JSXB instruction) are not supported by Georgia Tech C.

5. The Ratfor/Algol68 radix notation may be used to specify integer constants. In addition to using a leading 0 for specifying octal and 0x for hexadecimal, Georgia Tech C recognizes the Ratfor radix syntax for integer constants up to base 36. (For instance, "7r123" is 123 base 7, i.e. 66.)

6. Single quotes may be used to specify packed character strings as in Fortran. The Georgia Tech C compiler treats a single character enclosed in apostrophes as a character constant, while more than one character enclosed in apostrophes is considered a pointer to an array of integers containing a packed "hollerith" character constant.

7. The data type "long unsigned" is supported, giving access to 32-bit unsigned numbers.

8. Initialization of automatic aggregates is supported. (The code generator is not particularly smart about it, though, so initializing huge automatic arrays is incredibly space-inefficient.)

9. Macro definitions ("#define"s) can be specified on the command line using the "-D" compiler option.

10. Directories to be searched for include files may be specified on the command line using the "-I" compiler option.

11. The special macros "__FILE__" and "__LINE__" are supported to provide access to the source file name and source line number (as a string constant and an integer constant), respectively.

Compile Time Facilities

Include File Organization

The C compiler package comes with several standard header files which perform a number of functions for the C programmer. All include files are kept in the directory =incl=.

To maintain compatibility with the previous release of the C compiler, the file "=cdefs=" which the C compiler automatically includes (unless you use the "-f" option) is still "=incl=/swt_def.c.i". This is also in accordance with the Subsystem naming convention for other "standard" include files. However, *all* other C include files end in ".h" (for "header"), which is the Unix convention (this should make porting code a little easier as well).

The "=cdefs=" file itself has been considerably reorganized for the second release. This organization is discussed below.

=incl=/swt_def.c.i

This file now contains very few actual definitions. Instead, it **#include**s separate files to provide the same functionality as it previously did.

We have reorganized the include files to both increase the available functionality, and to separate the features into appropriate, self-contained, "modules". So that previous programs which depend on "=cdefs=" to contain everything need from breaking, "=cdefs=" includes the files it needs. All the include files have been organized so that they may be included more than once. The definitions will only take effect the first time.

The following identifiers are defined in "=cdefs=" for use in determining what kind of hardware and software environment the program will run in. This is useful for writing code designed to be ported to more than one computing system. The identifiers are:

```
#define gtswt   1       /* using Software Tools */
#define primos  1       /* os is primos */
#define prime   1       /* hardware is prime */
#define pr1me   1       /* another name for prime */
```

We have **#define**d the identifier "gtswt" instead of just plain "swt", since "swt" is the name of the routine you have to call in order to exit to the Subsystem.

You would use these identifiers for tailoring your code to different environments. For instance

```
#ifdef gtswt
     /* code to do nifty stuff for Software Tools */
#else
#ifdef unix
     /* code to do nifty stuff for Unix */
#else
     /* code to do nifty stuff in a generic environment */
#endif
#endif
```

"=cdefs=" then includes the following four files (discussed below).

```
#include "=incl=/stdio.h"
#include "=incl=/ctype.h"
#include "=incl=/swt.h"
#include "=incl=/ascii.h"
```

Following the discussion of these four main files, we briefly discribe the other include files which are available.

=incl=/stdio.h

This file contains the declarations and definitions needed to use the C Standard I/O Library.

The following functions, macros, and symbolic constants are available for the programmer to use. (There are others, whose names begin with '_', which the programmer should not need to use, so they aren't discussed here).

```
typedef ... FILE;   The standard type FILE.

/* declaration of functions */
extern long ftell();
extern FILE *fopen(), *fdopen(),  *freopen(), *popen();
extern char *fgets(), *gets();
extern char *strcat(), *strcpy(), *strncpy(), *strpbrk(), *strtok();
extern char *strchr(), *strrchr(), *index(), *rindex();

stdin         Standard Input.
stdout        Standard Output.
stderr        Standard Error Output (Standard Output 3).
stdin1        Subsystem name for stdin.
stdout1       Subsystem name for stdout.
stdin2        Standard Input Port 2.
stdout2       Standard Output Port 2.
stdin3        Standard Input Port 3.
stdout3       Standard Output Port 3 (Error Output).
tty           File pointer which is always the terminal.
errin         Another name for stdin3.
```

errout	Subsystem name for stderr.
EOF	End of file indicator (not a valid character).
NULL	The empty pointer.
BUFSIZ	A convenient buffer size (used a lot on Unix).
TRUE	Represents logical true.
FALSE	Represents logical false.
L_cuserid	The length of a string to hold a user id.
L_ctermid	The length of a string to hold a terminal name.
L_tmpnam	The length of a string to hold a temporary file name.
P_tmpdir	The (string) name of a directory for temporary files.
getchar()	Get a character from stdin.
putchar(ch)	Put a character on stdout.
feof(fp)	Did EOF happen on this file?
ferror(fp)	Did an error occur on this file?
clearerr(fp)	Clear all error flags for this file.
fileno(fp)	Give the SWT file descriptor for the FILE pointer.

Any functions which don't return **int**, and which are not declared in "=incl=/stdio.h", should be declared before they're used. (The type of each function in the C library is given in the chapter on the run time environment, below.)

=incl=/ctype.h

This file defines the character testing macros discussed in the chapter on the run time environment. These macros are very useful, and are often faster than writing an explicit test (e.g. islower(c) should be faster than (c >= 'a' && c <= 'z')). The macros have been rewritten to only evaluate their argument *once*, so that they won't bite you if the argument has side effects (.e.g. islower(*c++)).

=incl=/swt.h

This file provides most of the functionality that the Ratfor programmer obtains from "=incl=/swt_def.r.i". Some of the Ratfor specific declarations have been deleted (for example, the "dynamic memory" routines). The programmer is referred to Appendix D of the *User's Guide for the Ratfor Preprocessor* and the "swt.h" file itself for details.

One thing that may need clarifying: The SET_OF_?* macros are used in the following way:

```
        .
        .
switch (c = getchar()) {
case SET_OF_UPPER_CASE:
        /* stuff for upper case */
        break;

case SET_OF_LOWER_CASE:
        /* stuff for lower case */
        break;

case SET_OF_DIGITS:
        /* stuff for digits */
        break;

default:
        /* stuff for default */
        break;
}
        .
        .
```

In other words, you supply the leading **case** and the trailing colon; the macro supplies everything else.

=incl=/ascii.h

This file contains definitions for the ASCII mnemonics, as well as for the control characters. E.g. Both BEL and CTRL_G are defined as octal 0207. The synonyms BACKSPACE, TAB, BELL, RHT, and RUBOUT for other characters are also defined.

=incl=/assert.h

This file defines the 'assert' macro. It must be specifically included in order to use it. See the chapter on the run time environment for what the 'assert' macro does.

=incl=/debug.h

This file declares a macro "debug" which is useful for putting debugging code into your programs. For instance:

```
#include <debug.h>
...
debug (fprintf (stderr, "i == %d\n", i));
/* note the balanced parentheses */
...
```

If the symbol "DEBUG" has been defined *before* <debug.h> is included, then whatever occurs as an argument to the "debug" macro will be placed into the source code. Otherwise, "debug" becomes a null macro. The easiest way to turn debugging on is to

put "debug" statements in your code, and then do a "-DDEBUG" on the compiler command line. For larger blocks of code, you can do

```
#ifdef DEBUG
        /*
         * a lot of debugging code
         */
#endif
```

=incl=/keys.h

This file declares the symbolic keys for the Primos file system. It is the analogue of "=incl=/keys.r.i".

=incl=/lib_def.h

This file is analogous to the Ratfor include file "=incl=/lib_def.r.i". It contains symbolic constants and macros which are useful for dealing with the low level Software Tools Library routines.

=incl=/math.h

This file contains declarations for all the mathematical routines in the C library. These routines all return **double**.

=incl=/memory.h

This file contains declarations of mem?* functions. These functions are similar to the str?* functions, but work on arbitrary areas of memory, and do not care about the zero word, '\0'. This file should be included before using the functions, although you can always just declare each function before using it.

=incl=/setjmp.h

This file *must* be included if you intend to use the 'setjmp' and 'longjmp' non-local goto functions.

=incl=/swt_com.h

This file contains the necessary **#define**s and declarations for accessing the Software Tools common blocks from a C program. It has not been extensively tested. See the file for more details.

=incl=/varargs.h

This file contains definitions which allow you to portably write functions which expect a variable number of arguments. The macros are discussed below, in the chapter dealing with the run time environment. They have not been extensively tested, but do seem to work.

Loading C Programs For Bare Primos

Several of the routines in the C Library depend on the shared Subsystem libraries to do some of their work.

In order for you to write C programs to run under bare Primos, we have provided a second run time library with alternate versions of these few subroutines, as well as a second C start off routine. Most routines perform the same under both the Subsystem and bare Primos. Those few which behave differently under bare Primos are detailed in the chapter on run time facilities. In particular, they always return the value that indicates an error has occurred.

The alternate C start off routine and run time library are in the files =lib=/nc$main and =lib=/nciolib, respectively. Since loading programs for bare Primos is not simple, 'ld' does not have an option for loading C programs for running without the Subsystem. You must do it yourself, by hand.

To load a C program for use with bare Primos, follow this procedure:

1) Load the file =lib=/nc$main. This is the alternate startoff routine, which does some extra initialization.

2) Load your C binaries.

3) Load =lib=/nciolib. This library contains versions of the few routines which act differently under bare Primos. This library also contains a special version of 'getarg', to allow 'argc' and 'argv' to work properly. The environment pointer, 'envp' (see below), will be set to NULL when a program is loaded for running under bare Primos.

4) Load =lib=/ciolib. This contains the rest of the C run time library.

5) Load =lib=/vswtmath, if your C program uses the C math routines.

6) Load =lib=/nvswtlib. This is the non-shared version of the Subystem library, which does most of the real work.

7) Load any system libraries, e.g. the Fortran library.

You should actually be able to use 'ld' to load your program, following this outline:

```
] ld -dnu -l nc$main <binaries> -l nciolib -l ciolib _
    [-l vswtmath] -l nvswtlb -t -m -o <executable_file>
```

You will probably not have too many programs to be run under bare Primos, but we have provided for this possibility.

Run Time Environment

Calling Primos and Subsystem Routines

C programs have access to all Primos system and library subroutines and Software Tools library routines. Georgia Tech C follows Prime's established conventions for parameter passing, thus allowing C routines to call or be called by programs written in other high-level languages or in assembly language. For example the following C program uses the Ratfor subroutine 'putch' for output:

```
main ()      /* copy input to output until EOF */
{
        int c;

        while ((c = getchar()) != EOF)
                putch (c, STDOUT);
}
```

'Ccl' and 'ucc' both use the Subsystem loader interface 'ld'. When loading C programs, 'ld' automatically includes the C Standard I/O Library, "ciolib", the SWT math library, "vswtmath", the shared shell library, "vshlib", and the shared Subsystem I/O and utility library "vswtlb". However, if another library is required, e.g. one of your own making, "mylib", then the following command must be used:

```
] ccl <program_name> -l mylib
```

The Main Program

All complete C programs must have a function named 'main', which is where execution will begin. The 'main' function in Georgia Tech C Programs may have zero, one, two, or three arguments. If there are arguments, the first is an integer, which indicates the number of command line arguments there were (including the command name). The second is a pointer to an array of character strings containing the text of the arguments. The final element in the array will be equal to NULL. The third argument is a similar pointer to an array of character strings containing a list of *name*=*value* pairs. These are your shell variables and their values. (This is just like the Unix environment pointer, although shell variables aren't as heavily used under Software Tools.) Try this sample program (call it junk.c):

```
main (argc, argv, envp)
int argc;               /* argument count */
char **argv;            /* argument values */
char **envp;            /* environment pointer */
{
        int i;

        for (i = 0; i < argc; i++)
                printf ("%s\n", argv[i]);

        for (i = 0; envp[i] != NULL; i++)
                printf ("%s\n", envp[i]);
}
```

Compile and run it with:

```
] ccl junk.c
] junk foo bar baz
```

You should see something like:

```
junk
foo
bar
baz
HOME=/uc/arnold
_prt_dest=LPB
_search_rule=^int,^var,&,=ubin=/&,=lbin=/&,=bin=/&
```

The program printed its arguments, and then the names and values of any shell variables you may have set.

C Run Time Library

The Georgia Tech C Run Time Library, "ciolib", is a version of the C Standard I/O library. It is automatically loaded with C programs by 'ccl' and 'ucc'. This section describes the routines available in "ciolib".

We have attempted to provide almost all the routines in Section 3 of the *UNIX User's Manual*, for Release 1 of UNIX System V. In particular, "ciolib" contains all of the routines marked "3S" (the Standard I/O Library), most of the routines marked "3M" (the Math library), as many as possible of the routines marked "3C" (routines automatically loaded with every C compilation), and even some of the routines marked "3X" (routines from specialized libraries). In addition, there are routines to emulate some of the more useful (and easy to implement) Unix system calls. These should help when porting programs originally written to run under Unix. Finally, there are a few routines which are not provided under Unix at all, but which allow access to certain features of Primos, or which are just generally useful.

NOTE: The calling sequences of two routines, 'c$ctov' and 'c$vtoc', have changed since the first release of the C compiler. The original motivation for these routines was that the C end-of-string character ('\0') was different from the Subsystem EOS. Since they are now the same, these routines have been brought closer in line with the behavior of the other C string routines. If you need them the old way, take a look at 'ctov' and 'vtoc' in section 2 of the *Software Tools Subsystem Reference Manual*. No other routines have been changed in how they are called, although the functionality and/or implementation of a routine may have changed.

In the following, NULL denotes the null pointer (defined in "=incl=/stdio.h" as "(char *) 0"). Note that, on the Primes, ASCII NUL is represented as octal 0200, while '\0', the zero character, has the octal value 0.

Finally, remember that "=cdefs=" includes the files "=incl=/stdio.h", "=incl=/ctype.h", "=incl=/swt.h", and "=incl=/ascii.h", so their contents are automatically available, unless you specify the "-f" option.

UNIX System Calls

This section describes the routines in "ciolib" which are not part of the Standard I/O Library per se, but which emulate Unix system calls.

The Unix i/o system calls operate on integers, called file descriptors. Due to the similarity with Software Tools file descriptors, these routines usually act as interludes to their SWT counterparts, but return the values described in the Unix User's Manual.

. chdir --- change directory

Calling Information:

```
int chdir (path)
char *path
```

'Chdir' is used to change the current working directory. It uses the SWT routine 'getto' to actually change directory. 'Chdir' returns 0 if it succeeded, -1 if it failed.

Note that under Primos, if a program does a 'chdir', you will be in the new directory when the program exits, not where you were when the program started.

. **close --- close an open fd**

Calling Information:

```
int close (fd)
int fd;
```

'Close' closes a file associated with the file descriptor 'fd' returned by 'creat' or 'open'. 'Close' flushes any data buffers associated with the file and returns 0 if it was successful. If an error occurs, 'close' returns -1. This is not the same as SWT's 'close' (it's a macro), so "=incl=/stdio.h" must be included for 'close' to work as described.

. **creat --- create a file**

Calling Information:

```
int creat (name, mode)
char *name;
int mode;   /* protection mode; not used on Prime */
```

'Creat' creates and opens the file 'name' with WRITE access and returns a file descriptor. The new file has protection keys of "a/" (owner has all permissions). If the file 'name' already exists, 'creat' opens it for writing and truncates it to length 0. An existing file must have either "wt/" or "a/" protection keys (owner has **both** write and truncate permission). A return value of -1 indicates that the file cannot be created or that an attempt was made to 'creat' an existing file with the wrong protection keys. 'Mode' is ignored in this implementation.

. **open --- open a file, return a SWT fd**

Calling Information:

```
int open (name, mode)
char *name;
int mode;
```

'Open' provides a "Unix-style" call to open a file for reading and/or writing and returns a file descriptor. 'Mode' = 0 specifies read access, 1 specifies write access, and 2 specifies read/write access. A return value of -1 indicates that the file does not exist (as determined by filtst(2)), or cannot be opened (access mode does not match protection keys, or no free file descriptors are available) or that 'mode' was invalid. The C 'open' is not the same as SWT's 'open' (it's a macro), and requires that "=incl=/stdio.h" be included to function correctly.

. **exit --- exit from this program**

Calling Information:

```
int exit (exit_val)
int exit_val;   /* not used on the Primes */
```

'Exit' closes all open files and returns to the Subsystem (or to bare Primos). Temporary files that may have been created during program execution remain in directory "=temp=". 'Exit_val' is unused.

. **getpid --- return the current process number**

Calling Information:

```
int getpid()
```

'Getpid' returns the current process number. It uses the Subsystem routine 'date' to retrieve this information from Primos.

. **lseek --- position to a designated word in file**

Calling Information:

```
long lseek (fd, offset, origin)
int fd, origin;
long offset;
```

'Lseek' positions the read/write pointer for the file associated with file descriptor 'fd' (returned by 'creat' or 'open') to the word designated by 'offset' and 'origin'. If 'origin' = 0, 'offset' is the number of words from the beginning of the file. If 'origin' = 1, 'offset' is the number of words forward(backward) from the current position. If 'origin' = 2, 'offset' is the number of words past(before) the end of the file. See 'fseek' for further discussion.

If 'lseek' succeeds, it returns the current file position; otherwise it returns -1. 'lseek' calls 'markf', which will flush the buffers associated with 'fd'.

. **read --- read raw words from a file**

Calling Information:

```
int read (fd, buf, nw)
int fd, nw;
char *buf;
```

'Read' reads words from the file associated with the file descriptor 'fd' (returned by 'creat' or 'open') until 'nw' words have been read or until it encounters the end of file. If 'fd' is attached to a terminal device, 'read' will collect characters until it encounters a NEWLINE.

If an error occurred, 'read' returns -1; if 'read' encounters the end of the file or if a disk error occurred, it returns 0, so both -1 and 0 should be taken as error returns. Otherwise, 'read' returns the number of words transferred to 'buf'.

. **unlink --- delete a file**

Calling Information:

```
int unlink (path)
char *path;
```

Since Primos does not support links to files, 'unlink' always removes the file. If the file is open by any other user, or if the file does not have protection keys "t/" or "a/" (owner has truncate permission) 'unlink' will fail. 'Unlink' returns -1 on failure and 0 otherwise.

. **write --- write raw words to a file**

Calling Information:

```
int write (fd, buf, nw)
int fd, nw;
char *buf;
```

'Write' writes 'nw' words from 'buf' to the file associated with a file descriptor 'fd' (returned by 'creat' or 'open'). If an error occurs or if 'fd' is attached to "/dev/null", 'write' returns -1. Otherwise, 'write' returns the number of words written.

The C Standard I/O Library

The following routines are those listed as "3S", i.e. the actual Standard I/O Library. Input/Output operations in the Standard I/O Library occur on objects of type "FILE *". These are known variously as file pointers, or I/O streams.

The routines are listed below in roughly alphabetical order. However, logically associated routines (and/or macros) are grouped together.

. **ctermid --- return a filename for a terminal**

Calling Information:

```
char *ctermid (s)
char *s;
```

'Ctermid' returns the standard Georgia Tech SWT terminal name "/dev/tty". If 's' is not NULL, then 'ctermid' copies "/dev/tty" into it, and returns 's'. 'S' should be at least 'L_ctermid' characters long. 'L_ctermid' is defined in "=incl=/stdio.h".

. **cuserid --- return the user's login name**

Calling Information:

```
char *cuserid (s)
char *s;
```

'Cuserid' returns the user's login name consisting of 'L_cuserid' - 1 or fewer lower case non-blank ASCII characters followed by '\0'. ('L_cuserid' is defined in "=incl=/stdio.h".)

. **fclose --- close a stream**

Calling Information:

```
int fclose (stream)
FILE *stream;
```

'Fclose' closes the file and flushes the buffer associated with 'stream'. 'Fclose' returns 0 if the close was successful, EOF (-1) otherwise.

. **ferror --- indicate if an error has occurred on a given stream**

Calling Information:

```
int ferror (stream)
FILE *stream;
```

'Ferror' returns TRUE if an error has occurred while doing i/o on 'stream.' It returns FALSE otherwise. This is actually a macro in "=incl=/stdio.h".

. **feof --- indicate if EOF has occurred on a given stream**

Calling Information:

```
int feof (stream)
FILE *stream;
```

'Feof' returns TRUE if EOF has occurred on 'stream', and FALSE otherwise. 'Feof' should be used to find out if EOF has actually occurred, particularly when using 'fread' and 'fwrite'. This is actually a macro in "=incl=/stdio.h".

. **clearerr --- clear any errors associated with a given stream.**

Calling Information:

```
int clearerr (stream)
FILE *stream;
```

'Clearerr' will clear all of the error and EOF flags associated with 'stream'. This is actually a macro in "=incl=/stdio.h".

. **fileno --- return a Subsystem file descriptor**

Calling Information:

```
int fileno (stream)
FILE *stream;
```

'Fileno' is a macro in "=incl=/stdio.h" which returns the Software Tools Subsystem file descriptor associated with 'stream'. (Each FILE structure contains a Subsystem file descriptor, along with other information that the programmer should not need to access.) This permits you to use Subsystem routines that require a file descriptor rather than a file pointer, for instance:

```
FILE *fp;
...
fp = fopen ("file", "w");
/* do formatted i/o with a Subsystem routine */
print (fileno (fp), "i = *d*n", i);
```

. **fflush --- flush all buffers for a stream**

Calling Information:

```
int fflush (stream)
FILE *stream;
```

'Fflush' ensures that the contents and position of the open file reflect all output and positioning operations performed by the program. In other words, any in-memory C library and operating system buffers are flushed to disk, so that the permanent file matches the "logical" file (the file that the program has been working with). This is analogous to making changes to a file with the screen editor, and then issuing a "w" command to force the changes back out to the permanent file.

Please note that 'fflush' called on a disk file anywhere other than after a NEWLINE has been read or written may cause undesirable results, since Primos measures file positions in words and the i/o library writes in units of bytes. Flushing in the middle of a line can cause the compressed-blank count to be lost on an input file or can cause an additional '\0' (padding the last word) to be written to an output file. 'Fflush' also dumps the stream's 'ungetc' buffer.

'Fflush' returns EOF (-1) if the flush failed, 0 if it was successful.

. **fopen --- open an i/o stream**

Calling Information:

```
FILE *fopen (name, mode)
char *name, *mode;
```

'Fopen' opens a file 'name' and returns a pointer to an i/o stream. If the file does not exist, 'fopen' will create it. The stream has the mode specified in 'mode':

Mode	*equivalent SWT mode*
"r"	read, file not truncated
"r+"	read/write, file not truncated
"w"	write, file truncated
"w+"	read/write, file truncated
"a"	append for writing, file not truncated
"a+"	read/append for writing, file not truncated

Opening a file for write ("w", "w+") access truncates the file to length 0, while opening it for read ("r", "r+") or append ("a", "a+") access does not. The file pointer is positioned to the beginning of the file for both read and write access, and to the end of the file for append access. Append mode forces all writes to occur on the end of the

file, even if the file was opened for reading as well, and it is not currently at the end of the file.

'Fopen' returns NULL if no streams are available, if an invalid mode is supplied, if any of the arguments are bad, or if the access mode does not match the Primos protection keys.

. **freopen --- associate a new file with an opened stream**

Calling Information:

```
FILE *freopen (name, mode, stream)
char *name, *mode;
FILE *stream;
```

'Freopen' closes the file currently associated with 'stream', opens the file 'name' with mode 'mode' (same as in 'fopen'), and associates it with 'stream'. This function finds use in associating named files with the standard stream identifiers 'stdin', 'stdout', and 'stderr'.

'Freopen' returns NULL if the mode specified is invalid or if the file 'name' cannot be opened. If the file does not exist, 'freopen' will create it. In any case, 'stream' will be closed first.

Use of this routine is normally not possible in a non-Unix environment; Software Tools is an exception, since its file descriptors are very similar to those used by Unix i/o system calls.

. **fdopen --- associate a stream with an opened file**

Calling Information:

```
FILE *fdopen (fd, mode)
int fd;
char *mode;
```

'Fdopen' gets an i/o stream and associates it with the file descriptor 'fd' returned by 'creat' or 'open'. The stream has the mode specified in 'mode'; 'mode' may take the same values as the 'mode' argument to 'fopen'. This implementation of 'fdopen' **does not** check to make sure that modes of the file descriptor and the stream are the same.

The function returns NULL if an invalid mode is specified or if there are no free i/o streams. Successful execution returns a pointer to the newly assigned stream.

. **fread --- read raw words from a stream**

Calling Information:

```
int fread (ptr, itemsize, nitems, stream)
char *ptr;
int itemsize, nitems;
FILE *stream;
```

'Fread' reads 'nitems' * 'itemsize' words from 'stream' into the buffer addressed by 'ptr'. ('Itemsize' may be determined using the **sizeof** operator.) The function returns the number of items of size 'itemsize' read without error. If an error occurs or if it encounters end-of-file, 'fread' returns 0. Results are unpredictable if more words are requested than there is space in the buffer.

. **fwrite --- write raw words to a stream**

Calling Information:

```
int fwrite (ptr, itemsize, nitems, stream)
char *ptr;
int itemsize, nitems;
FILE *stream;
```

'Fwrite' writes 'itemsize' * 'nitems' words onto 'stream' from the buffer pointed to by 'ptr'. If an error occurs, 'fwrite' returns 0; otherwise it returns the number of successfully written items of size 'itemsize'.

. **fseek --- position to a designated word in a stream**

Calling Information:

```
int fseek (stream, offset, origin)
FILE *stream;
long offset;
int origin;
```

'Fseek' first flushes the stream buffers (including the 'ungetc' buffer) to clear up any pending i/o on 'stream' and then positions the read/write pointer to the word specified by 'offset' and 'origin'. If 'origin' = 0, 'offset' is the number of words from the beginning of the file. If 'origin' = 1, 'offset' is the number of words forward (backward) from the current position. If 'origin' = 2, 'offset' is the number of words past (before) the end of the file. 'Fseek' returns -1 if an error occurs or 0 if it succeeds.

There are several things to note about 'fseek'. First, it is not possible to seek past the end of a Primos file; zero words ('\0's) must be written to extend the file. Second, positioning to the end of a Primos sequential-format file requires Primos to read all of the blocks in the file (i.e. 'origin' = 2 can be quite slow). Third, since Primos text

files contain blank compression and '\0' padding, an 'fputs' of a 30-character string will probably not change the file pointer's position by 30; 'ftell' at the beginning of a line is the only reliable way to obtain a file position of a line in a text file. Finally, flushing the stream buffers may have undesirable results if it occurs during the formation of a line (i.e. before 'fputs', 'fprintf', etc. have put out a complete line).

. **rewind --- rewind to beginning of stream**

Calling Information:

```
int rewind (stream)
FILE *stream;
```

'Rewind' positions the read/write pointer associated with 'stream' to the beginning of the file. It is equivalent to "fseek (stream, 0L, 0)". Returns 0 if it was successful or -1 if an error occurred. (Under Unix, 'rewind' returns no value).

. **ftell --- return absolute position in a stream**

Calling Information:

```
long ftell (stream)
FILE *stream;
```

'Ftell' returns the current word position in 'stream' after flushing the stream buffers (and the 'ungetc' buffer) to clear any pending i/o on the stream.

Under Primos, 'ftell' may actually corrupt (slightly) an output text file when it flushes the stream buffers. On text files, 'markf' is the only reasonable way to determine the position of the beginning of a line. See the comments under 'fseek'. 'Ftell' returns -1 if an error occurs.

. **getc --- get a character from a stream**

. **getchar --- get a character from 'stdin'**

Calling Information:

```
int getc (stream)
FILE *stream;

int getchar()
```

'Getc' obtains the next character from 'stream'. If there are no more characters, or if an error occurs, 'getc' returns EOF (-1).

'Getchar' is a macro; it is defined as 'getc(stdin)' in "=incl=/stdio.h".

. **fgetc --- get next character from stream**

Calling Information:

```
int fgetc (stream)
FILE *stream;
```

'Fgetc' is another function which does what 'getc' does. It was initially created to serve as a real function that one could take the address of etc., since under Unix, 'getc' is a macro. On the prime, both 'getc' and 'fgetc' are functions.

. **getw --- get a machine word from a stream**

Calling Information:

```
int getw (stream)
FILE *stream;
```

'Getw' returns a single 16-bit word from a stream or EOF (-1) if an error occurred or no more characters were available. If 'stream' is attached to a terminal, 'getw' returns EOF when a NEWLINE is encountered. Since EOF could be an actual word value, 'feof' should be used to see if end of file has actually occurred.

. **gets --- get a string (up to a newline) from 'stdin'**

Calling Information:

```
char *gets (s)
char *s;
```

'Gets' copies the next line from 'stdin' into 's', discarding the NEWLINE and terminating 's' with '\0'. If 'gets' returns a line, the function return value is 's'; otherwise it is NULL.

. **fgets --- get a string from a stream**

Calling Information:

```
char *fgets (line, size, stream)
char *line;
int size;
FILE *stream;
```

'Fgets' fetches the next line from 'stream' by copying characters into 'line' from the stream buffer until 'size'-1 characters have been copied or until it encounters the next NEWLINE character. The NEWLINE character *is* kept. The function appends '\0' as the last character in 'line'. 'Fgets' returns 'line' if it obtained a line, and NULL otherwise.

. **popen --- initiate a "pipe" to/from a process**

. **pclose --- close a stream obtained from popen**

Calling Information:

```
FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;
```

'Popen' takes two string arguments. The first, 'command', is a command to be executed by the Software Tools shell. The second, 'type', is either "r" or "w", to read from the standard output of the command, or to write to the standard input of the command, respectively. The function return is a stream which can be treated like any other object of type "FILE *".

'Popen' will return NULL if 1) another "pipe" is still open, 2) the 'type' argument is invalid, 3) the shell could not execute the command, or 4) needed temporary files could not be created.

'Pclose' closes the stream obtained from 'popen'. It returns 0 if it could successfully close the "pipe", otherwise it returns -1.

For programs that use "=lib=/nciolib", 'popen' *always* returns NULL, and 'pclose' *always* returns -1.

See the help on shell(2) in the *Software Tools Subsystem Reference Manual* for some caveats when dealing with the shell.

. **printf --- formatted output to 'stdout'**

. **fprintf --- formatted output to a stream**

. **sprintf --- formatted memory to memory conversion**

Calling Information:

```
int printf (control [, arg1, arg2, ..., arg10])
char *control;
untyped arg1, arg2, ..., arg10;

int fprintf (stream, control [, arg1, arg2, ..., arg10])
FILE *stream;
char *control;
untyped arg1, arg2, ..., arg10;

int sprintf (string, control [, arg1, arg2, ..., arg10])
char *string;
char *control;
```

untyped arg1, arg2, ..., arg10;

'Printf' formats its arguments ('arg1', ..., 'arg10') according to conversion specifications in 'control' and outputs the resulting character string on 'stdout'. The arguments may be pointers (i.e., to strings) or names of variables (e.g., ints, floats, ...). 'Fprintf' does the same thing, but to the named 'stream'. 'Sprintf' places the formatted output into 'string'. These routines take care of accessing the arguments according to the specifications of the 'control' string. In the following discussion 'printf' should be taken as a generic name for all three functions.

The 'control' string contains literal characters, which are copied to the "output" directly, and up to 10 conversion specifiers, each of which must have a corresponding argument. These routines conform as closely as possible to the specifications given for 'printf' in the UNIX System V User's Manual (Release 1).

