Shells:
User's Guide

HP 9000 Computers

HP Part No. B2355-90046
Printed in USA August 1992

Second Edition
E0892

FINAL TRIM SIZE : 7.5 in x 9.0 in

Notices

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing or use of this material.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Printing History

New editions of this manual will incorporate all material updated since the previous edition.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

January, 1991 ... Edition 1. This Edition documents material related to shells relevant to the 8.X release of HP-UX.

- Bourne, C, and Korn replace the like-named parts of manual part number 97089-90062, Edition 3, dated October 1987. Changes were made to each of these parts to update incorrect or confusing information.
- Key Shell is new functionality for the 8.X release of HP-UX.
- PAM replaces the PAM chapter of manual part number 98515-90004, Edition 1, dated September, 1989. Changes were made to update incorrect or confusing information, and to reflect PAM software changes effective for the 8.X release of HP-UX.

August, 1992 ... Edition 2. This Edition documents material related to shells relevant to the 9.X release of HP-UX.

- POSIX Shell is new functionality for the 9.X release of HP-UX.
- PAM not supported in the 9.X release; PAM information removed.
- Shell comparison information added from *The Beginner's Guide to HP-UX*, part number B1862-90000.

Contents

Figures

Tables

1

Introduction to Shells

What are Shells?

For purposes of this User's Guide, a shell is the interface between HP-UX and you, the user. The shell interprets the text you type, and the keys you press, in order to direct the HP-UX operating system to take an appropriate action. A shell can also serve as a programming language.

What is Bourne Shell?

Bourne Shell is the oldest shell. It was written by Stephen Bourne at Bell Laboratories. The Bourne Shell has been the default shell for HP-UX users, and has been a de facto standard in the industry. The Bourne Shell has neither the interactive features, nor the complex programming constructs, of the C and Korn shells.

What is C Shell?

C Shell is a shell developed by Bill Joy at the University of California at Berkeley. The C Shell syntax resembles that of the C programming language. It has powerful interactive features like command history and file name completion.

What is Korn Shell?

Korn Shell is a newer shell developed by David Korn at Bell Laboratories, and is upwardly compatible with most Bourne shell features. It has interactive features like C Shell, but executes faster and has extended in-line command editing capability.

What is POSIX Shell?

POSIX Shell is based on the standard defined in *Portable Operating System Interface (POSIX)* - IEEE P1003.2. This standard is designed to be used by both application programmers and system administrators. This standard is intended to describe language interfaces and utilities in sufficient detail so an application developer can understand the required interfaces without access to the source code of existing implementations on which they are based.

Most of the POSIX Shell features are similar to the Korn Shell. This guide covers both shells in the same Part. In addition to the common features, this guide also covers the differences between the POSIX and Korn Shells and the new features introduced in the POSIX Shell.

The POSIX Shell(sh) is named the same as the Bourne Shell but is put in `/bin/posix` directory. The Bourne Shell in put in the `/bin` directory.

What is Key Shell?

Key Shell is a softkey interface for the Korn Shell. It was developed by Hewlett-Packard Company. It provides menus and online help to assist you in building commands to perform such tasks as viewing files, printing files, and listing contents of directories. It is also user-extensible, in that you can create your own softkeys and online help.

Choosing Between the Shells

- With the Key Shell, new users can comfortably start with the Korn Shell and use the Key Shell interface to ease into the syntax and power of the Korn Shell.
- If you feel comfortable with a command-line driven shell, the Bourne Shell might be the place to start. Remember that you can later migrate to the Korn Shell if you want its interactive or programming features.
- If you are a C programmer, the C Shell may be a good starting point, since you will already be familiar with the C Shell syntax.

- The POSIX Shell is the future HP-UX Shell and also the standard Shell in the HP-UX environment. It is fully compatible with HP-UX Korn Shell and also conforms to the standards defined in POSIX - IEEE P1003.2.

Table 1-1 lists some additional features which may help you make a decision on which shell would be best for the kind of work you are doing:

Table 1-1. Comparison of Shell Features

Features	Description	Bourne	POSIX Korn Key	C
Command history	A feature allowing commands to be stored in a buffer, then modified and reused.	No	Yes	Yes
Line editing	The ability to modify the current or previous command lines with a text editor.	No	Yes	No
File name completion	The ability to automatically finish typing file names in command lines.	No	Yes	Yes
`alias` command	A feature allowing users to rename commands, automatically include command options, or abbreviate long command lines.	No	Yes	Yes
Restricted shells	A security feature providing a controlled environment with limited capabilities.	Yes	Yes	No
Job control	Tools for tracking and accessing processes that run in the background. *See Shells: User's Guide*	No	Yes	Yes

Changing Shells

Determining Your Login Shell

Your system may already be configured with the shell you want to use. You can display the file name of the shell you entered when you logged in by typing:

```
$ echo $SHELL
```

The `echo` command displays the contents or value of a variable named `SHELL`. The `SHELL` variable contains the name of the file that contains the shell program that you are running. The system responds to your `echo $SHELL` command with something like the following:

```
/bin/posix/sh
$
```

In this case it is `/bin/posix/sh`, the file that contains the code for the POSIX Shell. Table 1-2 lists both the file name that displays for each shell and the default system prompt.

Table 1-2. Shell File Names and Default Prompts

Shell	File Name	Prompt
Bourne	`/bin/sh`	`$`
POSIX	`/bin/posix/sh`	`$`
Korn	`/bin/ksh`	`$`
Key	`/usr/bin/keysh`	`$`
C	`/bin/csh`	`%`
Restricted Bourne	`/bin/rsh`	`$`
Restricted Korn	`/bin/rksh`	`$`

Temporarily Changing Your Shell

Unless you are in a restricted shell, you can temporarily change your shell by using this command:

shell_name

where *shell_name* is the name of the shell (for example, `/bin/sh`, or `/bin/ksh`). Temporarily changing your shell lets you experiment in other shells. By typing the name of the shell you want to run, you *invoke* (enter) that shell, and the correct prompt is displayed. After experimenting in the new shell, return to your original shell by typing either `exit` or pressing CTRL-D.

The following example begins in the Bourne Shell, enters the Korn Shell, and returns to the Bourne Shell:

```
$ /bin/ksh                          Enter Korn Shell.
$ ps                                Execute the ps command.
   PID TTY      TIME COMMAND
  6009 tty01    0:00 ksh            Notice that both the Korn Shell and
  5784 tty01    0:00 sh             Bourne Shell processes are running.
  6010 tty01    0:00 ps
$ exit                              Exit Korn Shell.
$ _                                 Bourne Shell returns.
```

Permanently Changing Your Shell

To permanently change your *login shell* (the default shell you get when you log in), use the `chsh` (ch*ange* sh*ell*) command:

`chsh` *username full_shell_name*

where *username* is your user name and *shell_path_name* is the full path name (e.g., `/bin/posix/sh`) of the shell you want as your default. Table 1-2 contains the full path names for each of the shells. After you use the `chsh` command, you must log out and log in again for the change to take effect. For example, if `terry` changes the default login shell to the Korn Shell, the command reads:

```
$ chsh terry /bin/ksh
$ _
```

Part I
Bourne Shell

- The Bourne Shell
- Shell Commands
- Shell Scripts
- Basic Shell Programming
- Advanced Programming
- Programming Tips
- Detailed Reference
- Bourne Glossary

2

The Bourne Shell

The Bourne Shell is a "command interpreter"; it takes your commands and interprets them to the system. This tutorial will help you learn to *program* the shell to make your daily work easier.

For example, if you have to execute a series of commands every day, you may get tired of typing the commands each time. By programming the shell, you can create a **shell script**, a file containing all of the commands that need to be executed each day. To execute the commands, you only need execute the shell script.

This tutorial will discuss several concepts which are related to programming. If you are familiar with a programming language (such as C, Pascal, or BASIC) you should have no difficulty understanding the concepts in this tutorial. If you have never programmed before, you may wish to read about concepts such as **loops**, which are used to repeat a specific sequence of commands more than once, **condition statements**, which are used to select, based on existing values in the shell environment, which command(s) to execute, and **variables**, which are used to store alphanumeric values that may be used in more than one place and/or that may change over time. Computer literacy books and beginning programming books discuss these concepts.

UNIX System Structure

HP-UX is a fully compatible, enhanced version of UNIX™ System V. The structure of the system consists of several parts which work together to bring you the HP-UX operating system.

The **kernel** is the core of the HP-UX operating system. It controls the computer's resources and allots time to different users and tasks. The kernel keeps track of the programs being run and is in charge of starting each user on the system. However, the kernel does not interact with the user to interpret the commands. The **shell** is a program that the kernel runs for each user which sets up commands for execution. By having several shells and one kernel, HP-UX is able to support many users at the same time (the user's requests are not actually processed at the same time, but the kernel schedules processing time in a way which simulates concurrent processing). By having the kernel in control, it is also possible for one user to run several shells. The kernel remains in control of all shells and programs.

When you log on to the system, the kernel checks if your login identifier and password are correct. It then runs a shell program for you to interact with it (you never see this, only the shell after successful login). Most systems will start the POSIX Shell (`/bin/posix/sh`) as a default, but it is possible to run the Bourne Shell (`/bin/sh`), the C Shell (`/bin/csh`), or the Korn Shell (`/bin/ksh`). instead.

To give you an idea of processes and how the kernel schedules them, let's look at the `ps` command which lists the processes the kernel is currently coordinating. Type:

```
ps -ef
```

and receive a list similar to the following:

```
  UID   PID  PPID  C   STIME TTY  TIME COMMAND
davek 28125 28124  0 08:50:56 12  0:02 ps -ef
davek 28124 28091  0 08:50:55 12  0:00 sh -c ps -ef > temp
davek 28091 22022  0 08:23:17 12  0:51 vi programming
 root 27781     1  0 06:47:58 co  0:01 /etc/getty console H 0
 root 27097     1  0 23:51:47 05  0:03 /etc/getty tty05 H 0
 root 27092     1  0 23:50:37 04  0:02 /etc/getty tty04 H 0
 root 25740     1  0 11:59:58 03  0:01 /etc/getty tty03 H 0
```

```
  root 24970     1  0 Aug  3    ?  0:01 /etc/getty tty99 3 240
  root 22026     1  0 Aug  2   15  0:01 /etc/getty tty15 H 0
  root 22024     1  0 Aug  2   14  0:02 /etc/getty tty14 H 0
  root 22023     1  0 Aug  2   13  0:01 /etc/getty tty13 H 0
 davek 22022     1  0 Aug  2   12  0:08 -sh
```

The `UID` column refers to the user identifier (the person who executed this process). `PID` refers to the process identifier. There are several commands which use the `PID`, such as `kill`. For example,

```
kill -9 28125
```

will kill (terminate) process 28125 (the first entry in the above list).

`PPID` is the process identifier of the parent process (the process that calls this process). The first row shows 28124 as the parent process. Look in the `PID` column for 28124 to see what the parent process is (shown on the second row).

The `C` column shows processor utilization for scheduling. `STIME` is the starting time of the process. `TTY` is the controlling terminal for the process. `TIME` is the cumulative execution time for the process, and `COMMAND` is the command name. For more details on the `ps` command, see the *HP-UX Reference*.

Before we begin the discussion on the Bourne Shell, let us first define some terms.

Definitions

The following are some definitions which will be used in this tutorial.

filename	The name of a file.
command_list	Either a line containing a command or several commands in a pipe, or several lines containing commands (pipes will be discussed later).
`[]`	Brackets used in a command `syntax` indicate the items enclosed are optional.
word	A command name.
string	A string of characters.

Conventions

This tutorial contains several different types of fonts:

`computer`	Computer fonts are used in screen printouts, for actual command names and file names, and in examples to show anything you are to type (i.e., `varname=penguin` means you type the entire string).
underline	Underline designates actual user response in a computer dialogue.
italic	If you see italic font in command examples, it refers to something you need to substitute for the italicized word(s) (i.e., `varname=`*variable_name* means you actually type `varname=`, but you have to substitute a variable name in place of *variable_name*). Italics also indicate text being emphasized, text to which you should pay particular attention.
bold	Bold font indicates a term used for the first time in the chapter.

3

Shell Commands

This chapter will discuss methods for combining shell commands. You should already be familiar with executing single commands, like running the `date` command. In addition to simply typing a command and pressing Return, you have the ability to include *options* and *parameters* to the command.

Options to a command can be found in the *HP-UX Reference* under the description of the command. These options are usually preceded by a dash (`-`) and are separated from the command name, other options, and parameters by blanks. Parameters, or variables, are data the command needs to function properly. If you omit parameters from the `ls` command, the current directory is listed. But if you include a directory name (or path name) as a parameter, a listing of that directory is printed. Command syntax usually takes the following form:

command [*options*] [*parameters*]

Sequential Processing

When you enter commands line by line (pressing Return after each command), you are telling the system to complete the command (or program) before executing the next command. Executing:

```
date
ps -ef
who
```

will complete each command before going on to the next. You can place all of the commands on the same line by using the ";" separator. For example,

```
date; ps -ef; who
```

is equivalent to entering each command on a separate line. This process is called **sequential processing**. New programs or commands cannot be started until the preceding program or command has completed.

If parameters are required by the program, they are entered as usual. The semicolon is placed after the last parameter.

While a program is running as a sequential process, there is no response to keyboard activity until after the program has completed (other than the keyboard buffer delay).

Programs already in progress when a program with sequential processing is executed continue to run as usual. While a program is running as a sequential process, you have the option of waiting for the program to finish.

Nonsequential (Background) Processing

Programs can also be run *nonsequentially*, in which case, each program runs without waiting for the previous program to complete. This type of execution is more commonly called "running in the background." Follow the program name with `&` to specify background processing.

```
program1 & program2 & program3 &
```

This example runs `program1`, `program2`, and `program3`, in the background, and returns a prompt to the user immediately.