A conversion specifier may consist of the following:

required	%	Begins conversion specification.
optional	flag	Modifies the meaning of the conversion specifier.
optional	<number>	Minimum field width. More space will be used if needed. If <number> begins with "0", then "0" will be used as a padding character; otherwise, 'printf' pads the output field with blanks.
optional	.	Separates field width and precision.
optional	<number>	Precision: maximum number of characters to print from a string or maximum number of digits to the right of the decimal point in a real number.
optional	[lh]	Size of argument indicator. 'l' indicates a long integer, 'h' a short. The 'h' modifier is recognized, but has no effect on the conversion.
required		Conversion specifier.

A field width or precision may be a '*' instead of a decimal integer. In this case, the next argument in the argument list will be treated as an integer and used for the field width or precision.

The flag may be one of the following:

- Left justify the conversion within the field.

+ The result of a signed conversion is always signed. I.e. a '+' will be prepended if the result is positive.

<blank> If the first character of a signed conversion is not a minus sign, the result will be prepended with a blank. The '+' flag overrides the <blank> flag.

Convert the result to an alternate form. This flag has no effect on the **c**, **d**, **s**, and **u** conversion specifiers. For **o** conversion, the precision is increased so that the result has a leading 0. For **x** (**X**) conversion, the result will have a leading **0x** (**0X**). For **e**, **E**, **f**, **g**, and **G** conversions, the result should always have a decimal point, even if there are no digits following the decimal point.

The conversion characters and their meanings are:

d,o,u,x,X Interpret the corresponding argument as a decimal, octal (no leading "0"), unsigned decimal or hexadecimal (no leading "0x"), integer, respectively. For **x** conversion, the letters **abcdef** are used, while for **X** conversion, the letters **ABCDEF** are used.

c Interpret the argument as a character (unpacked).

s The argument is a '\0'-terminated string: output characters until the correct precision has been achieved, or until '\0' is encountered, whichever comes first.

e,E Interpret the corresponding argument as a *double-precision* floating-point number and print it in the form

[-]m{m}.n{n}**e**±xx.

E format will cause the exponent to start with **E** instead of **e**.

f Interpret the corresponding argument as a *double-precision* floating-point number and print it in the form

[-]m{m}.n{n}

with 'precision' digits to the right of the decimal point. If 'precision' is greater than 14, at most 6 significant digits will be printed.

g,G Use the shortest of %e and %f formats. **G** format indicates %E instead of %e.

If the character "%" follows the initial "%" of the control specifier, the pair is taken as a literal character "%". 'Printf' returns the number of successfully printed characters or EOF (-1) if an error occurred.

Note that the old style (undocumented) capital letter conversion specifiers, which indicated 'long' arguments (e.g. %D for **long int**), are not supported. Use %ld (for example) instead of %D.

. **putc --- put a character on a stream**

. **putchar --- put a character onto 'stdout'**

Calling Information:

```
int putc (ch, stream)
char ch;
FILE *stream;

int putchar (c)
char c;
```

'Putc' puts the single character in 'ch' on 'stream'. If an error occurs, 'putc' returns EOF (-1); otherwise it returns the character just written.

'Putchar' is a macro; it is defined as 'putc(c, stdout)' in "=incl=/stdio.h".

. **fputc --- put a character on a stream**

Calling Information:

```
int fputc (c, stream)
char c;
FILE *stream;
```

'Fputc' is another function which does what 'putc' does. It was initially created to serve as a real function that one could take the address of etc., since under Unix, 'putc' is a macro. On the prime, both 'putc' and 'fputc' are func-

tions.

. **putw --- put raw words on a stream**

Calling Information:

```
int putw (w, stream)
int w;
FILE *stream;
```

'Putw' writes a single 16-bit word on 'stream'. After a successful write 'putw' returns the word written, while it returns EOF (-1) if an error occurred. If 'stream' is attached to "/dev/null", 'putw' always returns EOF.

. **puts --- put a string on 'stdout'**

Calling Information:

```
int puts (s)
char *s;
```

'Puts' appends a NEWLINE to the '\0'-terminated string 's' and prints it on standard output. If an errors occurs, 'puts' returns EOF (-1).

. **fputs --- put a string on a stream**

Calling Information:

```
int fputs (s, stream)
char *s;
FILE *stream;
```

'Fputs' puts the '\0'-terminated string 's' on 'stream'. Note that 's' need not contain a NEWLINE character and that 'fputs' will **not** supply one. 'Fputs' returns EOF (-1) if an error occurred, zero otherwise.

. **scanf --- formatted input conversion from 'stdin'**

. **fscanf --- formatted input conversion from stream**

. **sscanf --- formatted input conversion from a string**

Calling Information:

```
int scanf (control [, arg1, arg2, ..., arg10])
char *control;
char *arg1, *arg2, ..., *arg10;

int fscanf (stream, control [, arg1, arg2, ..., arg10])
FILE *stream;
char *control;
char *arg1, *arg2, ..., *arg10;
```

```
int sscanf (string, control [, arg1, arg2, ..., arg10])
char *string;
char *control;
char *arg1, *arg2, ..., *arg10;
```

'Scanf' reads characters from 'stdin', formats them according to conversion specifications in the control string, and stores the results in the variables pointed to by corresponding arguments 1-10. 'Fscanf' reads its input from the named 'stream'. 'Sscanf' reads characters from the named 'string'. In the following discussion, 'scanf' should be taken as a generic name for all three functions.

The control string may contain white space, which is skipped, literal characters (which **must** match corresponding characters from the input stream) and at most 10 conversion specifiers consisting of the following:

required	%	Begins conversion specification.
optional	*	Suppresses assignment of input field (does **not** skip argument).
optional	<number>	Numeric field width.
optional	l	Read variable as 'long'.
optional	h	Read variable as 'short'.
required		Conversion specifier:

'['[^]<char>[-<char>]{<char>[-<char>]}']'
: Input a string of characters until finding a character not included in the bracketed set. E.g., "[a-zA-Z]" stops reading when a non-alphabetic character is encountered. If the first character in the set is '^', input characters are read until finding a character included in the bracketed set. E.g., "[^]" reads until a blank is found.

d,o,u,x
: Input decimal, octal, unsigned decimal, or hexadecimal integers.

c
: Read single character(s) including blanks.

s
: Input a string delimited by white space, or until <width> characters have been read. The variable in which the string is to be stored must be long enough to contain the string followed by a '\0'.

e,f,g Read a floating point number of the form

[±]m{m}.n{n}[**E**[±]x{x}].

The value returned in both cases is a **float**. Use the "l" option to specify a **double** value.

If the character '%' follows the initial '%' of the control specifier, the pair is taken as a literal character '%'.

'Scanf' conversions stop when EOF is seen, when the control string is exhausted, or when an input character conflicts with the control string.

'Scanf' returns the number of successfully assigned input items, or EOF (-1) if none were found.

. **setbuf --- set buffering on a stream**

Calling Information:

```
setbuf (stream, buf)
FILE *stream;
char *buf;
```

Under Software Tools and Primos, 'setbuf' is a null (do nothing) function. Under Unix, it allows the user to associate a character array as the buffer for a given stream. 'Setbuf' is provided to make porting of programs easier. It is an actual function, just in case there is code which takes its address, or does something else strange of this nature.

. **system --- pass a command to the Software Tools Shell**

Calling Information:

```
int system (cmd)
char *cmd;
```

'System' passes a command 'cmd' to the Software Tools shell to be executed, and returns TRUE if the call was successful. If the call failed, 'system' returns FALSE. See the help on shell(2) in the *Software Tools Subsystem Reference Manual* for some caveats when dealing with the shell.

For programs that use "=lib=/nciolib", 'system' *always* returns FALSE.

NOTE: This routine has changed from the previous release of the C compiler. Before Version 9 of Software Tools, it was not possible to call the Software Tools shell, so the 'system' routine called the Primos command interpreter. If you still need to call Primos, see the help on sys$$(2) in the *Software Tools Subsystem Reference Manual*.

. **tmpfile --- create a temporary file**

Calling Information:

```
FILE *tmpfile ()
```

'Tmpfile' returns a pointer to a temporary file opened with "w+" access. The name of the file is inaccessible from inside a program. The file actually created bears the process unique name "=temp=/tm###" (where ### ranges from 1-999) and remains in the "=temp=" directory after the creating process terminates. If no file can be created, 'tmpfile' returns NULL.

. **tmpnam --- return a filename for a temporary file**

Calling Information:

```
char *tmpnam (s)
char *s;
```

'Tmpnam' returns a unique temporary file name "=temp=/ct=pid=###" where ### is a process unique number 0-999 and =pid= is the current process id. Names are recycled after all 1000 have been used.

. **tempnam --- return a filename for a temporary file**

Calling Information:

```
char *tempnam (dir, pfx)
char *dir, *pfx;
```

'Tempnam' is designed to give the user a little more control over the name of his temporary file. The directory for the tempfile will be taken from the environment variable TMPDIR, if it exists. Otherwise, if 'dir' is not NULL, 'dir' will be used. If 'dir' is NULL, the directory will be P_tmpdir (defined in "=incl=/stdio.h").

If 'pfx' is not NULL, it will be used as the prefix for the file name. Otherwise, the prefix will be "ct".

The full file name will consist of the directory name, a '/', then the prefix, the process id number, and a number between 0 and 999. The number changes after each call to 'tempnam'. After all 1000 have been used, they will be recycled.

'Tempnam' uses 'malloc' to create space for the string containing the file name. The pointer returned by 'tempnam' can be used later in a call to 'free'.

'Tempnam' returns NULL if it could not allocate enough space for the string to hold the generated file name.

. **ungetc --- push a single character back on an input stream**

Calling Information:

```
int ungetc (ch, stream)
char ch;
FILE *stream;
```

'Ungetc' places 'ch' in a single-character buffer associated with 'stream'. The next call to 'getc' or 'fgetc' retrieves 'ch'. Attempting to push more than one character back onto the input stream or using 'ungetc' on a closed stream produces an error return of EOF (-1). Otherwise, 'ungetc' returns 'ch'.

. **ftrunc --- truncate a stream at the current position**

Calling Information:

```
int ftrunc (stream)
FILE *stream;
```

'Ftrunc' flushes all file and 'ungetc' buffers for the file associated with 'stream' and truncates the file at its current position. The file must be opened with write access. (If 'fopen' is used to open the file, then the only really useful values for 'mode' are "r+" and "a+", because read access is usually necessary to position the file correctly and because opening the file for write ("w", "w+") access always truncates the file.) 'Ftrunc' returns 0 if it succeeded, -1 otherwise.

'Ftrunc' is not part of the Standard I/O Library per se, but is provided in order to allow access to this capability of the Primos file system.

Unix Subroutines For C Programs

The following routines are those listed as "3C", i.e. the routines which are loaded along with every Unix C program, but which are not guaranteed to be on other non-Unix systems.

The character testing macros discussed below ('isalnum', 'isdigit', etc.) are valid on integers in the range -1 to 0377 (EOF to ASCII DEL). They merely return FALSE on characters in the range -1 to '\177'. The result of these macros on values less than -1 or greater than 0377 is *undefined*.

These macros do not necessarily return "true" values equal to the symbolic constant TRUE defined in "=incl=/stdio.h". Rather, they return logical true, i.e. *non-zero*, and logical false, i.e. *zero*. They should be used as conditions, not compared against TRUE and FALSE. In other words, use:

```
if (islower (c)) { /* stuff */ }
```

and not

```
if (islower (c) == TRUE) { /* stuff */ }
```

The routines are listed below in roughly alphabetical order. However, logically associated routines (and/or macros) are grouped together.

. **a64l --- convert base-64 string to long integer**

. **l64a --- convert long integer to base-64 string**

Calling Information:

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

'A64l' takes a '\0'-terminated string containing a base-64 representation, and returns the corresponding long. If the string has more than six characters, only the first six are used. 'L64a' takes a long integer, and returns a pointer to a string with the corresponding base-64 representation.

These routines use the following characters as digits in the base-64 notation. '.' for 0, '/' for 1, '0' through '9' for 2-11, 'A' through 'Z' for 12-37, and 'a' through 'z' for 38-63.

. **abort --- generate a "fault"**

Calling Information:

```
int abort ()
```

Under Unix, 'abort' generates a **SIGIOT** fault, which causes the program to exit and dump core. The user may catch this signal. Under Software Tools or Primos, this routine simply exits.

. **abs --- return integer absolute value**

Calling Information:

```
int abs (x)
int x;
```

'Abs' returns the absolute value of its integer argument. This is a fast, assembly language routine, local to the C library.

. **atof --- convert character string to double precision real**

Calling Information:

```
double atof (str)
char *str;
```

Converts a string of characters 'str' to a double precision real number. Conversion stops when 'atof' encounters a non-numeric character.

. **atoi --- convert character string to integer**

Calling Information:

```
int atoi (str)
char *str;
```

'Atoi' converts a string of characters 'str' to a base-10 integer. Conversion stops when 'atoi' encounters a non-numeric character. 'Atoi' uses 'gctoi', so it will recognize the Ratfor "radix notation" (e.g. 8r377).

. **atol --- convert character string to long integer**

Calling Information:

```
long atol (str)
char *str;
```

'Atol' converts a string of characters 'str' to a base-10 long (32-bit) integer. Conversion stops when 'atol' encounters a non-numeric character in 'str'. 'Atol' uses 'gctol', so it will recognize the Ratfor "radix notation" (e.g. 8r377).

. **strtol --- convert string to arbitrary base long integer**

Calling Information:

```
long strtol (str, ptr, base)
char *str;
char **ptr;
int base;
```

'Strtol' takes a '\0'-terminated string in 'str', and returns the long integer it represents. 'Base' is the base of the string. If base is less than zero or greater than 36, 'strtol' will return. If base is zero, it will attempt to determine the base from the string itself. A leading '0' indicates octal, '0x' or '0X' indicates hexadecimal; otherwise, the string is assumed to be in decimal.

If the value of 'ptr' is not (char **) 0, the address of the character which terminated the string will be placed in *ptr. If no integer can be converted from the string, *ptr

is set to 'str', and 0 is returned.

. **getcwd --- get pathname of current working directory**

Calling Information:

```
char *getcwd (buf, size)
char *buf;
int size;
```

'Getcwd' returns a pointer to a string containing the SWT path name of the current directory. If 'buf' is not NULL, 'getcwd' will use 'buf' a buffer in which to place the name. Otherwise, it will use 'malloc' to dynamically allocate a buffer. In this case, the returned pointer can be used later in a call to 'free'. 'Size' is the size of the buffer to be 'malloc'ed, so it must include room for the trailing '\0'.

'Getcwd' returns NULL if size is less than or equal to 1, if 'malloc' could not allocate enough memory, or if one of the SWT routines 'follow' or 'gcdir$' failed.

. **getenv --- return value for "environment" variable**

Calling Information:

```
extern char **environ;

char *getenv (var)
char *var;
```

'Getenv' scans the environment list of *name*=*value* pairs pointed to by the external variable 'environ'. If 'var' is found, 'getenv' returns a pointer to its value. Otherwise, it returns NULL.

For programs which use "=lib=/nciolib", 'getenv' will *always* return NULL, and 'environ' is always equal to NULL.

. **getlogin --- get the login name**

Calling Information:

```
char *getlogin()
```

If the current process is a phantom, 'getlogin' returns NULL. Otherwise, it returns the user's login name, as obtained from 'cuserid'.

. **getopt --- get option letter from argument vector**

Calling Information:

```
extern char *optarg;
extern int optind;

int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;
```

'Getopt' returns the next option letter in 'argv' that matches a letter in 'optstring'. If a letter in 'optstring' is followed by a colon, then that option is supposed to have an argument, that may or may not be separated from it by white space. 'Optarg' is set to point to the beginning of the argument to the current option when 'getopt' returns.

'Getopt' sets 'optind' to the index of the next argument in 'argv' to be processed. Since 'optind' is external, it is initialized to zero.

When all options have been processed (i.e. when the first non-option is encountered), 'getopt' returns EOF. The special option **--** can be used to delimit the end of the options. 'Getopt' will return EOF, and will skip the **--**.

'Getopt' returns a '?' and prints an error message on 'stderr' when it finds an option that is not in 'optstring'.

. **getpass --- read a password**

Calling Information:

```
char *getpass (prompt)
char *prompt;
```

'Getpass' disables echoing, and prints 'prompt' on 'stderr'. It then reads up to a newline or EOF from the terminal. It returns a pointer to a '\0'-terminated string of at most eight characters. If 'getpass' cannot use the TTY file descriptor, and if it cannot open "/dev/tty", it will read from 'stdin'. 'Getpass' turns echoing back on before returning.

. **isalnum --- indicate if a character is alphanumeric**

Calling Information:

```
int isalnum (ch)
char ch;
```

'Isalnum' returns TRUE if 'ch' falls in the range 'A' through 'Z', inclusive, in the range 'a' through 'z', inclusive, or if it lies between '0' and '9', inclusive. It

returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isalpha --- indicate if a character is alphabetic**

Calling Information:

```
int isalpha (ch)
char ch;
```

'Isalpha' returns TRUE if 'ch' lies between 'A' and 'Z', inclusive, or if it lies between 'a' and 'z', inclusive. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isdigit --- indicate if a character is a decimal digit**

Calling Information:

```
int isdigit (ch)
char ch;
```

'Isdigit' returns TRUE if 'ch' lies between '0' and '9', inclusive. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isxdigit --- indicate if a character is a hexadecimal digit**

Calling Information:

```
int isxdigit (ch)
char ch;
```

'Isxdigit' returns TRUE if 'ch' falls between '0' and '9', inclusive, or if it falls between 'A' and 'F' inclusive, or 'a' and 'f' inclusive. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isupper --- indicate if a character is an upper case letter**

Calling Information:

```
int isupper (ch)
char ch;
```

'Isupper' returns TRUE if 'ch' lies between 'A' and 'Z', inclusive. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **islower --- indicate if a character is an lower case letter**

Calling Information:

```
int islower (ch)
char ch;
```

'Islower' returns TRUE if 'ch' lies between 'a' and 'z', inclusive. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isprint --- indicate if a character is printable**

Calling Information:

```
int isprint (ch)
char ch;
```

'Isprint' returns TRUE if 'ch' is a printable character. This includes all punctuation, letters, digits, and the space character ' '. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isgraph --- indicate if a character is printable and visible**

Calling Information:

```
int isgraph (ch)
char ch;
```

'Isgraph' is similar to 'isprint' above, except that it excludes the space character ' '. It returns TRUE if 'ch' has a graphic representation, FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **ispunct --- indicate if a character is punctuation**

Calling Information:

```
int ispunct (ch)
char ch;
```

'Ispunct' returns TRUE if 'ch' is neither an alphanumeric nor a control character. To use this macro, the file "=incl=/ctype.h" must be included first.

. **iscntrl --- indicate if a character is a control character**

Calling Information:

```
int iscntrl (ch)
char ch;
```

'Iscntrl' returns TRUE if 'ch' is an ASCII control character, i.e., if 'ch' falls in the range '\200' to '\237', inclusive, or if 'ch' equals '\377' (DEL). It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isascii --- indicate if character is within the ASCII character set**

Calling Information:

```
int isascii (ch)
char ch;
```

'Isascii' returns TRUE if 'ch' lies in the range '\200' to '\377', inclusive (Prime's ASCII representation). It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **isspace --- indicate if a character is white space**

Calling Information:

```
int isspace (ch)
char ch;
```

'Isspace' returns TRUE if 'ch' is a space, a tab, a newline, a carriage return, a form feed or a vertical tab. It returns FALSE otherwise. To use this macro, the file "=incl=/ctype.h" must be included first.

. **malloc --- allocate memory**

. **alloc --- allocate memory (old name)**

Calling Information:

```
char *malloc (n)
int n;

char *alloc (n)
int n;
```

'Malloc' allocates 'n' words of memory and returns a pointer to the beginning of the storage block. If 64K words are requested or if zero words are requested, 'malloc' returns NULL.

'Alloc' is the name of the pre-Version 7 UNIX storage

allocator. It performs the same function as 'malloc'. Its use in new programs is *strongly* discouraged. It is provided to make porting code easier, and because it was in the first release of the C compiler.

. **calloc --- allocate memory for arrays or structures**

Calling Information:

```
char *calloc (n, size)
int n, size
```

'Calloc' allocates 'n' * 'size' words of memory for storing 'n' objects of 'size' words each. If successful, 'calloc' initializes all words to zero, and returns a pointer to the first word of the storage block. If 64K or more words are requested or if zero words are requested, 'calloc' returns NULL.

. **realloc --- change the size of previously allocated memory**

Calling Information:

```
char *realloc (ptr, size)
char *ptr;
int size;
```

'Realloc' reallocates a block of memory of 'size' words for a storage block previously allocated by 'malloc' or 'calloc' (0 < 'size' < 64K). The contents of the original storage block are preserved by copying to the newly allocated block. Therefore, the pointer 'ptr' passed as a parameter to 'realloc' **must** point to the beginning of a block allocated by 'malloc' or 'calloc' in order for the copy to work properly. Any existing pointers to the original data structure must be changed. (I.e. the contents of the old memory are preserved, but the same actual block of memory may not be used.)

'Realloc' returns a pointer to the first word of the new storage block or NULL if an error occurred.

. **free --- free allocated memory**

. **cfree --- free allocated memory (old name)**

Calling Information:

```
free (ptr)
char *ptr

cfree (ptr)
char *ptr;
```

'Free' frees a block of memory previously allocated by 'malloc', 'realloc', or 'calloc'. 'Free' will fail miserably if handed an arbitrary pointer; only pointers returned by 'mal-

loc', 'realloc' or 'calloc' are valid parameters.

'Cfree' is the name of the pre-Version 7 UNIX storage releaser. It performs the same function as 'free'. Its use in new programs is *strongly* discouraged. It is provided to make porting code easier, and because it was in the first release of the C compiler.

. **memccpy --- copy characters up to a character or some number**

Calling Information:

```
char *memccpy (s1, s2, c, n)
char *s1, *s2, c;
int n;
```

'Memccpy' copies characters (words, on the Prime) from the memory pointed to by 's1' into 's2' until it encounters 'c', or until 'n' characters have been copied. It returns a pointer to the character after the first occurrence of 'c', or NULL if 'c' was not found in the first 'n' characters of 's1'.

. **memchr --- return a pointer to a char within a memory area**

Calling Information:

```
char *memchr (s, c, n)
char *s;
int c, n;
```

'Memchr' returns a pointer to the first occurrence of 'c' within the first 'n' characters (words, on the Prime) of 's'. It returns NULL if 'c' does not occur.

. **memcmp --- compare arbitrary areas of memory**

Calling Information:

```
int memcpmp (s1, s2, n)
char *s1, *s2;
int n;
```

'Memcmp' looks only at the first 'n' words of its first two arguments. It returns an integer less than zero, equal to zero, or greater than zero, according as 's1' is lexicographically less than, equal to, or greater than 's2'.

. **memcpy --- copy arbitrary areas of memory**

Calling Information:

```
char *memcpy (s1, s2, n)
char *s1, *s2;
int n;
```

'Memcpy' copies 'n' characters from 's2' to 's1'. It returns 's1'.

. **memset --- initialize memory to a given value**

Calling Information:

```
char *memset (s, c, n)
char *s;
int c, n;
```

'Memset' sets the first 'n' characters (words) of 's' to 'c'. It returns 's'.

. **mktemp --- make a unique file name**

Calling Information:

```
char *mktemp (template)
char *template;
```

'Mktemp' replaces the contents of the string pointed to by 'template' with a unique file name, and returns 'template'. 'Template' should look like a file name with six trailing **X**s; 'mktemp' replaces the **X**s with a unique letter and the process id. (This implementation only requires four **X**s; six is recommended for portability to/from Unix systems.)

If there are no **X**s in the 'template', 'mktemp' returns NULL. The "unique" letter will recycled after 26 calls to 'mktemp'.

So that this routine does not conflict with the SWT 'mktemp', it is actually a macro, so "=incl=/stdio.h" must be included in order to use it.

. **rand --- return a random integer**

. **srand --- seed the random number generator**

Calling Information:

```
int rand()

srand (seed)
unsigned seed;
```

'Rand' uses 'rand$m' in the SWT Math Library. It returns a

number between 0 and 2^16-1. 'Srand' uses 'seed$m' to seed the random number generator. In keeping with the Unix semantics, if the user calls 'rand' before calling 'srand', the random number generator will be seeded with 1.

. **setjmp --- set up for non-local goto**

. **longjmp --- perform a non-local goto**

Calling Information:

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

longjmp (env, status)
jmp_buf env;
int status;
```

'Setjmp' saves the current stack frame in 'env' for later use. On every call, 'setjmp' returns 0. 'Longjmp' takes 'status' and performs a non-local "goto" to an environment ('env') saved by a previous call to 'setjmp'. The result of that operation allows execution to continue as if 'setjmp' had returned 'status' rather than 0 at the point of invocation. Use of these routines *requires* inclusion of "<setjmp.h>" before either of the routines is called. In particular "<setjmp.h>" does a **typedef** on the type 'jmp_buf', and contains a macro definition which is needed for 'setjmp' to return properly.

. **sleep --- sleep for the given number of seconds**

Calling Information:

```
int sleep (amount)
unsigned amount;
```

'Sleep' will sleep for 'amount' seconds. 'Sleep' simply calls the Primos 'sleep$' routine. It returns no value.

. **strcat --- concatenate two strings**

Calling Information:

```
char *strcat (t, s)
char *t, *s;
```

'Strcat' concatenates string 's' to string 't', terminating 't' with '\0'. The target string 't' is assumed to be large enough to accommodate all of the characters copied from 's'. 'Strcat' returns a pointer to 't', or NULL if either 's' or 't' is NULL.

. **strncat --- concatenate substring to string**

Calling Information:

```
char *strncat (t, s, n)
char *t, *s;
int n;
```

Concatenates at most 'n' characters of string 's' to string 't'. If 's' contains fewer than 'n' characters, then only "strlen (s)" characters will be copied. In any case, 'strncat' terminates 't' with '\0' and returns a pointer to 't'.

. **strcmp --- compare strings**

Calling Information:

```
strcmp (s1, s2)
char *s1, *s2;
```

'Strcmp' compares strings 's1' and 's2' and returns 0 if they are equal or if s1 = NULL and s2 = NULL. If *s1 > *s2 or if s2 = NULL, 'strcmp' returns a positive value; it returns a negative value if *s1 < *s2 or if s1 = NULL.

. **strncmp --- compare substrings**

Calling Information:

```
int strncmp (s1, s2, n)
char *s1, *s2;
int n;
```

'Strncmp' compares at most 'n' characters of 's1' and 's2'. It returns 0 if equal or if s1 = NULL and s2 = NULL; a positive value if *s1 > *s2 or if s2 = NULL; or a negative value if *s1 < *s2 or if s1 = NULL.

. **strcpy --- copy string**

Calling Information:

```
char *strcpy (t, s)
char *t, *s;
```

Copy string 's' to string 't'. 'Strcpy' assumes that 't' is large enough to receive all characters contained in 's'. If 't' is NULL 'strcpy' returns NULL; if 's' is NULL and 't' is non-NULL, 't' is set to the empty string ("").

. **strncpy --- copy substring to string**

Calling Information:

```
char *strncpy (s1, s2, n)
char *s1, *s2;
int n;
```

'Strncpy' copies at most 'n' characters from string 's2' to string 's1'. If 's2' contains more than 'n' characters, then 's1' will not be '\0'-terminated. If 's2' contains fewer than 'n' characters (including a terminal '\0'), then 's1' is '\0'-padded until it contains 'n' characters.

. **strlen --- return length of string**

Calling Information:

```
int strlen (s)
char *s;
```

'Strlen' returns the length of a string 's' *excluding* the terminating '\0' character.

. **strchr --- find character in string**

Calling Information:

```
char *strchr (s, c)
char *s, c;
```

Returns pointer to first occurrence of character 'c' in string 's'; if 'c' is not found 'strchr' returns NULL.

. **strrchr --- find character in string (last occurrence of)**

Calling Information:

```
char *strrchr (s, c)
char *s, c;
```

'Strrchr' returns a pointer to the last occurrence of the character 'c' in the string 's'. If 'c' does not occur in 's', NULL is returned. ('Strrchr' also works if 'c' = '\0'.)

. **strpbrk --- find one of a class of characters in a string**

Calling Information:

```
char *strpbrk (s1, s2)
char *s1, *s2;
```

Returns a pointer to the first character in 's1' matching any character in string 's2', or NULL if no character in 's2' is in 's1'. Both 's1' and 's2' must be non-NULL.

. **strspn --- find qualified substring**

Calling Information:

```
int strspn (s1, s2)
char *s1, *s2;
```

'Strspn' returns the length of the initial substring of 's1' that is made entirely of characters from 's2'. If either 's1' = NULL or 's2' = NULL, 'strspn' returns 0.

. **strcspn --- find qualified substring**

Calling Information:

```
int strcspn (s1, s2)
char *s1, *s2;
```

'Strcspn' returns the length of the initial substring of 's1' not having any characters contained in 's2'.

. **strtok --- find tokens in a string**

Calling Information:

```
char *strtok (s1, s2)
char *s1, *s2;
```

'Strtok' returns a pointer to the start of each token in 's1'. Tokens are defined as contiguous strings of characters delimited by separators. 'Strtok' skips over any leading separators. 'S2' contains 1 or more characters to be considered as token separators. When 'strtok' finds a token, it replaces the terminating delimiter with '\0'. If it can't find a token with the current set of delimiters, 'strtok' returns NULL; however, if 's1' has already been successfully searched for a token, the terminating '\0' in 's1' is considered a valid delimiter. Thus, the final token in 's1' can be retrieved without any special finagling (the '\0' of 's2' is never considered a valid separator during the scan of 's1' for delimiters).

On the first call to 'strtok', 's1' should point to a valid string, while on subsequent calls 's1' should be NULL so that the entire string is scanned. The characters in 's2' may change from call to call searching the same 's1'.

. **index --- find character in string**

Calling Information:

```
char *index (s, c)
char *s, c;
```

Same as 'strchr'. This is the V7 (and Berkeley) UNIX routine; the name was changed with UNIX System III.

. **rindex --- find the last occurrence of character in string**

Calling Information:

```
char *rindex (s, c)
char *s, c;
```

Same as 'strrchr'. This is the V7 (and Berkeley) UNIX routine; the name was changed with UNIX System III.

. **toascii --- convert a char/int to a valid ASCII value**

Calling Information:

```
char toascii (ch)
char ch;
```

'Toascii' returns its argument converted into a valid ASCII value. When unpacking packed character strings, 'toascii' can be used to obtain the high character after shifting the packed word right by 8 bits. The low character can be gotten by passing the whole integer to 'toascii'. To use this macro, the file "=incl=/ctype.h" must be included first.

. **toupper --- convert lower case character to upper case**

Calling Information:

```
char toupper (ch)
char ch;
```

If 'ch' is a lower case letter, 'toupper' returns the corresponding upper case letter. Otherwise it returns 'ch' unchanged. This is actually a fast, assembly language routine, declared in "=incl=/ctype.h".

. **tolower --- convert upper case character to lower case**

Calling Information:

```
char tolower (ch)
char ch;
```

If 'ch' is an upper case letter, 'tolower' returns the corresponding lower case letter. Otherwise it returns 'ch' unchanged. This is actually a fast, assembly language routine, declared in "=incl=/ctype.h".

. **_toupper --- blindly convert a character to upper case**

Calling Information:

```
char _toupper (ch)
char ch;
```

'_toupper' returns 'ch' converted to upper case. It does **not** check that its argument is indeed a lower case letter. This function performs the Unix Version 7 'toupper', which was changed to '_toupper' in Unix System III to accommodate those who may still want it. To use this macro, the file "=incl=/ctype.h" must be included first.

. **_tolower --- blindly convert a character to lower case**

Calling Information:

```
char _tolower (ch)
char ch;
```

'_tolower' returns 'ch' converted to lower case. It does **not** check that its argument is indeed an upper case letter. This function performs the Unix Version 7 'tolower', which was changed to '_tolower' in Unix System III to accommodate those who may still want it. To use this macro, the file "=incl=/ctype.h" must be included first.

. **ttyname --- return the name of the terminal**

Calling Information:

```
char *ttyname (fd)
int fd;
```

'Ttyname' takes an integer file descriptor as an argument. If the device which is attached to the file descriptor is a terminal, it returns the string "/dev/tty", otherwise it returns NULL. (Under Unix, it would return the actual device name.)

The associated 'isatty' function already exists in SWT. It may used as is.

The C Math Library

The following routines are those listed as "3M", i.e. the C Math Library. These routines all take arguments of type **double**, and return type **double**.

You should include the file "=incl=/math.h" which declares these routines, before using them. This can be done with the line:

```
#include <math.h>
```

Most of these routines simply call the corresponding routine in the SWT math library, "vswtmath". See the *SWT Math Library User's Guide* for details on how these routines work, and under what condition(s) they will raise an error condition.

- **acos --- take arc cosine of a real**

  Calling Information:

  ```
  double acos (x)
  double x;
  ```

  This routine returns the value obtained from 'dacs$m' in SWT math library.

- **asin --- take arc sine of a real**

  Calling Information:

  ```
  double asin (x)
  double x;
  ```

  This routine returns the value obtained from 'dasn$m' in SWT math library.

- **atan --- take arc tangent of a real**

  Calling Information:

  ```
  double atan (x)
  double x;
  ```

  This routine returns the value obtained from 'datn$m' in SWT math library.

- **atan2 --- take arc tangent of x/y**

  Calling Information:

  ```
  double atan2 (x, y)
  double x, y;
  ```

  This routine first divides x by y. It passes the result of the division to the SWT math library routine 'datn$m', returning its result.

- **ceil --- return smallest integer not less than x**

  Calling Information:

  ```
  double ceil (x)
  double x;
  ```

This routine uses the 'dint$m' routine in the SWT Math Library to remove the fractional part, and then adds 1.0 if its argument was positive.

. **cos --- take cosine of a real**

Calling Information:

```
double cos (x)
double x;
```

This routine returns the value obtained from the 'dcos$m' routine in the SWT math library.

. **cosh --- take hyperbolic cosine of a real**

Calling Information:

```
double cosh (x)
double x;
```

This routine returns the value obtained from the 'dcsh$m' routine in the SWT math library.

. **exp --- compute exponential (base e) of a real**

Calling Information:

```
double exp (x)
double x;
```

This routine returns the value obtained from the 'dexp$m' routine in the SWT math library (Raise *e* to 'x' power).

. **fabs --- compute the absolute value of a real**

Calling Information:

```
double fabs (x)
double x;
```

This routine returns the absolute value of its double precision argument. It is a fast, assembly language routine, local to the C library.

. **fmod --- do floating point modulus operation**

Calling Information:

```
double fmod (x, y)
double x, y;
```

'Fmod' returns 'x' if 'y' is zero. Otherwise, it returns a number *f*, of the same sign as *x*, such that $x = iy + f$ for some integer *i*, and $|f| < |y|$.

. **hypot --- return Euclidean distance function**

Calling Information:

```
double hypot (x, y)
double x, y;
```

'Hypot' returns

```
sqrt (x * x + y * y)
```

It does not check for overflow of the multiplication and addition operations (although it should).

. **floor --- return largest integer not greater than x**

Calling Information:

```
double floor (x)
double x;
```

This routine uses the 'dint$m' routine in the SWT Math Library to remove the fractional part, and then subtracts 1.0 if its argument was negative.

. **log --- take the natural log (base e) of a real**

Calling Information:

```
double log (x)
double x;
```

This routine returns the value obtained from the 'dln$m' routine in the SWT math library.

. **log10 --- take the log base 10 of a real**

Calling Information:

```
double log10 (x)
double x;
```

This routine returns the value obtained from the 'dlog$m' routine in the SWT math library.

. **pow --- provide exponentiation for C programs**

Calling Information:

```
double pow (a, b)
double a, b;
```

'Pow' computes 'a' to the 'b'. 'Pow' calls 'powr$m' in the SWT math library. It may raise 'SWT_MATH_ERROR$' if the first argument is negative, or if the first argument is zero, and the second is negative or zero. The exception

will also be raised if the results of the calculation would cause an overflow.

. **sin --- take sine of a real**

Calling Information:

```
double sin (x)
double x;
```

This routine returns the value obtained from the 'dsin$m' routine in the SWT math library.

. **sinh --- take hyperbolic sine of a real**

Calling Information:

```
double sinh (x)
double x;
```

This routine returns the value obtained from the 'dsnh$m' routine in the SWT math library.

. **sqrt --- take square root of a positive real**

Calling Information:

```
double sqrt (x)
double x;
```

This routine returns the value obtained from the 'dsqt$m' routine in the SWT math library.

. **tan --- take tangent of a real**

Calling Information:

```
double tan (x)
double x;
```

This routine returns the value obtained from the 'dtan$m' routine in the SWT math library.

. **tanh --- take hyperbolic tangent of a real**

Calling Information:

```
double tanh (x)
double x;
```

This routine returns the value obtained from the 'dtnh$m' routine in the SWT math library.

Unix Special Library Routines

The following routines are those listed as "3X", i.e. the routines which require special libraries and/or include files.

. **assert --- put assertions into C programs**

Calling Information:

```
#include <assert.h>

assert (expression)
int expression;
```

The 'assert' function is actually a macro which tests the boolean expression. If the expression is FALSE, 'assert' prints a message to 'stderr', and exits. The error message will contain the file name and line number of the 'assert' statement. The 'assert' macro is written in such a way that it can be used as a regular statement in a C program; it will not mess up **if-else** nesting, for instance.

Defining NDEBUG before including "<assert.h>" or on the 'cc' (or 'ccl' or 'ucc') command line will turn 'assert' into a null (empty) macro.

. **logname --- return login name of user**

Calling Information:

```
char *logname()
```

'Logname' returns a pointer to a string containing the user's login name. If the LOGNAME environment variable exists, 'logname' returns its value. Otherwise, if the template =user= has a value, that value is returned. If neither of those work, 'logname' returns NULL. Both of these methods are subject to counterfieting.

. **varargs --- portably write functions with a variable number of arguments**

Calling Information:

```
#include <varargs.h>

function (va_alist)
va_dcl
va_list pvar;
va_start (pvar);
f = va_arg (pvar, type);
va_end (pvar);
```

The file "=incl=/varargs.h" contains a set of macros which allow you to write functions which will have a variable number of arguments in a portable (if slightly opaque) fashion.

va_alist — is used in place of the argument list inside the parentheses of a function header, to declare a **va**riable **a**rgument **list**.

va_dcl — declares the "type" of the variable argument list. Note that there is *no* semicolon after the "va_dcl".

va_list — is a "type" for declaring the variable *pvar*. *Pvar* is a variable which will be used to step through the argument list. One such variable must be declared.

va_start (*pvar*) — initializes *pvar* to the beginning of the argument list.

va_arg (*pvar*, *type*) — returns the next argument in the list pointed to by *pvar*. It will be a value of type *type*. Variables of different types may be mixed, but it is up to the called routine to determine their types, since this cannot be done at compile time.

va_end (*pvar*) — is used to finish up.

The list can be traversed multiple times, as long as each traversal starts with a 'va_start' and ends with 'va_end'.

While 'varargs' originated at Bell Labs, and is available with System V, it is not documented there. Instead, its use was popularized with the Berkeley versions of Unix (which do document it). In any case, you should be able to use the 'varargs' macros to portably write functions which take a variable number of arguments (like 'printf' does).

The current implementation of 'varargs' allows a maximum of ten arguments in the 'va_alist'.

Other Routines Not From Unix

The following routines are not routines found on Unix, but are supplied in "ciolib", since they are generally useful.

. **basename --- return the file name part of a path name**

. **dirname --- return all but the last part of a path name**

Calling Information:

```
char *basename (str)
char *str;

char *dirname (str)
char *str;
```

'Basename' returns a pointer to the last part of the SWT path name contained in 'str'. If there are no slashes in 'str', it returns 'str', otherwise it returns a pointer to somewhere in the middle of 'str'.

'Dirname' returns a pointer to the directory part of the SWT path name contained in 'str'. It copies 'str' into a private buffer (of length MAXPATH), and then replaces the final slash with a '\0'. If there are no slashes, it changes no characters in the buffer. In all cases, it returns the address of the buffer; the original 'str' is not modified.

The following example should clariy what these routines do:

```
basename ("path/file");   returns   "file"
dirname ("path/file");    returns   "path"
```

. **c$ctov --- convert C string to PL/I string**

Calling Information:

```
int c$ctov (dest, src)
int *dest;
char *src;
```

Converts the C string in 'src' to a PL/I varying string in 'dest'. (A PL/I string is an array of integers. The first element contains the number of characters in the string. The rest of the array contains the characters, packed two to a word.) Conversion terminates when a '\0' is encountered in 'src'. The function return is the number of characters converted to 'var'. Like other C string routines, no bounds checking is performed (see ctov(2) in the *Software Tools Subsystem Reference Manual*, though).

NOTE: This routine has been changed from the previous release of the C compiler.

. **c$vtoc --- convert PL/I string to C string**

Calling Information:

```
int c$vtoc (dest, src)
char *dest;
int *src;
```

Converts a PL/I varying string 'src' to a C string 'dest'. The function returns the number of characters copied into to 'str'. Again, no bounds checking is done (see vtoc(2) in the *Software Tools Subsystem Reference Manual*).

NOTE: This routine has been changed from the previous release of the C compiler.

_^HC**_^H**o**_^H**n**_^H**v**_^H**e**_^H**r**_^H**s**_^H**i**_^H**o**_^H**n**

The Georgia Tech C compiler is based on the specifications contained in *The C Programming Language* by Kernighan and Ritchie. However the C compiler environment is not totally compatible with the Unix C implementation. Simulation of a Unix environment under Primos can be done only with an unreasonable loss of performance. Therefore, Unix C programs require some conversion to execute on Prime systems. (Programs that depend intimately upon the Unix process mechanism or the Unix file system layout are more difficult to convert. Likewise, programs that make heavy use of Unix inter-process 'signal' interfaces will be difficult to convert.)

C Program Checker

There exist the beginnings of a "C Program Checker" to flag possibly dangerous C program constructs when it encounters them; e.g. type mismatches. The "C Program Checker" can be called by using the "-y" option with 'cc', 'ccl', or 'ucc'. It currently reports on mismatched formal/actual parameters and misdeclared function return values.

Incompatabilities With PDP-11 C

The C compiler is compatible with PDP-11 C where possible. The following list enumerates those features of the Georgia Tech C compiler which are not compatible with PDP-11 C.

Include Statements

The compiler will complain about semicolons appearing at the end of include statements.

Note that the Georgia Tech C compiler automatically includes the standard definitions in "=cdefs=" so that the typical Unix-style "#include <stdio.h>" is optional. The compiler will search for an include file starting with the the current working directory, through the directories listed with the "-I" compiler option in the order listed, and ending with the system include directory "=incl=". Use of angle brackets (e.g., <filename>) rather than double quotes (e.g., "filename") in the include statement directs the compiler to skip the search of the current working directory.

Pointers

It is currently not possible to make pointers and **int**s the same length. Pointers are 32 bits, **int**s are 16 bits. The compiler tries to warn of pointer truncation, but cannot always detect it.

If NULL pointers are to be passed as arguments, they must be of type pointer (e.g. you cannot pass 0 or 0L as a NULL pointer. Use the symbolic constant NULL which is defined in "=incl=/stdio.h" to be "(char*) 0").

Pointers to dynamically linked objects cannot be compared. Pointers to dynamically linked objects (currently only functions are dynamically linked) are actually faulting pointers to character strings. At run time, these pointers are filled in with the correct linkage address (the links are "snapped") the first time the pointer is referenced indirectly. The C compiler must generate a constant pointer to each external object in each C object file. If relocatable files are linked together, during execution it is possible to have one file's constant pointer snapped, and the other's untouched. The object code generated by the compiler to compare these pointers does not reference through the pointers; it merely treats them as 32-bit integers. Because of this, comparisons of pointers to dynamically linked objects may give inconsistent results. A significant performance penalty would be required to guarantee consistent results in such a limited case.

Program and Data Object Size Restrictions

No source file may require more than 65536 words of static data. The static data for each C source file is compiled into a single linkage frame, and the linkage frame size restriction is imposed by the system architecture.

If you do require very large data objects, you may be able to get around this restriction with some work. You must declare the data object as an **extern** and write a Fortran subroutine that declares the data object name as a common block. Then when accessing the contents of this large block you must somehow insure that an object **never** crosses a segment boundary (start it at the beginning of the next segment just as Fortran does). If you attempt to address an object (such as a **double**) across a segment boundary, part of your reference simply wrap around to the beginning of the segment you are trying to reference beyond.

No source file may require more than 65536 words of procedure text. The compiler generates all procedures in the same PMA (Prime Macro Assembler) module. Currently PMA restricts the module size to 65536 words.

No function may generate more than 65536 words of internal-format PMA (currently around 16K statements). This is a code-generator workspace restriction. It has only been encountered

with output from YACC -- functions this huge are just not normally found around PDP-11s. (YACC is an LALR(1) parser generator. Its reads a BNF grammar, and produces a C function which will parse the grammar. This generated output has many large tables.)

Functions

In C, all arguments are passed by value. In Georgia Tech C, as long as arguments match in type they are, in all outward appearances, also passed by value. However, the internal mechanism for parameter passing is different from Unix C and will give different side effects if arguments do not match in type and in number.

The Prime architecture maintains a stack for local variables and provides a 64V mode procedure call argument transfer primitive for passing pointers, but not data values. We have used this mechanism to take advantage of its speed. Therefore, pointers are passed by value, just as in Unix C, but data values are not passed by value; a pointer to the data value is passed into the stack frame of the called procedure; the data value is then copied into the local stack frame by the procedure initialization code. This scheme is transparent as long as there are no type mismatches. For this reason, an attempt to cast a pointer argument to a non-pointer type will fail.

A variable number of arguments can be used, but not in the same manner as in Unix. The strategy is to declare as many arguments as you will ever need (make them pointers so that the compiler does not try to copy them). You will actually ignore all but the first of these names in the function. This trick forces the compiler to leave enough room for your arguments in the procedure's local stack frame. When the function is called, you will find the first argument pointer at the address of the first argument, the second argument pointer at the address of the first argument plus 3, the third at the address of the first argument plus 6, etc. Note that because of software conventions, i.e., the procedure initialization code, functions that are declared with zero arguments must be called with exactly zero arguments; and functions that are declared with one or more arguments must *not* be called with zero arguments.

Programs that depend on the order of parameter evaluation will fail.

You cannot call a function with single precision floating point arguments nor can you ever expect a function to return a single precision floating point argument. Remember, C turns them into double precision.

If a structure is to be a return value, the compiler adds on an additional first argument through which it passes a pointer to a temporary area in the calling procedure for the return value. Needless to say, type or length mismatches could cause

significant nastiness.

The side effects of type mismatches are quite predictable and can be useful for calling non-C procedures. For example, if you pass a non-pointer argument to a pointer argument, it will behave exactly as if a pointer had been passed (i.e. possibly allowing the supposed "value" argument to be modified). If you pass a pointer argument to a simple variable argument it behaves just like you passed the value of the argument instead.

Be wary of non-C routines which modify their arguments (particularly Subsystem routines like 'ctoi'); if you pass a constant, the "constant" might end up with a different value in it than it had before the routine was called!

Arrays

Although it is possible to index outside of array bounds, doing so is very dangerous. In 64V mode, indexed instructions are much faster than 32-bit pointer arithmetic. As a consequence, the compiler generates 16-bit indexed instructions wherever possible. The only side effect of this performance improvement is that indexing outside the bounds of arrays may not give the expected results.

Identifiers --- Naming Restrictions

Because the C compiler originally generated symbolic assembly language which was then processed by PMA, the Prime Macro Assembler, variable and function names had to follow PMA's naming conventions which require that names begin with an alphabetic character. To achieve the necessary compatibility, variable and function names beginning with an underscore are prefixed with "z$". Even though 'vcg' now generates object code directly, this naming restriction is still in effect.

Field names within **struct**s must be unique since the C compiler does not maintain a separate symbol table for each **struct**. This behavior is in accordance with K&R and the V7 Unix C compiler. (Berkeley Unix, System III, and System V, all keep separate symbol tables for each structure.)

Character Representation and Conversion

Character values run from 128-255, not 0-127.

Characters are not sign extended when promoted to integers.

Numerical

Programs that use data of type **double** may lose precision in trade for increased magnitude.

See the *SWT Math Library User's Guide* for more details on Prime's floating point hardware and software.

Library Incompatibilities

The Unix call 'fork' cannot be efficiently implemented because of operating system restrictions, and is therefore not available with Georgia Tech C.

'Read' and 'write' calls that do not use 'sizeof' to compute the buffer length will probably have to be changed.

Programs that open other users' terminals can not be supported.

Unix File System Incompatibilities

Programs that depend intimately on the Unix directory structure ('..', directory layout, links) will not be easily converted.

Programs that depend on the order and behavior of Unix file descriptors will not be easily converted.

You cannot depend on file descriptors 0, 1, and 2 always being connected to standard input, standard output, and standard error respectively. Instead, use the macros STDIN, STDOUT, and STDERR (defined in "=incl=/swt.h", which is automatically included by "=cdefs=").

Tabs

Tabs are not supported in exactly the same manner on the Prime as in Unix. C programs which produce tabs in their output should be run piping their output into the Subsystem program 'detab' ('detab +8' is recommended).

Static Initializers

Initializers for static data objects which involve the "address-of" operator may only consist of "&objectreference". For example, while the statement "static char *x = &A" is okay, the statement "static char *x = &A+1" cannot be handled by the Georgia Tech C compiler. The restriction arises from the inability of PMA/SEG to handle address expressions of external symbols when forming 32-bit pointers.

Registers

In 64V mode, the Prime is essentially a single accumulator machine. Thus, while the compiler recognizes the **register**

keyword, there is no effect on the size or speed of the generated code.

The Type void

Berkeley and System III Unix introduced the new type **void** into the C language. A **void** function is one which is guaranteed not to return a value (i.e. a true procedure). Only functions may be declared to be of type **void**, although you may also cast a function call to **void**. Georgia Tech C does not directly support **void**, but you may get around it with the simple statement:

```
#define void int
```

which should allow you to port practically any code which uses **void**. Admittedly, this defeats some of the type checking that the new type provides, but it will allow you to port code, without having to modify it.

_^H**B**_^H**u**_^H**g**_^H**s**

Known Bugs

The following is a list of known problems with the C compiler, as well as important enhancements that need to be made.

1. In certain instances, the compiler's attempt at parsing error correction fails to accept an input token. This can result in an infinite loop in the parser as it encounters and reports the same error repeatedly. A good example is placing an extra semicolon after the right brace of the statement in an **if**, before the corresponding **else**. The compiler will halt after reaching a limit of 50 such messages.

2. Bit fields must be initialized by execution time assignments; compile time initialization does not work correctly.

3. If "f" and "g" are type float, then "f*=g" is performed in single precision, whereas "f=f*g" is performed in double precision. We have not made a detailed analysis of the ramifications of this situation; it may be that no loss of precision can be detected. Regardless, because of the structure of the code generator, it will be very difficult to alter this situation.

4. The preprocessor does not support the "#if" construct.

5. The parameter-checking option ("-y") does not check calls to the C library.

6. A duplicate **case** in a **switch** is not detected by the first pass of the compiler. It will cause an error (with no information regarding the location of the error!) message to be reported by 'vcg'. In some cases, no error is reported, but 'vcg' generates unreasonable code.

7. There are several problems involving duplicated declarations for an external/global identifier (e.g. "extern a; int a;"). Most reasonable redeclarations are handled correctly, but some of the more obscure cases are probably not handled the way the Unix compilers handle them. In general, correct handling of these odd cases is not described explicitly--to find out how they "should" be handled, you have to ask a Unix compiler (and they often give different answers).

8. The sequence of declarations "extern int a[]; int a[5];" generates a warning message that "a" is being redeclared improperly. This is caused by the differing array bounds confusing the compiler into thinking that the second declaration is unreasonable. This is a definite bug (as

opposed to a question of interpretation).

9. The constant -2^-31 (smallest 32-bit negative number) is mishandled in all bases ('gctoi' goofs on it). For the time being, instead of using "0x80000000", use "(1<<31)" or "(~0x7FFFFFF)". These will give identical results because the constant folder gives correct results.

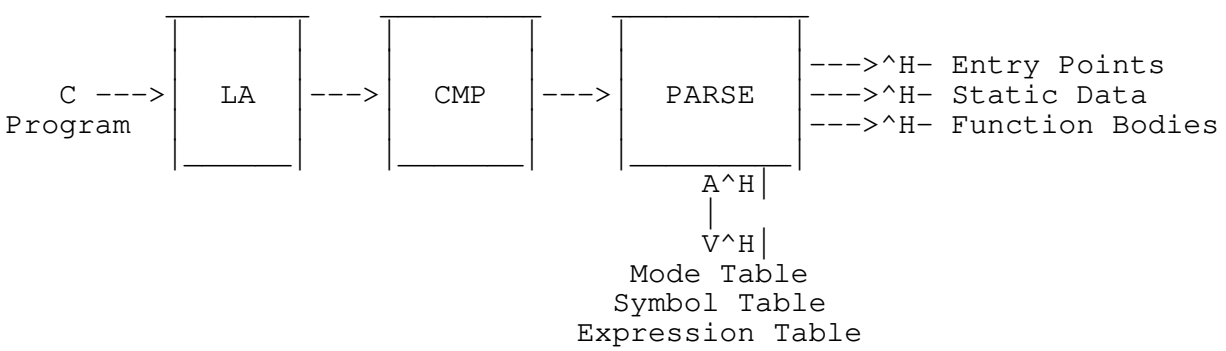
10. The construct "p++->x" confuses the compiler and causes it to complain about missing parentheses. This is because "->" is of higher precedence than "++" and thus confuses the recursive-descent parser. You should write the expression as "(p++)->x".

11. The new version of the code generator still has some bugs in it. If it produces an object file which causes an error from the loader, you may wish to compile the program with the "-s" option, to generate PMA in a ".s" file. Then use 'pmac' to assemble it, and load the new binary. This will usually work; it will simply take longer to compile.

Technical Information

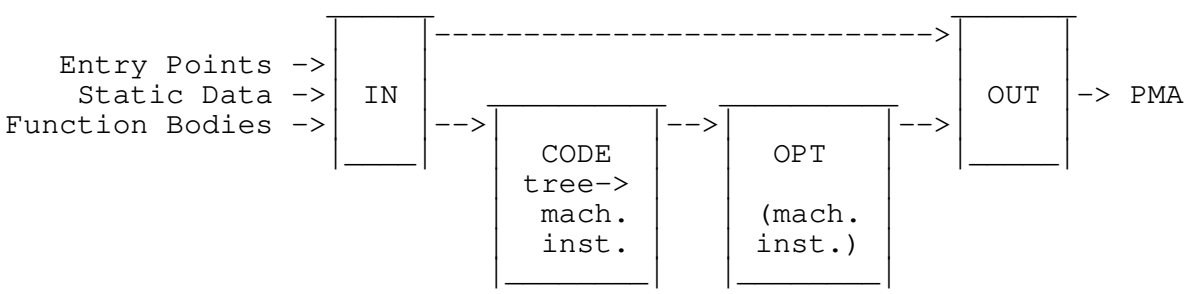
Implementation

The C compiler accepts C source code as input and in two "passes" produces 64V-mode relocatable object code as output. The first compilation pass is implemented by a Ratfor program known as the "front end," and the second pass by a Ratfor program known as the "back end." The front end is the Subsystem program ’c1’ and the back end is ’vcg’.



The "Front End." LA is the lexical analyzer, or scanner. CMP is the C macro preprocessor. PARSE is the parser and intermediate code generator.

The front end is a classical recursive-descent compiler, employing a lexical analyzer (to break the stream of input characters into tokens), a preprocessor (to handle macro definitions and source file inclusion), and a parser (to analyze the program, diagnose syntactic and semantic errors, and produce an "intermediate form" output stream).



The "Back End." IN reads the intermediate code produced by the front end. CODE attempts to "intelligently" generate machine instructions. OPT performs some simple peephole optimizations to remove redundant loads and stores. OUT converts the internal instruction form to 64V-mode object code, or, optionally, to Prime Macro Assembly Language.

The back end is a reusable general-purpose code generator. It accepts the linearized intermediate form tree produced by the front end, rebuilds the tree internally, converts the tree to a linked list of machine instruction representations, performs peephole optimizations on that list, and then produces 64V-mode object code, ready for link editing and subsequent execution. 'Vcg' has an option for producing symbolic assembly language instead of object code. The assembly language that 'vcg' produces is suitable for processing by the Prime Macro Assembler, PMA.

For those of you wishing to supply your own front-ends to the code generator, there is a *V-mode Code Generator User's Guide* (use the Subsystem command 'guide') and a Reference Manual entry for 'vcg'.

Performance

The C compiler requires parts of 5 segments to run. The previous version of the C compiler, which used to call PMA, ran almost twice as fast as Prime's Fortran 77 and Pascal compilers (700 lines per minute vs 400 lines/minute on a PRIME 550 running under Primos 18.1). Hand inspection and informal benchmarks indicate that the code produced is superior to that produced by Pascal, PL/1 and Fortran 77; in particular, fewer base register loads are generated, and operations on packed data structures are performed without resorting to the field manipulation instructions.

The compile time requirements for each phase were approximately as follows: 'c1': 23%, 'vcg': 27%, 'pma': 50%. Roughly half of 'vcg's time was spent in the assembly-language output routines.

For the second release, 'vcg' has been changed to produce 64V-mode object modules directly. This substantially reduces compile time. We have not measured the new version of the compiler, but the compile time requirements for each phase are about equal. The total compile time is now approximately half of what it was, since PMA is not involved in the process.

Subsystem Managers Section

The machine-readable text of the *User's Guide for the Georgia Tech C Compiler* is in the file "=doc=/fguide/cc" (already formatted) and in the directory "=doc=/guide/cc" (unformatted) (assuming that you have already installed the C compiler according to the directions below).

Installation Procedure

The C compiler and its support programs are intended to be part of the Subsystem. Source, documentation and executable versions "drop in" to appropriate Subsystem directories so that they are accessible as standard Subsystem tools. This section covers the procedures necessary for installation of the C compiler.

Georgia Tech C Installation Package

The C Installation Package as sent from Georgia Tech contains the following items:

1 Release Tape
1 Copy of the C User's Guide

Release Tape Contents

The C Release Tape contains all files and directories necessary for proper operation of the C compiler under the Software Tools Subsystem. It is in standard MAGSAV/MAGRST format and contains 5 "logical tapes." Each logical tape contains a number of files that "drop in" to Subsystem directories.

Logical Tape 1

The first logical tape contains executable files that are to be placed in "=bin=",

cc ccl compile ucc vcg vcgdump

'Cc' is the Subsystem C compiler, 'ccl' is a shell file that compiles and loads a C program, 'ucc' is a 'Unix-like' C compiler and 'vcg' is the V-mode code generator. 'Vcg' is used by the C

compiler but can also be used separately by those users who have their own "front ends." 'Vcgdump' reads the intermediate files produced by 'c1', and prints a human-readable version of the intermediate form tree. 'Compile' is a general purpose compiler interlude.

Logical Tape 2

The second logical tape contains libraries for "=lib=",

```
ciolib   c$main   nciolib   vcglib   vcg_main
```

"Ciolib" contains the executable version of the C run time library; "c$main" is a small startup program that must be loaded with every C main program. "Nciolib" is the version of the C run time library for programs which are to run under bare Primos.

'Vcglib' is a library of regular and shortcall routines for range testing and other purposes. 'Vcg' generates calls to these routines for their operations, instead of generating code. 'Vcg_main' is a small general purpose start off routine. These are not used by C programs, but are necessary if you wish to provide your own "front end" for 'vcg'.

Logical Tape 3

The third logical tape contains files for "=extra=",

```
bin/(c1   cc   cck1   cck2   compile   ucc)

incl/(swt_def.c.i    ascii.h    assert.h    ctype.h
      debug.h        lib_def.h  keys.h      math.h
      memory.h       setjmp.h   stdio.h     swt.h
      swt_com.h      varargs.h)

incl/(vcg_defs.h   vcg_defs.p.i   vcg_defs.r.i)
```

'C1' is the "front end" for the C compiler and is called by 'cc', 'ccl', and 'ucc'. 'Compile' is a general purpose compiler interlude. 'Ucc' calls it. 'Cck1' and 'cck2' are the "trouble spotters" for C programs. They will flag potentially dangerous constructs in a C program and are invoked by compiling a program with "ucc -y". Subsystem definitions for the C compiler are contained in "swt_def.c.i". The ?*.h files are other header files, discussed above in the chapter on the compile time environment.

The vcg_defs.?* files contain constant definitions for use in writing "front ends" for 'vcg'.

Logical Tape 4

The fourth logical tape contains documentation for "=doc=",

```
man/s1/(cc.d    ccl.d       ucc.d
        vcg.d   vcgdump.d   compile.d)

man/s5/(c1.d   cck1.d   cck2.d)

fman/s1/(cc.d    ccl.d       ucc.d
         vcg.d   vcgdump.d   compile.d)

fman/s5/(c1.d   cck1.d   cck2.d)

guide/(cc vcg)

fguide/(cc vcg)
```

Logical Tape 5

The fifth logical tape contains source files for "=src=",