Note that programs that write to the terminal or require input are poor choices for background execution, since the output will be intermixed on the screen, or the input may not be read by the correct program.

Redirecting Input and Output

Every program has at least three data *paths* associated with it: standard input, standard output, and standard error output. Programs use these data paths to interact with you. By default, standard input (`stdin`) is your keyboard. The default destination for both standard output (`stdout`) and standard error (`stderr`) is your screen.

Redirecting input and output is a convenient way of selecting what files or devices a program uses. The output of a program that is normally displayed on the screen can be sent to a printer or to a file. Redirection does not affect the

functioning of the program because the destination of output from the program is changed at the *system* level. The program is unaware of the change.

I/O redirection enables you to change a specific data path of a program while leaving its other data paths unchanged. For example, `stdout` can be stored in a file instead of written to your screen.

How to Redirect Input and Output

I/O redirection symbols are entered on the shell command line, or from a shell program. The program begins executing with the data paths specified by the redirection symbols. To specify I/O redirection for a program, each file name is preceded by a redirection symbol, as in:

`programA <` *file_name*
`programB >` *file_name*

Spaces between the redirection symbols and the file names are optional. The symbol identifies the name that follows it as a file for input or output. The redirection symbols are listed in Table 3-1.

Table 3-1. Redirection Symbols

Symbol	Function	Example
<	Read *standard input* from an existing file.	`program1 <input.data`
>	Write *standard output* to a file.	`program2 >output.data`
>>	Append *standard output* to an existing file.	`sample.prog >>output.data`

Note Using > destroys any previous contents of the file specified to receive the output. If a file's contents must be preserved, use >>.

Note Be careful not to use the same file for standard input and standard output. When input and output operations access the same file, the results are unpredictable.

If a file you specify with a redirection symbol is not in the current directory, you should use a path name to identify it. The following actions are taken when the system does not locate files named with the redirection symbols:

- If a file specified for input with the < symbol is not located, an error message is displayed.
- If a file specified for output with the > or >> symbol is not located, it is created and used for program output.

Examples

The following examples show how the data paths of programs, commands, or utilities can be modified with the redirection symbols.

```
CHItest < data1
```

Runs the program `CHItest` using the file *data1* as input.

```
date >> syslog
```

Adds the current time and date to the end of the file *syslog*.

Pipes

Two or more programs or commands can be connected so the output of one program is used as the input of another program. The data path that joins the programs is called a **pipe**. Pipes allow you to redirect program input and output without the use of temporary files.

When programs are connected with pipes, the shell coordinates the input and output between the programs. The pipes only transfer data in one direction, from the standard output of one program to the standard input of another program.

How to Connect Programs With Pipes

The vertical bar (|) is the "pipe" symbol. Parameters for the program are listed after the program name, but before the | symbol. Spacing between the program names and vertical bars is optional. The syntax used for connecting programs with pipes is as follows:

program_a | *program_b* | *program_c*

where *word* is a command or executable program. Pipes operate on or transform data by separate programs in stages. For example, *word_a* might require input that you type from the keyboard. *word_a* could collect this data and then direct it to `stdout`. This output would be passed through the first pipe to become the input to *word_b*. *word_b* might check that data for validity and process it in some way, perhaps sort it. The processed data would then go to `stdout` and be passed through the second pipe to become the input to *word_c*. *word_c* might format that input into a report.

Here are some examples.

To print the number of files in the current directory, type:

```
ls | wc
```

To print a listing of each file in the directory, and paginate it for convenient screen viewing, type:

```
ls | more
```

To send the contents of `file` to `pr`, which formats the data and then passes it to `lp` for printing on the line printer, type:

```
cat file | pr | lp
```

Redirection in Pipes

The redirection symbols can be used for programs connected with pipes. However, only the data paths not connected with pipes can be changed. If you specify a change to a data path being used with a pipe, then an error occurs. The following changes are permitted:

- The standard input of the first program using a pipe can be redirected with the `<` symbol.
- The standard output of the last program using a pipe can be redirected by using the `>` symbol or appended to an existing file with the `>>` symbol.

Examples

The following commands show how programs can be connected with pipes and how additional changes can be made to data paths with redirection symbols.

The first example takes the standard output from `test_prog1` and uses it as standard input to `/usr/output_prog`.

```
test_prog1 | /usr/output_prog
```

The next example runs four programs connected with pipes and puts the output of the fourth program in `store_file`.

```
get_it | check_it | process_it | format_it > store_file
```

Pipe Example

The following pipe uses several of the symbols we just discussed. Try to figure out what will happen before you read the description below.

```
sort pdir; (( pr pdir | lpr )& (sort local)& ); cat local >>pdir
```

This pipeline will run three sets of commands sequentially. The first command is to sort the *pdir* file. When it is completed, the second command set is executed. The parentheses separate the commands so the shell knows which command to associate with a symbol (for more on command grouping, see "Command Grouping" in Chapter 8). Therefore, the two commands (`pr` and `sort`) are run nonsequentially. So, at the same time, the *pdir* file is formatted and sent to the printer, and the *local* file is sorted. Finally, the `cat` command is run which appends the *local* file to the *pdir* file.

File Name Generation

A helpful way to reduce typing is to use patterns to match file names. If you are in a directory with a file "`programming`" you can see a listing with either:

```
ls programming
```

or you can use a pattern to match:

```
ls p*
```

where "`*`" will match any character or string of characters. If you have another file beginning with "`p`", it too will be listed. Table 3-2 shows the file generation symbols you can use:

Table 3-2. File Generation Symbols

Symbol	Description
*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any one of the characters enclosed in the brackets. A pair of characters separated by a minus will match any character between the pair (lexically).

```
[a-z]?cubit*.[ca]
```

will match a file which begins with any character `a` through `z` (lower case), followed by any single character, followed by the string "`cubit`", followed by any number of characters, and which ends in "`.c`" or "`.a`".

4

Shell Scripts

Introduction to Shell Scripts

Simple Scripts

Stringing commands together on a line with sequential processing, background processing or pipes is an extremely useful tool for a limited number of commands. To save typing the commands repetitively, in the case where you use the same sequence of commands often, you can place the command line(s) into a file. This file is called a *shell script*. You create a file containing the commands, tell the system you want the file to be executable (so it can be run as a program), and then type the name of the file to execute the commands in the shell script.

A simple shell script could contain the following command line:

```
date; who; ps -ef; du /users
```

which executes each command only when the previous command has completed. To create the script, enter an editor (`vi` for example) and type the above command line. Save the file.

To run the script, you have two methods: the `sh` command, or changing the permissions on the file. The `sh` command will create a new shell to run the script. As mentioned in the beginning of this tutorial, it is possible to have several shells running at the same time (with the kernel in control). The `sh` command creates a new shell to execute the file you specify (if you don't specify a file, it creates a new shell similar to the one you are already in). To execute the script with the `sh` command, type:

`sh` *scriptname*

Where *scriptname* is the name of the file you placed the command line in.

The common way to run a script or program, however, is to declare the file executable with the `chmod` command. `chmod` is used to alter the permissions on a file. For our purposes, we will declare the file to be executable by everyone on the system, but only you can update the file. Type:

```
chmod +x scriptname
```

Now the file is executable, and you only need enter the file name to run the script (simply type the *scriptname* as if it was a command). Your script will execute, and you will see a large output. Both methods of executing *scriptname* have the same net effect, they just behave differently at first. For details on the `chmod` command, see the *HP-UX Reference*.

Scripts With More Than One Line

The example above just uses one command line for the script. You can, however, make the script easier to read and contain more than one line of commands. Each line of commands is executed in *sequential* order (the previous line must complete before the next line is executed). So, we can take the previous example:

```
date; who; ps -ef; du /users
```

and spread the command line into four lines which accomplish the same thing:

```
date
who
ps -ef
du /users
```

When this script is executed, you get the same results as before.

Echo and Redirection in Scripts

If you have a large output from a script like in the above example, you may wish to place some headers or comments in the output and place the output into a file. The `echo` command will print titles or comments for you. It works in the following manner:

```
echo "string"
```

where *string* is a string of characters.

Modify your example script to look like:

```
echo "Current date and time: \c"
date
echo "Users logged in:\n"
who
echo "\nCurrent processes:"
ps -ef
echo "\nUser disk usage:"
du /users
```

where "`\c`" causes the next line of output to be printed on the same line, and "`\n`" causes an extra carriage return and line feed (for more detail see "Echo" in Chapter 5).

Next you can execute the file using the redirection symbols to append the output to another file. For example, let's say our file is called `status1`, and the file we wish to place the output in is called `status_file`:

```
status1 >> status_file
```

Each time you monitor the system, you can have the output added to a file.

The .profile File

The Bourne Shell runs a script automatically when you login, called `.profile`. This script sets the "environment" in which you work: it sets up certain variables which tell the system where to look for a command, what the prompt should look like, where to get the mail, and other variables. The `.profile` file is usually set up by the system administrator, but you can customize it as you learn shell programming techniques. Here is a sample `.profile` file:

```
PATH=/docs/tools:/bin:/usr/bin:/usr/contrib/bin:/users/hpux/davek:.
PATH=$PATH:/usr/local/bin:/users/hpux/davek/bin:/d1/usr/informix/bin:
PATH=$PATH:/d1/usr/informix/lib:/d1/usr/informix
MAIL=/usr/mail/$LOGNAME
TERM=2623
export TERM PATH MAIL HOME
stty kill '^c'
stty sane
tabs -T$TERM
if mail -e
then
 echo
```

```
    echo "You have mail."
    echo
  fi
```

The script sets some essential definitions for shell variables and makes them global to the system. For example, the `PATH` variable sets up a search path for commands. When you execute a command in the shell, it looks at the `PATH` parameter. The `PATH` parameter gives the shell several directories in which to look for the command. If you execute a program that is not in one of the directories specified by `PATH`, you will receive an error message.

Let's go line by line and describe the entries in this sample `.profile` file:

- `PATH` sets up the search path for the shell. Each directory in the path is separated with a colon (`:`). Once the shell has read this `.profile`, any command you execute is searched for first in the `/docs/tools` directory, then the `/bin` directory, and so on. Notice the last entry in the first line is a dot (`.`). This indicates the current directory at the time you execute the command, whatever that happens to be.

  The second and third line are continuations of the `PATH` parameter. To add to the path, you set the variable `PATH` to its previous value (`$PATH`) followed by a colon, then continue listing the directories.

  As you learn more about shell programming and develop several programs, you may wish to call these programs from any directory. One way to do this is to create a "library", a directory which contains all of these shell programs. Then place the path to the library into the `PATH` variable. This directory will always be searched when you type the program name.

- `MAIL` sets the file in which to look for new mail.

- `TERM` sets the terminal type. This example is using an HP 2623 Graphics computer terminal.

- The `export` command marks parameters for exporting their values to the environment. The `export` command can be thought of as a way of letting other commands know the value of a variable. If you do not export a parameter, other processes will not know its value.

- The `stty` command sets characteristics for your terminal. Setting the `kill` characteristic to `^c` (control `c`) tells the computer to interrupt the current process when control `c` is pressed.

- `stty sane` resets all modes to some predefined reasonable values.
- `tabs` will set the tabs to the default format for your terminal. The `-T` option is followed by the terminal type (here it is `$TERM` which is a parameter we set earlier to `2623`).
- The last six lines construct a *condition* (we will learn the details of conditions later). These lines check if you have received any mail. If you have, the message "`You have mail.`" will appear on the screen.

Customizing .profile

If you wish to customize your `.profile` script, you can add any of the items discussed in the shell programming sections. The following are some system parameters and commands you can add to your `.profile` script which may be of interest:

- `PS1` is a system parameter which sets the value of the system prompt. The default is `$`, but you can change that to anything by using the following format:

  `PS1="`*string*`"`

  where *string* is any character string.
- To have the script clear the screen, include a line with the `clear` command on it.
- To have anything printed on the screen, include a line with the `echo` command:

  `echo "`*string*`"`

  where *string* is what you want to appear on the screen.

Here is a list of some system parameters:

Table 4-1. Shell Parameters

Parameter	Description
HOME	The default directory for the `cd` command.
PATH	The search path for commands.
CDPATH	The search path for the `cd` command.
MAIL	If this parameter is set to the name of a mail file, and the `MAILPATH` parameter is not set, the shell tells you when mail arrives.
MAILCHECK	This parameter tells how often (in seconds) the shell will check for mail. The default is 600 seconds. If set to 0, the shell will check before each prompt.
MAILPATH	The search path for mail files. The shell informs the user when mail arrives.
PS1	Primary system prompt. The default is "`$`".
PS2	Secondary system prompt. The default is "`>`".
IFS	"Internal Field Separators" which are normally *space*, and *tab*.
SHACCT	Write an accounting record in the writable file set by this parameter.
SHELL	If an 'r' is contained in the basename (last entry in a path), the shell becomes a restricted shell.

Note See "The .profile File" section earlier in this chapter for an example `.profile` file.

5

Basic Shell Programming

Once you have mastered simple shell scripts, you can do more complex shell programming. This chapter introduces ways to pass information to a shell program, how to execute commands conditionally, and how to get data from the keyboard during the execution of a shell program.

All of the constructs of shell programming can be executed in two ways: you can type the commands into a file so they will all be executed when the file name is entered (after changing the permission), or you can enter the commands directly into the shell (just as you enter commands like "`date`").

When you enter shell constructs directly into the shell, you can either type them on the same line (and press return to execute them), or you can type them over several lines. For example, we can type the following construct two ways (don't worry what the construct actually does, just how it can be typed). First on one line:

```
if test -d /d1; then echo "/d1 is a directory"; fi
```

Then on several lines:

```
if test -d /d1
then
  echo "/d1 is a directory"
fi
```

Typing the command on one line is simple to do in the shell. If you type the command on several lines, you will receive a *secondary* prompt (which you can define in the `PS2` variable). The secondary prompt is usually a "`>`". So, if you were to type the above command on several lines, the screen would look like:

```
$ if test -d /d1
> then
>   echo "/d1 is a directory"
```

```
> fi
/d1 is a directory
$
where "$" is the system prompt.
```

What is more, you can create shells from programs such as `notes`, `mail` and most editors (such as `vi`), and execute shell commands from these shells. Running a shell from another program is usually called "forking" a shell. It may be useful if you are writing a program and wish to test the program: you can edit the program in `vi`, fork a shell from `vi` (by typing "`:sh`"), execute the program to see if it works, exit the new shell (by typing CTRL-d), and be right back in the editor to make any changes.

Parameters

In addition to shell parameters, you can create parameters of your own. The format for user-created parameters is:

parameter `=` *value*

Note that there must be no blanks between the *parameter*, equal sign (`=`), and the *value*. You can create these parameters while you are in the shell, and they will help you save typing. Look at an example:

```
x=phantom
```

When you type in the above statement, the variable `x` is created and the value "`phantom`" is assigned. To access the variable `x`, you will need to precede the variable name with a dollar sign (`$`). Try this:

```
echo $x
```

The `echo` command writes the value of `x` on the screen. One possible use of parameters is to assign a long pathname to a variable so you do not have to type the whole pathname each time you wish to use it. For example:

```
dir1=/users/hpux/davek/projects/shellp
```

So, to list the contents of this directory, type:

```
ls $dir1
```

Using Parameters in Shell Programs

You can use parameters within your shell programs in the same way. On one line, define the variable with the same format. When you wish to refer to the value of the parameter, you precede the parameter name with a dollar sign (`$`).

One advantage of using parameters in a program is that you can combine them with each other, or with file names, to create a variety of file paths with less typing. Let's say you define a parameter to be the path to a directory:

```
dir2=/users/hpux/dave/projects/memos
```

If you want to print the contents of a file in the above directory, you would use the `cat` command as follows:

```
cat ${dir2}/junememo
```

where the braces differentiate between the parameter and the characters following it and `junememo` is the name of a file (note we had to include a slash before the filename or "`junememo`" would have been concatenated directly to "`memos`" and we would have received an error message). What has happened is called **parameter substitution** and will be discussed next.

Parameter Substitution

When you wish to include the value of a parameter into a string or statement, you must precede the parameter with a dollar sign (`$`). Also, the following conventions hold:

`${`*parameter*`}`	The value of the parameter in the brackets is substituted. Use the brackets `{}` when the parameter is followed by a letter, digit, or underscore which is not part of the parameter. Example: `${dir1}123_file` will substitute the value for `dir1` and append the characters `123_file`.

`${`*parameter*`:-`*word*`}`	If the parameter is set and non-null, the value will be substituted. Otherwise, the *word* will be substituted. Example: `${dir1:-/usr/bin}`. If `dir1` is null, then `/usr/bin` will be substituted.
`${`*parameter*`:=`*word*`}`	If the value of *parameter* is not set or null, then set the value to *word* and substitute that value. Example: `${dir1:=/usr/bin}`. If `dir1` is null, its new value is `/usr/bin`.
`${`*parameter*`:?`*word*`}`	Does the same as `:-` except the shell program will be exited if the parameter is null. If *word* is left off, the message "`parameter null or not set`" is printed. Example: `${dir1:?/usr/bin}` will perform the substitution with `/usr/bin` if `dir1` is null, and then exit the shell.
`${`*parameter*`:+`*word*`}`	If the parameter is set and is non-null, then substitute *word*. Otherwise, substitute nothing.

Positional Parameters

When you execute a shell program, you can include parameters on the command line. When you do, each parameter must be separated with a blank, like:

```
scopy file1 file2 file3
```

where `scopy` is a shell program with three parameters.

When the shell program runs, you can access the value of these parameters (each separated by a blank) with *positional* parameters named `$0`, `$1`, `$2` ... `$9`. If your list of values exceeds nine parameters, the values are placed in a buffer, and you can access the values with the `shift` command (discussed later in this chapter).

```
scopy personnel fileA
```

has positional parameters `$1` equal to "`personnel`" and `$2` equal to "`fileA`". The positional parameter `$0` is always the command name, "`scopy`" in the example above.

If you need to know the number of positional parameters (let's say you wish to see if the user included any parameters at all) you use `$#`.

If you need a parameter which contains all of the positional parameters separated by blanks, use `$*` (this is useful if the positional parameters constitute a sentence or even a command line).

Positional parameters are accessed within the body of the script. When the script is executed, the parameters are assigned values only for the execution of the script. To make the parameters retain their values in the current shell, see "The . (dot) Command" in Chapter 6.

Here is an example script using positional parameters:

```
echo "Searching for $1 in $2"
grep $1 $2
echo "Done"
```

This shell program expects two positional parameters. The first parameter is a string, and the second is a file. The `grep` command searches the file for each occurrence of the string. Here is an example of how we would type in the shell program for execution (let's call the program "`search`"):

```
search "widget 20809" /users/dave/datafile
```

The `$1` parameter is "`widget 20809`" and the `$2` parameter is "`/users/dave/datafile`". Notice the quotes around the first parameter. If you need to include a blank in a positional parameter, you have to quote the expression. Quoting is discussed later in this chapter.

Shift

In the last section we learned how to access the positional parameters by using the numbers `$1` through `$9`. However, if we access these by name, we must already know what to expect. In other words, we cannot have the positional parameters in an arbitrary order nor more than nine.

The `shift` operation helps alleviate problems with positional parameters. Let's think of the positional parameters as a stack, with `$1` at the bottom and `$9` at the top (if there are more than nine parameters, the remainder would be stacked on top).

`Shift` will remove the value of `$1` and replace it with the value of `$2`, move the value of `$3` to `$2`, and so on. It is like removing the bottom entry of the stack and letting the values fall one position down. Let's look at a graphic representation of this idea:

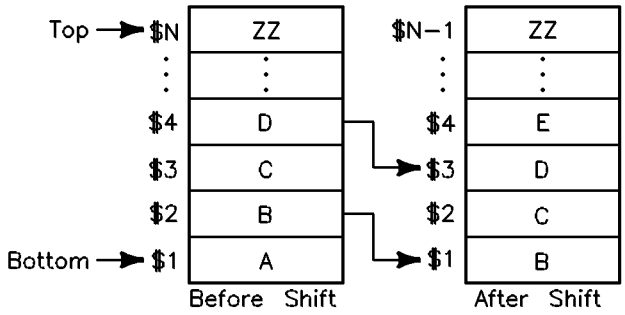

Figure 5-1. Shifting Positional Parameters

You can use shift in loops, which we will discuss next, or you can use it sequentially like in the following example named "`list`":

```
if [ $1 = yes ]
then
  shift
  cat $1
  exit
else
  shift
  echo "file called $1 was rejected"
fi
```

If the first positional parameter (`$1`) is equal to "`yes`", then the contents of the filename (the second positional parameter) will be listed. The first time `$1` is used for the test, it may have the value "`yes`". After the shift, the value that was `$2` is shifted to `$1`, and `$1` would be the file name. To execute this script, you would type "`list yes filename`", or even "`list no file2`".

Echo

We have already mentioned the `echo` command as a method to display text on the screen. The `echo` command can be used to prompt the user for input (see `read`), or to indicate something has been done. You can also use parameter substitution in the `echo` command.

One helpful item for the `echo` command is the `\c` (backslash "`c`"). If you add `\c` to the end of an `echo` statement, the default linefeed is suppressed. This means you can prompt the user to input on the same line (see the example at the end of this section). Another helpful item is `\n` which adds an extra new line. For further information on the `echo` command, see the *echo*(1) entry in the *HP-UX Reference*.

Quoting

Since the shell is full of special characters (with special meanings), we need a way to suppress the meaning of a special character. If we have a string which contains a special character we may not want it treated as such.

If you were to assign a string of characters to a parameter, and the characters contained blanks and characters with a dual meaning (blanks in this case would indicate the end of the parameter assignment), you may receive an error message. When you quote a character or string of characters with single quotes ('), you suppress any special meaning. Other quote marks have different effects.

The Backslash

The backslash (\) will cancel, or **escape**, the special meaning of the next character:

```
echo \$dir1
```

will echo "`$dir1`" instead of the parameter value of "`dir1`" because the dollar sign is told to have no special meaning. In this example, the `$` has been *escaped*. Note that there's no harm done in escaping non-special characters.

The Double Quote

The double quote (") quotes anything enclosed in two double quotes except \ $ " ' and ‘ (grave accent). For example:

```
echo "$dir1 is an \"old directory\""
```

The dollar sign interprets `dir1` as a parameter; the backslash (\) ignores the following double quote (in other words, it does not end the echo string but includes the double quote as part of it).

The Single Quote

The single quote (') will quote everything enclosed in two single quotes except the single quote itself. So the above example could be represented as:

```
echo $dir1' is an "old directory"'
```

Notice where the single quotes begin. If we place `$dir1` inside the single quotes, the value of `dir1` will not be printed, rather the exact characters `$dir1` since the dollar sign would be ignored as a special character.

Note If you leave off a quote when entering commands in the shell, you will receive a secondary prompt (usually a "`>`"). This just means you need to type in the closing quote.

Command Substitution

The grave accent (‘) indicates a command substitution.

Note Pay particular attention to the difference between the grave accent (‘) and the single quote ('). The single quote is usually located below the double quote (") on the 46020A and 46021A keyboards, and the grave accent is under the tilde (~).

Command substitution means you can substitute a shell command's output into a string like the `echo` string. The command is a shell command and must be enclosed between grave accents.

The following example shows a command substitution in an `echo` command, with the output from the command substitution appearing on the same line as the echoed words:

```
echo  "The current date and time is ‘date‘."
```

This command will print something similar to the following:

```
The current date and time is Wed May 30 15:24:35 MDT 1990.
```

The following example shows a command substitution in an `echo` command with the output from the command substitution appearing on lines following the line with the echoed words:

```
echo  "People currently on the system:\n\n ‘who‘"
```

This command will print something similar to the following:

```
People currently on the system:

billa      console      May 30 09:45
stu        tty02        May 30 12:02
jth        tty03        May 30 07:24
michael    tty04        May 30 12:31
clarke     tty07        May 30 08:15
richard    pty/ttyp0    May 30 12:19
```

If you need to quote characters within grave accents, make sure you use a different quote character than the enclosing quote. In the following, we use double quotes to enclose the entire string, and single quotes within the grave accents:

```
echo "The banner command,\n ‘banner ’ banner’‘"
```

The result of this command will generate the following:

```
The banner command,

  #####     ##    #    #  #    #  ######  #####
  #    #   #  #   ##   #  ##   #  #       #    #
  #####   #    #  # #  #  # #  #  #####   #    #
  #    #  ######  #  # #  #  # #  #       #####
  #    #  #    #  #   ##  #   ##  #       #   #
  #####   #    #  #    #  #    #  ######  #    #
```

Be sure to try these commands yourself.

Conditions: The if Statement

Your shell programs may need to execute a command or set of commands only if a certain condition exists. Let's say you want to "execute the sort command only if the file exists, otherwise print an error message". Your statement would look like:

```
if test -f $1
then
  sort $1
else
  echo "file does not exist"
fi
```

where `$1` is a filename passed in from the command line. The `if` statement checks the status of the command following it (in the above case, the test command follows). The `else` statement is executed if the command in the `if` statement fails. For the case of "if this then that, else if this then that, etc" we can use the `elif` statement which means "else if".

The format for the `if` construct looks like:

```
if command_list1
then command_list2
elif command_list3
then command_list4
   .
   .
```

```
    .
else command_listn
fi
```

It is helpful to indent to indicate parts of the `if` construct. *Make sure you end the construct with* `fi`.

Let's look at an example to better clarify this construct:

```
if grep jones personnel
then
  echo "jones" >> available
elif grep castle personnel
  then
    echo "castle" >> available
  else
    echo "empty" >> available
fi
```

This construct will attempt the first command list: `grep jones personnel`. If the string `jones` is found in the `personnel` file then the command `echo "jones" >> available` will be executed. If the search for `jones` fails, we go to the `elif` statement and try the `grep "castle" personnel` command. If this is successful, the command `echo "castle" >> available` will be executed. If this `grep` command is unsuccessful, we go to the `else` statement and execute the "`echo "empty" >> available`" command.

Test

An often used command is the `test` command. You can use the `test` command in the `if` construct to test conditions such as equality. There are many options we will not mention here, so you may wish to refer to the *test*(1) entry in the *HP-UX Reference*.

Here are two examples to explain the use of the `test` command:

```
dir1=/usr/bin
if test $dir1 = /usr/bin
then
  echo "directory found"
fi
```

This construct "tests" if the value for `dir1` (notice how we used parameter substitution) is equal to the string "`/usr/bin`".

```
if test $# -eq 0
then
  echo "no positional parameters"
fi
```

The `-eq` option is used to test the numeric equivalence of the `$#` and the value zero. Remember `$#` is the number of positional parameters passed to the script.

To make typing easier, you can use an abbreviation for `test`. The square brackets enclosing the options and parameters do the same as the `test` command. For example:

```
if [ $# -eq 0 ]
```

Note: This is the most common usage.

has the same meaning as the first line in the above example (`if test $# -eq 0`).

Note Be sure to separate the square brackets from any characters with a blank. If you do not, the brackets will be assumed to be part of the options.

Testing files is another use for `test`. This command, in addition to its many other capabilities, enables you to check if a file is a directory, is readable, or is writable.

Read

If you wish to receive input during the execution of a shell program, you can use the `read` statement with the following format:

`read` [*parameter...*]

where [*parameter...*] means a list of one or more parameters. When the computer executes this statement, it gets input from the keyboard (unless you use redirection symbols to get input from a file). Each word (words are separated by blanks) typed in is assigned to the respective parameter in the

list, with the leftover words assigned to the last parameter. To see how this is used, see the example at the end of this chapter.