```
std.sh/(cc.sh   ccl.sh   ucc.sh   compile.sh)

ext.c   # new directory with source files for C interludes

ext.r/(cck1.r   cck2.r   cck2_com.r.i   cck2_def.r.i)

lib/(cio   c$main    nc$main   vcg    vcg_main)

spc/(c1.u   vcg.u)

std.r/(vcgdump.r   vcgdump_com.r.i)
```

If you do not have a source license then you will not recieve any of the source files. In fact, the "src" directory will not be on the tape.

Loading the Tape

To load the release tape, follow the instructions below:

1. Assign a tape drive:

   ASSIGN MT0

2. Mount the release tape on the assigned drive.

3. Attach to directory "=bin=":

   ATTACH BIN <owner-password>

or if the tape is being restored to an ACL or priority ACL protected partition, type

ATTACH BIN

4. Load the contents of the first logical tape with MAGRST:

MAGRST
Tape Unit (9 Trk): **0**
Enter logical tape number: **1**
<tape label information>
Ready to Restore: **yes**

(This loads the files "cc", "ccl", "ucc", "compile", "vcg", and "vcgdump".)

5. Attach to directory "=lib=":

6. Load the contents of the next logical tape (i.e., reply "0" to the "Enter logical tape number:" prompt) with MAGRST. (This loads the library files for 'cc' and 'vcg'.)

7. Attach to directory "=extra=":

8. Load the contents of the next logical tape with MAGRST. (This loads the support programs for the compiler interfaces.)

9. Attach to directory "=doc=":

10. Load the contents of the next logical tape with MAGRST. (This loads the formatted and unformatted 'vcg' and C compiler guides, and the formatted and unformatted Reference Manual (help) entries.)

11. Attach to directory "=src=":

12. Load the contents of the next logical tape with MAGRST. (This loads the source code for the C compiler, the run_time library, the compiler interfaces, the V-mode code generator, and the vcg support routines.) If you do not have a source license, and/or you have received a demonstration tape, this logical tape will not be present.

This completes the loading of the C compiler from tape.

Installation

Once you have loaded the tape, the C compiler is ready to use. However, for the C compiler programs to appear in the "help" index, you must rebuild it by executing 'man_index' in "=doc=/build".

Finally, for the 'locate' and 'source' commands to work correctly, you have to rebuild the =srcloc= file. To do this, 'cd' to "=src=/misc", and execute the file "make_srcloc". This completes the integration of the C compiler with the rest of the Subsystem.

TABLE OF CONTENTS

Getting Started

Features of Georgia Tech C

Compile Time Facilities

Run Time Environment

Conversion

Bugs

Technical Information

Subsystem Managers Section

A Re-Usable Code Generator for Prime 50-Series Computers

User's Guide

T. Allen Akin

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

March, 1983

Foreword

Although the School of Information and Computer Science has operated Prime 400 and 550 computers for over four years, as yet there has been no successful local attempt to produce a compiler for them. The main reasons for this failure are the irregularity of the architecture and existing system software, the complexity of Prime's standard object code format, and the lack of documentation on matters of importance to compiler writers.

This paper discusses the design, implementation, and usage of a re-usable code generator. This program can serve as a common "back-end" for a number of language translators, producing 64V-mode assembly language code suitable for execution on the P400 and higher numbered processors in Prime's "50" series. Furthermore, it could be tailored to match specific front-ends, when needs for special optimizations arise.

A preliminary version of the code generator is available for general use.

How to Use This Guide

The first chapter of this **Guide** is the *Overview*. The *Overview* is a brief summary of the design and construction of the code generator. This chapter may be of general interest, but it is not necessary to read it in order to learn to use the code generator.

The *Code Generator Usage* chapter describes the location of the code generator and its associated run-time support libraries, as well as the Software Tools Subsystem commands necessary to access them. Recommended procedure is to study this section, then generate command language programs to do the low-level file access operations.

Input Data Stream Formats gives a bird's-eye view of the formats of the three code generator input streams. This chapter merits some study, although it is supplemented by the *Extended Examples*.

The three operator definitions chapters (*Operators Useful in the Static Data Stream*, *Operators Useful in the Procedure Definition Stream*, *Operators Useful in Procedure Definitions*) provide a detailed reference for the intermediate form operators interpreted by the code generator. One or two readings through this chapter are desirable; thereafter, it can be used as a reference with the *Operator/Function Index* and the Table of Contents used as entry points.

The *Extended Examples* are comprised of several short (but complete) programs written in the language C. These examples include the original C code, annotated versions of the three code generator input streams, and an annotated listing of the code generator's assembly language output. The chapter should be useful in learning how the various intermediate form operators work together, and may be used as a reference when building a new front end.

'Drift' is a very small expression-based language whose structure closely mimics the code generator's internal world-model. *The 'Drift' Compiler* is a complete, working compiler using the code generator as a back-end. It serves as an example of one way to construct a front-end for the VCG.

For ease of reference, all the intermediate form operators have been organized by subject in the *Intermediate Form Operator/Function Index*. Typically, one would look up some function (e.g., "subscripting") in the *Index*, find the name of the appropriate intermediate form operator (e.g., INDEX_OP), then look up that operator in the table of contents to find its complete description.

Overview

Philosophy

Design Considerations

The design of the code generator (hereinafter referred to as VCG, for "V-mode Code Generator") was driven by a number of considerations:

. For experimental language translators, code generation should be fast and straightforward. This is necessary both for fast turnaround and ease of debugging in the development stage, and for fast turnaround in typical educational applications.

. The VCG should insulate front-ends from details of storage allocation and data format selection, as well as instruction generation. This encourages inter-language compatibility at the object code level, as well as providing a framework for easily retargetable front-ends.

. The intermediate form (IMF) that is processed by the VCG should be simple to generate and display (for debugging purposes). Furthermore, it should not unduly restrict extension for additional functionality or optimization.

. The output object code should conform to Prime's current standards, and should include at least minimal provisions for separate compilation and run-time debugging.

Implementation Approaches

After some time, consideration of the goals above led to the following approaches to the implementation of the VCG:

. The basic IMF handled by both the front end and the VCG should be a tree structure. A tree is easily generated from the information available on the semantic stack during a bottom-up parse, and can be generated directly without an explicit stack during a top-down parse. A number of operations like constant folding, reordering of operands of commutative operators, and global context propagation are readily performed on a tree structure. Furthermore, use of a tree can eliminate the need for generation and tracking of temporary variables in the front end.

. The IMF operators should be close to the constructs used in an algorithmic language of the level of, say, Pascal. This permits straightforward translation of most algorithmic

languages, and provides enough additional context to simplify many optimization tasks. For example, the IMF resembles the program's flow graph closely enough that simple global register allocation can be performed without graph reduction.

. One of the basic functions of the VCG is the mapping of data descriptions supplied by the front end into physical storage layouts. The goal of complete machine data structure independence in the front end cannot be met without compromising the code generator's utility for languages that allow storage layout specification (C and Ada are notable examples). Therefore, the IMF should contain descriptions of data structures in terms of a small set of primitive data modes that can easily be parameterized in front-end code. Simple variables, structures, and arrays defined in the front end must be converted to single or multiple instances of the following basic machine data modes: 16-bit signed integer, 16-bit unsigned integer, 32-bit signed integer, 32-bit unsigned integer, 32-bit floating point, and 64-bit floating point.

. The IMF tree should be linearized and passed to the VCG as a stream of data in prefix Polish notation. The linearized form partly reflects the usual Software Tools methodology of expressing even complex data transformations as "filters." However, there are other advantages, particularly in storing and interpreting the IMF for debugging. Prefix Polish was chosen because it can be generated easily from the internal representation of the tree, and because it minimizes the amount of state information that must be explicitly maintained by both the front end and the VCG in order to output or input the IMF.

. The final output of the VCG should be a stream of Prime Macro Assembly Language source code. Although the time required to assemble this source imposes a significant penalty on code generator performance, it appears to be unavoidable if the compiler writer is to be insulated from Prime's object code format. (In addition, Prime has scheduled object code format changes, and it would not be wise to invest heavily in the present format.)

Structure

The VCG "main loop" simply reads each module present on its input, rebuilds the tree represented by the input, transforms the tree to a linked list of machine instructions, performs register tracking optimizations on that list, and finally converts the list to assembly language and outputs it.

The input and output routines are straightforward and relatively uninteresting.

The optimization routines amount to about 13 pages of Ratfor code, and work by simulating the effect of each machine instruction on the contents of the six registers that are tracked. At the moment, three types of optimization are performed: redundant loads are eliminated, some memory references are eliminated in favor of register-to-register transfers, and general instruction sequences are replaced with special-case code.

The heart of the code generator is the set of transformation routines that convert the tree representation to the doubly-linked list of machine instructions. The transformation routines exhibit a great deal of knowledge about the machine architecture, but actually employ only very simple algorithms for code generation.

IMF operators may appear in one of several "contexts," identified internally by the following terms:

Reach. An operator evaluated in reach context yields the address of a word in memory containing the result of the operation, if possible. At present, only the object, constant, pointer dereferencing, array indexing, and structure member selection operators yield addresses. All other operators behave as if they were evaluated in "load" context.

Load. An operator evaluated in load context yields a value in a machine register. The particular register used depends only on the basic machine data mode of the operation. Most IMF operators are evaluated only in this context.

Void. An operator evaluated in void context yields side effects only. In a very few cases, this results in an opportunity to exploit special-case machine instructions that perform some calculation without making the result available in a register (incrementing a memory location, for example).

Flow. An operator evaluated in flow context yields a change in flow-of-control rather than a value. For example, a "test for equality" operator would return 1 or 0 in a load context, but in flow context would cause a jump to a given label depending on the outcome of the test.

AP. An operator evaluated in AP context yields an "argument pointer" rather than a value. Argument pointers are used to pass parameters to procedures.

Context information is propagated top-down by the code generator as it scans the IMF tree. Additional information in the form of register requirements is propagated from the bottom up during the same scan. Together, context and register usage determine with fair accuracy the optimal code sequence to be generated for a given operator.

Input/Output Semantics

Input Structure

The IMF passed to the VCG consists of a sequence of *modules*. A module is a sequence of procedure definitions, static data definitions, and entry point declarations. The static data definitions build a data area that is shared by all procedures in the module, while the procedure definitions build code and data areas that are strictly local to each procedure, and the entry point declarations make the static data area or procedures visible to Prime's link editor.

Prime's Fortran compilers currently generate code that is equivalent to one procedure per module under this scheme; Prime's PL/I and Pascal compilers generate code that is equivalent to a single IMF module. The VCG module structure permits compatibility with either of these alternatives, as well as compromise forms that are more suitable for other languages.

> Note: Separate compilation capability directly affects module structure. At present, there is no way for separately compiled procedures to share a static data area. Furthermore, separately-compiled static objects must be referenced by a unique 8 or fewer character name made visible to the loader. A Fortran COMMON block definition can be used to reduce the number of such external symbols, but COMMON definitions must match exactly in all separately-maintained modules. In addition, note that Prime's current loader software requires that external objects be referenced through an indirect address, which can cause a significant reduction in performance.

Each *static data definition* allocates space for an object and may specify an initial value for the object. A *static data declaration* names an object that is defined outside the current module, but provides no other information about the object.

Each *procedure definition* consists of information associated with a closed routine defined by the front end. In particular, the procedure's argument types and code tree are included.

The bulk of the IMF will be in subtrees defining the code associated with procedures. Most storage allocators, arithmetic operators, and flow controllers are straightforwardly expressed in tree form; a description of these IMF components is available elsewhere.

Output Structure

Each VCG input module generates a single PMA input module, terminated by an END pseudo-op. The PMA input module may be

assembled, link-edited, and subsequently executed. The concatenation of all static data definitions and declarations forms a *link frame* that is shared by all procedures in the module. Each procedure definition yields an entry control block (ECB) and a chunk of machine code that implements the function of the procedure, including the allocation of space in the procedure's *stack frame* for local variables.

Code Generator Usage

The code generator currently resides in the file =bin=/vcg. The three input streams can be read from the three standard inputs, or from three files (if a standard naming convention is used). The PMA output stream is produced on standard output 1, and should be redirected to a file for assembly.

Assume temporary files will be used for communications between the front end and the code generator. The temporary files must have names of the form "xxx.ct1" (for IMF stream 1), "xxx.ct2" (for IMF stream 2), and "xxx.ct3" (for IMF stream 3), where "xxx" is completely arbitrary but must be the same for all of the three temporary files in a given run. When the code generator is invoked, the string "xxx" must be passed to it as a command line argument.

To use the code generator, first run the front end to produce the temporary files:

```
front_end
```

Say, for example, this produces files "temp.ct1", "temp.ct2", and "temp.ct3". Next, run the code generator and produce the assembly language output:

```
vcg temp >temp.s
```

Run the assembler to convert the PMA source to relocatable binary code:

```
pmac temp.s
```

Finally, run the link editor to load the VCG main program, the binary code for your program, and all required library routines:

```
ld =lib=/vcg_main temp.b =lib=/vcglib -o program
```

This produces an object program (in the file "program") which may be executed simply by typing its name:

```
program
```

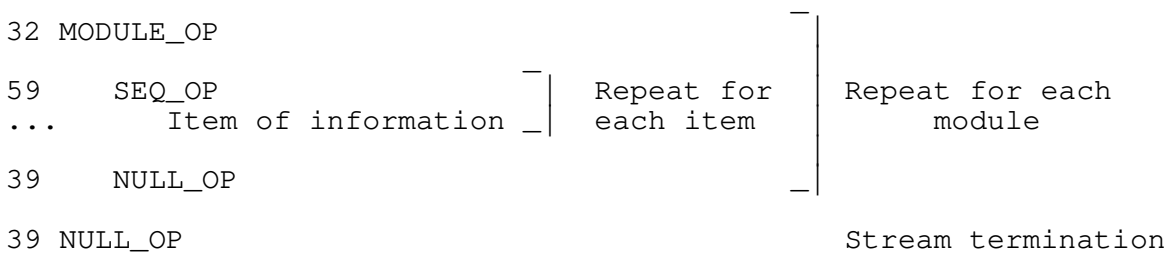
All run-time support routines called by the output of the code generator are available in the library =lib=/vcglib. The stub main program in =lib=/vcg_main calls a procedure named MAIN; therefore, the user's main program must be named MAIN. (This is the usual case in C environments.)

One miscellaneous note: if the front end is being written in Ratfor, the complete set of macro definitions for the intermediate form operators can be obtained by simply including

the file "=incl=/vcg_defs.r.i". If the front end is being writ-
ten in Pascal, the complete set of constant definitions for the
intermediate form operators can be obtained by including the file
"=incl=/vcg_defs.p.i".

Input Data Stream Formats

This section describes the formats of the three code generator input streams. Note that all three have the same basic format:



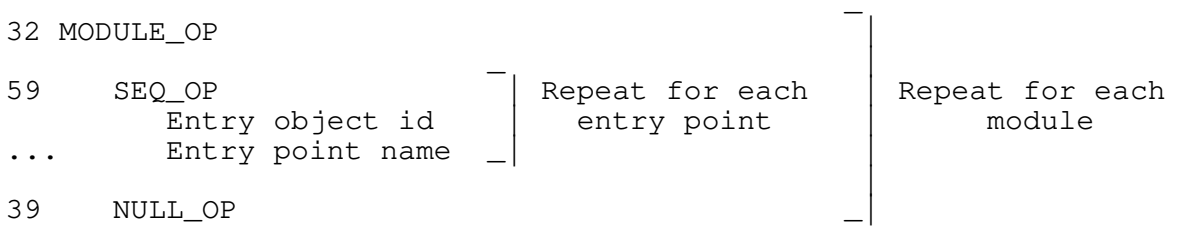
Detailed examples of the code generator input can be found in the "Extended Examples" section of this guide.

Stream 1 --- Entry Point Declarations

The first intermediate form stream consists of one or more *modules*. Each module consists of a MODULE_OP, a list of *entry point declarations* separated by SEQ_OPs, and a NULL_OP terminating the list of entry point declarations. The list of modules is terminated by a final NULL_OP.

Each entry point declaration is an object identification number followed by a character string, expressed as the length of the string followed by the ASCII character codes for the characters in the string. Each such string is assumed to be the name of a location defined in the current input module, and is made available to the link editor for resolving references made by other modules.

A template for stream 1 would look something like this:

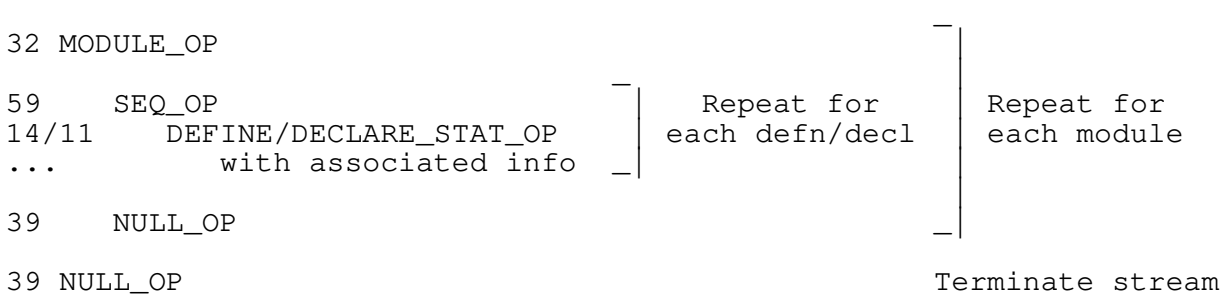


Stream 2 --- Static Data Declarations/Definitions

In C terminology, a data "definition" reserves storage space for an object and possibly initializes that space, whereas a data "declaration" simply indicates that the storage space for an object resides outside the current module. The second intermediate form input stream defines or declares static data (objects that are not automatically allocated on the stack when a procedure is entered).

The input stream consists of a series of *modules*, terminated by a NULL_OP. Each module contains a sequence of *DEFINE_STAT_OP*s and *DECLARE_STAT_OP*s, terminated by a NULL_OP.

A template for the static data stream would look something like this:

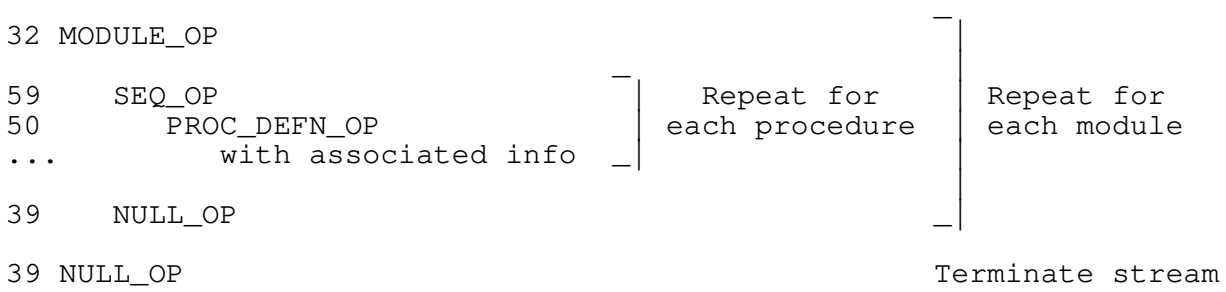


Stream 3 --- Procedure Definitions

The third intermediate form input stream consists of one or more *modules*, terminated by a NULL_OP. Each module contains a list of *PROC_DEFN_OP*s, separated by SEQ_OPs and terminated with a NULL_OP.

Each PROC_DEFN_OP causes a procedure to be defined and code for it to be generated.

A template for stream 3 would look something like this:



Primitive Data Modes

The following primitive data modes are presently handled by the code generator:

INT_MODE 1

Integer objects are one 16-bit word in size. They have integral values in the range (-2**15) to (2**15 - 1), inclusive.

LONG_INT_MODE 2

Long integer objects are two 16-bit words in size. They have integral values in the range (-2**31) to (2**31 - 1), inclusive.

UNS_MODE 3

Unsigned objects are nominally one 16-bit word in size. They have integral values in the range 0 to (2**16 - 1). Bit fields (see FIELD_OP) can be of mode UNSIGNED, and may range from 1 bit to 16 bits in length (with consequent change in the range of values they can represent).

LONG_UNS_MODE 4

Long unsigned objects are nominally two 16-bit words in size. They have integral values in the range 0 to (2**32 - 1). Machine addresses (pointers) are represented as long unsigned quantities. Bit fields (see FIELD_OP) can be of mode LONG UNSIGNED, and may range from 1 bit to 32 bits in length (with consequent change in the range of values they can represent).

FLOAT_MODE 5

Floating point objects are two 16-bit words in size.

LONG FLOAT_MODE 6

Long floating point objects are four 16-bit words in size.

STOWED_MODE 7

STOWED mode is the mode assigned to structured objects like arrays and structs (Pascal "records"). STOWED objects may be any size from 1 to 65536 16-bit words; IMF operators that need to know the size of a STOWED object invariably have a "length" or "size" parameter to carry that information.

Operators Useful in the Static Data Stream

DECLARE_STAT_OP 11

```
int 11
int object_id
string external_name
```

DECLARE_STAT informs the code generator that an object defined outside the current module will be referenced by a given integer object id. The parameter 'external_name' is a character string, represented in the IMF by a length followed by a stream of ASCII characters (one per word, right justified, zero filled). The external name is used by the link editor and the loader to resolve actual references to the object.

Example: extern int abc
where 'abc' is assigned the object id 6

```
11       DECLARE_STAT_OP
6           Object id of 'abc'
3           Length of name 'abc'
225            character 'a'
226            character 'b'
227            character 'c'
```

DEFINE_STAT_OP 14

```
int 14
int object_id
tree init_list
int size
```

This operator causes storage for the object identified by 'object_id' to be allocated in the current link frame (static data area). 'Object_id' must be used in all subsequent references to the object, and the object's definition with DEFINE_STAT must precede all such references. The init_list is a list of initializers whose values will be assigned to successive portions of the newly-declared object. The size parameter specifies the amount of storage to be reserved for the object, in words. (Slightly fewer than 65,535 words are available for static storage in each module.)

Example: static int abc[100]
where abc is assigned the object id 6

```
14       DEFINE_STAT_OP
6           Object id for 'abc'
39          NULL_OP (no initializers present)
```

```
100         Object is 100 words long
```

Operators Useful in the Procedure Definition Stream

PROC_DEFN_OP 50

```
int 50
int object_id
int number_of_args
string proc_name
tree argument_list
tree code
```

Each procedure to be generated by the code generator is defined by a PROC_DEFN_OP. The 'object_id' is an integer identifier that must be used on calls to the procedure and other references to its entry control block (for example, pointers to functions as used in C). 'Number_of_args' should be self-explanatory. 'Proc_name' is a string (in the IMF, a length followed by ASCII character values) giving the internal name of the procedure. (This information is used to print trace information during debugging.) Each formal parameter (argument) is described by a PROC_DEFN_ARG_OP; 'argument_list' is simply a linked list of those descriptions. 'Code' is a subtree containing the body of the procedure: local variable definitions and expressions to be evaluated.

Example: the following C function

```
main (argc, argv)
int argc;
char **argv;
   {
   int i;
   i = 4;
   }
```

```
50       PROC_DEFN_OP
1           Procedure is object number 1
2           Procedure has 2 arguments
4           Procedure name is 4 characters long
237            m
225            a
233            i
238            n
49          PROC_DEFN_ARG_OP
2              Argument is object number 2
1              INT_MODE
0              VAL_DISP; pass argument by value
1              Argument is 1 word long
49          PROC_DEFN_ARG_OP
3              Argument is object number 3
4              LONG_UNS_MODE (a pointer)
1              REF_DISP; pass argument by reference
2              Argument is 2 words long
```

```
39          NULL_OP; end of argument description list
59          SEQ_OP; beginning of procedure code
13             DEFINE_DYNM_OP
4                 Object id is 4
39                NULL_OP; no initializers
1                 Object is 1 word long
59          SEQ_OP; procedure code continues
5              ASSIGN_OP
1                 INT_MODE
40                OBJECT_OP
1                    INT_MODE
4                    Object id is 4
9                 CONST_OP
1                    INT_MODE
1                    Constant is 1 word long
4                    Constant has value 4
1                 Assignment transfers 1 word
39          NULL_OP; end of procedure code
```

PROC_DEFN_ARG_OP 49

```
int 49
int object_id
int mode
int disposition
int length
tree next_argument
```

Formal parameters to procedures are described by this operator. The 'object_id' is an integer identifier that must be supplied on subsequent references to the parameter (see OBJECT_OP). The 'mode' is the machine data type of the parameter. 'Disposition' indicates how the argument is to be treated on the call; the two alternatives at the moment are 0 (VALUE_DISP) for pass-by-value (aka copy in) and 1 (REF_DISP) for pass-by-reference. 'Length' gives the size of the argument in 16-bit words; it is primarily necessary for handling of STOWED arguments that are passed by value. 'Next_argument' is simply a link to the next PROC_DEFN_ARG_OP in a procedure's argument descriptor list, or a NULL_OP.

See PROC_DEFN_OP for examples of PROC_DEFN_ARG_OP.

Operators Useful in Procedure Definitions

ADDAA_OP 1

```
int 1
int mode
tree left
tree right
```

The result of this operator is an rvalue, the sum of the values of the left and right operands. As a side effect, the sum is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the ADDAA operation. The operation mode may not be STOWED.

ADDAA stands for "add and assign." This operator is normally used to implement the addition assignment operator ("+=" in C, "+:=" in Algol 68).

Example: i += 1 (where i is an integer object with object id 12)

```
1               ADDAA_OP
1                  INT_MODE
40                 OBJECT_OP
1                     INT_MODE
12                    Object id 12
9                  CONST_OP
1                     INT_MODE
1                     length is 1 word
1                     value of first word is 1
```

ADD_OP 2

```
int 2
int mode
tree left
tree right
```

The result of this operator is an rvalue, the sum of the values of the left and right operands. Both operands must have the same mode as the ADD, and STOWED mode is not allowed.

ADD is used to implement simple addition of fixed or floating point values.

Example: i + 1 (where i is an integer object with object id 12)

```
2               ADD_OP
1                  INT_MODE
40                 OBJECT_OP
1                     INT_MODE
```

```
12                   Object id 12
9                 CONST_OP
1                    INT_MODE
1                    length is 1 word
1                    value of first word is 1
```

ANDAA_OP 3

```
int 3
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise logical "and" of the values of the left and right operands. As a side effect, the conjunction is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the ANDAA operation; the only allowable modes are INT, UNSIGNED, LONG INT, and LONG UNSIGNED.

ANDAA stands for "'and' and assign." ANDAA_OP is used to implement the logical-and assignment operator ("&=" in C).

Example: i &= 1 (where i is an integer object with object id 12)

```
3               ANDAA_OP
1                  INT_MODE
40                 OBJECT_OP
1                     INT_MODE
12                    Object id 12
9                  CONST_OP
1                     INT_MODE
1                     length is 1 word
1                     value of first word is 1
```

AND_OP 4

```
int 4
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise logical-"and" of the values of the left and right operands. Both operands must have the same mode as the AND operation; the only allowable modes are INT, LONG INT, UNSIGNED, and LONG UNSIGNED.

AND_OP is normally used to implement the bitwise logical conjunction of integers ("&" in C). Although AND_OP can be used to implement conjunction in Boolean expressions, the short-circuit

conjunction operator (SAND_OP) is probably a better choice, since it guarantees evaluation order and prevents undesirable side effects.

```
Example:  i & 1 (where i is an integer object with object id 12)
         4              AND_OP
         1                 INT_MODE
         40                OBJECT_OP
         1                    INT_MODE
         12                   Object id 12
         9                 CONST_OP
         1                    INT_MODE
         1                    length is 1 word
         1                    value of first word is 1
```

ASSIGN_OP 5

```
int 5
int mode
tree left
tree right
int length
```

The result of this operator is an rvalue, namely the value of the right operand. As a side effect, the result is stored into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the ASSIGN operation. Any mode is allowable, but the parameter 'length' must be set to the operand length, in 16-bit words.

ASSIGN implements the semantics of assignment statements in most algorithmic languages. Note that STOWED mode values are allowed, so things like Pascal record assignment can be handled.

```
Example:  i = 1 (where i is an integer object with object id 12)
         5              ASSIGN_OP
         1                 INT_MODE
         40                OBJECT_OP
         1                    INT_MODE
         12                   Object id 12
         9                 CONST_OP
         1                    INT_MODE
         1                    length is 1 word
         1                    value of first word is 1
         1                 length of assigned quantity is 1 word
```

BREAK_OP 6

```
int 6
int levels
```

BREAK_OP yields no result value, but causes an exit from one or more enclosing loops or multiway-branch ("switch," in C terminology; "case" in Pascal) statements. The operand 'levels' is an integer giving the number of nested loops and multiway branches to terminate. Obviously, 'levels' must be between 1 and the number of nested loops and multiway branch statements currently active, inclusive.

BREAK is mainly intended to implement premature loop exits. Because of (inadequate) historical reasons, a BREAK is also required to force control out of a multiway-branch alternative to the end of the statement. Thus, in implementing a Pascal-style case statement with the SWITCH_OP described below, each alternative would end with a BREAK_OP with 'levels' equal to 1. If the BREAK_OP was missing, control would fall through from case to case, as it does in C.

```
Example:  break 2  (terminate 2 enclosing loops)
          6              BREAK_OP
          2                 Levels to break
```

CASE_OP 7

```
int 7
tree value
tree actions
tree next_case
```

CASE is used to label an alternative in a multiway branch statement (like 'switch' in C or 'case' in Pascal). The 'value' parameter is the case label value for the alternative; it must be a CONST_OP node of the same mode as the switch expression (see SWITCH_OP). The mode may not be STOWED. The 'actions' parameter is the code to be executed for the given case label. The 'next_case' operand is a DEFAULT_OP or another CASE_OP or a NULL_OP (for the last alternative in the multiway-branch).

CASE_OP is simply a structural device; it organizes the alternatives in a multiway-branch so that variable-sized SWITCH operators are not necessary.

```
Example:  case 10: i += 1   (i is an integer with object id 12)
          7              CASE_OP
          9                 CONST_OP
          1                    INT_MODE
          1                    length is 1 word
          10                   value of first word is 10
          1                 ADDAA_OP
          1                    INT_MODE
          40                   OBJECT_OP
          1                       INT_MODE
          12                      Object id 12
          9                    CONST_OP
```

```
1                       INT_MODE
1                       length is 1 word
1                       value of first word is 1
7 or 12 or 39     CASE_OP or DEFAULT_OP or NULL_OP,
                  depending on next element of SWITCH
```

CHECK_LOWER_OP 72

```
int 72
int mode
tree expression
tree lower_bound
int source_line_number
```

The result of this operator is an rvalue, the value of the parameter 'expression'. The expression must have the mode given by the parameter 'mode', and may not be FLOAT, LONG_FLOAT, or STOWED. If at run time the value of the expression is less than the value of the expression given by the parameter 'lower_bound', an error message is printed and a RANGE_ERROR exception raised. The parameter 'source_line_number' is printed as part of the error message, and is identified as the number of the source code line that caused the range check to be generated.

This operator would normally be used in a situation that permitted optimized range checking, like assignment of one integer subrange variable to another.

Example: var i: 0..100; j: 1..100; begin ...; j := i; ... end
(where i has object id 12 and j has object id 13,
and the code above appears on line 14)

```
5        ASSIGN_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
13             Object id for j
72          CHECK_LOWER_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object id for i
9              CONST_OP
1                 INT_MODE
1                 Length is 1 word
1                 Value is 1
14             Line number in source code
1           Length of assigned quantity is 1 word
```

CHECK_RANGE_OP 70

```
int 70
int mode
tree expression
tree lower_bound
tree upper_bound
int source_line_number
```

The result of this operator is an rvalue, the value of the parameter 'expression'. The expression must have the mode given by the parameter 'mode', and may not be FLOAT, LONG_FLOAT, or STOWED. If at run time the value of the expression is less than the value of the expression given by the parameter 'lower_bound' or greater than the value of the expression given by the parameter 'upper_bound' an error message is printed and a RANGE_ERROR exception raised. The parameter 'source_line_number' is printed as part of the error message, and is identified as the number of the source code line that caused the range check to be generated.

This operator would normally be used where a complete range check was necessary (an array subscripted by an unconstrained integer variable, for example).

Example: var a: array 1..10 of integer; i: integer; ...a[i]...
where 'a' has object id 4, 'i' has id 12,
and the subscripting operation appears on line 97
of the source code:

```
25       INDEX_OP
1           INT_MODE (element type of a)
40          OBJECT_OP; this is the base address of 'a'
7              STOWED_MODE
4              Object id of 'a'
70          CHECK_RANGE_OP; this is the index expression
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object id of 'i'
9              CONST_OP; this is the lower bound
1                 INT_MODE
1                 Length of constant is 1 word
1                 Value of constant is 1
9              CONST_OP; this is the upper bound
1                 INT_MODE
1                 Length of constant is 1 word
10                Value of constant is 10
97             Range check is on line 97
1           Array element size is 1 word
```

CHECK_UPPER_OP 71

```
int 71
int mode
tree expression
tree upper_bound
int source_line_number
```

The result of this operator is an rvalue, the value of the parameter 'expression'. The expression must have the mode given by the parameter 'mode', and may not be FLOAT, LONG_FLOAT, or STOWED. If at run time the value of the expression is greater than the value of the expression given by the parameter 'upper_bound', an error message is printed and a RANGE_ERROR exception raised. The parameter 'source_line_number' is printed as part of the error message, and is identified as the number of the source code line that caused the range check to be generated.

Like CHECK_LOWER, this operator is normally used in situations that permit optimized range checks.

Example: var i: 1..100; j: 1..10; begin ...; j := i; ... end
(where i has object id 12 and j has object id 13,
and the code above appears on line 14)

```
5        ASSIGN_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
13             Object id for j
71          CHECK_UPPER_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object id for i
9              CONST_OP
1                 INT_MODE
1                 Length is 1 word
10                Value is 10
14             Line number in source code
1           Length of assigned quantity is 1 word
```

COMPL_OP 8

```
int 8
int mode
tree operand
```

The result of this operator is an rvalue, the bitwise complement of the operand. The operand must have the same mode as the COMPL operation; the only allowable modes are INT, LONG INT, UNSIGNED, and LONG UNSIGNED.

This operator implements bitwise complementation in languages that support bit operations (e.g. the "~" operator in C). In most cases, it should not be used for logical negation; the NOT_OP is more appropriate.