Exit

Each command returns a status when it terminates. If it is unsuccessful, it returns a code which tells the shell to print an error message. You can use the `exit` command to leave a shell program with a certain exit status (see below for a table of the codes).

The default `exit` (no arguments) will exit the shell program with the status of the last command executed. You can exit with a different exit status; see the *HP-UX Reference* pages for the exit statuses of each command. The usual exit statuses are:

Table 5-1. Exit Status

Value	Description
0	Success.
1	A built-in command failure.
2	A syntax error has occurred.
3	Signal received that is not trapped (see the `trap` command).

For example, the statement

```
exit 0
```

will give the instructions to leave the shell program successfully.

Comments

To add to a shell program comments, simply start the line with a pound sign (`#`). For example:

```
# this line is a comment
```

Or you can add a comment after a statement as long as you precede it with the pound sign.

Note *Do not* start your shell script (your file containing the script) with the pound sign (`#`). If the first character in a script file is "`#`", the system will think the script is a "C" shell (csh) script. You may choose to always start your shell script files with a blank line, or always include a space before you use the pound sign in comments.

Example: Moving Files

The following example uses all of the concepts we just discussed. You should try the example yourself, and then try writing one yourself (to get you started on your own, try writing a shell program that will do the same thing the `cp` (copy) command does, except have it prompt the user for input). The name of the example script is `move`.

Remember to leave the first line blank, or precede any comments with a blank space or tab.

```
#################################################################
# This shell program will prompt the user for moving files. #
#################################################################

#(1) test if there are any arguments
if [ $# -eq 0 ]
then
  echo "No arguments: include file name."
  exit
fi

#(2) ask if file is to be moved to directory or file
echo "Move to directory or new file name (d or f)?\c"
read x

#(3) test if x is a directory; if not, leave script
if [ ${x:?} = d ]
then
  echo "Enter directory name ->\c"
  read dir1
# perform the move
  mv $1 $dir1
  echo "$1 moved to $dir1"
  exit

#(4) test if it is a file
elif [ $x = f ]
then
  echo "Enter new file name ->\c"
  read file2
# perform the move
  mv $1 $file2
  echo "$1 now named $file2"
  exit

#(5) response is not d or f
else
  echo "$x not a correct response."
```

```
    exit
  fi
```

Discussion of Example: Moving Files

The shell program was created in the `vi` editor. When the file was typed, the permission was changed to allow the file to be executed: `chmod +x move`. To execute the command, you would type `move` followed by the name of a file you want moved.

#(1) test if there are any arguments. The first few lines which are preceded with a `#` are comments. Then the next few lines comprise an `if` construct. This construct uses the `test` command indicated by the square brackets, which compares the number of positional parameters to zero. If there are no positional parameters, then an error message is printed and the shell is exited.

#(2) ask if file is to be moved to directory or file. After the next comment, the main body of the program begins. The user is prompted to type a "d" or "f" to indicate whether the file is to be moved to a directory or to another file. The `read` statement accepts input from the keyboard.

#(3) test if x is a directory; if not, leave script. Next, the parameter `x` is tested to see if it is equal to "d". The construct `${x:?}` will exit with an error message (`parameter null or not set`) if the user just hits return. If a "d" is typed, then the user is prompted to enter the name of the directory, and the move command is executed using `$1` (the positional parameter the user typed after the shell program name) and the `$dir1` parameter (the directory the user typed when prompted).

#(4) test if it is a file. If `x` was instead "f", the user is prompted to enter the new filename.

#(5) response is not d or f. If the user typed in neither "d" or "f", then an error message is printed. In all of the above three cases, the `exit` command is used to terminate the shell program. Pay attention to how positional parameters are used, and how you match up `if`'s, `else`'s, `elif`'s and `fi`'s.

6

Advanced Programming

The example at the end of this chapter will be a script created with the information you will have learned. The example is similar to an HP-UX command.

Looping

Many times sequential processing in a program is just not enough. We need a mechanism which will allow us to repeat the same set of commands using a different set of parameter values. To accomplish this in shell programming you can choose between three looping constructs: `for`, `while`, and `until`.

For

The `for` construct allows you to execute a set of commands once for every new value assigned to a parameter. Look at the following format:

```
for parameter [ in wordlist]
do command-list
done
```

where *parameter* is any parameter name, *wordlist* is a set of one or more values to be assigned to *parameter*, and *command-list* is a set of commands to be executed each time the loop is performed. If the wordlist is omitted (and also "`in`"), then the parameter is assigned the value of each positional parameter.

The word list is a versatile quantity in the `for` construct. It can be a list which you specifically type (separated with blanks), or it can be a shell command (using grave accents) which generates a list. Let's look at some examples.

```
for i in `ls`
```

```
do
  cp $i /users/rhonda/$i
  echo "$i copied"
done
```

This example will assign one file at a time from the current directory (the values are generated by the `‘ls‘` command) to the "`i`" parameter. The loop's command list will copy the file to another directory, then report the success of the copy. You can use file name generation (see "File Name Generation" in Chapter 3) to match files. Instead of the first line of the above loop being "`for i in ‘ls‘`", you could use: "`for i in *`".

```
for direc in /dev /usr /users/bin  /lib
do
  num=‘ls $direc | wc -w‘
  echo "$num files in $direc"
done
```

This example lists the values to be given to `direc` in the loop. The command list then lists each respective directory (the parameters) and assigns a word count (`wc`) to the `num` parameter. Then the word count is printed out.

```
for i
do
  sort -d -o ${i}.srt $i
done
```

This final example will assign each positional parameter respectively to "`i`" (since the `in` clause was omitted). If the positional parameters are file names, the script will sort the file and place the result in a file having the same name as the unsorted file with "`.srt`" appended to it. It will then get the next positional parameter until all have been accessed.

(You can also use pattern matching to specify the word list. Pattern matching is discussed in the section entitled "Case," and in the section entitled "Using Shell Expansions.")

While

The `while` construct repeatedly executes a list of commands in the following format:

```
while command-list1
  do command-list2
done
```

All of the commands in *command-list1* are executed. If the last command in the list is successful (indicated by an exit status of `0` from the command), then the commands in *command-list2* are executed. Then we loop back to execute *command-list1* until the last command in the list is unsuccessful, and then the `while` loop terminates.

```
while [ -r "$1" ]
do
  cat $1 >> composite
  shift
done
```

This example tests the positional parameter to see if it exists and is a readable file. If it is, it appends the contents of the file to the `composite` file, shifts the positional parameters (what was `$2` is now `$1`) and tests the new file. When the file is not readable, or there are no more positional parameter values (`$1` is null) the `while` loop is terminated. (Note that without double quotes(") around $1 in the test, the test will respond with an `argument expected` syntax error when $1=NULL.)

Until

The `until` construct is basically the same as the `while` construct except that the commands in the loop are executed until the conditions are true (instead of false like in the `while` loop). Here is the format:

```
until command-list1
do command-list2
done
```

If the last command in *command-list1* is *unsuccessful*, then the commands in *command-list2* are executed. When the last command in *command-list1* is

successful, the `until` loop is terminated. Let's use the same operation in the `while` section to illustrate:

```
until [ ! -r $1 ]
do
  cat $1 >> composite
done
```

Notice the subtle difference with the `while` loop. The `!` negates the test conditions. We execute the loop *until* the condition is true (or successful). The `while` loop executes the commands *while* a condition is true (or successful).

Case

The `case` construct is an expansion of the `if` construct. If you have a condition which may have several possible responses, you can either string together many `if`'s or you can use the `case` construct:

```
case parameter in
pattern1 [ | pattern2...] ) command-list1 ;;
pattern2 [ | pattern3...] ) command-list2 ;;
    .
    .
    .
esac
```

After the first line (which asks if *parameter* matches one of the following conditions) is listed all of the possibilities for *parameter*. Each of these lines contains a *pattern* (or value for *parameter*). The brackets (*[|| pattern2...]*) refer to other values that may be valid. The vertical bar (`|`) represents "or". Finally, the pattern(s) are followed by a close parenthesis `)`, and then by a list of commands to be executed if the patterns match.

An example may better illustrate:

```
case $i in
  -d | -r ) rmdir $dir1
            echo "option -d or -r" ;;
  -o )      echo "option -o" ;;
  -* )      echo "incorrect response";;
esac
```

Here the first positional parameter is compared to several patterns. If `$1` is "`-d`" or "`-r`", then `$dir1` is removed, and a statement printed. If `$1` is "`-o`", a statement is printed. The last pattern uses the `*` metacharacter to "catch" all other option possibilities and flag them as errors. Remember to end each command list with `;;` and to end the entire `case` construct with `esac`.

Note Be aware that the order of patterns in a `case` is important. The *specific* case should precede the *general* case. For instance, if you used `-*)` before `-g)`, there could never be any matches for the `-g)` because the `-*)` would have matched them all, already.

The . (dot) Command

Normally, when you execute a shell program, a subshell is created in which to execute it. Therefore, if you define variables in the program, they are only good for as long as the program is executing (when the program is done, you return to the current shell's environment). If you wish to have the shell program executed in the current shell (and thus make the defined variables good for the current shell's environment), use the "dot" command:

. *scriptname*

Make sure there is a space between the dot (.) and the script name (otherwise the system will assume it is part of the script name). Let's look at an example.

Create a file with the following commands:

```
echo $dog
dog=tired
echo $dog
```

Make the script executable with the chmod command ("`chmod +x dogsample`", where "`dogsample`" is the name of the script). Next, define the variable `dog` to be:

```
dog=rover
```

Run the script (by typing `dogsample`) without the dot command. The results will be:

```
rover
tired
```

Now, check to see what the value of `dog` is:

```
$ echo $dog
rover
$
```

The original value for `dog` appears. This is because the shell was executed in a subshell. Now, try the dot command:

```
$ . dogsample
rover
tired
$
```

and then test the value of `dog`:

```
$ echo $dog
tired
$
```

The value of `dog` was changed because the script was run in the current shell.

The eval Command

The `eval` command reads its arguments as input to the shell, and the resulting commands are executed. The format is:

`eval` [*arg ...*]

where *arg ...* is one or more arguments which are shell commands or shell programs. Here is an example:

```
eval "grep jones $p_file | set | echo $1 $2 $4"
```

`eval` will execute the pipe contained in double quotes in the shell.

If you use the following:

```
s='date &'; $s
```

you would receive an error message from date. The '&' is ignored as a special character (due to the single quotes). So, to make the command function as expected, use `eval`:

```
s='eval date &'; $s
```

and the `eval` will reparse the string and thus attach the special meaning to '&'.

Using Shell Expansions

You read about pattern matching in "File Name Generation" in Chapter 3. Here are some examples which will simplify some of the constructs you just learned.

When you generate lists for your `for` constructs (or any other construct where you are trying to generate filenames without needing to type in each file name), you are free to use pattern-matching characters. For example:

```
for i in *.c
do
  mv $i /tmp
done
```

Here we generate a loop in which `i` is set to each filename in the current directory that ends in ".c."

```
case $i in
  ?[dD].c ) echo $i ;;
  *[!nN]  ) mv $i .. ;;
  *       ) exit ;;
esac
```

This case construct will match the value of `i` on the first pattern line if it begins with any single character (`?`), followed by either "`d.c`" or "`D.c`". The second pattern line matches any string (including the null) ending in any letter other than "n" or "N". The last expression matches anything left over.

Helpful Tips

Let's wrap up this section with a couple of helpful items.

- To print a character that will "beep" to alert a user, use CTRL-G in an `echo` command.
- To add control characters to the `vi` editor, you must first type CTRL-V, then type the control string.
- To break from a "`for`" or "`while`" loop, use the `break` statement. If you want to break out of a certain number of levels in a nested loop construct, add `break` *n*, where *n* in the number of levels of nesting. As an example, consider:

```
for i
do
  while true
  do
   ⋮
    break 2
```

This "break 2" gets you beyond the second `done`*. It breaks you out of two enclosing loops (*`for` *and* `while`*).*

```
  done
done
```

- To continue with the next iteration of the enclosing loop, use the `continue` statement. To continue at the next iteration of the nth enclosing loop, use: `continue` *n*.
- For more items, look in the *sh*(1) entry in the *HP-UX Reference*. Some of these features will be discussed in the next section.

Example: Groupcopy

```
bool='n'
query='n'
dir='n'
#####################################################################
# This shell program copies all of the files in the current         #
# directory to the specified directory.                             #
#                                                                   #
# Usage: To copy all files to a specified directory, type the       #
#           directory as the parameter.                             #
#        To be prompted for file copy, type the -q option           #
#           immediately following the gp command, then the          #
#           directory as the second parameter.                      #
#        To include files in subdirectories, use the -d option.     #
#####################################################################

#(1) test to make sure the directory parameter is included
if [ $# -eq 0 ]
then
  echo "gp [-opt] to_directory"
  echo "Usage: include options and a directory name"
  echo "options: -q,  query each file"
  echo "         -d,  include files in subdirectories"
  exit 1
fi
```

```
#(2) look for options
for i
do
  case $i in
    -q) query='y' ;;
    -d) dir='y' ;;
    -*) echo "unknown option; available options are -q, -d"
        exit 1 ;;
  esac
done
newdir=$1

#(3) test if parameter is a directory
if [ -d $1 ]
then
# look to see if parameter is in current directory

for g in *
  do
    if [ $1 = $g ]
    then
      bool='y'
    fi
  done


# if parameter is in current directory,
#   fill in full path name
if [ $bool = y ]
then
  newdir=`pwd`/$1
fi
```

```
#(4) begin main loop
for f in *
do
if [ $f != $1 ]
then
# test if file is a directory or regular file
  if [ ! -d $f ]
  then
# test if query option is used
    if [ $query = y ]
    then
# prompt user to respond 'y' to copy,
#   or anything else to ignore
      echo "copy $f? \c"
      read copy

# test if user wants file copied
      if [ $copy = y ]
      then
        cp $f $newdir
        echo $f copied to $newdir
      fi
    else

# query option not used
      cp $f $newdir
      echo $f copied to $newdir
    fi
  else

# test for -d option
    if [ $dir = y ]
    then
```

```
# test if user wants to copy from subdirectories
      echo "copy subdirectory files from $f?\c"
      read dcpy
      if [ $dcpy = y ]
      then
        if [ $query = y ]
        then
          curdir=‘pwd‘
          cd $f
          gp -q -d $newdir
          cd $curdir
        else
          curdir=‘pwd‘
          cd $f
          gp -d $newdir
          cd $curdir
        fi
      fi
    fi
  fi
fi
done

#(5) parameter is not a directory
else
  echo "$1 is not a directory"
  exit 1
fi
```

Discussion of Example: Groupcopy

Since this is a rather lengthy example, we have provided comments throughout to explain its function. The example is really a new command you can use, and you may find it quite useful. The example, called `gp`, (groupcopy) copies files from one directory into another. This will save you time in typing each file individually as you copy the files.

The file has several options, you include options by typing a `-` (minus) followed by a letter: `-q` will prompt you as each file is about to be copied, and you can choose not to copy it; `-d` will look in subdirectories if that directory has any, and then copy it to the new directory. If no options are included, all files in the directory (not including subdirectories) are copied to the new directory. The format for the command is:

`gp` [*options*] *directory*

where *options* are those described above, and *directory* is the directory to where you want the files copied. The program looks in the current directory for files to copy.

#(1) test to make sure the directory parameter is included. The first condition (`if [ $# -eq 0 ]`) looks to see if the user included any options or a directory. If they did not, they are told how the `gp` command is used and the program ends.

#(2) look for options. The next section (look for options) is a `for` loop with a `case`. This construct looks for options. If none are found, the default is assumed: copy all files from the current directory to the directory specified. If options are found, an appropriate flag is set, and the positional parameters are shifted.

#(3) test if parameter is a directory. If the parameter is a directory, check if it is in the current directory, and set the "`bool`" flag (then in the next construct concatenate the entire pathname to the parameter name; this is needed when a subdirectory is being accessed).

#(4) begin main loop. The main loop tests several options and executes the appropriate action. For example, if the query option (`-q`) is set, it asks the user if he/she wants a file to be copies or not.

#(5) parameter is not a directory. Finally, if the parameter supplied is not a directory, an error message is returned.

Study the example and read the comments in the code. Then type it into a file and try to run the program yourself. By typing it in, you may come to understand the constructs and how they operate better than just reading the code on a page in a manual. Some additions you may wish to try are to selectively copy files that have a `.c` suffix (C source files).

7

Programming Tips

This chapter presents some tips for programming in the Bourne Shell.

Debugging

When you use pipes in shell programs, it becomes difficult to debug since you do not see output from commands in the pipe. One suggestion to help debug pipes is to add `cat` statements in the pipes to show you what the intermediate output would be. For example, you could add a `cat` command followed by an `exit` at one point in a pipe. The pipe will then list the output at that stage, and it will exit the program (to avoid further errors, and to indicate exactly where in the pipe you are).

Now, when you are ready to test the program, you need not exit the editor (which we are assuming is `vi`), run the program, see the output, then enter `vi` again to make changes. Rather there is a more convenient way to debug: save the program using `vi` command "`:w`"; run the program from `vi` by using the command:

`:!` *script* [*arguments*]

The `:!` command executes commands in the shell *outside* of `vi`. When you see the output, you then go back to `vi` (when prompted) make any necessary changes, and try it again. You can also execute a shell from `vi` (by typing "`:sh`") then execute the script.

For making this process quicker, you can: add the "`cat`" statements in the program, save the program, run the program from `vi`, return to `vi` and use the `u` (`undo`) command which will get rid of the "`cat`" statements (as long as you do not execute any other text manipulation commands since the last insert).

Another suggestion is to use the "`tee`" command instead of "`cat`". "`tee`" will transcribe the standard input to the standard output and makes a copy in a file(s) which are arguments to the command. The format is:

`tee` [`-i`][`-a`][*file ...*]

where the `-i` option ignores interrupts, and the `-a` option causes the output to be appended to the *files* rather than overwriting them. More than one file can be specified.

Creating Optional Pieces in a Pipe

There may come a time when you need a pipe with an optionally inserted piece. In other words, you wish to execute "`a | c`" if one condition exists and "`a | b | c`" if another condition exists. To do this, consider the following example:

```
optional="
if [ condition ]
then
  optional='b ||'
fi

eval "a | $optional c"
```

If *condition* is true, `optional` becomes "`b |`", and thus the `eval` statement executes "`a | b | c`". Otherwise, "`a | c`" is executed.

You can also use this same idea in optional redirection statements.

Halting Background Processes

If you are running several background processes and a foreground process, you may wish to be able to terminate all processes at the same time (instead of using the `kill` command for each). This may be helpful for instrumentation related work.

Let's say you wish to use the Break key to terminate three processes: the one in the foreground (terminated automatically) and two in the background. Here is a script which would accomplish this:

```
# initialize the process list
proc=

echo starting process 1
process1
# add process number to list
proc="$proc $!"

echo starting process 2
process2
# add process number to list
proc="$proc $!"

# the BREAK key will kill everything
trap "kill $proc;trap 2;exit" 2

echo starting process 3
# foreground process
process3
```

The first line initializes a parameter `proc` to a null value. The next two sections start the two background processes: first a comment is echoed to the screen so you know the process was started, then the process is started in the background (using the `&` operator), then the parameter `proc` is set to the process id of the process just run (the parameter `$!` is the process id if you recall).

The line containing the `trap` command looks for the signal 2 (which means interrupt). When this signal is received, it executes the commands in the double quotes: `kill $proc` will kill the two background processes since `$proc` is a list of the process ids.

The last command section starts the foreground process.

So, when this script is executed, the three processes are run. If you press the Break key, the trap is activated killing the two background processes (`process1` and `process2`). The foreground process (`process3`) is automatically terminated.

8

Detailed Reference

This chapter will cover the remaining concepts and commands associated with Bourne Shell programming. So far you have learned how to write a shell program with conditions, loops, user prompts and other options. This section discusses executing commands, defining functions, input/output, special commands, return values and executing the `sh` command.

Command Separators

When you execute commands in a shell program separated by newlines (Return's), the commands are executed *sequentially* or in the order they appear in the file. The following separators allow you to control the sequence of command execution.

The && Separator

This separator is a conditional separator. It will execute the next command in the command line only if the previous command executes successfully.

```
test -d /users/rhonda/tools && cd /users/rhonda/tools
```

This command line will first test to see if `/users/rhonda/tools` is a directory. If it is, the `cd` command is executed. If not, no further action is taken.

The || Separator

The double vertical bar separator will execute the next command only if the previous command was unsuccessful.

```
test -d /users/michael/projects || echo "directory does not exist"
```

This command line will test to see if the directory `/users/michael/projects` exists. If the test fails, the `echo` command is executed.

Mixing Separators

Here is an example which mixes the above separators:

```
test -d /tools && cd /tools; test -z "$fn" || sort -o $fn $fn &
```

The shell uses `;` and `&` to terminate a command sequence. Thus there are two command sequences: `test -d /tools && cd /tools`, and `test -z "$fn" || sort -o $fn $fn`. The first sequence is executed before the second (because of the `;` separator). If the first test is successful, the `cd` command is executed. The second command sequence is then executed in the background (due to the terminating `&`). The second test is performed, and if unsuccessful, the sort is performed.

Command Grouping

You can group a sequence of commands together using parentheses `()` or braces `{}`. If you group a series of commands with parentheses, a *sub-shell* is created to run the commands.

```
(who; ls)
```

This command grouping is executed separately from the current shell program. The current shell program only sees the results of the command grouping. The advantage of command grouping is you can place a series of commands in the background, or use other command separators to achieve a variety of results. Here's another example:

```
test -f $file && (cat $file > temp; sort -o temp temp; lp temp; rm temp)&
```

This command sequence will test if "`file`" is an ordinary file. If it is, it runs a command grouping in the background (note the terminating `&`). This command line could be simplified to read:

```
test -f $file && sort -o < $file | pr | lp
```

Another helpful way to group commands is with braces, {}. This command grouping is used primarily for redirecting combined output. You can group a series of commands together and use the resulting output:

```
{
date
ls
who
} > contents
```

All of the commands in the braces are executed, and the resulting output from all of the commands is placed in a file called `contents`.

Note Please be aware that *redirecting* output from `{}` causes a subshell to execute. Hence, any shell variables set, created, or changed from within the `{}` won't be effective outside of it. This is *not* true if output is not redirected.

Defining Functions

The more complicated your shell programs get, the more you will want to modularize them by using functions. This way you can create generic functions which can be re-used and eliminate repetitive code.

To define a function, use the following syntax:

name`()` `{`*list*`;}`

where *name* is the name of the function, and *list* is a list of commands used in the function.

Here is an example to show how functions are defined:

```
stat() {
  if [ -d $1 ]
  then
    echo "$1 is a directory"
    return 0
  else
    echo "$1 is not a directory"
    return 1
  fi;
}
```

This function tests the filename to see if it is a directory. If it is it returns a status of `0` (see "Return Values" later in this chapter). Otherwise it returns status `1`. Do not forget to place the semi-colon (;) at the end of the last line.

You can type your function in its entirety at the beginning of the shell program. When you wish to access it, you use the following format:

name [*parameter* ...]

where *name* is the name of the function, and [*parameter...*] refers to any optional positional parameters you wish to include.

You should note the following things concerning Bourne Shell functions:

- When calling a function, no parentheses are used; when defining a function, parentheses are necessary.
- When specifying parameters in a function call, be aware that the positional parameters ($1, $2, ...) for the *entire shell program* will be reset to these parameters, and the original values will be lost.

Input/Output

The common redirection symbols can be used in shell programs (`>` for redirecting output to a file, `>>` for appending output to a file, `<` for redirecting input to a command from a file). In addition are these redirection conventions:

`<<`[`-`] *word*

This redirection construct, called a *here document*, causes all lines after this one, and up to a line consisting only of *word*, to be used as input data.

Let's look at a sample section from a script file:

```
cat <<marker
These words are
to be printed with the
cat command, until the
line with "marker" is found.
marker
echo "End of text."
```

The text down to (but not including) "`marker`" will be printed on the screen when this script is run. Then the `echo` command is executed, giving an output of:

```
These words are
to be printed with the
cat command, until the
line with "marker" is found.
End of text.
```

Be sure to include quotes in *word* if the line contains special characters for command and parameter substitution, because they will be interpreted if not quoted. Notice it does not just look for the word `"marker"` but rather the line containing only the word `"marker"`. `<<` is particularly useful for multi-line input to commands (usually *ed*(1) commands). It is also useful because it eliminates, in many cases, the need for separate input files.

If you add the optional `-` after `<<`, then all leading tabs in the here document are stripped.

`<&` *digit*

This input redirection symbol uses the file descriptor associated with the descriptor *digit*. Most programs have standard input as 0, standard output as 1, and standard error as 2 (`stdin`, `stdout`, and `stderr` respectively). All programs which work properly with pipes observe 0 and 1 (and consequently 2). Other programs may not.

`>&`*digit*

is the format for using descriptors, where *digit* can be any single digit (0, ... ,9). The most commonly used redirection of this form is `1>&2` or `2>&1`. For example,

```
echo File $name not found 1>&2
```

The output of this line is redirected to the standard error (your terminal). So, in effect, you are creating your own error message and redirecting it in the same manner as an error from the shell. You can use this capability to ensure messages in a shell file reach the user. In the same manner `2>&1` merges the standard error into the standard output. And, you can use the `<&` capability in a similar manner to use as standard input.

The order in which you place the redirections is significant. The shell evaluates redirections from left to right:

```
1>fileA 2>&1
```

will first associate file descriptor 1 (thus it is no longer associated with standard output) with the file `fileA`. Then file descriptor 2 is associated with the file with file descriptor 1 (which is `fileA`). If we had placed the `2>&1` first, file descriptor 2 would be associated with file descriptor 1 (the terminal), and then file descriptor 1 would be associated with `fileA`.

To force *both* the standard output and error output into the same file, you usually use a statement like:

```
>file  2>&1
```

To close standard input use: `<&-`. To close standard output use: `>&-`.

Special Commands

The following are commands which may be of help to you in your shell programs.

Exec

The `exec` command allows you to replace the current shell with a new shell or another program. With the syntax:

`exec` [*arg ...*]

where *arg...* can be a sequence of commands or shell programs. This command can be helpful in cases where you do not wish to create subshells. You could have no desire to return to the parent shell, or you may be recursing and do not wish to keep parent shells active. A good example is a shell script which calls itself.

Expr

The `expr` command is very useful for performing arithmetic operations in shell programs. It also has other operations useful for string manipulation.

With the form:

`expr` *expression* { `+ -` } *expression*

you can add or subtract integers.

```
a=15
expr $a + 5
```

will return the string `20`.

To modify variables, you can use a similar format to:

```
a=‘expr $a + 1‘
```

using command substitution (grave accents) to place the new value in the variable `a`.

The symbols for multiplication, division, and remainder of integer-valued arguments are: `\*`, `/`, and `%`, respectively. Note the `*` is preceded by a backslash (`\`) to escape the shell's interpretation of the asterisk.

To compare integers, use the following format:

`expr` *expression* { `=,\>,\>=,\<,\<=,!=` } *expression*

where `!=` is "not equal to", and the other symbols represent mathematical comparisons (again, note the backslash before the special characters `<` and `>`). The function will return 0 if the comparison is successful, and 1 if it is not.