```
Example:  ~i (i is an integer object with id 12)
         8                  COMPL_OP
         1                     INT_MODE
         40                    OBJECT_OP
         1                        INT_MODE
         12                       Object id is 12
```

CONST_OP 9

```
int 9
int mode
int length
int word[1]
int word[2]
...
int word[length]
```

The result of this operator is an rvalue, equivalent to the value of the constant it defines. 'Length' is the length of the constant in 16-bit machine words. 'Mode' may take on any of the operand mode values, although STOWED constants are not of much use outside initializers.

CONST_OP is the only operator whose IMF representation varies in length depending on its contents. Most literals in a source language program eventually are expressed as CONST_OPs in the IMF.

```
Example:  14 (an integer constant)
         9                  CONST_OP
         1                     INT_MODE
         1                     length is 1 word
         14                    first word has value 14
```

CONVERT_OP 10

```
int 10
int source_mode
int destination_mode
tree operand
```

The result of this operator is an rvalue, namely the value of the operand converted to the data mode specified by 'destination_mode'. The operand mode must be the same as 'source_mode'. STOWED mode is not permissible in either mode

parameter. Note that in most cases, no range checking is performed; it is possible, for example, to convert an UNSIGNED quantity into an negative INT quantity. Floating point to integer conversions are performed by truncation.

CONVERT is the only means of converting data from one mode to another; the code generator never coerces data from one mode to another, unless the coercion is called for by a CONVERT operator.

```
Example:  x = i (x is a FLOAT object, with id 6;
                 i is an INT object, with id 12)
         5                ASSIGN_OP
         5                   FLOAT_MODE
         40                  OBJECT_OP
         5                      FLOAT_MODE
         6                      Object id is 6
         10                  CONVERT_OP
         1                      from INT_MODE
         5                      to FLOAT_MODE
         40                     OBJECT_OP
         1                         INT_MODE
         12                        Object id is 12
```

DECLARE_STAT_OP 11

```
int 11
int object_id
string external_name
```

See "Operators useful in the Static Data Stream".

DEFAULT_OP 12

```
int 12
tree actions
tree next_case
```

This operator is used to label the default action in a multiway-branch statement. (In C, the default action is labeled "default"; in Pascal, it is labeled "otherwise".) The DEFAULT_OP need not be the last alternative in the list of alternatives following a SWITCH. A DEFAULT_OP behaves much like a CASE_OP, in that control will fall through to the next alternative unless the actions conclude with a BREAK_OP.

```
Example:  default: i += 1 (where i is an integer object with id 12)
         12             DEFAULT_OP
         1                 ADDAA_OP
         1                    INT_MODE
```

```
40                      OBJECT_OP
1                          INT_MODE
12                         Object id 12
9                       CONST_OP
1                          INT_MODE
1                          length is 1 word
1                          value of first word is 1
7 or 39              CASE_OP or NULL_OP, depending on structure
                     of SWITCH
```

DEFINE_DYNM_OP 13

```
int 13
int object_id
tree init_list
int size
```

This operator causes storage for the object identified by 'object_id' to be allocated in the current stack frame. It is generated for local variable declarations and for temporary variables allocated by the front end. 'Object_id' must be used in all subsequent references to the object, and the object's definition with DEFINE_DYNM must precede all such references. The init_list is a list of expressions whose values will be assigned to successive words of the newly-declared object (see INITIALIZER_OP and ZERO_INITIALIZER_OP). The size parameter specifies the amount of storage to be reserved for the object, in 16-bit words. (Slightly fewer than 65,535 words are available for local storage in each procedure.)

When processing a declaration, the front-end should assign each declared variable an integer "object id." To be safe, the object id should be unique within an IMF module. This object id must be used whenever the variable being declared is referenced.

Example: int blank = 160; (a local declaration; assume 'blank' is assigned the object id 4)

```
13                DEFINE_DYNM_OP
4                    Object id is 4
26                   INITIALIZER_OP
1                       INT_MODE
9                       CONST_OP
1                          INT_MODE
1                          Length is 1 word
160                        Value of first word is 160
39                      NULL_OP (end of init list)
1                    Size is 1 word
```

DEFINE_STAT_OP 14

```
int 14
int object_id
tree init_list
int size
```

This operator causes storage for the object identified by 'object_id' to be allocated in the current link frame (static data area). It is normally generated by the front end for global variable declarations. 'Object_id' must be used in all subsequent references to the object, and the object's definition with DEFINE_STAT must precede all such references. The init_list is a list of *constants* whose values will be assigned to successive words of the newly-declared object (see INITIALIZER_OP and ZERO_INITIALIZER_OP). The size parameter specifies the amount of storage to be reserved for the object, in 16-bit words. (Slightly fewer than 65,535 words are available for static storage in each module.)

Any storage reserved by a DEFINE_STAT_OP that is not filled by an initializer will be set to zero.

When processing a declaration, the front-end should assign each declared variable an integer "object id." To be safe, the object id should be unique within an IMF module. This object id must be used whenever the variable being declared is referenced.

Example: int blank = 160; (a global declaration; assume 'blank' is assigned the object id 4)

```
14              DEFINE_STAT_OP
4                  Object id is 4
26                 INITIALIZER_OP
1                     INT_MODE
9                     CONST_OP
1                        INT_MODE
1                        Length is 1 word
160                      Value of first word is 160
39                    NULL_OP (end of init list)
1                  Size is 1 word
```

DEREF_OP 15

```
int 15
int mode
tree operand
```

The result of this operator is an lvalue, the object whose address is given by the value of the operand. The operand must yield a 32-bit LONG INT or LONG UNSIGNED value. The operation mode is not restricted.

DEREF is one of the few operators that yield an lvalue, and are

therefore allowed as left-operands of assignments. DEREF is normally used for indirection through pointers in languages that support them explicitly (eg "^" operator in Pascal, or unary "*" in C), although it is also useful in obtaining the value of a variable that is passed to a procedure by reference.

```
Example:  i = *p  (or i = p^ in Pascal)
                  (i is an integer object with id 12;
                  p is a long unsigned object with id 32)
          5              ASSIGN_OP
          1                 INT_MODE
          40                OBJECT_OP
          1                    INT_MODE
          12                   Object id is 12
          15                DEREF_OP
          1                    INT_MODE
          40                   OBJECT_OP
          4                       LONG_UNS_MODE
          32                      Object id is 32
```

DIVAA_OP 16

```
int 16
int mode
tree left
tree right
```

The result of this operator is an rvalue, the quotient of the value of the left operand divided by the value of the right. As a side effect, the quotient is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the DIVAA operation; any mode other than STOWED is acceptable.

DIVAA stands for "divide and assign." The operator is usually used to implement the division assignment operator ("/=" in C, "/:=" or "divab" in Algol 68).

If the operation mode is UNSIGNED or LONG UNSIGNED and the right operand is a power of 2, the division will be performed by a right logical shift.

```
Example:  i /= 10 (where i is an integer object with object id 12)
          16             DIVAA_OP
          1                 INT_MODE
          40                OBJECT_OP
          1                    INT_MODE
          12                   Object id 12
          9                 CONST_OP
          1                    INT_MODE
          1                    length is 1 word
          10                   value of first word is 1
```

DIV_OP 17

```
int 17
int mode
tree left
tree right
```

The result of this operator is an rvalue, the quotient of the value of the left operand divided by the value of the right. Both operands must have the same mode as the DIV operation, and the mode STOWED is not allowed.

DIV is used to implement simple division.

If the operation mode is UNSIGNED or LONG UNSIGNED and the right operand is a power of 2, the division will be performed by a right logical shift.

Example: i / 10 (where i is an integer object with object id 12)

```
17              DIV_OP
1                  INT_MODE
40                 OBJECT_OP
1                     INT_MODE
12                    Object id 12
9                  CONST_OP
1                     INT_MODE
1                     length is 1 word
10                    value of first word is 1
```

DO_LOOP_OP 18

```
int 18
tree body
tree condition
```

This operator implements a test-at-the-bottom loop. 'Body' specifies the operations to be performed in the loop. The loop is performed until the value of the expression specified by 'condition' is non-zero. A BREAK_OP may be used to terminate execution of the loop from within the body, and a NEXT_OP may be used to cause an immediate transfer to the condition test from within the body.

It is not kosher to use a DO_LOOP as a value-returning construct.

Example: do i *= 2 until (i > j)
(where i and j are integer objects, with ids 12 and 60)

```
18              DO_LOOP_OP
33                 MULAA_OP
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
12                       Object id is 12
```

```
9                    CONST_OP
1                       INT_MODE
1                       Length is 1 word
2                       Value is 2
23                GT_OP
1                    INT_MODE
40                   OBJECT_OP
1                       INT_MODE
12                      Object id is 12
40                   OBJECT_OP
1                       INT_MODE
60                      Object id is 60
```

EQ_OP 19

```
int 19
int mode
tree left
tree right
```

The result of this operator is an rvalue: 1 if the value of the left operand equals the value of the right, and 0 otherwise. Both operands must have the mode specified by the parameter 'mode', but note that the result mode of EQ is *always* INTEGER. The operation mode may not be STOWED.

EQ is used to implement test-for-equality for both expressions yielding Boolean values and for control flow tests. The restriction against STOWED operands will hopefully be lifted in the near future.

Example: i == 1 (where i is an integer object with object id 12)

```
19             EQ_OP
1                 INT_MODE
40                OBJECT_OP
1                    INT_MODE
12                   Object id 12
9                 CONST_OP
1                    INT_MODE
1                    length is 1 word
1                    value of first word is 1
```

FIELD_OP 69

```
int 69
int mode
int offset_from_msb
int length_in_bits
tree base_address
```

FIELD is used to select a partial field of a word or double word. It may be used on the left hand side of assignments, to cause the right hand side value to be placed in the field, or as an rvalue, to yield the value stored in the field. The operation mode must be INT, UNSIGNED, LONG INT, or LONG UNSIGNED. The parameter 'base_address' is an lvalue which specifies the first 16-bit word containing any portion of the bit field. The parameter 'offset_from_msb' gives the offset, in bits, of the beginning of the field from the left-hand (most significant) bit of the first word. The parameter 'length_in_bits' gives the length of the bit field. Bit fields may be 1 to 32 bits in length, and must be aligned so as not to cross more than one word boundary.

FIELDs behave like lvalues in most circumstances; for instance, they can be used in left-hand-sides of assignments. However, bit fields cannot be addressed, so they may not be passed by reference on procedure calls or used as an operand of the REFTO operator. FIELDs can always be used as rvalues.

Bit fields may not cross more than one word boundary, since this would require 48 bit shifts for field extraction. Formally, this means that 'offset_from_msb' + 'length_in_bits' must be less than or equal to 32.

Example: Fetching the right-hand byte of a 16-bit word in the integer object i, with id 12:

```
69       FIELD_OP
1           INT_MODE
8           Bit field begins 8 bits from the most
            significant bit
8           Bit field is 8 bits long
40          OBJECT_OP; the base address of the field
1              INT_MODE
12             Object id for 'i'
```

FOR_LOOP_OP 20

```
int 20
tree init
tree cond
tree reinit
tree body
```

The FOR_LOOP_OP implements the general-purpose C 'for' loop. The parameters 'init', 'reinit', and 'body' correspond to statements; 'cond' corresponds to a Boolean expression. The for-loop

```
for (init; cond; reinit) statement
```

is equivalent to

```
          init; while cond do begin statement; reinit end
```

A typical application in languages other than C might be the construction of an arithmetic loop like the Pascal 'for' or the Fortran 'do'.

Within the body of the loop, a BREAK_OP may be used to cause early loop termination, and a NEXT_OP may be used to cause an immediate jump to the 'reinit' code in preparation for another iteration.

It is not reasonable to use a FOR_LOOP as a value-returning construct.

Example:
```
for (i = 1; i <= n; i += 1)
    j *= i;
```
(where i, j, n are integers with object ids 12, 60, 44)

```
20              FOR_LOOP_OP
5                  ASSIGN
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
12                       Object id 12
9                     CONST_OP
1                        INT_MODE
1                        Length is 1 word
1                        Value is 1
1                     Assign 1 word
28                 LE_OP
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
12                       Object id 12
40                    OBJECT_OP
1                        INT_MODE
44                       Object id 44
1                  ADDAA_OP
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
12                       Object id 12
9                     CONST_OP
1                        INT_MODE
1                        Length is 1 word
1                        Value is 1
33                 MULAA_OP
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
60                       Object id 60
40                    OBJECT_OP
1                        INT_MODE
12                       Object id 12
```

GE_OP 21

```
int 21
int mode
tree left
tree right
```

The result of this operator is an rvalue, 1 if the value of the left operand is greater-than-or-equal-to the value of the right, 0 otherwise. Both operands must have the mode given in the parameter 'mode'; note, however, that the result of GE is *always* of mode INTEGER. The operation mode may not be STOWED. Note that if the operands are unsigned, a "magnitude" comparison is performed to insure correct results.

GE_OP implements the test for greater-or-equal in both Boolean expressions and flow-of-control tests. The restriction against STOWED operands may be lifted someday.

Example: i >= 1 (where i is an integer object with object id 12)

```
21              GE_OP
1                  INT_MODE
40                 OBJECT_OP
1                     INT_MODE
12                    Object id 12
9                  CONST_OP
1                     INT_MODE
1                     length is 1 word
1                     value of first word is 1
```

GOTO_OP 22

```
int 22
int object_id
```

GOTO_OP is used to implement unrestricted 'goto' statements in languages that support such nonsense. The parameter 'object_id' is the integer object identifier of the label which is the target of the goto. (See LABEL_OP).

The stack is *not* adjusted if the target label is outside the current procedure.

Example: goto label (where 'label' has object id 99)

```
22              GOTO_OP
99                 Object ID of target label
```

GT_OP 23

```
int mode
tree left
tree right
```

The result of this operator is an rvalue, 1 if the value of the left operand is greater than the value of the right, 0 otherwise. Both operands must have the mode given by the parameter 'mode'; but note that GT always returns a value of mode INTEGER. The operation mode may not be STOWED. Note that if the operands are of mode unsigned, a "magnitude" comparison will be performed to insure correct results.

GT implements the test for greater-than for Boolean expressions and expressions in flow-of-control context. The restriction against STOWED operands might be lifted if the public demands it.

Example: i > 1 (where i is an integer object with object id 12)

```
23              GT_OP
1                  INT_MODE
40                 OBJECT_OP
1                     INT_MODE
12                    Object id 12
9                  CONST_OP
1                     INT_MODE
1                     length is 1 word
1                     value of first word is 1
```

IF_OP 24

```
int 24
int mode
tree condition
tree then_part
tree else_part
```

IF can be used to implement conditional expressions or conditional evaluation of statements; it always returns an rvalue. If the value of the condition is non-zero, the 'then_part' will be evaluated; otherwise, the 'else_part' will be evaluated. Either 'then_part' or 'else_part' may be omitted (ie, replaced by a NULL_OP). The operation mode may not be STOWED; if the operator is used to return a value (as in a conditional expression) then the modes of both the 'then_part' and the 'else_part' must be the same as the operation mode.

IF is most often used to implement conditional statements (eg the 'if' statement of most algorithmic languages). Since the code generator tends to view operators as value-returning, IF may also be used to implement conditional expressions ('if'-'then'-'else' in the Algol family, or '?:' in C).

```
Example:  if a < b then m = a else m = b
         (where a, b, m are floating point objects with id's 1, 2, 13)
         24              IF_OP
         5                  FLOAT_MODE
         31                 LT_OP
         5                     FLOAT_MODE
         40                    OBJECT_OP
         5                        FLOAT_MODE
         1                        Object id 1
         40                    OBJECT_OP
         5                        FLOAT_MODE
         2                        Object id 2
         5                  ASSIGN_OP
         5                     FLOAT_MODE
         40                    OBJECT_OP
         5                        FLOAT_MODE
         13                       Object id 13
         40                    OBJECT_OP
         5                        FLOAT_MODE
         1                        Object id 1
         2                     Length is 2 words
         5                  ASSIGN_OP
         5                     FLOAT_MODE
         40                    OBJECT_OP
         5                        FLOAT_MODE
         13                       Object id 13
         40                    OBJECT_OP
         5                        FLOAT_MODE
         2                        Object id 2
         2                     Length is 2 words
```

INDEX_OP 25

```
int 25
int mode
tree array_base
tree index_expression
int element_size
```

The result of this operator is an lvalue, one member of a vector of identical objects. The parameter 'array_base' is the base of the vector; it is typically a simple OBJECT_OP, although it may be an expression yielding the base address of the vector (a dereferenced pointer, for example). It must be an lvalue. The 'index_expression' selects the particular vector element desired; it should have a value greater than or equal to zero and less than the number of elements in the vector. (Note that this implies zero-origin addressing.) (Note furthermore that there is no subscript checking.) The 'index_expression' must be of mode INTEGER or UNSIGNED (indexing across 64K-word segment boundaries produces incorrect results in V mode). 'Element_size' is the size of one element of the vector, in 16-bit words. The operation mode must be the same as the mode of the vector elements,

but is otherwise unrestricted; in particular, STOWED mode is allowed.

INDEX is used to implement array subscripting. The operator has deliberately been made rather primitive, to allow the front-end greater freedom in selecting storage layouts. For example, multidimensional arrays may be implemented by treating arrays as vector elements, and subsuming the additional addressing calculations in the 'index_expression'. This allows a compiler to select row- or column-major addressing. Note that subscripting is vastly more efficient if vector elements are a power of 2 words in length, and furthermore that lengths 1, 2, and 4 are most efficient.

Example: a[i + 1] (where a is a floating point object with id 1, and i is an integer object with id 12)

```
25              INDEX_OP
5                  FLOAT_MODE
40                 OBJECT_OP
7                     STOWED_MODE
1                     Object id 1
2                  ADD_OP
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
12                       Object id 12
9                     CONST_OP
1                        INT_MODE
1                        Length is 1 word
1                        Value is 1
2                  Array element is 2 words long
```

INITIALIZER_OP 26

```
int 26
int mode
tree expression
tree next_initializer
```

Initializers are the initial-value expressions that appear in definitions of variables in C (see DEFINE_DYNM_OP and DEFINE_STAT_OP). In the case of local variables, which are reinitialized whenever they are allocated, these expressions are arbitrary. In the case of static variables, these expressions must be constants or REFTO operators whose operands are constants or OBJECT_OPs.

Initializers are formed by linking a number of INITIALIZER_OPs and ZERO_INITIALIZER_OPs together through their 'next_initializer' fields. ZERO_INITIALIZER_OP is a compact representation of an initializer consisting of all zeros.

Any mode is allowable in an INITIALIZER. INT and UNSIGNED

initializers cause one word to be filled; LONG INT, LONG UNSIGNED, and FLOAT cause two words to be filled; LONG FLOAT causes four words to be filled; STOWED expressions fill as many words as the size of the expression allows (STOWED mode CONST_OPs are particularly useful here).

```
Example:  int ai[3] = {1, 2, 3}
          (a local declaration, where ai is assigned object id 8)
          13          DEFINE_DYNM_OP
          8              Object has id 8
          26             INITIALIZER_OP
          1                 INT_MODE
          9                 CONST_OP (the init. expression)
          1                    INT_MODE
          1                    Constant has length 1
          1                    Constant has value 1
          26                INITIALIZER_OP
          1                    INT_MODE
          9                    CONST_OP
          1                       INT_MODE
          1                       Constant has length 1
          2                       Constant has value 2
          26                   INITIALIZER_OP
          1                       INT_MODE
          9                       CONST_OP
          1                          INT_MODE
          1                          Constant has length 1
          3                          Constant has value 3
          39                      NULL_OP (end of initializers)
          3              Object has size 3 words
```

As an alternative,

```
          13          DEFINE_DYNM_OP
          8              Object has id 8
          26             INITIALIZER_OP
          7                 STOWED_MODE
          9                 CONST_OP
          7                    STOWED_MODE
          3                    Constant is 3 words long
          1                    First word is 1
          2                    Second word is 2
          3                    Third word is 3
          39                NULL_OP (end of initializers)
          3              Object is 3 words long
```

LABEL_OP 27

```
int 27
int object_id
```

LABEL_OP is used to place the target label for 'goto' statements. The parameter 'object_id' is the integer identifier used by

GOTO_OPs to identify their target labels.

```
Example:  label lab;.... lab:
          (assume the label declaration causes 'lab' to be assigned
          the object id 6)
         27          LABEL_OP
         6              The object ID of the label
```

LE_OP 28

```
int 28
int mode
tree left
tree right
```

The result of this operator is an rvalue, 1 if the value of the left operand is less than or equal to the value of the right, 0 otherwise. Both operands must have the mode specified by the parameter 'mode'; STOWED mode is not allowable. Note that LE *always* returns a value of mode INTEGER. Magnitude comparisons are generated for unsigned operands, to insure correct results.

Use LE_OP to implement all tests for less-than-or-equal-to, whether they appear in boolean expressions or flow-of-control tests. The restriction against STOWED operands may be lifted if the author feels sufficiently threatened.

Example: i <= 1 (where i is an integer object with id 12)

```
         28       LE_OP
         1           INT_MODE
         40          OBJECT_OP
         1              INT_MODE
         12             Object id 12
         9           CONST_OP
         1              INT_MODE
         1              Constant has length 1
         1              Constant has value 1
```

LSHIFTAA_OP 29

```
int 29
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand shifted logically (zero-fill) left the number of bit places specified by the value of the right operand. As a side effect, the result is stored back into the left operand. The

left operand must be an lvalue or a bit field (see FIELD_OP). The operation mode may be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the left operand must have the same mode. The right operand must be of mode INT or UNSIGNED, and really should have a value between 0 and the length of the left operand, inclusive. (Reasonable results outside this range are not guaranteed.)

LSHIFTAA stands for "left-shift and assign." The operator is used to implement "<<=" in C.

Example: i <<= 1 (where i is an integer object with id 12)

```
29       LSHIFTAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

LSHIFT_OP 30

```
int 30
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand shifted left logically (zero-fill) the number of bit places specified by the value of the right operand. The operation mode may be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the left operand must have the same mode. The right operand must be of mode INT or UNSIGNED, and really should have a value between 0 and the length of the left operand, inclusive. (Reasonable results outside this range are not guaranteed.)

LSHIFT is used to implement the "<<" operator in C.

Example: i << 1 (where i is an integer object with id 12)

```
30       LSHIFT_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

LT_OP 31

int 31
int mode
tree left
tree right

The result of this operator is an rvalue, 1 if the value of the left operand is less than the value of the right, 0 otherwise. Both operands must have the mode given in the parameter 'mode'. Note that LT always returns a value of mode INTEGER, no matter what the operation mode was. The operation mode may not be STOWED. Magnitude comparisons are used if the operands are unsigned, to insure correct results.

LT is used to implement the test for less-than, in both Boolean expressions and flow-of-control expressions. The restriction against STOWED operands may be removed if an angry armed mob storms the author's office.

Example: i < 1 (where i is an integer object with id 12)

```
31       LT_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

MODULE_OP 32

int 32

This operator is not used in procedure definitions; it is used strictly to separate modules in input streams.

MULAA_OP 33

int 33
int mode
tree left
tree right

The result of this operation is an rvalue, the product of the value of the left operand and the value of the right. As a side effect, the product is stored into the left operand. The left

operand must be an lvalue or a bit field. Both operands must have the same mode as the operation, and that mode may not be STOWED.

MULAA stands for "multiply and assign." It is used to implement the multiplication assignment operators ("*=" in C, "*:=" or "mulab" in Algol 68). When either operand is known to be a power of 2, the multiplication will be replaced by a left logical shift.

Example: i *= 10 (where i is an integer object with id 12)

```
33       MULAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
10             Constant has value 10
```

MUL_OP 34

```
int 34
int mode
tree left
tree right
```

The result of this operation is an rvalue, the product of the value of the left operand and the value of the right. Both operands must have the same mode as the operation, and that mode may not be STOWED.

MUL_OP is used to implement simple multiplication. When either operand is known to be a power of 2, the multiplication will be replaced by a left logical shift.

Example: i * 2 (where i is an integer object with id 12)

```
34       MUL_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
2              Constant has value 2
```

NEG_OP 35

```
int 35
int mode
tree operand
```

The result of this operator is an rvalue, the additive inverse of the value of the operand. Unsigned operands are subtracted from 2**n, where n is the number of bits used to represent them (16 or 32, in this implementation). The operation mode must be the same as the mode of the operand, and may not be STOWED.

NEG_OP implements the unary minus (negation) operator for all the primitive arithmetic data modes.

Example: -i (where i is an integer object with id 12)

```
35       NEG_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object has id 12
```

NEXT_OP 36

```
int 36
int levels
```

NEXT_OP yields no result value, but causes an immediate restart of a particular enclosing loop. 'Levels' - 1 enclosing loops are terminated (see BREAK_OP) and then a branch is taken to the proper restart point in the next enclosing loop. For the FOR_LOOP, the restart point is the re-initialization statement at the end of the body. For DO_LOOPs and WHILE_LOOPs, the restart point is the evaluation of the iteration condition.

Example: next 2 (break 1 loop, continue the next outermost)

```
36       NEXT_OP
2           Levels
```

NE_OP 37

```
int 37
int mode
tree left
tree right
```

The result of this operator is an rvalue, 1 if the value of the left operand does not equal the value of the right, 0 otherwise.

The modes of both operands must match the mode of the operation, and STOWED mode is not allowed. Note that NE_OP always returns a value of mode INTEGER, no matter what operation mode is specified.

NE implements the test for inequality in all contexts. Use of nuclear weapons might be enough to convince the author to lift the restriction against STOWED operands.

Example: i <> 1 (where i is an integer object with id 12)

```
37       NE_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

NOT_OP 38

```
int 38
int mode
tree operand
```

The result of this operator is an rvalue, the logical negation of the operand value. (Ie, if the operand has value zero, the result of the NOT_OP will be 1; if the operand is non-zero, the result of the NOT_OP will be zero.) The mode of the operand must be the same as the mode of the operation, and STOWED mode is not allowed. The result of a NOT_OP is *always* of mode INTEGER, no matter what the operation mode.

NOT_OP is normally used to implement Boolean negation. For bitwise complementation, use COMPL_OP.

Example: !i (where i is an integer object with id 12)

```
38       NOT_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object has id 12
```

NULL_OP 39

```
int 39
```

The null operator is usually used to terminate lists constructed with the sequence operator SEQ_OP, or to indicate that a subtree has been omitted. For example, if a conditional has no else_part, the missing subtree must be represented by a NULL_OP. SEQ also acts as a delimiter at several places in the input stream.

Example:

```
        39      NULL_OP
```

OBJECT_OP 40

```
int 40
int mode
int object_id
```

The result of this operator is an lvalue, corresponding to a variable defined by the front end. 'Mode' is unrestricted; objects may have any primitive data mode, including STOWED (for arrays and records). The 'object_id' parameter gives the identification number that was supplied in the definition or declaration of the object.

Normally, each occurrence of a variable in the source program produces an OBJECT_OP in the intermediate form. OBJECTs are the primitive lvalues from which all other lvalue-producing constructs are derived.

Each object that is referenced in the intermediate form must be identified by a simple integer known as the "object id." Typically these ids are assigned at declaration time (for variables) or at time of first reference (for locations, like procedure names or statement labels). Object ids should be unique within each IMF module.

Example: i (where i is an integer object, with object id 12)

```
        40      OBJECT_OP
        1          INT_MODE
        12         Object id 12
```

ORAA_OP 41

```
int 41
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise inclusive-or of the values of the left and right operands. As a side

effect, the result is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). The operation mode must be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the modes of both operands must match the operation mode.

ORAA stands for "logical or and assign." It is used to implement the C assignment operator "|=".

Example: i |= 1 (where i is an integer object with id 12)

```
41       ORAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

OR_OP 42

```
int 42
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise inclusive-or of the values of the left and right operands. The operation mode must be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the modes of both operands must match the operation mode.

OR is used to implement bit-oriented logical operations, like the "|" operator of C. Although OR can be used in Boolean expressions, the sequential-OR operator SOR_OP is usually more appropriate.

Example: i | 1 (where i is an integer object with id 12)

```
42       OR_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

POSTDEC_OP 43

```
int 43
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand before the operator is executed. As a side effect, the left operand is decremented by the value of the right operand. The left operand must be an lvalue or a bit field (see FIELD_OP), and must have the same mode as the operation. The right operand must be a CONST_OP, with the same mode as the operation.

The POSTDEC operator corresponds to the C postfix autodecrement construct.

Example: p-- (where p is a long unsigned (pointer) object with object id 15, and p is intended to point to integers)

```
43       POSTDEC_OP
4           LONG_UNS_MODE
40          OBJECT_OP
4              LONG_UNS_MODE
15             Object id of p
9           CONST_OP
4              LONG_UNS_MODE
2              Constant has length 2
0              Constant has value...
1              ...1, expressed as a long integer
```

POSTINC_OP 44

```
int 44
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand before the operator is executed. As a side effect, the left operand is incremented by the value of the right operand. The left operand must be an lvalue or a bit field (see FIELD_OP), and must have the same mode as the operation. The right operand must be a CONST_OP, with the same mode as the operation.

The POSTINC operator corresponds to the C postfix autoincrement construct.

Example: p++ (where p is a long unsigned (pointer) object with object id 15, and p is intended to point to integers)

```
44       POSTINC_OP
4           LONG_UNS_MODE
```

```
40          OBJECT_OP
4              LONG_UNS_MODE
15             Object id of p
9           CONST_OP
4              LONG_UNS_MODE
2              Constant has length 2
0              Constant has value...
1              ...1, expressed as a long integer
```

PREDEC_OP 45

```
int 45
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand decremented by the value of the right operand. As a side effect, the result is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP), and must have the same mode as the operation. The right operand must be a CONST_OP, with the same mode as the operation.

The PREDEC operator corresponds to the C prefix autodecrement construct.

Example: --p (where p is a long unsigned (pointer) object with object id 15, and p is intended to point to integers)

```
45       PREDEC_OP
4           LONG_UNS_MODE
40          OBJECT_OP
4              LONG_UNS_MODE
15             Object id of p
9           CONST_OP
4              LONG_UNS_MODE
2              Constant has length 2
0              Constant has value...
1              ...1, expressed as a long integer
```

PREINC_OP 46

```
int 46
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand incremented by the value of the right operand. As a side effect, the result is stored back into the left operand. The

left operand must be an lvalue or a bit field (see FIELD_OP), and must have the same mode as the operation. The right operand must be a CONST_OP, with the same mode as the operation.

The PREINC operator corresponds to the C prefix autoincrement construct.

Example: ++p (where p is a long unsigned (pointer) object with object id 15, and p is intended to point to integers)

```
46       PREINC_OP
4           LONG_UNS_MODE
40          OBJECT_OP
4              LONG_UNS_MODE
15             Object id of p
9           CONST_OP
4              LONG_UNS_MODE
2              Constant has length 2
0              Constant has value...
1              ...1, expressed as a long integer
```

PROC_CALL_ARG_OP 47

```
int 47
int mode
tree expression
tree next_argument
```

Procedure call arguments are specified in a linked list of PROC_CALL_ARG_OPs attached to a PROC_CALL_OP. An argument expression is specified by the parameter 'expression'; its mode must be given by the parameter 'mode'. The parameter 'next_argument' is simply the next procedure argument in the list. Any mode expression is allowable as an argument, since the Prime procedure call convention passes a fixed-size pointer to the actual argument, rather than the argument itself.

Note that arguments (with the exception of bit fields) are always passed by reference. If arguments are to be copied on procedure entry or exit, the *called* procedure must do the copying. (See PROC_DEFN_ARG_OP; an argument will be copied automatically if it is given the disposition VALUE_DISP.) Bit fields are an exception; they are not addressable objects, and so are always passed by value.

See PROC_CALL_OP for examples of PROC_CALL_ARG_OP.

PROC_CALL_OP 48

```
int 48
int mode
```

```
tree procedure
tree argument_list
```

The PROC_CALL_OP is used to generate a call to a procedure. The parameter 'mode' is the mode of the return value of the procedure, if any. The parameter 'procedure' is an lvalue representing the address of the procedure to be called; the most common case is simply an OBJECT_OP with an object id equal to the id of a declared procedure (see PROC_DEFN_OP). The parameter 'argument_list' is a singly-linked list of expressions to be passed as arguments to the procedure; each expression in the argument list is contained in a PROC_CALL_ARG_OP subtree, and the entire list is terminated with a NULL_OP.

PROC_CALL implements invocation of both "procedures" and "functions" (or "value-returning procedures").

Example: l = strlen (s)
where l is an integer object with id 13,
s is a STOWED object (an array of integers) with id 14,
and strlen is a procedure with id 50.

```
5        ASSIGN_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
13             object id for l
48          PROC_CALL_OP
1              INT_MODE
40             OBJECT_OP; this gives the procedure address
7                 STOWED_MODE
50                Object id for strlen
47             PROC_CALL_ARG_OP; description of first arg
7                 STOWED_MODE
40                OBJECT_OP
7                    STOWED_MODE
14                   Object id for s
39                NULL_OP; ends list of arguments
1           Number of words transferred by ASSIGN
```

PROC_DEFN_ARG_OP 49

```
int 49
int object_id
int mode
int disposition
int length
tree next_argument
```

This operator cannot be used as part of the code of a procedure. See "Operators Useful in the Procedure Definition Stream".

PROC_DEFN_OP 50

```
int 50
int object_id
int number_of_args
string proc_name
tree argument_list
tree code
```

This operator cannot be used as part of the code of a procedure. See "Operators Useful in the Procedure Definition Stream".

REFTO_OP 51

```
int 51
int mode
tree operand
```

The result of this operator is an rvalue, the virtual memory address of the operand. The operand must be an lvalue, but it can have any mode. In particular, the operand *may not be a bit field*. The operation mode must be LONG INT or LONG UNSIGNED.

REFTO implements the unary "&" operator in C.

Example: &i (where i is an integer object with id 12)

```
51      REFTO_OP
4          LONG_UNS_MODE; pointers are generally of this mode
40         OBJECT_OP
1             INT_MODE
12            Object id for i
```

REMAA_OP 52

```
int 52
int mode
tree left
tree right
```

The result of this operation is an rvalue, the remainder resulting from division of the value of the left operand by the value of the right. As a side effect, the result is stored back into the left operand. The left operand must be an lvalue or a bit field. Both operands must have the same mode as the operation, and the operation mode may not be STOWED, FLOAT, or LONG FLOAT. (The restriction against floating point operands may be lifted in the near future.)

Note that this operator produces the *remainder* resulting from the division; the remainder may be negative. If a true modulus is desired, the absolute value of the left operand should be remaindered by the right operand, instead.

Example: i %= 2 (where i is an integer object with id 12)

```
52       REMAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for i
9           CONST_OP
1              INT_MODE
1              Length of constant is 1 word
2              Value of constant is 2
```

REM_OP 53

```
int 53
int mode
tree left
tree right
```

The result of this operation is an rvalue, the remainder resulting from division of the value of the left operand by the value of the right. Both operands must have the same mode as the operation, and the operation mode may not be STOWED, FLOAT, or LONG FLOAT. (The restriction against floating point operands may be lifted in the near future.)

Note that this operator produces the *remainder* resulting from the division; the remainder may be negative. If a true modulus is desired, the absolute value of the left operand should be remaindered by the right operand, instead.

Example: i % 2 (where i is an integer object with id 12)

```
53       REM_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for i
9           CONST_OP
1              INT_MODE
1              Length of constant is 1 word
2              Value of constant is 2
```

RETURN_OP 54

int 54
int mode
tree operand

The operand is evaluated and returned as the result of the current procedure. If the operand is absent (represented by a NULL_OP) a procedure return takes place, but no effort is made to return a particular value. The operation mode may not be STOWED.

This operator is used to implement the "return" statement in many algorithmic languages. All procedures should end with a RETURN_OP.

Example: return (0)

```
54       RETURN_OP
9           CONST_OP
1              INT_MODE
1              Constant has length 1
0              Constant has value 0
```

RSHIFTAA_OP 55

int 55
int mode
tree left
tree right

The result of this operator is an rvalue, the value of the left operand shifted right the number of bit places specified by the value of the right operand. As a side effect, the result is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). The operation mode may be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the left operand must have the same mode. The right operand must be of mode INT or UNSIGNED, and really should have a value between 0 and the length of the left operand, inclusive. (Reasonable results outside this range are not guaranteed.)

RSHIFTAA stands for "right-shift and assign." The operator is used to implement ">>=" in C. If the operation mode is UNSIGNED or LONG UNSIGNED, the vacated bits on the left are zero-filled (logical shift); if the operation mode is INT or LONG INT, the vacated bits on the left are sign-filled (arithmetic shift).

Example: i >>= 1 (where i is an integer object with id 12)

```
55       RSHIFTAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
```

```
12             Object id for i
9           CONST_OP
1              INT_MODE
1              Length of constant is 1 word
1              Value of constant is 1
```

RSHIFT_OP 56

```
int 56
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand shifted right the number of bit places specified by the value of the right operand. The operation mode may be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the left operand must have the same mode. The right operand must be of mode INT or UNSIGNED, and really should have a value between 0 and the length of the left operand, inclusive. (Reasonable results outside this range are not guaranteed.)

This operator is used to implement ">>" in C. If the operation mode is UNSIGNED or LONG UNSIGNED, the vacated bits on the left are zero-filled (logical shift); if the operation mode is INT or LONG INT, the vacated bits on the left are sign-filled (arithmetic shift).

Example: i >> 1 (where i is an integer object with id 12)

```
56       RSHIFT_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for i
9           CONST_OP
1              INT_MODE
1              Length of constant is 1 word
1              Value of constant is 1
```

SAND_OP 57

```
int 57
int mode
tree left
tree right
```

The result of this operation is an rvalue. The left operand is evaluated first. If it is zero, the result of the operation is zero and evaluation is terminated. If it is non-zero, then the

value of the right operand is returned as the result of the operator. The modes of both operands must be the same as the mode of the result.

SAND is used to implement sequential ("short-circuit") logical conjunctions.

Example: i && j (where i, j are integer objects with ids 12 and 13)

```
57       SAND_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for i
40          OBJECT_OP
1              INT_MODE
13             Object id for j
```

SELECT_OP 58

```
int 58
int mode
int offset
tree structure
```

The result of this operator is an lvalue, one member of a heterogeneous data structure (ala the Pascal "record" or the C "struct"). The parameter 'mode' is the mode of the element selected; it is unrestricted. The parameter 'structure' is an lvalue expression yielding the base address of the structure. Typically it is an OBJECT_OP with an object_id field equal to the object id of a STOWED object defined by DEFINE_STAT or DEFINE_DYNM.

Example: rec.field
(rec is a record with object id 4;
field is an integer field offset 3 words from the beginning of the record)

```
58       SELECT_OP
1           INT_MODE
3           Offset from beginning of struct
40          OBJECT_OP
7              STOWED mode
4              Object id of 'rec'
```

SEQ_OP 59

```
int 59
tree left
```

tree right

SEQ causes the left operand to be evaluated, then the right operand. The result is the result of the right operand.

SEQ_OP corresponds roughly to the "," operator in C and the semicolon statement separator in Pascal.

Example: i = 1; j = 2
(where i, j are integer objects with ids 12, 13)

```
59       SEQ_OP
5           ASSIGN_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object id of 'i'
9              CONST_OP
1                 INT_MODE
1                 Constant length is 1 word
1                 Constant value is 1
1              Assignment transfers 1 word
5           ASSIGN_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
13                Object id of 'j'
9              CONST_OP
1                 INT_MODE
1                 Constant length is 1 word
1                 Constant value is 1
1              Assignment transfers 1 word
```

A frequently-used alternative to the above is

```
59       SEQ_OP
5           ASSIGN_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object id of 'i'
9              CONST_OP
1                 INT_MODE
1                 Constant length is 1 word
1                 Constant value is 1
1              Assignment transfers 1 word
59       SEQ_OP
5           ASSIGN_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
13                Object id of 'j'
9              CONST_OP
1                 INT_MODE
1                 Constant length is 1 word
1                 Constant value is 1
```

```
1              Assignment transfers 1 word
39       NULL_OP; end of sequence
```

SOR_OP 60

```
int 60
int mode
tree left
tree right
```

The result of this operator is an rvalue. The left operand is evaluated first. If it is non-zero, it is returned as the result of the operation. If it is zero, the value of the right operand is returned as the result of the operation. The mode of the operation result is always INTEGER. The operands may be of any mode other than STOWED.

SOR is used to implement sequential ("short-circuit") logical disjunctions.

Example: i || j (where i, j are integer objects with ids 12 and 13)

```
60       SOR_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for i
40          OBJECT_OP
1              INT_MODE
13             Object id for j
```

SUBAA_OP 61

```
int 61
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand minus the value of the right operand. As a side effect, the difference is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the operation, and the mode may not be STOWED.

SUBAA stands for "subtract and assign." It is used to implement the "-=" operator of C and the "-:=" or "minusab" operator of Algol 68.

Example: i -= 1 (where i is an integer object with id 12)

```
61       SUBAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for 'i'
9           CONST_OP
1              INT_MODE
1              Constant is of length 1
1              Constant has value 1
```

SUB_OP 62

```
int 62
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand minus the value of the right operand. Both operands must have the same mode as the operation, and that mode may not be STOWED.

Example: i - 1 (where i is an integer object with id 12)

```
62       SUB_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for 'i'
9           CONST_OP
1              INT_MODE
1              Constant is of length 1
1              Constant has value 1
```

SWITCH_OP 63

```
int 63
int mode
tree selector
tree alternative_list
```

SWITCH_OP is used to generate a multiway-branch statement, like the 'switch' of C or the 'case' of Pascal. When the SWITCH is used as a value-returning construct, the modes of all the CASESs must match the operation mode, and must not be STOWED. The parameter 'selector' is an expression to be evaluated and compared with all alternative values in CASE_OPs. 'Alternative_list' is a singly-linked list of CASE_OPs and at most one DEFAULT_OP, terminated with a NULL_OP.

Note that there is no automatic jump from the end of an alternative to the end of the switch; if one is desired, a BREAK_OP should be used. This behavior allows construction of alternatives with multiple case labels, as illustrated in the example below.

Example: The following Pascal 'case' statement, assuming i and j are integer variables with object ids 12 and 13, respectively

```
case i of
   1: j := 6;
   2, 4: j := 10;
   otherwise j := 9;
   end;
```

```
63       SWITCH_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for 'i'
7           CASE_OP; the first alternative
9              CONST_OP
1                 INT_MODE
1                 Length of constant is 1
1                 Value of constant is 1
59             SEQ_OP; actions for first CASE
5                 ASSIGN_OP
1                    INT_MODE
40                   OBJECT_OP
1                       INT_MODE
13                      Object id for 'j'
9                    CONST_OP
1                       INT_MODE
1                       Length of constant is 1 word
6                       Value of constant is 6
1                    Assignment transfers 1 word
59             SEQ_OP; continuing CASE actions
6                 BREAK_OP
1                    1 Level (the SWITCH)
39             NULL_OP; end of CASE actions
7           CASE_OP; second alternative
9              CONST_OP
1                 INT_MODE
1                 Constant has length 1
2                 Constant has value 2
39             NULL_OP; no actions, control falls through
7           CASE_OP; second case of second alternative
9              CONST_OP
1                 INT_MODE
1                 Constant has length 1
4                 Constant has value 4
59             SEQ_OP; beginning of actions
5                 ASSIGN_OP
1                    INT_MODE
40                   OBJECT_OP
1                       INT_MODE
```

```
13                      Object id for 'j'
9                    CONST_OP
1                       INT_MODE
1                       Constant has length 1
10                      Constant has value 10
1                    Assignment transfers 1 word
59                SEQ_OP; actions continue
6                    BREAK_OP
1                       1 Level
39                NULL_OP; end of actions
12             DEFAULT_OP; default actions for SWITCH
59                SEQ_OP; beginning of actions
5                    ASSIGN_OP
1                       INT_MODE
40                      OBJECT_OP
1                       INT_MODE
13                      Object id for 'j'
9                    CONST_OP
1                       INT_MODE
1                       Length 1
9                       Value 9
1                    Assignment transfers 1 word
59                SEQ_OP; default actions continue
6                    BREAK_OP
1                       1 Level
39                NULL_OP; end of default actions
39             NULL_OP; end of alternatives for SWITCH
```

UNDEFINE_DYNM_OP 64

int 64
int object_id

UNDEFINE_DYNM is used to release space assigned to an object allocated in the current local storage area. The parameter 'object_id' is the object identifier used in the DEFINE_DYNM_OP that assigned space to the object.

This operator is rarely used; it is normally unnecessary unless the language supported by the front-end allows nested blocks or the front-end generates and deallocates temporary variables explicitly.

Example: If object number 44 has been allocated by the front end as a temporary, it can be deallocated with

```
64          UNDEFINE_DYNM_OP
44             ID of object to be deallocated
```

WHILE_LOOP_OP 65

```
int 65
tree condition
tree body
```

WHILE_LOOP_OP generates a test-at-the-top loop. The parameter 'condition' must be an expression yielding a result of zero (for loop termination) or non-zero (for loop continuation). The parameter 'body' is the body of the loop (which may contain BREAK ops for early termination or NEXT ops for explicit continuation).

Example: while (i < j) do i <<= 1;
where i, j are integer objects with ids 12 and 13

```
65       WHILE_LOOP_OP
31          LT_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object ID for i
40             OBJECT_OP
1                 INT_MODE
13                Object ID for j
29          LSHIFTAA_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object ID for i
9              CONST_OP
1                 INT_MODE
1                 Length 1
1                 Value 1
```

XORAA_OP 66

```
int 66
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise exclusive-or of the values of the left and right operands. As a side effect, the result is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the operation, and only modes INT, LONG INT, UNSIGNED, and LONG UNSIGNED are allowable.

XORAA stands for "exclusive-or and assign." It is used to implement the "^=" operator of C.

Example: i ^= 1 (where i is an integer object with id 12)

```
66       XORAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for 'i'
9           CONST_OP
1              INT_MODE
1              Constant is of length 1
1              Constant has value 1
```

XOR_OP 67

```
int 67
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise exclusive-or of the values of the left and right operands. Both operands must have the same mode as the operation, and only modes INT, LONG INT, UNSIGNED, and LONG UNSIGNED are allowable.

Example: i ^ 1 (where i is an integer object with id 12)

```
67       XOR_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for 'i'
9           CONST_OP
1              INT_MODE
1              Constant is of length 1
1              Constant has value 1
```

ZERO_INITIALIZER_OP 68

```
int 68
int size
tree next_initializer
```

Initializers are the initial-value expressions that appear in definitions of variables in C (see DEFINE_DYNM_OP and DEFINE_STAT_OP). Local variables are reinitialized whenever the procedure containing them is entered; global (static) variables are initialized only when the program containing them is loaded.

ZERO_INITIALIZER provides a compact way of specifying an all-zeros initializer. The parameter 'size' is the number of 16-bit zero words to be generated; 'next_initializer' is simply a link to the next INITIALIZER or ZERO_INITIALIZER in a variable's

initial-value list.

Example: int a[3] = {0, 0, 0} (a global declaration where 'a' has object id 1)

```
14       DEFINE_STAT_OP
1           Object id for 'a'
68          ZERO_INITIALIZER_OP
3              Fill 3 words with zero
39             NULL_OP; no more initializers
3           Size of 'a', in 16-bit words
```

Extended Examples

These examples should illustrate some global aspects of using the code generator. They include a segment of C source code, the (annotated) intermediate form code produced by the C front end, and the (annotated) assembly language generated by the VCG.

Basic VCG Input

C Code

```
extern int e1, e2;      /* defined outside this module */

int v1, v2;             /* defined here, visible outside */

static int s1, s2;      /* defined here, not visible outside */

proc1 ()                /* procedure defined here, visible outside */
   {
   }

proc2 ()                /* more of the same */
   {
   }
```

IMF Stream 1

```
32    A MODULE_OP; begins the input module
59       SEQ_OP; initiates the sequence of entry points
7           Object number 7 is an entry point...
5              whose name is 5 characters long...
240               p
242               r
239               o
227               c
177               1
59       SEQ_OP; next member of the list of entry points
8           Object number 8 is an entry point...
5              whose name is 5 characters long...
240               p
242               r
239               o
227               c
178               2
59       SEQ_OP; next member of the list of entry points
3           Object number 3 is an entry point...
2              whose name is 2 characters long...
246               v
```

```
177              1
59       SEQ_OP; next member of the list of entry points
4           Object number 4 is an entry point...
2              whose name is 2 characters long...
246               v
178               2
39       NULL_OP; terminates the list of entries in this module
39    NULL_OP; terminates the list of modules in the input
```

IMF Stream 2

```
32  MODULE_OP; beginning of this input module
59     SEQ_OP; beginning of static data declarations list
14        DEFINE_STAT_OP; reserve space for an object
3            Object ID is 3
39           NULL_OP; there are no initializers for this object
1            Its size is 1 word
59     SEQ_OP; next element of declarations list
14        DEFINE_STAT_OP; reserve space for an object
4            Object ID is 4
39           NULL_OP; there are no initializers for this object
1            Its size is 1 word
59     SEQ_OP; next element of declarations list
14        DEFINE_STAT_OP; reserve space for an object
5            Object ID is 5
39           NULL_OP; there are no initializers for this object
1            Its size is 1 word
59     SEQ_OP; next element of declarations list
14        DEFINE_STAT_OP; reserve space for an object
6            Object ID is 6
39           NULL_OP; there are no initializers for this object
1            Its size is 1 word
59     SEQ_OP; next element of declarations list
11        DECLARE_STAT_OP; declare object defined outside this module
1            Object ID is 1
2            Name has 2 characters...
229             e
177             1
59     SEQ_OP; next element of declarations list
11        DECLARE_STAT_OP; declare object defined outside this module
2            Object ID is 2
2            Name has 2 characters...
229             e
178             2
39     NULL_OP; end of static data definition/declaration list
39  NULL_OP; end of modules in input stream
```

IMF Stream 3

```
32    MODULE_OP; beginning of next module in input stream
59       SEQ_OP; first element of procedure definitions list
50          PROC_DEFN_OP; procedure definition follows
7              Procedure is object number 7
0              Procedure has no arguments
```

```
5              Procedure name is 5 characters long...
240               p
242               r
239               o
227               c
177               1
39             NULL_OP; empty argument description list
39             NULL_OP; no code for this procedure
59       SEQ_OP; next element of procedure definitions list
50          PROC_DEFN_OP; procedure definition follows
8              Procedure is object number 8
0              Procedure has no arguments
5              Procedure name is 5 characters long...
240               p
242               r
239               o
227               c
178               2
39             NULL_OP; empty argument description list
39             NULL_OP; no code for this procedure
39       NULL_OP; end of procedure definitions list (and this module)
39    NULL_OP; end of modules in this input stream
```

PMA Code

```
 SEG              Assemble in 64V mode
 RLIT             Place literals in procedure frame
 SYML             Allow 8-character external names
 ENT PROC1,L7_    PROC1 is an entry point with address L7_
 ENT PROC2,L8_    Similarly for PROC2,
 ENT V1,L3_          V1,
 ENT V2,L4_          and V2
 LINK             Output data in link (static) frame
L3_ EQU *
 BSZ '1           Reserve one word for L3_, init to zero
 PROC             Output data in proc (procedure) frame
 LINK
L4_ EQU *
 BSZ '1           Reserve one word for L4_, init to zero
 PROC
 LINK
L5_ EQU *
 BSZ '1           Reserve one word for L5_, init to zero
 PROC
 LINK
L6_ EQU *
 BSZ '1           Reserve one word for L6_, init to zero
 PROC
 LINK
 EXT E1           Declare symbol E1 external to this module
L1_ EQU *
 IP E1            Generate a pointer for the loader to fill in
 PROC
 LINK
 EXT E2           Declare symbol E2 external to this module
```

```
L2_ EQU *
 IP E2            Generate a pointer for the loader
 PROC
 PROC
L65535_ EQU *     Beginning of a procedure
 EAL L7_          Set up stack frame owner pointer for debugging
 STL SB%+18
 LDA =’4000
 STA% SB%
 PRTN             "Procedure Return" at end of procedure
L7_ ECB L65535_,,SB%+’0,0,’24    Entry control block for procedure
 DATA ’5          PL/I character varying form procedure name
 DATA ’170362
 DATA ’167743
 DATA ’130405
 PROC
L65534_ EQU *     Beginning of second procedure
 EAL L8_          Set up stack frame owner pointer
 STL SB%+18
 LDA =’4000
 STA% SB%
 PRTN
L8_ ECB L65534_,,SB%+’0,0,’24    Entry control block
 DATA ’5          Procedure name
 DATA ’170362
 DATA ’167743
 DATA ’131370
 END              End of this module
```

Storage Allocation

C Code

```
int   i,           /* a static integer variable */
      ii [10];     /* a static integer array */

struct
   {
   int f1, f2;
   } s;            /* a static structure with two integer fields */

main (argc, argv)  /* a non-trivial procedure, with arguments */
int argc;             /* integer argument */
char **argv;          /* pointer-to-pointer-to-character argument */
   {
   int li,            /* a local integer variable */
       lii [10];      /* a local integer array */

   struct
      {
      int m1, m2;
      } ls;           /* a local structure with two integer fields */

   i;                 /* use of various things in expressions */
   ii [0];
   s.f1;
   li;
   lii [0];
   ls.m1;
   argv;
   argc;
   }
```

IMF Stream 1

```
32    MODULE_OP; beginning of next module in input stream
59       SEQ_OP; beginning of entry point declaration list
1           Object number 1 is an entry point...
1              whose name is 1 character long...
233               i
59       SEQ_OP; next member of entry point list
3           Object number 3 is an entry point...
1              whose name is 1 character long...
243               s
59       SEQ_OP; next member of entry point list
4           Object number 4 is an entry point...
4              whose name is 4 characters long...
237               m
225               a
233               i
238               n
59       SEQ_OP; next member of entry point list
2           Object number 2 is an entry point...
```

```
2              whose name is 2 characters long...
233               i
233               i
39       NULL_OP; end of entry point list (and this module)
39    NULL_OP; end of modules in this input stream
```

IMF Stream 2

```
32    MODULE_OP; beginning of next module
59       SEQ_OP; beginning of static data declarations/definitions
14          DEFINE_STAT_OP; reserve space for a static variable
1              Object ID is 1
39             NULL_OP; no initializers for this variable
1              Object size is 1 word
59       SEQ_OP; next member of static data list
14          DEFINE_STAT_OP; reserve space for a static variable
2              Object ID is 2
39             NULL_OP; no initializers for this variable
10             Object size is 10 words
59       SEQ_OP; next member of static data list
14          DEFINE_STAT_OP; reserve space for a static variable
3              Object ID is 3
39             NULL_OP; no initializers for this variable
2              Object size is 2 words
39       NULL_OP; end of static data list
39    NULL_OP; end of modules in this input stream
```

IMF Stream 3

```
32  MODULE_OP; beginning of next module in input stream
59     SEQ_OP; beginning of procedure definition list
50        PROC_DEFN_OP; procedure definition follows
4            Object ID of procedure is 4
2            Procedure has 2 arguments
4            Procedure name is 4 characters long...
237             m
225             a
233             i
238             n
49           PROC_DEFN_ARG_OP; description of first argument
5               Argument has object ID 5
1               Argument has mode 1 (INTEGER)
0               Argument has disposition 0 (pass-by-value)
1               Argument is 1 word long
49              PROC_DEFN_ARG_OP; description of second argument
6                  Argument has object ID 6
4                  Argument has mode 4 (LONG UNSIGNED, or pointer)
1                  Argument has disposition 1 (pass-by-reference)
2                  Argument is 2 words long
39                 NULL_OP; end of argument descriptor list
59           SEQ_OP; beginning of procedure code
13              DEFINE_DYNM_OP; reserve space for local variable
7                  Object ID 7
39                 NULL_OP; no initializers
```

```
1                  Size 1 word
59              SEQ_OP; next element of procedure code
13                 DEFINE_DYNM_OP; reserve space for local variable
8                     Object ID 8
39                    NULL_OP; no initializers
10                    Size 10 words
59              SEQ_OP; next element of procedure code
13                 DEFINE_DYNM_OP; reserve space for local variable
9                     Object ID 9
39                    NULL_OP; no initializers
2                     Size 2 words
59              SEQ_OP; next element of procedure code
40                 OBJECT_OP; (this is actually an expression subtree)
1                     Mode 1 (INTEGER)
1                     Object ID 1
59              SEQ_OP; next element of procedure code
25                 INDEX_OP; again, the top of an expression subtree
1                     Mode 1 (INTEGER)
40                    OBJECT_OP; the base address of the array
7                        Mode 7 (STOWED)
2                        Object ID 2
9                     CONST_OP; this one is the index expression
1                        Mode 1 (INTEGER)
1                        Length is 1 word
0                        Value of word is 0
1                     Array element size is 1 word
59              SEQ_OP; next element of procedure code
58                 SELECT_OP; again, the top of an expression subtree
1                     Mode 1 (INTEGER)
0                     Field to be selected has word offset 0 from base
40                    OBJECT_OP; the base address of the structure
7                        Mode 7 (STOWED)
3                        Object ID 3
59              SEQ_OP; next element of procedure code
40                 OBJECT_OP; an expression, again
1                     Mode 1 (INTEGER)
7                     Object ID is 7
59              SEQ_OP; next element of procedure code
25                 INDEX_OP; using an array element as an expression
1                     Mode 1 (INTEGER)
40                    OBJECT_OP; the base of the array being indexed
7                        Mode 7 (STOWED)
8                        Object ID is 8
9                     CONST_OP; this is the subscript expression
1                        MODE 1 (INTEGER)
1                        Length of constant is 1 word
0                        Value of constant is 0
1                     Array element size is 1 word
59              SEQ_OP; next element of procedure code
58                 SELECT_OP; using struct field as an expression
1                     Mode 1 (INTEGER)
0                     Offset of selected field is 0 words from base
40                    OBJECT_OP; the base address of the structure
7                        Mode 7 (STOWED)
9                        Object ID is 9
59              SEQ_OP; next element of procedure code
```

```
40              OBJECT_OP; just the top of an expression tree
4                  Mode 4 (LONG_UNSIGNED, or pointer)
6                  Object ID is 6
59           SEQ_OP; next element of procedure code
40              OBJECT_OP; an expression, again
1                  Mode 1 (INTEGER)
5                  Object ID is 5
39           NULL_OP; end of procedure body code (and proc defn)
39     NULL_OP; end of procedure defn list (and this module)
39  NULL_OP; end of this input stream
```

PMA Code

```
 SEG               Assemble in 64V mode
 RLIT              Place literals in procedure frame
 SYML              Allow 8-character external symbols
 ENT I,L1_         I is an entry point, with address L1_
 ENT S,L3_         S is an entry point, with address L3_
 ENT MAIN,L4_      MAIN is an entry point, with address L4_
 ENT II,L2_        II is an entry point, with address L2_
 LINK              Emit data in link (static data) frame
L1_ EQU *
 BSZ '1            Reserve 1 word for L1_
 PROC
 LINK
L2_ EQU *
 BSZ '12           Reserve 10 words ('12 octal) for L2_
 PROC
 LINK
L3_ EQU *
 BSZ '2            Reserve 2 words for L3_
 PROC
 PROC
L65535_ EQU *      Beginning of a procedure
 ARGT              Transfer arguments from caller
 EAL L4_           Set up stack frame owner pointer for debugging
 STL SB%+18
 LDA ='4000
 STA% SB%
 LDA SB%+'24,*     Make copy of pass-by-value arguments
 STA SB%+'24
 LDA LB%+'400      Evaluate expression 1,
 LDA LB%+'401                          2,
 LDA LB%+'413                          3,
 LDA SB%+'25                           4,
 LDA SB%+'32                           5,
 LDA SB%+'44                           6,
 LDL SB%+'27                           7,
 LDA SB%+'24                           8
 PRTN              Return from the procedure
L4_ ECB L65535_,,SB%+'24,2,'46      Entry control block
 DATA '4           PL/I char varying procedure name
 DATA '166741
 DATA '164756
 END               End of this PMA module
```

String Copy

C Code

```
strcpy (s, t)      /* copy string s to string t */
char s[], t[];
   {
   int i;          /* a local integer variable, for indexing */

   i = 0;          /* start at first char */
   while ((t[i] = s[i]) != '\0') /* copy until a zero char is seen */
      i += 1;      /* incrementing the index each time */
   }
```

IMF Stream 1

```
32    MODULE_OP; begins the input module
59       SEQ_OP; begins sequence of entry points
1           Object number 1 is an entry point
6              whose name is 6 characters long...
243               s
244               t
242               r
227               c
240               p
249               y
39       NULL_OP; terminates entry point list
39    NULL_OP; terminates list of modules in the input
```

IMF Stream 2

```
32    MODULE_OP; begins the input module
39       NULL_OP; terminates the sequence of static data definitions
39    NULL_OP; terminates list of modules in the input
```

IMF Stream 3

```
32  MODULE_OP; begins next module in the input stream
59     SEQ_OP; first procedure definition follows
50        PROC_DEFN_OP; procedure definition follows
1            Procedure is object number 1
2            There are 2 arguments, described below.
6            Procedure name is 6 characters long...
243             s
244             t
242             r
227             c
240             p
249             y
49           PROC_DEFN_ARG_OP; description of argument number 1
2               Argument is object number 2
4               LONG_UNS_MODE; argument is a pointer
```

```
1              REF_DISP; argument is passed-by-reference
2              Argument is 2 words long
49             PROC_DEFN_ARG_OP; description of argument number 2
3                 Argument is object number 3
4                 LONG_UNS_MODE; argument is a pointer
1                 REF_DISP; argument is passed-by-reference
2                 Argument is 2 words long
39                NULL_OP; end of argument descriptions
59          SEQ_OP; beginning of procedure code list
13             DEFINE_DYNM_OP; declare a local variable
4                 Variable has object id 4
39                No initializers
1                 Variable is 1 word in length
59          SEQ_OP; next element of code list
5              ASSIGN_OP
1                 INT_MODE
40                OBJECT_OP
1                    INT_MODE
4                    Object id is 4
9                 CONST_OP
1                    INT_MODE
1                    Constant has length 1
0                    Constant has value 0
1                 Assignment transfers 1 word
59          SEQ_OP; next element of code list
65             WHILE_OP
37                NE_OP
1                    INT_MODE
5                    ASSIGN_OP
1                       INT_MODE
25                      INDEX_OP; the LHS of the assignment
1                          INT_MODE
15                         DEREF_OP; this is the base address
7                             STOWED_MODE
40                            OBJECT_OP
4                                LONG_UNS_MODE
3                                Object id is 3
40                         OBJECT_OP; this is the subscript
1                             INT_MODE
4                             Object id is 4
1                          Array element size is 1 word
25                      INDEX_OP; the RHS of the assignment
1                          INT_MODE
15                         DEREF_OP; the base address expression
7                             STOWED_MODE
40                            OBJECT_OP
4                                LONG_UNS_MODE
2                                Object id is 2
40                         OBJECT_OP; the subscript, again
1                             INT_MODE
4                             Object id is 4
1                          Array element size is 1 word
1                       Assignment transfers 1 word
9                    CONST_OP; the right operand of the NE_OP
1                       INT_MODE
1                       Constant is 1 word long
```

```
0                        Constant has value 0
59                   SEQ_OP; beginning of the body of the WHILE loop
1                       ADDAA_OP
1                          INT_MODE
40                         OBJECT_OP
1                             INT_MODE
4                             Object id is 4
9                          CONST_OP
1                             INT_MODE
1                             Length is 1 word
1                             Value is 1
39                   NULL_OP; end of the body of the WHILE loop
39             NULL_OP; end of the statements for the current procedure
39     NULL_OP; end of the procedure definitions in this module
39  NULL_OP; end of this input stream
```