Here is an example of how a comparison might be used:

```
if expr $a \<= $b
then
  echo "$a is less than or equal to $b"
fi
```

Conditions

`expr` *expression* `\|` *expression*

will return the first *expression* if it is neither null nor 0. Otherwise it will return the second expression.

`expr` *expression* `\&` *expression*

will return the first *expression* if neither *expression* is null nor 0. Otherwise it will return 0.

Expr and Strings

`Expr` can also be used in string manipulation (the strings can be arithmetic):

`expr` *expression* : *expression*

will compare the first argument with the second argument which must be a **regular expression** (There is a discussion of Regular Expressions in *Text Processing: User's Guide* and in *The Ultimate Guide to the vi and ex Text Editors*). The ^ symbol is not a special character, however, because all patterns are anchored (begin with "^"). Normally, the matching operator returns the number of characters matched, and 0 on failure.

`expr length` *expression*

will return the length of the *expression* (number of characters).

`expr substr` *expression expression expression*

will return a substring of the first *expression*, starting at the character specified by the second *expression*, and for the length given by the third *expression*. For example:

```
a=batman
expr substr $a 1 3
```

returns the string "`bat`". And,

```
a=batman
expr substr $a 4 3
```

returns the string "`man`". (Note that the second and third expressions must be numeric.)

`expr index` *expression expression*

will return the position in the first *expression* which contains a character found in the second *expression*:

```
a=batman
expr index $a m
```

returns the value 4.

Set

The `set` command has a variety of uses. It is mainly used to *set* the value of a parameter. Let's begin with using `set` without arguments. If you type `set`, you get a list of all the parameters the system knows. These will include system parameters set by your `.profile` file, and any parameters you define.

You can define, or set, positional parameters easily with `set`. Simply follow the `set` command with the values for the positional parameters `$1`, `$2`, and so on. Here is an example:

```
set camp town ladies
```

Now `$1` has the value "`camp`", `$2` has the value "`town`", and `$3` has the value "`ladies`". You can also use command substitution with the `set` command:

```
set ‘date‘
```

where `$1`="`Thu`", `$2`="`Jun`", and so on for a date output of "`Thu Jun 26 09:34:01 MDT 1986`".

There are several options you can use with `set`. Preceding the option with `-` will turn the flag on. Preceding the option with `+` will turn the flag off. The format is as follows:

`set` [[*--aefhkntuvx*] *arg...*]

where the options are shown in the following table.

Table 8-1. Options to the `set` Command

Option	Description
`-a`	Mark variables which are modified or created for export.
`-e`	Exit immediately if a command exists with a non-zero exit status.
`-f`	Disable the file name generation.
`-h`	Locate and remember function commands as functions are defined.
`-k`	All keyword arguments are placed in the environment for a command, not just those that precede the command name.
`-n`	Read commands but do not execute them.
`-t`	Exit after reading and executing one command.
`-u`	Treat unset variables as an error when substituting.
`-v`	Print shell input lines as they are read.
`-x`	Print commands and their arguments as they are executed.
`--`	Do not change any of the flags.

These options can also be used with the `sh` command.

Unset

This command will remove the specified variable or function. The format is:

`unset` [*name...*]

where *name...* is a list of variables or functions *except* `PATH`, `PS1`, `PS2`, `MAILCHECK`, `IFS`.

Trap

The `trap` command waits until signals are sent to the shell program, and traps them. Instead of performing the default action, you can have the signals processed any way you wish. In other words, you use the trap command to wait for certain signals from the shell (which may be an unsuccessful command execution). When the trap sees a signal, it executes a list of predefined commands you generate. The syntax is:

`trap` [*command_list*] [*n*]

where *n* is the signal (or signals) `trap` looks for, and when they are found, *command_list* is executed. If *n* is `0`, then the command list is executed when the shell is exited. If you type `trap` with no arguments, a list of commands associated with each signal number is printed. An attempt to trap signal 11 (memory fault) produces an error.

Table 8-2 is a list of the signal numbers, their descriptions, and whether they can be trapped. To trap for signals 0, 1, 2, 3, 15 and execute a certain set of commands, you would use a command similar to:

```
trap "echo 'removing temp file'; rm temp" 0 1 2 3 15
```

Signals 1, 2, and 3 cannot be trapped if the script is run in the background (using nonsequential processing symbol '`&`'). Signal 9 should not be used as an argument to trap because it can never be caught, as well as signal 14 (it is used internally by *sh*(1)).

Table 8-2. Signals

Signal	Description	Trap Characteristic
00	Success	Trappable
01	hangup	Trappable (unless in background)
02	interrupt	Trappable (unless in background)
03	quit	Trappable (unless in background)
04	illegal instruction	Trappable
05	trace trap	Trappable
06	software generated (sent by `abort`(3C))	Trappable
07	software generated	Trappable
08	floating point exception	Trappable
09	kill	Cannot be trapped
10	bus error	Trappable
11	segmentation fault	Cannot be used as argument to trap
12	bad argument to system call	Trappable
13	write on a pipe with no one to read it	Trappable
14	alarm clock	Cannot be trapped.
15	software termination signal	Trappable
16	user defined signal	Trappable
17	user defined signal	Trappable
18	death of a child process	Cannot be used as argument to trap
19	power fail	Trappable
20	virtual timer alarm	Trappable
21	profiling timer alarm	Trappable
22	reserved	Cannot be used as argument to trap
23	window change or mouse signal	Cannot be used as argument to trap

Hash

The format for `hash` is:

`hash` [*-r*] [*name...*]

where *name...* is a list of command names.

The `hash` command makes searching for a command faster. Usually the shell will look in your search path (indicated in the shell parameter `PATH`) and go through each directory searching for the first occurrence of the command. `hash` will place the command in a table and include a pointer to the directory in which it resides. Thus, when you call the command, the hash table is first checked. If the command is in the hash table, it will be able to go directly to the directory instead of through all of the directories in the search path.

If you wish to delete the remembered locations in the hash table, include the `-r` option.

The default for `hash` (no options or parameters) is to print a listing of all commands used since login. The list includes two columns: `hits` which are the number of times the command has been invoked by the shell process, and *cost* which is the measure of work required to locate a command in the search path. The default `hash` command is used more for information, to see how the performance of the hash table is compared to the search path.

If you wish to see if a command is in a hash table, you can use the `type` command.

Type

The `type` command will tell you where a command is located in the directory structure. It will also indicate if the command is hashed (see `hash` above). The format is:

`type` [*name...*]

where *name...* is a list of commands.

Readonly

The `readonly` command is used to set the value of a parameter permanently. The format is as follows:

`readonly` [*name...*]

where *name...* is a list of parameters. When you use the readonly command on a parameter, it places the parameter into a set of parameters which are marked so they cannot be changed. No attempts to change the value of the parameter are allowed. For example, let's say we specify these parameters to be readonly:

```
dogs=rover
knuckles=chuckles
readonly dogs knuckles
```

If we attempt to change the value of, say, `dogs`,

```
dogs=spot
```

we get the message:

```
dogs: is read only
```

and the value remains at "`rover`". If you type in `readonly` with no parameters (the default), you get a list of all parameters which are readonly:

```
readonly dogs knuckles
```

Newgrp

You can change your group identification with `newgrp`. You remain logged in, but access permissions to files are done according to the new group environment. With `newgrp` you are always given a new shell even if the command terminates unsuccessfully. The format is as follows:

`newgrp` [`-`][*group*]

where *group* is the new group, and the `-` option will cause the environment to be changed to what it would be if you logged in again (you lose your old shell and get a new one). With no arguments, the group is changed back to what your password entry file indicates. For more information, see *newgrp*(1) in the *HP-UX Reference*.

Times

This command prints the accumulated user and system times for processes run from the shell. The times are precise to units of $1/HZ$ seconds, where HZ is processor dependent. The output looks like:

```
0m37s 0m25s
```

For more information, read `times`(2) in the *HP-UX Reference*.

Ulimit

This command provides control over process limits. The format is:

`ulimit` [*-fp*] [*n*]

where *n* is the size limit imposed by `ulimit`.

The `-f` option imposes a size limit of *n* blocks on files written by child processes (with no argument the current limit is printed). The `-p` option changes the pipe size to *n*. If no option is given, the `-f` option is assumed.

Wait

The `wait` command will **wait** until the specified process is finished, and then report its termination status. To specify the process, use this format:

`wait` [*n*]

where *n* is the process id. Most of the time you will not know the process number, but if you look ahead to the section "Parameters set by the shell," you will notice one entry (`!`) refers to the process number of the last background command executed. So, to wait for that background process to terminate, you would use:

```
wait $!
```

`wait`, without parameters, waits for all child processes to terminate.

Return Values

When a function or a command terminates, it sets a flag indicating the status of the termination. In other words, if the function or command was successful in executing, it returns a value indicating its success. The values (or error codes) normally used are listed in the `exit` section. These values are only conventions; shell scripts normally use these conventions, but programs in general do not.

When you execute a shell command incorrectly, you usually get an error message. What usually happens is the shell command returns an error code. If the error code is, say, 2, you will receive a message indicating a syntax error has occurred.

You can return error codes from your shell programs and functions in two ways. The `exit` statement can return any value you specify by using the following format:

```
exit n
```

where *n* can be an integer from 0 to 255. You can return error codes from functions by using the `return` statement:

```
return n
```

To check the return value of the last command you executed, you can use a parameter called `$?`.

Parameters Set by the Shell

You can access several special parameters that are set automatically by the shell. As mentioned above in the "Return Values" section, `$?` holds the return value of the executed command or function. Other such parameters you can use are:

Table 8-3. Parameters Set by the Shell

Parameter	Description
`$#`	The number of positional parameters.
`$-`	Flags supplied to the shell on invocation or by the `set` command.
`$?`	The return value sent by the previously executed command.
`$$`	The process number of this shell.
`$!`	The process number of the last background command.

Example

If it became necessary to `kill` some processes, and you especially didn't want to kill the shell you were in, you could get the `PID` of that shell by typing:

```
echo $$
```

The response to that command might be something like:

```
1899
```

Options for the sh Command

If the `sh` command is used to invoke shells or shell programs, you have several options available. You can use the options in the following table, and you can also use the options described in Table 8-1 (Options to `set` Command).

Table 8-4. Options for sh Command.

Option	Description
`-c string`	Read commands from `string`
`-s`	(or if no arguments are specified) Read commands from standard input. Any remaining arguments become positional parameters.
`-i`	This specifies an **interactive** shell: `TERMINATE` is ignored (`kill 0` does not kill an interactive shell) and `INTERRUPT` is caught and ignored (so that `wait` is interruptible).
`-r`	Make the shell **restricted** (see below).

Restricted Bourne Shell

Making a shell **restricted** (or `rsh`) causes the following actions to be disallowed:

- changing the directory (`cd`)
- setting the value of `PATH`
- specifying path or command names containing `/`
- redirecting output (`>` and `>>`).

The restricted shell is useful when you wish users to have limited access to the system. *Make sure the directories in which the restricted users are placed do not give them access to subdirectories in which they may do damage. Also make sure they do not have commands available to them, such as* `chsh` *and* `csh`*, that let them escape the restricted shell.* (See *sh*(1) about provisions to ensure this.)

9

Bourne Glossary

background process

A process that has been scheduled nonsequentially (background processes are generally transparent to the user).

control key (CTRL)

Used with other keys (in the same manner as the Shift key) to generate special characters.

cursor

A visual position indicator which moves with characters entered with the keyboard or with cursor movement keys (▲, ▼, ▶, ◀).

device file

The file associated with an I/O device. Device files are read and written just like ordinary files, but requests to read or write result in activation of the associated device. These files normally reside in the `/dev` directory.

disk

A platter for recording and storing information. A disk can be either a flexible disk or a hard disk. In this manual, when the term "disk" is used alone, it refers to a hard disk.

driver number

A pointer to the part of the kernel needed to use the device. The driver number is used in the `mknod` command when setting up a device file.

editing

Making changes in a file containing text, data or a program.

environment

System defaults which affect shell operation.

execution

Carrying out the instructions of a program or command.

file

A collection of computer information: program or data residing on a mass storage medium (e.g., a hard disk).

file types

Several file types are recognized. The file type is established at the time of the file's creation. The types are:

Regular files - Contains a stream of bytes. Characters can be either ASCII or non-ASCII. This is generally the type of file a user considers to be a file: object code, text files, nroff files, etc.

Directory - Treated as regular files, with the exception that writing directly to directories is not allowed. Directories contain information about other files.

Block special files - Device files that buffer the I/O. Reads and writes to block devices are done in block mode.

Character special files - Device files that do not buffer the I/O. Reads and writes to character devices are in raw mode.

Pipes - A temporary file used with command pipelines. When you use a pipeline, the shell creates a temporary buffer to store information between the two commands. This buffer is a file, and is called a pipe.

FIFO - A named pipe. A FIFO (First In/First Out) has a directory entry and allows processes to send data back and forth.

function key, for example f1

A key on the keyboard which, when pressed, executes a specified computer function.

HP-UX

The computer's operating system. HP-UX is an HP value-added version of UNIX System V.

input

Data read by any program, whether from a keyboard, file or pipe.

internal memory

Electronic data storage located in the computer for program and computer operations execution.

kernel

The core of the HP-UX operating system. The kernel is the compiled code responsible for managing the computer's resources; it performs such functions as allocating memory and scheduling programs for execution. The kernel resides in RAM (Random Access Memory).

message

An item of information generated by the computer to inform the user of an operation or error resulting from a command.

nonsequential

In no particular order (at the same time).

operating system

The part of the system that interacts with the user and executes the user's commands.

output

The data that results from a program or computer process.

parameter

The second (and subsequent) words/data after a command or program. Parameters are used to pass information to a program or command.

parse

Separating statements into basic units for translating into machine language or for interpreting.

path

An ordered sequence of steps from origin to destination.

path name

A series of directory names separated by / characters, and ending in a directory name or a file name.

permission

Operation allowed to a specified type of user.

pipe

The name given to a command line where the output of one command becomes the input to another command. The commands must be connected by a "|" character.

process

A process is the environment in which a program (or command) executes. It includes the program's code, data, status of open files, and value of variables. For example, whenever you execute a shell command, you are creating a process; whenever you log in, you create a process.

program

A sequence of instructions performing a task.

redirection

Changing the default path of input or output (sending output to a file instead of to the screen, for example).

screen

The device with which the user sees computer output (the CRT or terminal).

script file

A file containing commands (each on a separate line). When the entire file is executed, the commands are executed in the order in which they appear in the file.

sequential

In order (not at the same time).

shell

A program that interfaces between the user and the operating system. HP-supported shells are:

```
/bin/sh       /bin/ksh
/bin/csh      /bin/rksh
/bin/rsh      /bin/posix/sh
```

variable

See Parameter.

vi

The vi editor (visualize).

Index

S

T

U

Part II
C Shell

- Preparing to Use C Shell
- Command History
- Aliases
- Metacharacters
- Shell Variables
- Commands, Jobs and Scripts

10

Preparing to Use the Shell

Introduction

csh, pronounced "C Shell", is an HP-UX command language interpreter and a high-level programming language. It is used to translate command lines typed into the system into system actions, such as running programs, moving between directories, and controlling the flow of information between programs.

`csh` has several useful features, including:

- Command History Buffer and associated history substitution facility. Recently executed commands can be modified and re-executed with ease.
- an aliasing mechanism. Useful statements can be referenced with a short alias.
- an extensive, C-like command and control capability.

For additional information about HP-UX shells, consult the Bourne Shell part of this manual.

Note This software and documentation is based in part on the fourth Berkeley Software distribution under license from the Regents of the University of California. We acknowledge the following individuals and institutions for their role in its development: William Joy.

This document uses the following conventions:

- All examples assume the C Shell prompt has been changed to show the current command event number by entering the following set command in either `$HOME/.cshrc` or `$HOME/.login`.

```
set prompt = "[\!] % "
```

This prompt will appear as:

```
[23] % _
```

- `Computer` font is used to show what should appear on the screen. For example, to activate the C Shell, type `csh`. Terminating command sequences with Return is assumed.
- Actual file names, like `sh_history`, and variable names, like `histsize`, are in computer font.
- Placeholders for file names, like *filename*, and variables, like *printername*, are in italic font.

Several HP-UX commands are useful in setting up and verifying shell operation. They include `chsh` (change login shell), `rlogin` (used to access remote systems over a local area network), `printenv` (lists currently defined environment variables with their corresponding values), `set` (sets or lists system variables), `setenv` (used to set shell environment variables to a given value).

Refer to the *HP-UX Reference*, section 1, for detailed information about these commands.

HP-UX Standard Shells

HP-UX systems support the Bourne Shell, the C Shell, the POSIX Shell, and the Korn Shell command interpreters.

The normal shell prompt for the Bourne, POSIX and Korn Shells is the dollar sign (`$`). When C Shell is active instead, the default prompt becomes the percent (`%`) symbol. The prompts for any shells can be changed to any character(s) you want, but more about that later.

Shell Startup and Termination

Running C Shell From the Bourne Shell

The name of the C Shell program is `/bin/csh`. To run C Shell from the Bourne Shell, type:

```
csh
```

Your prompt changes to the C Shell prompt, %, unless you have redefined the C Shell prompt.

Making C Shell Your Login Shell

To make C Shell your default login shell, type in:

`chsh` *login_name* `/bin/csh`

The argument *login_name* is your login name.

The command `chsh` means *change shell*. When you change shells, the new shell is your default login shell until you use `chsh` again. `chsh` changes your login shell, *not* your current working shell. To change to the new login shell, exit from your current shell, then log in again.

C Shell is stored in `/bin/csh`. The Bourne Shell is stored in `/bin/sh`. To make the Bourne Shell your login shell, type:

`chsh` *login_name*

If no shell pathname is specified on the `chsh` command line, the login shell is set to default (Bourne).

Terminating C Shell

Various ways can be used to terminate C Shell, depending on the current value of the boolean flag `ignoreeof`. To determine the current value of `ignoreeof`, type `set` without arguments. This lists all currently defined variables and their values. Boolean variables are listed only if set. For example:

```
[25] % set
argv    ()
autologout      15
cwd /users/login_name
history         15
home /users/login_name
ignoreeof                                   ignoreeof is set for this example
noclobber
prompt [!] %
shell /bin/csh
status          0
term hp2622
path (/bin/posix /bin /usr/bin /usr/local/bin /etc/users/login_name . )
[26] % _
```

`exit` (or `logout`, if in a login shell) can be used to exit C Shell at any time if a prompt is being displayed. If `ignoreeof` is not set, you can also use CTRL-D.

Returning to a Parent Shell

If you started C Shell from another shell with `ignoreeof` set, type:

```
exit
```

to return to the original shell. If you use CTRL-D and `ignoreeof` is set, the error message:

```
Use "exit" to leave csh.
```

results. You will know that you have returned to the Bourne or Korn Shell because the shell prompt changes to your Bourne or Korn Shell prompt.

If `ignoreeof` is not set, you can use CTRL-D or `exit` to obtain the same result.

Logging Off the System

If C Shell is your default login shell and you have not set the system variable `ignoreeof`, you can terminate C Shell and log off the system by typing `exit` or `logout`, or by pressing CTRL-D

The system variable `ignoreeof` is discussed later. If a file `$HOME/.logout` (a file named `.logout` in your home directory) exists, it is executed as part of the log-off process.

Terminating C Shell with `ignoreeof` Set

If C Shell is your default login shell and the system variable `ignoreeof` is set, you cannot terminate C Shell and log off the system by typing:

CTRL-D

If you attempt to do so, the system responds with the message:

```
Use "logout" to logout.
```

C Shell Startup

Depending on whether it is your default login shell, C Shell looks for one or all three of the following files and executes them as indicated in the order indicated, if they exist:

/etc/csh.login	If C Shell is your login shell and this file exists, it is executed.
.cshrc	If this file exists in your home (login) directory, it is executed every time C Shell starts, whether at login or when C Shell is spawned from another shell.
.login	If C Shell is your login shell and this file exists in your home directory, it is executed.

While none of these files is required, if present, they provide a convenient means for customizing the shell environment to fit your needs.

Setting Environment and Shell Variables

Two kinds of variables can be set in the `.cshrc` and `.login` files:

Environment variables	These variables are global (used by the login shell process and any processes spawned by the shell process). They are usually represented by uppercase letters.
Shell variables	Shell variables are local (used by the login shell process only) and are not inherited by spawned processes. They are usually represented by lowercase letters.

Environment variables are usually defined by using the `setenv` command, while shell variables are typically defined by the `set` command. However, three of the most commonly used environment variables – USER, TERM, and PATH – are automatically imported to and exported from three corresponding variables – user, term, and path. Thus, if you execute:

```
set path=(/bin/posix /bin /usr/bin)
```

the value of the environment variable PATH also becomes `/bin/posix:/bin:/usr/bin` (note the difference in syntax between the two variables).

The commands `set` and `setenv` can be executed interactively from a terminal, or they can be placed in the `.cshrc` or `.login` files.

The .cshrc Shell Script File

Whenever a C Shell starts during your session, it searches for the file *.cshrc* in your home directory and executes it if it exists. The information in this file is used to set variables and operating parameters that are local to the shell process.

Since every C Shell created executes this file, it is customary to use it for setting shell variables by including `set` commands in the file. If the *.cshrc* file does not exist in your home directory, HP-UX spawns C Shell using default values for needed variables.

To verify your current shell environment, execute `set`. A listing similar to the following is printed on the display:

```
[25] % set
argv    ()
autologout      15
cwd /users/login_name
history         15
home /users/login_name
ignoreeof                               ignoreeof is set for this example
noclobber
prompt [!] %
shell /bin/csh
status          0
term hp2622
path (/bin/posix /bin /usr/bin /usr/local/bin /etc/users/login_name . )
[26] % _
```

Some of the commands commonly used in the `.cshrc` file and their meanings are shown in Table 10-1.

Table 10-1. `.cshrc` File Commands

Command	Meaning
`set ignoreeof`	Traps CTRL-D's to avoid accidental system log off. Use the `logout` or `exit` command.
`set prompt = "[\!] %"`	This command causes your C Shell prompt to be the current event number in square brackets followed by a percent sign. This is very helpful when using the command history buffer.
`set history=15`	Sequentially keeps a buffer of your last (15 in this case) events.
`set savehist=15`	This command saves the last (15 in this case) events when you log off your system. When you log back onto your system, the event history is restored.
`set noclobber`	This command stops C Shell from overwriting and destroying the information in an existing file.

You can suppress execution of the `.cshrc` file by using the `-f` option in the `csh` command as follows:

```
csh -f
```

The .login Shell Script File

When you activate C Shell by logging onto the system, C Shell looks for the shell script file `.login` in your home directory and executes it if it exists. This shell script file contains **global commands**, **variables**, and **parameters** that you want executed or set up automatically at the beginning of your session. Some of the commonly used commands you might want to include in this file and their meanings are shown below. The term `login_name` refers to your login name.

Command	Meaning
`setenv TERM hp2622`	Sets the system variable TERM to recognize the HP 2622 as your terminal.
`setenv TZ MST7MDT`	This command sets the time zone variable. The example specifies U.S. Mountain Standard Time/Mountain Daylight Savings Time Zone.
`setenv PATH /bin/posix:/bin:/usr/bin:/usr/local/bin:/etc:/users/`*login_name*`:.`	This command sets the search pattern the system uses for finding commands.
`set mail=/usr/mail/login_name`	Required to receive mail for HP-UX.
`alias h history`	Make the character `h` an alias for your command history file.
`alias bye logout`	For some, `bye` is easier to remember than `logout` as a session termination order.
`news | more`	Pipe the news through `more`.

C Shell Termination

When C Shell is your default login shell and you log off of the system (not when you return to another shell that spawned C Shell), C Shell looks for a file `.logout` in your home directory and executes it if it exists. Commands that are typically included in a logout shell script include the following:

Table 10-2. Logout Script Commands

Command	Meaning

`echo ' '`
`echo '***** You are logged out now. *****'`
`echo ' '`
	Print logout message to your standard output (`stdout`) device.

`date`	Prints your log out date and time.
`sync`	Put all information stored in all buffers onto the system disk.

11

Command History

The Command History Buffer

`csh` maintains a Command History Buffer capable of holding one or more of your most recent commands. By setting the *history* variable to some integer value, the history buffer can hold many commands. These saved commands, sometimes called **events**, can be accessed in many useful ways. Commands can be quite complex, so the term **event** is used to refer to commands stored in the Command History Buffer from now on. A buffer size of 10 to 20 is about right for most situations.

You can make use of the history buffer by using the C Shell history substitution facility, which enables you to use words from previous commands as parts of new commands, repeat command events, repeat arguments from a previous command in the current command event, and fix spelling and typographical errors in previous events.

History substitutions begin with an exclamation point (`!`) and cannot be nested.

To see how this all works, place the following lines in a file named *.cshrc* or *.login* in your home directory.

```
set history = 15
set savehist = 15
set prompt = "[\!] % "
```

These commands:

- create a fifteen-event Command History Buffer.
- save the last 15 events in your command history buffer when you log off the system and restore them the next time you log on the system.
- cause your C Shell prompt to display the event number of each event.

All of the capabilities that you are about to see can be used without this special prompt, but they are easier to manipulate if you have a prompt that provides event numbers of each event executed.

To see what is in your history buffer, type in the command `history` without arguments. Your display may appear as shown below:

```
[6] % history
      1  ls -als
      2  cat junk
      3  pr memo | lp
      4  mail jd < memo
      5  vi .cshrc
      6  history
[7] % _
```

Re-executing Events

You can re-execute a previous event by referencing the event in your history buffer. Events can be referenced by:

- event number.
- relative location from the current event.
- the text of the event.

As a special case, the immediately previous event can be referenced by two successive exclamation points (!!). The first activates the substitution facility; the second references the most recent previous command.

Referencing by Event Number

One way to re-execute an event stored in the history buffer is to reference its event number. For example:

```
[7] % !2
cat junk
This is the contents of the file junk.
[8] %
```

re-executes event number *2*. Notice that the event to be re-executed is echoed on the terminal before it is executed, so you can verify that you are referencing the correct event.

Referencing by Relative Location

Another way to re-execute an event is to reference its position in the history buffer relative to the current event. For example:

```
[8] % !-4
mail jd < memo
[9] %
```

executes event four (8-4=4), in this case sending a memo to `jd` again.

Referencing by Event Text

You can re-execute an event by entering the first few characters of that event's command line. If you have previously executed *history*, you can see what the current history buffer contains by using:

```
[9] % !h
```

The history substitution facility searches backward through the buffer until it finds an event whose command line begins with the letter "h". When it finds the event with the *history* command line, it re-executes it, producing:

```
[9] % !h
     1  ls -als
     2  cat junk
     3  pr memo | lp
     4  mail jd < memo
```

```
     5  vi .cshrc
     6  history
     7  cat junk
     8  mail jd < memo
     9  history
[10] %
```

Reusing Command Arguments

The history substitution facility enables you to use parts of previous commands as building blocks of new commands. Each command argument in a command event is numbered. To reference a command argument, specify the event with one of the methods described previously in "Re-executing Events," then use a colon (:) followed by the argument's position number.