PMA Code

```
 SEG
 RLIT
 SYML
 ENT STRCPY,L1_
 PROC
L65535_ EQU *
 ARGT             Transfer arguments from caller
 EAL L1_          Set up stack frame owner pointer for debugging
 STL SB%+18
 LDA =’4000
 STA% SB%
 CRA              Load A with zero
 STA SB%+’32      Store in i
 JMP L65533_      Enter the WHILE loop at the test
 FIN              (dump literals here)
L65532_ EQU *     Top of the WHILE loop body
 IRS SB%+’32      Increment i
 RCB                 (takes two instructions on this turkey machine)
L65533_ EQU *     WHILE loop test starts here
 LDX SB%+’32      Load index register with i
 LDA SB%+’24,*X   Load next character in string s
 STA SB%+’27,*X   Store it in next position in string t
 BNE L65532_      If it’s non-zero, go back for more characters
L65534_ EQU *     Exit label for the WHILE loop
 PRTN             Return from string copy procedure
L1_ ECB L65535_,,SB%+’24,2,’33
 DATA ’6
 DATA ’171764
 DATA ’171343
 DATA ’170371
 END
```

Tree Print

C Code

```
/* recursive tree-printing routine */

#define NULL 0 /* a nil pointer */

struct TNODE   /* the data structure out of which the tree is built */
   {
   int value;
   struct TNODE *left, *right;
   };
typedef struct TNODE tnode;  /* create a new type, for convenience */


treeprint (t)
tnode *t;
   {
   if (t != NULL)
      {
      treeprint (t->left);
      printf ("%4d\n", t->value);      /* output the 'value' field */
      treeprint (t->right);
      }
   }
```

IMF Stream 1

```
32    MODULE_OP; beginning of next module in input stream
59       SEQ_OP; beginning of list of entry point declarations
1           Object number 1 is an entry point
9              whose name is 9 characters long...
244               t
242               r
229               e
229               e
240               p
242               r
233               i
238               n
244               t
39       NULL_OP; end of entry point list for this module
39    NULL_OP; end of modules in this input stream
```

IMF Stream 2

```
32    MODULE_OP; beginning of next module in this input stream
59       SEQ_OP; beginning of list of static data definitions
11          DECLARE_STAT_OP; declare an externally-defined object
3              Object has object id 3
6              Name of object is 6 characters long...
240               p
```

```
242                r
233                i
238                n
244                t
230                f
39          NULL_OP; end of static data for this module
39       NULL_OP; end of modules in this input stream
```

IMF Stream 3

```
32       MODULE_OP; beginning of next module in input stream
59          SEQ_OP; beginning of list of procedure definitions
50             PROC_DEFN_OP; procedure definition follows
1                 Procedure has object id 1
1                 Procedure has 1 argument
9                 Procedure name is 9 characters long...
244                  t
242                  r
229                  e
229                  e
240                  p
242                  r
233                  i
238                  n
244                  t
49                PROC_DEFN_ARG_OP; description of procedure argument
2                    Argument has object id 2
4                    Argument has mode LONG_UNS (it's a pointer)
1                    Argument has REF disposition (pass-by-reference)
2                    Argument is 2 words long
39                   NULL_OP; no further argument descriptions
59                SEQ_OP; beginning of statement list
24                   IF_OP
1                       INT_MODE
37                      NE_OP
4                          LONG_UNS_MODE
40                         OBJECT_OP
4                             LONG_UNS_MODE
2                             Object has id 2
9                          CONST_OP
4                             LONG_UNS_MODE
2                             Constant has length 2
0                             First word of constant is 0
0                             Second word of constant is 0
59                      SEQ_OP; then-part of IF statement follows
48                         PROC_CALL_OP (for treeprint)
1                             INT_MODE
40                            OBJECT_OP; this is the base address
7                                STOWED_MODE; arbitrary, for procs.
1                                Object id of procedure is 1
47                            PROC_CALL_ARG_OP
7                                STOWED_MODE
15                               DEREF_OP
7                                   STOWED_MODE
58                                  SELECT_OP
```

```
4                                    LONG_UNS_MODE
1                                    Field offset is 1 word
15                                   DEREF_OP
7                                       STOWED_MODE
40                                      OBJECT_OP
4                                          LONG_UNS_MODE
2                                          Object id is 2
39                              NULL_OP; end of argument list
59                           SEQ_OP; next element of IF body follows
48                              PROC_CALL_OP (for printf)
1                                  INT_MODE
40                                 OBJECT_OP; the base address
7                                     STOWED_MODE; ignored in this case
3                                     Object id of procedure is 3
47                                 PROC_CALL_ARG_OP
1                                     INT_MODE
15                                    DEREF_OP
1                                        INT_MODE
51                                       REFTO_OP
4                                           LONG_UNS_MODE (pointer to char)
9                                           CONST_OP; this is the string
7                                              STOWED_MODE
5                                              Length is 5 words
165                                            Value is %
180                                                      4
228                                                      d
138                                                      newline
0                                                        0
47                                    PROC_CALL_ARG_OP
1                                        INT_MODE
58                                       SELECT_OP
1                                           INT_MODE
0                                           Field is at offset 0
15                                          DEREF_OP; base address of struct
7                                              STOWED_MODE
40                                             OBJECT_OP
4                                                 LONG_UNS_MODE
2                                           Object id is 2
39                                       NULL_OP; end of argument list
59                           SEQ_OP; next element of body of IF follows
48                              PROC_CALL_OP
1                                  INT_MODE (ignored)
40                                 OBJECT_OP; the procedure address
7                                     STOWED_MODE
1                                     Object id is 1
47                                 PROC_CALL_ARG_OP
7                                     STOWED_MODE
15                                    DEREF_OP
7                                        STOWED_MODE
58                                       SELECT_OP
4                                           LONG_UNS_MODE
3                                           Field has offset 3 words
15                                          DEREF_OP
7                                              STOWED_MODE
40                                             OBJECT_OP
4                                                 LONG_UNS_MODE
```

```
2                                        Object id is 2
39                              NULL_OP; end of treeprint args
39                     NULL_OP; end of then-part of IF
39                     NULL_OP; omitted else-part of the IF
39               NULL_OP; end of statements in this procedure
39         NULL_OP; end of procedure definition list (and this module)
39      NULL_OP; end of modules in this input stream
```

PMA Code

```
 SEG
 RLIT
 SYML
 ENT TREEPRINT,L1_   Make 'treeprint' available outside this module
 LINK
 EXT PRINTF
L3_ EQU *
 IP PRINTF           Use 'printf', defined outside this module
 PROC
 PROC
L65535_ EQU *        Beginning of 'treeprint'
 ARGT                Transfer arguments from caller
 EAL L1_             Set up stack frame owner pointer for debugging
 STL SB%+18
 LDA ='4000
 STA% SB%
 LDL SB%+'24         If the argument...
 BLEQ L65534_        ...is nonzero...
 LDX ='1                we first get the pointer in the 'left' field
 EAXB SB%+'24,*X        by addressing the field with XB
 LDL XB%+'0             then loading the value of the pointer into L
 STL SB%+'27            then storing it in a temporary
 PCL L1_                call 'treeprint' recursively
 AP SB%+'27,*SL         using the temporary to pass the value
 LINK
 DATA '245              This is the format string for 'printf'...
 DATA '264              ...value is "%4d\n\0"
 DATA '344
 DATA '212
 DATA '0
 PROC
 PCL LB%+'400,*         Here's the call to 'printf'
 AP LB%+'402,S             passing the formatting string
 AP SB%+'24,*SL            and the value field of the current tnode
 LDX ='3                Now we get the pointer in the 'right' field
 EAXB SB%+'24,*X        pretty much as we did it before
 LDL XB%+'0
 STL SB%+'27
 PCL L1_                and call 'treeprint',
 AP SB%+'27,*SL         passing it the pointer to the right subtree
L65534_ EQU *
 PRTN                   return from 'treeprint'
L1_ ECB L65535_,,SB%+'24,1,'31
 DATA '11
 DATA '172362
```

```
DATA '162745
DATA '170362
DATA '164756
DATA '172041
END
```

The 'Drift' Compiler

The 'Drift' Language

Description

'Drift' is an extremely simplified programming language for computers with Von Neumann-style architectures. While too restrictive to be generally useful, it does have a few interesting features. It is an expression-oriented language rather than statement-oriented; non-declarative constructs generally yield a value of some sort. The syntax is intended to be conducive to simple error recovery schemes (particularly to panic-mode symbol skipping) while retaining a reasonable degree of cleanliness and human engineering (for example, statements are terminated by end-of-line, rather than some delimiter like a semicolon; continuations across lines are represented explicitly by an '&'). The semantics of the language closely reflect the expression-oriented semantics of the VCG itself.

'Drift' programs are composed of variable declarations, function declarations, and expressions. Variables may be global in scope or restricted to the function in which they are declared. Function declarations may not be nested. All variables represent floating point quantities; all functions return floating point quantities. The return value of a function is the return value of the last expression in the expression series that comprises its body. Functions may be recursive and need not be defined before use. The function named 'main' is assumed to be the main program, and will be invoked by whatever environment supports 'drift' programs.

Expressions are made of the four standard operators (+, -, *, /), assignment ('=', treated uniformly as an arithmetic operator yielding the value of its right-hand-side), two-way selection ('if'), and a loop ('while'). Variables in expressions yield their values (or take on new ones if used as the left operand of an assignment operator). They must be declared before they are used. Function calls in expressions cause parameters to be passed by value to the named function; the value returned by the function then takes the place of the call in the expression. The quad ('#') is a pseudovariable used for input/output. When used in the right-hand-side of an assignment, it causes input of a floating point value from standard input; when used in the left-hand-side, it causes output of the right-hand-side to standard output.

BNF

The syntax of 'drift' presented below employs the extended BNF used throughout the Software Tools Subsystem documentation.

Alternatives enclosed in curly braces {} may be repeated any number of times, including zero. Alternatives enclosed in square brackets [] may be used once or not at all.

```
program ->
   newlines {declaration newlines} eof

declaration ->
      global_variable_declaration
   |  function_declaration

global_variable_declaration ->
   'float' identifier {',' newlines identifier}

identifier ->
   letter {letter | digit | '_'}

newlines ->
   {NEWLINE}

function_declaration ->
   'function' identifier '(' formal_parameters ')' newlines
      {local_variable_declaration newlines}
      series newlines
   'end_function'

formal_parameters ->
   [identifier {',' newlines identifier}]

local_variable_declaration ->
   'float' identifier {',' newlines identifier}

series ->
   expression newlines {expression newlines}

expression ->
   sum {'=' sum}

sum ->
   term {('+' | '-') term}

term ->
   primary {('*' | '/') primary}

primary ->
      '#'
   |  'null'
   |  number
   |  identifier
   |  identifier '(' actual_parameters ')'
   |  loop
   |  conditional
   |  '(' series ')'

loop ->
```

```
   'while' newlines series newlines
      'do' newlines
      series newlines
      'od'

conditional ->
   'if' newlines series newlines
   'then' newlines series newlines
   ['else' newlines series newlines]
   'fi'

actual_parameters ->
   [series {',' newlines series}]
```

Examples

The following programs compute the value of a base raised to a positive integer exponent. The first is iterative, while the second is recursive.

```
-- A sample program in 'drift'

   float x, y

   function power (base, exponent)

      float result

      result = 1
      while exponent       -- that is, while exponent <> 0
         do
         result = result * base
         exponent = exponent - 1
         od

      result
   end_function

   function main ()
      x = #
      y = #
      # = power (x, y)
   end_function




-- The same sample, only done recursively

   float x, y

   function power (base, exponent)
      if exponent
```

```
      then base * power (base, exponent - 1)
      else 1
      fi
   end_function

   function main ()
      x = #
      y = #
      # = power (x, y)
   end_function
```

The Compiler

The 'drift' compiler was implemented in Ratfor under the Software Tools Subsystem in about two man-days. Conceptually, it generates intermediate form code for the VCG in two passes: the first (lexical and syntactic) generates an internal form used only by the front end, while the second (semantic) does semantic checking and converts the internal form to IMF.

The lexical analyzer used in the compiler is a fairly standard one employed in a number of Software Tools Subsystem programs because of its compactness and high speed. It resides almost entirely in the subroutine 'getsym'.

The parser code is input to 'stacc', a recursive-descent parser generator that is part of the Software Tools package. The production for 'program' is actually the main routine of the compiler. Note that very little attempt is made to recover from syntactic errors; the purpose of the compiler is the demonstration of code generation, not parsing. The parser drives the compilation process, making calls on the lexical analyzer and internal form code generation routines as necessary.

The IMF generation process is concentrated in the subroutine 'semantic_analysis' and its descendents. This routine invokes 'void_context', 'lvalue_context', and 'rvalue_context' to propagate contextual information during a traversal of the internal form tree. The bulk of the IMF generation takes place in 'rvalue_context', since most operators yield floating point values. Special cases are handled in the other two contexts: left-hand-sides of assignments by 'lvalue_context' and constructs that don't yield values by 'void_context'. Since the internal form is tree structured, the translation to IMF is straightforward.

Global Variable Definitions

```
# global variables for 'drift' compiler

# dynamic storage used by symbol table routines:
   DS_DECL (Mem, MEMSIZE)
```

```
# symbol tables:
   pointer Functions, Globals, Locals, Reserved_words
   common /stcom/ Functions, Globals, Locals, Reserved_words

# lexical stuff:
   character Inbuf (INBUFSIZE), Symtext (MAX_SYM_LEN)
   integer Symbol, Ibp, Current_line
   real Symval
   common /lexcom/ Inbuf, Symtext, Symbol, Ibp, Current_line, Symval

# files for I/O:
   filedes In_stream, Ent_stream, Data_stream, Code_stream
   common /filcom/ In_stream, Ent_stream, Data_stream, Code_stream

# internal form memory:
   integer Ifmem (INTERNAL_FORM_MEMSIZE), Next_ifmem
   common /if1com/ Ifmem
   common /if2com/ Next_ifmem

# semantic stack:
   ifpointer Stack (SEMANTIC_STACK_SIZE)
   integer Sp
   common /semcom/ Sp, Stack

# other junk:
   integer Next_obj_id, Ex$in_id, Ex$out_id, Error_count
   common /miscom/ Next_obj_id, Ex$in_id, Ex$out_id, Error_count
```

Parser Source Code

```
# 'stacc' parser for drift

.common "drift_com.r.i";  # file containing global variables

.symbol Symbol;           # "current symbol" variable

.scanner getsym;          # name of lexical analysis routine

.state state;             # "parse state" variable

.terminal                 # terminal symbols
   256                       # start higher than largest char value
   FLOAT_SYM
   ID_SYM
   FUNCTION_SYM
   END_FUNCTION_SYM
   NULL_SYM
   NUMBER_SYM
   WHILE_SYM
   DO_SYM
   OD_SYM
   IF_SYM
   THEN_SYM
   ELSE_SYM
```

```
   FI_SYM
   ;


   NEWLINE
   EOF
   ;



program ->
                  !  call begin_program
   nls
   {
      declaration
      nls
      }
   EOF.
                  !  call end_program
                  ?  call pmr ("EOF expected*n"p, state)
   ;



declaration ->
      global_variable_declaration
   |
      function_declaration
   ;



global_variable_declaration ->
   FLOAT_SYM
   ID_SYM
                  !  call declare_global_variable (Symtext)
                  ?   call  pmr  ("missing identifier*n"p, state,
state)
   {
      ',,'
      nls
      ID_SYM
                  !  call declare_global_variable (Symtext)
                  ?  call pmr ("missing identifier*n"p, state)
      }
   ;



nls ->
   {
      NEWLINE
      }
   ;
```

```
function_declaration ->
   FUNCTION_SYM
   ID_SYM
                  !  call begin_function (Symtext)
                  ?  call pmr ("missing function name*n"p, state)
   '('
                  !  call make_null
                  ?  call pmr ("missing parameters*n"p, state)
   formal_parameters
                  !  call make_function_parameters
   ')'
                  ?  call pmr ("missing ')'*n"p, state)
   nls
                  !  call make_null
   {
      local_variable_declaration
      nls
      }
   series
                  !  call make_function_body
                  ?  call pmr ("missing function body*n"p, state)
   nls
   END_FUNCTION_SYM
                  !  call end_function
                  ?   call  pmr   ("missing   'end_function'*n"p,
state)
   ;


formal_parameters ->
   ID_SYM
                  !  call declare_formal_parameter (Symtext)
   {
      ','
      nls
      ID_SYM
                  !  call declare_formal_parameter (Symtext)
                  ?  call pmr ("missing identifier*n"p, state)
      }
   |
      epsilon
   ;


local_variable_declaration ->
   FLOAT_SYM
   ID_SYM
                  !  call declare_local_variable (Symtext)
                  ?  call pmr ("missing identifier*n"p, state)
   {
      ','
      nls
      ID_SYM
```

```
                    !  call declare_local_variable (Symtext)
                    ?  call pmr ("missing identifier*n"p, state)
      }
   ;


series ->
   expression
   nls
   {
      expression
                    !  call sequentialize
      nls
      }
   ;


expression ->
   sum
   {
      '='
      sum
                    !  call make_dyad (ASSIGN_NODE)
                    ?    call  pmr  ("missing  right-hand-side*n"p,
state)
      }
   ;


sum ->
                    !  integer node
   term
   {
      (
            '+'
                    !  node = ADD_NODE
         |
            '-'
                    !  node = SUBTRACT_NODE
         )
      term
                    !  call make_dyad (node)
                    ?  call pmr ("missing right operand*n"p, state)
      }
   ;


term ->
                    !  integer node
   primary
   {
      (
```

```
            '*'
                  !  node = MULTIPLY_NODE
         |
            '/'
                  !  node = DIVIDE_NODE
         )
      primary
                  !  call make_dyad (node)
                  ?  call pmr ("missing right operand*n"p, state)
      }
   ;


primary ->
                  !  character id (MAX_SYM_LEN)
      '#'
                  !  call make_quad
   |
      NULL_SYM
                  !  call make_null
   |
      NUMBER_SYM
                  !  call make_constant (Symval)
   |
      ID_SYM
                  !  call scopy (Symtext, 1, id, 1)
      (
            '('
            actual_parameters
            ')'
                  !  call make_call (id)
                  ?  call pmr ("missing ')'*n"p, state)
         |
            epsilon
                  !  call make_object (id)
         )
   |
      loop
   |
      conditional
   |
      '('
      series
      ')'
                  ?  call pmr ("missing ')'*n"p, state)
   ;


loop ->
   WHILE_SYM
   nls
   series
                  ?   call  pmr  ("missing  loop   condition*n"p,
state)
```

```
   nls
   DO_SYM
                  ?  call pmr ("missing 'do'*n"p, state)
   nls
   series
                  ?  call pmr ("missing loop body*n"p, state)
   nls
   OD_SYM
                  !  call make_loop
                  ?  call pmr ("missing 'od'*n"p, state)
   ;


conditional ->
   IF_SYM
   nls
   series
                  ?    call  pmr  ("missing  'if'  condition*n"p,
state)
   nls
   THEN_SYM
                  ?  call pmr ("missing 'then'*n"p, state)
   nls
   series
                  ?  call pmr ("missing then_part*n"p, state)
   (
         ELSE_SYM
         nls
         series
                  ?  call pmr ("missing else_part*n"p, state)
         nls
      |
         nls
                  !  call make_null
   )
   FI_SYM
                  !  call make_conditional
                  ?  call pmr ("missing 'fi'*n"p, state)
   ;


actual_parameters ->
                  !  call make_null
      series
                  !  call make_actual_parameter
      {
         ','
         nls
         series
                  !  call make_actual_parameter
                  ?  call pmr ("missing parameter after  ','*n"p,
state)
         }
   |
```

```
      epsilon
   ;
```

Remainder of Compiler Source Code

```
# drift --- sample compiler for VCG demonstration

define (GLOBAL_VARIABLES,"drift_com.r.i")

define (MAX_SYM_LEN, MAXLINE)
define (MEMSIZE, 4096)
define (SEMANTIC_STACK_SIZE, 100)
define (INTERNAL_FORM_MEMSIZE, 20000)
define (INBUFSIZE, 300)
define (PBLIMIT, 150)

define (UNDEFINED, 0)
define (DEFINED, 1)

define (ifpointer, integer)
define (unknown, integer)

# Types of internal form nodes:
define (ADD_NODE,1)
define (ARG_NODE,2)
define (ASSIGN_NODE,3)
define (CALL_NODE,4)
define (COND_NODE,5)
define (CONSTANT_NODE,6)
define (DECLARE_VAR_NODE,7)
define (DIVIDE_NODE,8)
define (FUNCTION_NODE,9)
define (IO_NODE,10)
define (LOOP_NODE,11)
define (MULTIPLY_NODE,12)
define (NULL_NODE,13)
define (PARAM_NODE,14)
define (SEQ_NODE,15)
define (SUBTRACT_NODE,16)
define (VAR_NODE,17)
define (LAST_NODE_TYPE,VAR_NODE)

# Elements of internal form records:
define (ARG_EXPR (n), Ifmem (n + 2))
define (ARG_LIST (n), Ifmem (n + 3))
define (COND (n), Ifmem (n + 2))
define (ELSE_PART (n), Ifmem (n + 4))
define (FUNC_BODY (n), Ifmem (n + 5))
define (LEFT (n), Ifmem (n + 2))
define (LINE_NUM (n), Ifmem (n + 1))
define (LOOP_BODY (n), Ifmem (n + 3))
define (NODE_TYPE (n), Ifmem (n))
define (NPARAMS (n), Ifmem (n + 4))
define (OBJ_ID (n), Ifmem (n + 2))
define (PARAM_LIST (n), Ifmem (n + 3))
```

```
define (RIGHT (n), Ifmem (n + 3))
define (THEN_PART (n), Ifmem (n + 3))
define (WORD1 (n), Ifmem (n + 2))
define (WORD2 (n), Ifmem (n + 3))

include "drift.stacc.defs"              # macro defns. produced by 'stacc'
include "/uc/allen/vcg/vcg_defs.r.i"    # macro defns. for IMF operators



   integer state

   call program (state)
   if (state ~= ACCEPT)
      call error ("syntactically incorrect program"p)
   stop
   end



include "drift.stacc.r"    # Ratfor source code produced by 'stacc'



# begin_function --- set up environment for compiling a function

   subroutine begin_function (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   pointer mktabl

   integer info2 (2)
   integer lookup, gen_id

   ifpointer func_node
   ifpointer ialloc

   Next_ifmem = 1          # initialize internal form memory
   Locals = mktabl (1)     # initialize local variable symbol table
   Sp = 0                  # initialize semantic stack pointer

  # Place function name in 'Functions' table, if it's not already there
   if (lookup (name, info2, Functions) == YES)
      if (info2 (2) == DEFINED)
         call warning ("function *s multiply defined*n"p, name)
      else {
         info2 (2) = DEFINED
         call enter (name, info2, Functions)
         }
   else {
      info2 (1) = gen_id (1)
      info2 (2) = DEFINED
      call enter (name, info2, Functions)
      }
```

```
   # Output an entry point definition for the procedure:
    call emit (SEQ_OP, Ent_stream)
    call emit (info2 (1), Ent_stream)      # object id of function
    call emit_string (name, Ent_stream)    # function name

   # Put function node on semantic stack:
    func_node = ifalloc (FUNCTION_NODE)
    NPARAMS (func_node) = 0
    OBJ_ID (func_node) = info2 (1)
    call push (func_node)

    return
    end



# begin_program --- do pre-program initialization

    subroutine begin_program

    include GLOBAL_VARIABLES

    pointer mktabl

    filedes create, open

    character infile (MAXARG)

    integer getarg, gen_id

    call dsinit (MEMSIZE)      # init. dynamic storage
    Functions = mktabl (2)     # symbol table for function names
    Globals = mktabl (1)       # symbol table for global variables
    Reserved_words = mktabl (1)   # symbol table for reserved words
    Next_obj_id = 1            # for object id generator
    Error_count = 0
    Ibp = 1                    # buffer pointer...
    Inbuf (Ibp) = EOS          # ...and input buffer used by lexer
    Current_line = 0

   # open input file specified on command line:
    if (getarg (1, infile, MAXARG) == EOF)
       In_stream = STDIN
    else {
       In_stream = open (infile, READ)
       if (In_stream == ERR)
          call cant (infile)
       }

   # create temporary files for passing the IMF to the code generator:
    Ent_stream = create ("_drift_.ct1"s, READWRITE)
    Data_stream = create ("_drift_.ct2"s, READWRITE)
    Code_stream = create ("_drift_.ct3"s, READWRITE)
    if (Ent_stream == ERR || Data_stream == ERR || Code_stream == ERR)
       call error ("can't open temporary files _drift_.ct[1-3]"p)
```

```
   call emit (MODULE_OP, Ent_stream)
   call emit (MODULE_OP, Data_stream)
   call emit (MODULE_OP, Code_stream)

  # define object id's for the two run-time routines we'll need:
   Ex$in_id = gen_id (1)            # run-time routine for input
   call emit (SEQ_OP, Data_stream)
   call emit (DECLARE_STAT_OP, Data_stream)
   call emit (Ex$in_id, Data_stream)
   call emit_string ("EX$IN"s, Data_stream)

   Ex$out_id = gen_id (1)           # run-time routine for output
   call emit (SEQ_OP, Data_stream)
   call emit (DECLARE_STAT_OP, Data_stream)
   call emit (Ex$out_id, Data_stream)
   call emit_string ("EX$OUT"s, Data_stream)

  # build the reserved-word table used by the lexical analyzer:
   call enter ("do"s, DO_SYM, Reserved_words)
   call enter ("else"s, ELSE_SYM, Reserved_words)
   call enter ("end_function"s, END_FUNCTION_SYM, Reserved_words)
   call enter ("fi"s, FI_SYM, Reserved_words)
   call enter ("float"s, FLOAT_SYM, Reserved_words)
   call enter ("function"s, FUNCTION_SYM, Reserved_words)
   call enter ("if"s, IF_SYM, Reserved_words)
   call enter ("null"s, NULL_SYM, Reserved_words)
   call enter ("od"s, OD_SYM, Reserved_words)
   call enter ("then"s, THEN_SYM, Reserved_words)
   call enter ("while"s, WHILE_SYM, Reserved_words)

  # fire up lexical analysis:
   call getsym

   return
   end



# declare_formal_parameter - put param name in table, assign obj id

   subroutine declare_formal_parameter (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   integer obj_id
   integer lookup, gen_id

   ifpointer param_node
   ifpointer ifalloc

   if (lookup (name, obj_id, Locals) == YES) {
      call warning ("*s:  multiply declared*n"p, name)
      return
      }
```

```
   obj_id = gen_id (1)
   call enter (name, obj_id, Locals)

  # create new parameter node and combine it with previous sequence
  #   on the semantic stack:
   param_node = ifalloc (PARAM_NODE)
   OBJ_ID (param_node) = obj_id
   call push (param_node)
   call sequentialize
   NPARAMS (Stack (Sp - 1)) += 1

   return
   end



# declare_global_variable - put name in global table, assign object id

   subroutine declare_global_variable (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   integer obj_id
   integer lookup, gen_id

   if (lookup (name, obj_id, Globals) == YES) {
      call warning ("*s:  multiply declared*n"p, name)
      return
      }

   obj_id = gen_id (1)
   call enter (name, obj_id, Globals)

  # go ahead and reserve space in the static data storage area for
  #   the variable we just declared:
   call emit (SEQ_OP, Data_stream)
   call emit (DEFINE_STAT_OP, Data_stream)
   call emit (obj_id, Data_stream)
   call emit (NULL_OP, Data_stream)    # no initializers
   call emit (2, Data_stream)          # 2 words for a floating object

   return
   end



# declare_local_variable - enter name in local table, assign object id

   subroutine declare_local_variable (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   integer obj_id
```

```
   integer lookup, gen_id

   ifpointer decl_var_node
   ifpointer ifalloc

   if (lookup (name, obj_id, Locals) == YES) {
      call warning ("*s:  multiply declared*n"p, name)
      return
      }

   obj_id = gen_id (1)
   call enter (name, obj_id, Locals)

  # make new variable declaration node and put it into a sequence
  #   following all previously declared variables:
   decl_var_node = ifalloc (DECLARE_VAR_NODE)
   OBJ_ID (decl_var_node) = obj_id
   call push (decl_var_node)
   call sequentialize

   return
   end



# emit --- place value on an output stream

   subroutine emit (val, stream)
   integer val
   filedes stream

   call print (stream, "*i*n"s, val)

   return
   end



# emit_string - place length of string and string on an output stream

   subroutine emit_string (str, stream)
   character str (ARB)
   filedes stream

   integer i
   integer length

   call emit (length (str), stream)
   for (i = 1; str (i) ~= EOS; i += 1)
      call emit (str (i), stream)

   return
   end
```

```
# end_function --- clean up after parse of a function is completed

   subroutine end_function

   include GLOBAL_VARIABLES

   call semantic_analysis (Stack (Sp))
   call rmtabl (Locals)    # get rid of all local variable information

   return
   end


# end_program --- clean up after the entire program is parsed

   subroutine end_program

   include GLOBAL_VARIABLES

   pointer position

   integer info2 (2)
   integer sctabl

   character sym (MAX_SYM_LEN)

   logical first

   call close (In_stream)

  # terminate IMF streams by ending sequence of definitions, then
  #   ending sequence of modules:
   call emit (NULL_OP, Ent_stream);  call emit (NULL_OP, Ent_stream)
   call emit (NULL_OP, Data_stream); call emit (NULL_OP, Data_stream)
   call emit (NULL_OP, Code_stream); call emit (NULL_OP, Code_stream)

   call close (Ent_stream)
   call close (Data_stream)
   call close (Code_stream)

  # check function table for names that were referenced but not
  #   declared; presumably these are externally compiled
   first = TRUE
   position = 0
   while (sctabl (Functions, sym, info2, position) ~= EOF)
      if (info2 (2) == UNDEFINED) {
         if (first) {
            call print (STDOUT, "External symbols:*n"p)
            first = FALSE
            }
         call print (STDOUT, "*s*n"p, sym)
         }

   return
   end
```

```
# gen_id --- generate new object identifiers

   integer function gen_id (num_ids)
   integer num_ids

   include GLOBAL_VARIABLES

   gen_id = Next_obj_id
   Next_obj_id += num_ids

   return
   end



# getsym --- get next symbol from input stream

   subroutine getsym

   include GLOBAL_VARIABLES

   procedure getchar forward
   procedure putback (c) forward
   procedure empty_buffer forward

   character c

   integer i
   integer getlin, lookup

   real ctor

   logical too_long, continuation

   continuation = FALSE    # true if we want to ignore a line boundary
   repeat { # until we find a legal symbol

      repeat
         getchar
         until (c ~= ' 'c)

      select (c)

         when (NEWLINE) {
            Current_line += 1
            Symbol = NEWLINE
            if (~continuation)
               break
            }

         when (';'c) {
            Symbol = NEWLINE     # but no line number advance
            if (~continuation)
```

```
            break
         }

      when ('-'c) {
         getchar
         if (c == '-'c) {     # -- begins comments
            empty_buffer
            Current_line += 1
            Symbol = NEWLINE
            if (~continuation)
               break
            }
         else {
            putback (c)
            Symbol = '-'c
            break
            }
         }

      when ('&'c)
         continuation = TRUE

      when ('+'c, '*'c, '/'c, '#'c, '('c, ')'c, ','c, '='c, EOF) {
         Symbol = c
         break
         }

      when (SET_OF_LETTERS) { # a-z or A-Z; starting an identifier
         too_long = FALSE
         i = 1
         while (IS_LETTER (c) || IS_DIGIT (c) || c == '_'c) {
            Symtext (i) = c
            i += 1
            if (i > MAX_SYM_LEN) {
               i -= 1
               too_long = TRUE
               }
            getchar
            }
         putback (c)
         Symtext (i) = EOS
         if (too_long)
            call warning("symbol beginning *s is too long*n"p,Symtext)
         if (lookup (Symtext, Symbol, Reserved_words) == NO)
            Symbol = ID_SYM
         break
         }

      when ('.'c, SET_OF_DIGITS) {
         putback (c)
         Symval = ctor (Inbuf, Ibp)    # advances Ibp
         Symbol = NUMBER_SYM
         break
         }

   else
```

```
            call warning ("'*c':  unrecognized character*n"p, c)

         }  # repeat until a valid symbol is found

      return


      # getchar --- get the next character from the input stream

         procedure getchar {

            if (Inbuf (Ibp) == EOS)       # time to read a new buffer?
               if (getlin (Inbuf (PBLIMIT), In_stream) == EOF)
                  c = EOF
               else {
                  c = Inbuf (PBLIMIT)     # pick up the first char read
                  Ibp = PBLIMIT + 1
                  }
            else {                        # text was already available
               c = Inbuf (Ibp)
               Ibp += 1
               }

            }


      # putback --- push a character back onto the input stream

         procedure putback (c) {
            character c

            if (Ibp <= 1)
               call error ("too many characters pushed back"p)
            else {
               Ibp -= 1
               Inbuf (Ibp) = c
               }

            }


      # empty_buffer --- kill remainder of line in input buffer

         procedure empty_buffer {

            Inbuf (Ibp) = EOS          # will force a read in 'getchar'

            }

      end



   # ifalloc - allocate space for a particular type node in
   # internal form memory

      ifpointer function ifalloc (node_type)
```

```
   integer node_type

   include GLOBAL_VARIABLES

  # These declarations assume that the internal form node types form
  #   a dense ascending sequence of integers from 1 to LAST_NODE_TYPE:
   integer sizeof (LAST_NODE_TYPE)
   data sizeof / _
      4,     # ADD_NODE
      3,     # ARG_NODE
      4,     # ASSIGN_NODE
      4,     # CALL_NODE
      5,     # COND_NODE
      4,     # CONSTANT_NODE
      3,     # DECLARE_VAR_NODE
      4,     # DIVIDE_NODE
      6,     # FUNCTION_NODE
      2,     # IO_NODE
      4,     # LOOP_NODE
      4,     # MULTIPLY_NODE
      2,     # NULL_NODE
      3,     # PARAM_NODE
      4,     # SEQ_NODE
      4,     # SUBTRACT_NODE
      3 _    # VAR_NODE
      /

   if (node_type < 1 || node_type > LAST_NODE_TYPE)
      call error ("ifalloc received bad node type"p)

   if (Next_ifmem + sizeof (node_type) > INTERNAL_FORM_MEMSIZE)
      call error ("insufficient internal form memory"p)

   ifalloc = Next_ifmem
   Next_ifmem += sizeof (node_type)

   NODE_TYPE (ifalloc) = node_type
   LINE_NUM (ifalloc) = Current_line

   return
   end



# lvalue_context --- generate VCG code for constructs used as lvalues
#     (assumes I/O quads have already been eliminated from LHS's)

   subroutine lvalue_context (node)
   ifpointer node

   include GLOBAL_VARIABLES

   select (NODE_TYPE (node))

      when (VAR_NODE) {
         call emit (OBJECT_OP, Code_stream)
```

```
            call emit (FLOAT_MODE, Code_stream)
            call emit (OBJ_ID (node), Code_stream)
            }

         when (SEQ_NODE) {
            if (NODE_TYPE (RIGHT (node)) == NULL_NODE)
               call lvalue_context (LEFT (node))
            else {
               call emit (SEQ_OP, Code_stream)
               call void_context (LEFT (node))
               call lvalue_context (RIGHT (node))
               }
            }

      else
         call warning("assignment on line *i has an illegal left side*n"p,
            LINE_NUM (node))

      return
      end



   # make_actual_parameter --- link actual parameter expression to list

      subroutine make_actual_parameter

      include GLOBAL_VARIABLES

      ifpointer act_param
      ifpointer ifalloc, pop

      act_param = ifalloc (ARG_NODE)
      ARG_EXPR (act_param) = pop (0)
      call push (act_param)
      call sequentialize

      return
      end



   # make_call --- generate a call to a function

      subroutine make_call (name)
      character name (ARB)

      include GLOBAL_VARIABLES

      integer info2 (2)
      integer lookup, gen_id

      ifpointer call_node
      ifpointer ifalloc, pop

     # if function name is in Functions table, all is well; if not,
```

```
  #   we add it provisionally (it may be defined later).
   if (lookup (name, info2, Functions) == NO) {
      info2 (1) = gen_id (1)
      info2 (2) = UNDEFINED
      call enter (name, info2, Functions)
      }

   call_node = ifalloc (CALL_NODE)
   OBJ_ID (call_node) = info2 (1)
   ARG_LIST (call_node) = pop (0)
   call push (call_node)

   return
   end



# make_conditional --- make conditional (if-then-else-fi) node

   subroutine make_conditional

   include GLOBAL_VARIABLES

   ifpointer cond
   ifpointer if_alloc, pop

   cond = if_alloc (COND_NODE)
   ELSE_PART (cond) = pop (0)
   THEN_PART (cond) = pop (0)
   COND (cond) = pop (0)

   call push (cond)
   return
   end



# make_constant --- make constant node from given value

   subroutine make_constant (val)
   real val

   include GLOBAL_VARIABLES

   real rkluge
   integer ikluge (2)
   equivalence (rkluge, ikluge) # used to unpack floating point constants

   ifpointer cnode
   ifpointer ifalloc

   cnode = ifalloc (CONSTANT_NODE)
   rkluge = val
   WORD1 (cnode) = ikluge (1)
   WORD2 (cnode) = ikluge (2)
```

```
   call push (cnode)
   return
   end


# make_dyad --- make node for a dyadic operator (=, +, -, *, /)

   subroutine make_dyad (node_type)
   integer node_type

   include GLOBAL_VARIABLES

   ifpointer node
   ifpointer ifalloc, pop

   node = ifalloc (node_type)
   RIGHT (node) = pop (0)
   LEFT (node) = pop (0)
   call push (node)

   return
   end


# make_function_body --- add function body to function definition node

   subroutine make_function_body

   include GLOBAL_VARIABLES

   ifpointer body
   ifpointer pop

   call sequentialize      # combine declarations and code
   body = pop (0)          # note deep-stack addressing makes sequencing
   FUNC_BODY (Stack (Sp)) = body       # necessary...

   return
   end


# make_function_parameters --- add params to function definition node

   subroutine make_function_parameters

   include GLOBAL_VARIABLES

   ifpointer params
   ifpointer pop

   params = pop (0)     # note:  deep-stack addressing makes use of
   PARAM_LIST (Stack (Sp)) = params    # a particular sequence necessary
```

```
   return
   end


# make_loop --- pop cond and body off stack, generate a loop node

   subroutine make_loop

   include GLOBAL_VARIABLES

   ifpointer loop
   ifpointer ifalloc, pop

   loop = ifalloc (LOOP_NODE)
   LOOP_BODY (loop) = pop (0)
   COND (loop) = pop (0)
   call push (loop)

   return
   end


# make_null --- push new "null operator" node on stack

   subroutine make_null

   include GLOBAL_VARIABLES

   ifpointer ifalloc

   call push (ifalloc (NULL_NODE))

   return
   end


# make_object --- push node referencing a variable on the stack

   subroutine make_object (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   ifpointer node
   ifpointer ifalloc

   integer obj_id
   integer lookup

   node = ifalloc (VAR_NODE)

   if (lookup (name, obj_id, Locals) == NO
     && lookup (name, obj_id, Globals) == NO) {
```

```
      call warning ("*s:  undeclared identifier*n"p, name)
      obj_id = 0
      }

   OBJ_ID (node) = obj_id
   call push (node)

   return
   end



# make_quad --- generate an input/output operation node

   subroutine make_quad

   include GLOBAL_VARIABLES

   ifpointer ifalloc

   call push (ifalloc (IO_NODE))

   return
   end



# output_arguments --- output IMF for procedure call arguments

   subroutine output_arguments (arg_node)
   ifpointer arg_node

   include GLOBAL_VARIABLES

   select (NODE_TYPE (arg_node))

      when (ARG_NODE) {
         call emit (PROC_CALL_ARG_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call rvalue_context (ARG_EXPR (arg_node))
         }

      when (NULL_NODE)
         ;

      when (SEQ_NODE) {
         call output_arguments (LEFT (arg_node))
         call output_arguments (RIGHT (arg_node))
         }

   else
      call error ("in output_argument:  shouldn't happen"p)

   return
   end
```

```
# output_params --- output IMF for procedure formal parameter definitions

   subroutine output_params (param_node)
   ifpointer param_node

   include GLOBAL_VARIABLES

   select (NODE_TYPE (param_node))

      when (PARAM_NODE) {
         call emit (PROC_DEFN_ARG_OP, Code_stream)
         call emit (OBJ_ID (param_node), Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call emit (VALUE_DISP, Code_stream)
         call emit (2, Code_stream) # FLOATs are 2 words long
         }

      when (NULL_NODE)
         ;

      when (SEQ_NODE) {
         call output_params (LEFT (param_node))
         call output_params (RIGHT (param_node))
         }

   else
      call error ("in output_param:  shouldn't happen"p)

   return
   end



# pmr --- panic mode recovery for parser

   subroutine pmr (message, state)
   character message (ARB)
   integer state

   include GLOBAL_VARIABLES

   call warning (message)
   state = ACCEPT

   while (Symbol ~= EOF && Symbol ~= ')'c && Symbol ~= NEWLINE
     && Symbol ~= END_FUNCTION_SYM && Symbol ~= THEN_SYM
     && Symbol ~= ELSE_SYM && Symbol ~= FI_SYM && Symbol ~= DO_SYM
     && Symbol ~= OD_SYM && Symbol ~= ','c)
      call getsym

   return
   end
```

```
# pop --- pop a node pointer off the semantic stack

   ifpointer function pop (dummy)
   integer dummy     # needed to satisfy FORTRAN syntax requirements

   include GLOBAL_VARIABLES

   if (Sp < 1)
      call error ("semantic stack underflow"p)

   pop = Stack (Sp)
   Sp -= 1

   return
   end



# push --- push a node pointer onto the semantic stack

   subroutine push (node)
   ifpointer node

   include GLOBAL_VARIABLES

   if (Sp >= SEMANTIC_STACK_SIZE)
      call error ("semantic stack overflow"p)

   Sp += 1
   Stack (Sp) = node

   return
   end



# rvalue_context --- generate VCG code for constructs used as rvalues

   subroutine rvalue_context (node)
   ifpointer node

   include GLOBAL_VARIABLES

   select (NODE_TYPE (node))

      when (ADD_NODE, SUBTRACT_NODE, MULTIPLY_NODE, DIVIDE_NODE) {
         select (NODE_TYPE (node))
            when (ADD_NODE)
               call emit (ADD_OP, Code_stream)
            when (SUBTRACT_NODE)
               call emit (SUB_OP, Code_stream)
            when (MULTIPLY_NODE)
               call emit (MUL_OP, Code_stream)
            when (DIVIDE_NODE)
               call emit (DIV_OP, Code_stream)
```

```
      call emit (FLOAT_MODE, Code_stream)
      call rvalue_context (LEFT (node))
      call rvalue_context (RIGHT (node))
      }

   when (ASSIGN_NODE) {
      if (NODE_TYPE (LEFT (node)) == IO_NODE) {
        # fake up output by calling 'ex$out' at run time:
         call emit (PROC_CALL_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call emit (OBJECT_OP, Code_stream)
         call emit (STOWED_MODE, Code_stream)
         call emit (Ex$out_id, Code_stream)
         call emit (PROC_CALL_ARG_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call rvalue_context (RIGHT (node))
         call emit (NULL_OP, Code_stream)
         }
      else {
         call emit (ASSIGN_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call lvalue_context (LEFT (node))
         call rvalue_context (RIGHT (node))
         call emit (2, Code_stream)    # assign 2 words
         }
      }

   when (CALL_NODE) {
      call emit (PROC_CALL_OP, Code_stream)
      call emit (FLOAT_MODE, Code_stream)
      call emit (OBJECT_OP, Code_stream)
      call emit (STOWED_MODE, Code_stream)
      call emit (OBJ_ID (node), Code_stream)
      call output_arguments (ARG_LIST (node))
      call emit (NULL_OP, Code_stream)
      }

   when (COND_NODE) {
      call emit (IF_OP, Code_stream)
      call emit (FLOAT_MODE, Code_stream)
      call rvalue_context (COND (node))
      call rvalue_context (THEN_PART (node))
      if (NODE_TYPE (ELSE_PART (node)) == NULL_NODE)
         call warning ("'if' on line *i needs an 'else' part*n"p,
            LINE_NUM (node))
      call rvalue_context (ELSE_PART (node))
      }

   when (CONSTANT_NODE) {
      call emit (CONST_OP, Code_stream)
      call emit (FLOAT_MODE, Code_stream)
      call emit (2, Code_stream)    # 2-word floats
      call emit (WORD1 (node), Code_stream)
      call emit (WORD2 (node), Code_stream)
      }
```

```
      when (DECLARE_VAR_NODE) {
         call emit (DEFINE_DYNM_OP, Code_stream)
         call emit (OBJ_ID (node), Code_stream)
         call emit (NULL_OP, Code_stream)    # no initializers
         call emit (2, Code_stream)    # size is 2 words
         }

      when (IO_NODE) {
        # fake up input by calling 'ex$in' at run time:
         call emit (PROC_CALL_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call emit (OBJECT_OP, Code_stream)
         call emit (STOWED_MODE, Code_stream)
         call emit (Ex$in_id, Code_stream)
         call emit (NULL_OP, Code_stream)    # no arguments
         }

      when (LOOP_NODE)
         call warning("while-loop at line *i is used as an rvalue*n"p,
            LINE_NUM (node))

      when (NULL_NODE)
         call emit (NULL_OP, Code_stream)

      when (SEQ_NODE) {
         if (NODE_TYPE (RIGHT (node)) == NULL_NODE)
            call rvalue_context (LEFT (node))
         else {
            call emit (SEQ_OP, Code_stream)
            call void_context (LEFT (node))  # can never yield a value
            call rvalue_context (RIGHT (node))
            }
         }

      when (VAR_NODE) {
         call emit (OBJECT_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call emit (OBJ_ID (node), Code_stream)
         }

   else
      call error ("in rvalue_context:  shouldn't happen"p)

   return
   end



# semantic_analysis --- check function and output VCG code for it

   subroutine semantic_analysis (func)
   ifpointer func

   include GLOBAL_VARIABLES

  # output the procedure definition node:
```

```
   call emit (SEQ_OP, Code_stream)
   call emit (PROC_DEFN_OP, Code_stream)
   call emit (OBJ_ID (func), Code_stream)
   call emit (NPARAMS (func), Code_stream)
   call emit_string (EOS, Code_stream) # we'll ignore this for now

  # take care of the formal parameter declarations:
   call output_params (ARG_LIST (func))
   call emit (NULL_OP, Code_stream)

  # finally, take care of local variables and the function code:
   call rvalue_context (FUNC_BODY (func))

   return
   end



# sequentialize --- combine two nodes with a "sequence" node

   subroutine sequentialize

   include GLOBAL_VARIABLES

   ifpointer seq_node
   ifpointer ifalloc, pop

   seq_node = ifalloc (SEQ_NODE)
   RIGHT (seq_node) = pop (0)
   LEFT (seq_node) = pop (0)
   call push (seq_node)

   return
   end



# void_context - generate VCG code for constructs that yield no value

   subroutine void_context (node)
   ifpointer node

   include GLOBAL_VARIABLES

   select (NODE_TYPE (node))

      when (COND_NODE) {      # an 'if' used as a statement
         call emit (IF_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call rvalue_context (COND (node))
         call void_context (THEN_PART (node))
         call void_context (ELSE_PART (node))
         }

      when (LOOP_NODE) {
         call emit (WHILE_LOOP_OP, Code_stream)
```

```
            call rvalue_context (COND (node))
            call void_context (LOOP_BODY (node))
            }

         when (SEQ_NODE) {
            call emit (SEQ_OP, Code_stream)
            call void_context (LEFT (node))
            call void_context (RIGHT (node))
            }

      else
         call rvalue_context (node)

      return
      end



# warning --- print warning message

      subroutine warning (format, a1, a2, a3, a4, a5, a6, a7, a8, a9)
      character format (ARB)
      unknown a1, a2, a3, a4, a5, a6, a7, a8, a9

      include GLOBAL_VARIABLES

      call print (ERROUT, "*i:  "s, Current_line)
      call print (ERROUT, format, a1, a2, a3, a4, a5, a6, a7, a8, a9)
      Error_count += 1

      return
      end
```

Run-Time Support Routines Source Code

```
* RUN-TIME SUPPORT FOR 'DRIFT'
*           UNFORTUNATELY, EX$IN MUST BE WRITTEN IN ASSEMBLER SINCE
*           FORTRAN DOESN'T ALLOW FUNCTIONS WITHOUT ARGUMENTS.
*           AN ALTERNATIVE WOULD BE TO HAVE THE COMPILER GENERATE
*           A DUMMY ARGUMENT ON THE CALL; THEN EX$IN AND EX$OUT
*           COULD BE WRITTEN IN RATFOR, PASCAL, OR WHAT HAVE YOU.


* EX$IN --- READ A LINE FROM STANDARD INPUT, CONVERT FROM CHARACTER
*           TO REAL, AND RETURN VALUE

            SEG
            RLIT
            SYML

            SUBR     EX$IN

EX$IN       ECB      EX$IN$

            DYNM     LINE(100),I
```

```
EX$IN$      EQU     *
            CALL    GETLIN          READ NEXT INPUT LINE
            AP      LINE,S
            AP      =-10,SL
            BGT     CVT_IN          IF WE HIT EOF,
            CALL    SWT                JUST QUIT
CVT_IN      EQU     *
            LT                      OTHERWISE,
            STA     I
            CALL    CTOR               CONVERT TO REAL
            AP      LINE,S
            AP      I,SL
            PRTN                       AND RETURN WITH VALUE IN F

            END


* EX$OUT - WRITE REAL VALUE TO STANDARD OUTPUT, RETURN VALUE UNCHANGED

            SEG
            RLIT
            SYML

            SUBR    EX$OUT
EX$OUT      ECB     EX$OUT$,,VAL,1

            DYNM    VAL(3)

EX$OUT$     EQU     *
            ARGT
            CALL    PRINT           JUST USE SWT I/O TO OUTPUT VALUE
            AP      =-11,S             ON STDOUT
            AP      =C'*r*n.',S
            AP      VAL,*SL
            FLD     VAL,*           RETURN VALUE IN F SO THIS FUNCTION
            PRTN                       BEHAVES LIKE A PSEUDO-VARIABLE

            END
```

Intermediate Form Operator/Function Index

```
bit fields
   FIELD_OP

bitwise logical operators
   ANDAA_OP, AND_OP, COMPL_OP, LSHIFTAA_OP, LSHIFT_OP, ORAA_OP,
   OR_OP, RSHIFTAA_OP, RSHIFT_OP, XORAA_OP, XOR_OP

boolean operators
   NOT_OP, SAND_OP, SOR_OP

bounds checking
   CHECK_RANGE_OP, CHECK_LOWER_OP, CHECK_UPPER_OP

branch
   GOTO_OP, LABEL_OP

break (loop termination)
   BREAK_OP, NEXT_OP

byte access
   FIELD_OP

call
   procedures, functions, subroutines:  PROC_CALL_OP,
      PROC_CALL_ARG_OP

case statement
   SWITCH_OP

character operations
   FIELD_OP

checking
   CHECK_RANGE_OP, CHECK_UPPER_OP, CHECK_LOWER_OP

choice
   boolean:  IF_OP
   arithmetic:  SWITCH_OP

coercions
   CONVERT_OP

common blocks
   DECLARE_STAT_OP

comparison operators
   EQ_OP, GE_OP, GT_OP, LE_OP, LT_OP, NE_OP

complement
   COMPL_OP, NOT_OP

conditional expressions
   IF_OP

conjunction
   AND_OP, ANDAA_OP
```

```
constants
   CONST_OP

continuation of loops
   NEXT_OP

control flow
   BREAK_OP, DO_LOOP_OP, FOR_LOOP_OP, GOTO_OP, IF_OP, LABEL_OP,
   NEXT_OP, PROC_CALL_OP, RETURN_OP, SEQ_OP, SWITCH_OP,
   WHILE_LOOP_OP

conversions
   CONVERT_OP

copy
   ASSIGN_OP

data
   CONST_OP, INITIALIZER_OP, ZERO_INITIALIZER_OP

deallocation
   UNDEFINE_DYNM_OP

declarations
   DEFINE_DYNM_OP, DEFINE_STAT_OP, DECLARE_STAT_OP

decrement
   POSTDEC_OP, PREDEC_OP, SUBAA_OP

default case
   DEFAULT_OP

define
   procedures:  PROC_DEFN_OP
   storage:  DEFINE_DYNM_OP, DEFINE_STAT_OP

dereferencing
   DEREF_OP

descriptor
   address:  REFTO_OP

difference
   SUBAA_OP, SUB_OP

disjunction
   ORAA_OP, OR_OP

disposition of arguments
   PROC_DEFN_ARG_OP

division
   DIVAA_OP, DIV_OP, RSHIFTAA_OP, RSHIFT_OP

do loop
   C-style:  DO_LOOP_OP
```

```
   Fortran-style:  FOR_LOOP_OP

double precision
   LONG_FLOAT_MODE

dynamic variablesa
   DEFINE_DYNM_OP, UNDEFINE_DYNM_OP

element
   of an array:  INDEX_OP
   of a structure or record:  SELECT_OP

else
   IF_OP

entry points
   See descriptions of Intermediate Form stream 1

equality
   EQ_OP, NE_OP

exception
   No exception handling, yet

exclusive-or
   XORAA_OP, XOR_OP

exit
   from procedures:  RETURN_OP
   from loops:  BREAK_OP, NEXT_OP

external symbols
   DECLARE_STAT_OP

false
   zero

fields
   of words:  FIELD_OP
   of structures or records:  SELECT_OP

fixed-point modes
   INT_MODE, LONG_INT_MODE, UNS_MODE, LONG_UNS_MODE

floating-point modes
   FLOAT_MODE, LONG_FLOAT_MODE

flow of control
   BREAK_OP, DO_LOOP_OP, FOR_LOOP_OP, GOTO_OP, IF_OP, LABEL_OP,
   NEXT_OP, PROC_CALL_OP, RETURN_OP, SEQ_OP, SWITCH_OP,
   WHILE_LOOP_OP

formal parameters
   PROC_DEFN_ARG_OP
```

```
functions
   declaration:  PROC_DEFN_OP
   call:  PROC_CALL_OP

global variables
   declaration:  DECLARE_STAT_OP
   definition:  DEFINE_STAT_OP

goto
   GOTO_OP

greater-than
   GT_OP

guarantees
   None here.

immediate operands
   CONST_OP

inclusive-or
   ORAA_OP, OR_OP

incrementation
   ADDAA_OP, POSTINC_OP, PREINC_OP

indexing
   INDEX_OP

indirection
   DEREF_OP

inequality
   EQ_OP, NE_OP

initialization
   INITIALIZER_OP, ZERO_INITIALIZER_OP

integer
   modes:  INT_MODE, LONG_INT_MODE, UNS_MODE, LONG_UNS_MODE
   conversion:  CONVERT_OP

inverse
   additive:  NEG_OP
   bitwise:  COMPL_OP
   boolean:  NOT_OP

invocation
   of procedures:  PROC_CALL_OP

iteration
   DO_LOOP_OP, FOR_LOOP_OP, WHILE_LOOP_OP

jump
   GOTO_OP
```

```
labels
   LABEL_OP

layouts
   of storage:  FIELD_OP; also see data modes

less-than
   LT_OP

literals
   CONST_OP

local variables
   DEFINE_DYNM_OP, UNDEFINE_DYNM_OP

locations
   REFTO_OP

logical operators
   ANDAA_OP, AND_OP, COMPL_OP, NOT_OP, ORAA_OP, OR_OP,
   SAND_OP, SOR_OP, XORAA_OP, XOR_OP

long data modes
   LONG_INT_MODE, LONG_UNS_MODE, LONG_FLOAT_MODE

loops
   DO_LOOP_OP, FOR_LOOP_OP, WHILE_LOOP_OP

lower bound checking
   CHECK_RANGE_OP, CHECK_LOWER_OP

lvalues
   DEREF_OP, INDEX_OP, OBJECT_OP, SELECT_OP

magnitude comparisons (unsigned arithmetic)
   GE_OP, GT_OP, LE_OP, LT_OP

member
   of an array:  INDEX_OP
   of a structure or record:  SELECT_OP

minus
   SUBAA_OP, SUB_OP

modes
   INT_MODE, LONG_INT_MODE, UNS_MODE, LONG_UNS_MODE, FLOAT_MODE,
   LONG_FLOAT_MODE, STOWED_MODE

modulus
   REMAA_OP, REM_OP

multidimensional arrays
   INDEX_OP

multiplication
   MULAA_OP, MUL_OP, LSHIFTAA_OP, LSHIFT_OP
```

multiway branch
   SWITCH_OP

negation
   NEG_OP

objects
   OBJECT_OP

or (logical)
   ORAA_OP, OR_OP, XORAA_OP, XOR_OP

otherwise
   in Pascal case statement: DEFAULT_OP

packed data structures
   arrays: no support
   structures: FIELD_OP

parameters
   formal: PROC_DEFN_ARG_OP
   actual: PROC_CALL_ARG_OP
   pass-by-value: see VALUE_DISP in PROC_DEFN_ARG_OP
   pass-by-reference: see REF_DISP in PROC_DEFN_ARG_OP

partial fields
   FIELD_OP

passing parameters
   PROC_CALL_ARG_OP
   by value: see VALUE_DISP in PROC_DEFN_ARG_OP
   by reference: see REF_DISP in PROC_DEFN_ARG_OP

pointers
   obtaining them: REFTO_OP
   indirection through them: DEREF_OP

portions of a machine word
   FIELD_OP

postdecrement
   POSTDEC_OP

postincrement
   POSTINC_OP

predecrement
   PREDEC_OP

preincrement
   PREINC_OP

primitive data modes
   INT_MODE, LONG_INT_MODE, UNS_MODE, LONG_UNS_MODE, FLOAT_MODE,
   LONG_FLOAT_MODE, STOWED_MODE

```
procedure
   calling:  PROC_CALL_OP
   definition:  PROC_DEFN_OP

public symbols
   See description of IMF stream 1
   DECLARE_STAT_OP

quotient
   DIVAA_OP, DIV_OP

range checking
   CHECK_RANGE_OP, CHECK_LOWER_OP, CHECK_UPPER_OP

real
   FLOAT_MODE, LONG_FLOAT_MODE

records
   STOWED_MODE
   SELECT_OP

reference (pass-by)
   see REF_DISP in PROC_DEFN_ARG_OP

references
   REFTO_OP

remainder
   REMAA_OP, REM_OP

reserving storage
   DEFINE_DYNM_OP, DEFINE_STAT_OP

returning from procedures/function/subroutines
   RETURN_OP

semicolon
   SEQ_OP

sets
   bit vector implementations:  FIELD_OP

shift
   left:  LSHIFTAA_OP, LSHIFT_OP
   right:  RSHIFTAA_OP, RSHIFT_OP

short data modes
   INT_MODE, UNS_MODE, FLOAT_MODE

sign change
   NEG_OP

stack
   allocating storage on:  DEFINE_DYNM_OP
   deallocating storage on:  UNDEFINE_DYNM_OP
```

```
statements
   ASSIGN_OP, BREAK_OP, DO_LOOP_OP, FOR_LOOP_OP, GOTO_OP, IF_OP,
   NEXT_OP, PROC_CALL_OP, RETURN_OP, SWITCH_OP, WHILE_LOOP_OP

static variables
   DEFINE_STAT_OP, DECLARE_STAT_OP

storage
   allocation:  DEFINE_DYNM_OP, DEFINE_STAT_OP, DECLARE_STAT_OP
   deallocation:  UNDEFINE_DYNM_OP

structures
   STOWED_MODE
   SELECT_OP

subscripting
   INDEX_OP

subtraction
   SUBAA_OP, SUB_OP, PREDEC_OP, POSTDEC_OP

sum
   ADDAA_OP, ADD_OP, POSTINC_OP, PREINC_OP

switch
   SWITCH_OP, CASE_OP, DEFAULT_OP

target label
   LABEL_OP

temporary variables
   DEFINE_DYNM_OP, UNDEFINE_DYNM_OP

termination
   of procedures:  RETURN_OP

tests
   EQ_OP, GE_OP, GT_OP, LE_OP, LT_OP, NE_OP

transfers
   GOTO_OP

true
   non-zero

truncation
   CONVERT_OP

type
   primitive types:  INT_MODE, LONG_INT_MODE, UNS_MODE,
      LONG_UNS_MODE, FLOAT_MODE, LONG_FLOAT_MODE, STOWED_MODE

unary
   minus:  NEG_OP
   complementation:  COMPL_OP, NOT_OP
```

unsigned data modes
   UNS_MODE, LONG_UNS_MODE

upper bound checking
   CHECK_RANGE_OP, CHECK_UPPER_OP

use list
   DECLARE_STAT_OP

value (pass-by)
   see VALUE_DISP in PROC_DEFN_ARG_OP

variables
   OBJECT_OP

vector element selection
   INDEX_OP

zeros
   ZERO_INITIALIZER_OP

ADDENDUM

Arnold D. Robbins

August, 1984

Introduction

With the second release of the Georgia Tech C Compiler, 'vcg' has been changed in two ways. This addendum describes those changes.

Object Code Produced Directly

'Vcg' has been changed to directly generate 64V-mode relocatable object code, instead of symbolic assembly language. This *enormously* speeds up code generation time, since the Prime Macros Assembler, PMA, is no longer needed to turn the assembly code into binary.

As an option, 'vcg' will still produce PMA, which can be assembled normally. This is occasionally useful, since the object code routines still have some bugs buried deep within them. See the help on 'vcg' for details on producing assembly code.

Shift Instructions

Whenever a shift instruction is needed, 'vcg' used to generate code to build an instruction and then XEC it. Now, 'vcg' will generate a shortcall into a table of shift instructions. This table is included in the "vcglib" library, and in the "ciolib" library for C programs. This change saves code space for programs with a lot of shift instructions.

TABLE OF CONTENTS