The first argument, usually the command, is argument number zero (0). The second argument is argument number one (1), etc. The last argument is given the special reference of the dollar sigh ($). The second argument, usually the first argument after a command word is given the special reference of the circumflex (^). To see how this works, begin with the example shown below.

```
[10] % nroff -man csh.1 | lp &
```

To see what the last argument in this event is, type in:

```
[11] % !10:$
&
[12] %
```

The last argument in event 10 is the ampersand (&). The history mechanism extends the normal meaning of "argument" to include important metacharacters. The argument specified by a circumflex (^) is `-man`. To see if this is true, type in:

```
[12] % echo !10:^
echo -man
-man: Command not found.
[13} %
```

The referenced argument can be made part of another command. A range of event arguments can also be specified by using a dash (-) to separate the range endpoints. For example:

```
[13] % echo !10:3-$
echo | lp &
[1] 18634 18635
[14] %
```

Note that the example generated a new C Shell with the event number `[1]` and two process IDs `18634 18635`. This new shell is called a **background process**. The output from `echo` is printed on the line printer (by piping it to the printer spooler). Jobs and job numbers are discussed later in this tutorial.

If you want to reuse all of the arguments of an event that follow an initial command, you can use an asterisk (`*`):

```
[14] % mkdir /users/bill /users/pete /users/mary
[15] % rmdir !14:*
rmdir /users/bill /users/pete /users/mary
```

Modifying Previous Events

As you use C Shell, you will find that re-executing a previous event with minor modifications reduces typing. To modify and re-execute a previous event, form the new command line by using a combination of the following steps:

1. Start the command with the re-execution character (!), followed by a reference to the previous event. The previous event reference can be the event number, location relative to the current event, or text contained in the event's command line as discussed earlier.
2. Optionally, you can specify particular words on the chosen event's command line as discussed earlier under "Reusing Command Arguments." This specification is usually separated from the event reference (Step 1) by a colon (:).
3. Finally, specify how you want the previous event altered by selecting from the list of modifiers that follows. If you skipped Step 2, the modifier applies to the entire event. If particular words were selected during Step 3, the

modifier applies to those words. Modifiers are always prefixed by a colon (:) and several can be used in sequence.

The following list of modifiers can be used to alter or replace event arguments prior to re-execution.

Table 11-1. Previous Event Modifiers

Modifier	Definition	Effect
s/old/new	substitute	Substitute *new* for *old*. Any character may be used as the delimiters between the substitution strings. An ampersand (&) in the new string is replaced with the entire old string. Note that this affects only the first occurrence of *old* on an event's command line. Use the "gs" combination if you want the effect to be global.
g	global	Use in combination with another modifier to make the effect of the modifier global for an event's entire command line. For example, `gs/old/new` replaces all occurrences of *old* with *new*. NOTE: Only one substitution can be made per argument in an event. For example, the effect of `gs/joe/mary` on the path name `/users/joe/joe_file` would be to make the following modification: `/users/mary/joe_file`.
h	head	Use only the directory path name from a specified argument in a previous event by removing its final path name component (that is, use only the path name's head).
p	print	Print the event specified, but do not execute it. This is useful if you just want to verify what a particular event was. For example: `[10] % !3:p` prints event number 3 on your terminal without executing it.
q	quote	Quote the modifications so that no further modifications can take place.
r	root	Remove the filename extension. If a file name's tail ends with a "." followed by one or more characters, the "." and the characters that follow it are dropped (thus, the `.o` is removed from filename `file.o` leaving `file`).

Continued on next page ...

Table 11-1. Previous Event Modifiers (continued)

Modifier	Definition	Effect
t	tail	Remove all elements of a path name except the last element (i.e., the path name's tail).
&	repeat	Do the previous substitution again. The history substitution facility keeps track of the last substitution you performed with the `s` modifier, thus enabling you to easily perform the same change on various events that you want to re-execute.

For example, suppose we enter the following commands:

```
[14] % car /users/jack/documents/memo
car: Command not found.
[15] %
```

The `cat` command in event 14 was misspelled. To fix this, type:

```
[15] % !14:s/car/cat
cat /users/jack/documents/memo

This is a test.
[16] %
```

This executes the command correctly, without retyping the whole path name of the file that you want to look at. To look at a file called "list" in the same directory, you can now enter:

```
[16] % !15:s/memo/list
cat /users/jack/documents/list

apples
oranges
bananas
pineapples
strawberries
plums
[17] %
```

Now, suppose that you want to move to the directory containing the files that you just looked at. You can do this with:

```
[17] % cd !!:^:h
cd /users/jack/documents
```

This is quite a complex command, but typing is still saved. The double exclamation marks specified the immediately previous event, the circumflex (^) argument specifier selected the second argument on the event's command line, and the h modifier used only the head of the specified argument is used ("/users/jack/documents").

To return to your home directory, type:

```
[18] % cd
[19] %
```

An Example

To see how this all comes together, let's try to debug the following C program. To do this example, use an editor to create the file `bug.c` as shown by event [22] below.

```
[22] % cat bug.c          Prompt set to show current command number.

main()
{
    printf("hello);
}
[23] % cc !$              Compile file named in last event.
cc bug.c

"bug.c",line 3: unterminated string or character constant
"bug.c",line 4: unterminated string or character constant
"bug.c",line 5: unexpected <end-of-file>
"bug.c",line 5: syntax error at <end-of-file>
```

```
[24] % ed !$                    Edit file named in last event.
ed bug.c

31
3s/);/"&/p
    printf("hello");
w
32
q

[24] % !c                       Do last event that began with small c
cc bug.c                        character.

[25] % a.out

hello [26] % !e                 Not right, run ed again.
ed bug.c

32
3s/lo/lo\\n/p
    printf("hello\n");
w
34
q

[26] % !c -o bug                Do the last c event and append the -o
cc bug.c -o bug                 option and word "bug".

[27] % size a.out bug

a.out: 792 + 408 + 84 = 1284
bug: 792 + 408 + 84 = 1284

[28] % ls -l !*                 Prefix last event's arguments with an
ls -l a.out bug                 ls -l command.
```

```
-rwxr-xr-x 1 jerry     1751 Feb 29 09:00 a.out
-rwxr-xr-x 1 jerry     1751 Feb 29 09:01 bug

[29] % bug
hello
[30] % pq -n !!:s/g/g.c
pq -n bug.c
pq: Command not found.

[31] % !!:s/pq/pr          Correct spelling in last event from "pq"
pr -n bug.c                to "pr".

  1 main()
  2 {
  3    printf("hello\n");
  4 }

[32] % !! | lp             Execute last executable event (!!)
pr -n bug.c | lp           and pipe to line printer spooler.

[33] %
```

12

Aliases, Command Substitution, Metacharacters

Aliases

C Shell provides an alias facility so you can customize commands. With aliasing, you can define new commands or make standard commands perform nonstandard functions. The alias facility is similar to a macro facility; when an alias is detected, it is replaced by the alias definition.

To list existing aliases, enter **alias** without arguments. For example:

```
[41] % alias
cd       cd !* ; ls
h        history
print    pr !* | lp
w        who ; echo You are ....... ; who am i
dir      (ls -als)
```

You can create the above aliases interactively from the terminal keyboard or by placing alias commands in a shell script.

Aliasing Existing Commands

You can alias HP-UX commands so that they perform nonstandard functions. Suppose you like to get a directory listing whenever you change directories. Do this by aliasing `cd` in the following way:

```
[42] % alias cd 'cd \!* ; ls'
```

Using a command statement in the alias of the command is acceptable.

The entire alias definition is placed inside single quotes to prevent interpretation of the semicolon as a metacharacter and to avoid unwanted substitutions.

The backslash (\) in front of the exclamation point prevents the exclamation point from being interpreted as a history substitution. As a result, the string `\!*` substitutes the entire argument list to the pre-aliasing `cd` command.

The semicolon separates the `cd` and `ls` commands so that they are executed sequentially.

Creating Custom Commands

C Shell's alias facility can also be used to create new commands. Suppose you want to get a long, alphabetical listing of your current working directory showing the size of each file. You could type in:

```
ls -als
```

each time, but you want to make up your own command

```
dir
```

and get the same results. To do this, type in:

```
alias dir ls -als
```

Alias Substitution

After a command line is scanned, it is parsed into distinct command arguments. The first word of each command, left-to-right, is checked to see if it has an alias. If it does, the alias string replaces the aliased word. The process begins again. The substituted alias string is marked to avoid looping and does not modify the rest of the command word's arguments.

Alias and the history facility both use the same substitution scheme. A single exclamation point represents the current event and is preceded by a backslash so that the shell does *not* interpret it but instead passes it on to alias. History modifiers also work in alias statements.

Alias Use Restrictions

There are two basic restrictions that you must adhere to when using the alias facility:

- Although you can alias the `alias` command to be called something else, you cannot alias any command to be called `alias`. If you attempt to do so, an error message is generated.
- To prevent the formation of an alias loop, C Shell allows a particular alias string to appear only once in another alias definition. Also, the command that is being aliased can appear only once in its own alias definition. For example:

  ```
  [32] % alias ls alias
  ```

  works, but:

  ```
  [33] % alias ls 'ls ; ls'
  ```

  does not. If you try to execute `ls` after it has been aliased with event 33 above, you see:

  ```
  [34] % ls
  Alias loop.
  [35] %
  ```

Unaliasing an Alias

Assume that when you type alias at the `%` prompt, the following aliases are active in your shell. (Your system administrator might have created them, or you could have added them to your `.cshrc` file yourself.)

```
[41] % alias
cd      cd !* ; ls
h       history
print   pr !* | lp
w       who ; echo You are ....... ; who am i
dir     (ls -als)
```

To `unalias` the change directory command (`cd`), type in:

```
[42] % unalias cd
```

```
[42] % alias
h       history
print   pr !* | lp
w       who ; echo You are ..... ; who am i
dir     (ls -als)
```

Command Substitution

A command enclosed in single back-quote (‘), also called a grave accent, characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to:

```
[43] % set pwd=‘pwd‘
```

to save the current directory in the variable `pwd`. You can now print the value of the `pwd` variable with:

```
[44] % echo $pwd
/users/joe/documents
[45] %
```

Command substitution also provides a way of generating arguments for other commands. For example:

```
ex ‘grep -l TRACE *.c‘
```

runs the editor `ex`, supplying as arguments those files whose names end in `.c` and contain lines which contain the string `TRACE`.

Metacharacters in C Shell

C Shell recognizes a number of characters as having special meaning. Because they have syntactic and semantic meaning to C Shell, these special characters are called **metacharacters**.

Metacharacters affect C Shell operation only as the characters are read into the shell. (C Shell displays an `&` as a prompt when reading.) Metacharacters normally recognized by C Shell are ignored by C shell when running another program, such as `vi` or `mailx`. Thus, you can include metacharacters in text being processed by such programs without concern for their significance to C shell.

Syntactic Metacharacters

`;` separates commands to be executed sequentially.

`|` separates commands in a pipeline. Commands in a pipeline execute sequentially with the output of one command being fed as input to the next command.

`()` isolates commands separated by ";" or pipelines such that the result appears as a single command. Thus, pipelines enclosed in parentheses can be used as components in another pipeline. Commands enclosed within parentheses are always executed in a subshell.

`&` indicates command(s) must be executed as a background process. For example, to print the file *letter* as a background process on the system printer spooler, type:

```
pr letter | lp &
```

`||` separates commands or pipelines in such a manner that the second is performed only if the first fails.

`&&` separates commands or pipelines in such a manner that the second is performed only if the first succeeds.

Filename Metacharacters

If a file name contains one of the metacharacters listed below, the name is a candidate for file name substitution. File name metacharacters can represent patterns or identify abbreviations. Characters representing patterns indicate that the name is a pattern which the shell should replace with all file names in the specified (or current if not specified) directory that match it. Characters that identify abbreviations cause C Shell to expand the file name, based on the abbreviation provided.

Metacharacters that represent patterns include:

`?` expansion character matching any single character when specifying a filename. For example, to collect the files `filea.o`, `fileb.o` and `filec.o` in the file named `total.o`, type in:

```
cat file?.o > total.o
```

`*` expansion character matching any sequence of characters, including the empty sequence. To remove all files beginning with the word `old`, type in:

```
rm old*
```

`[]` expansion matching of any single character or range of characters separated with a dash (`-`) listed within the brackets. For example, to list all the files with the same root name `file` followed by any single lower case character, type:

```
ls file[a-z]
```

This could produce:

```
fileo  filep
```

Metacharacters that identify abbreviations include:

`{}` abbreviating a set of words which have common parts. For example, the files list, last and lost can be listed with:

```
ls l{aio}st
```

`~` substitutes that path name of the specified user's home directory. Syntax is a tilde followed by the login name of the desired user. If the tilde is followed immediately by a slash (`~/`) and a file or path name, your home directory is substituted instead (tilde can be used alone with the `cd` command to change to your home directory). If a `~` appears in the middle of a word it is not interpreted as a metacharacter and is left undisturbed.

The slash (`/`) character also has special significance in file names:

`/` separates components of a file's pathname. For example, `/bin/csh` is the pathname to the file `csh`. The first slash in a pathname or a single slash aliases the system's root directory.

Quotation Metacharacters

`\` prevents interpretation of the character which follows it as a metacharacter. For example, typing

```
ls *
```

prints a list of all files and directories and in the current directory. Typing:

```
ls \*
```

prints

```
* not found
```

`'` prevents interpretation of a string of characters as commands or metacharacters. For example, if you set a variable to contain a command string, the command string may in turn contain metacharacters. Thus, whenever the variable is referenced, there is a risk that the metacharacters could be inappropriately processed. By enclosing the string within single quotes, unwanted processing of any metacharacters in the string is avoided.

" prevents interpretation of metacharacters in a string, while allowing normal command and variable expansion. Double quotes are similar to single quotes except that only metacharacters are left unprocessed

Input/Output Metacharacters

`<`*name* indicates redirected input from *name*. For example,

```
mail boss < memo &
```

sends the file `memo` to the `boss`.

`>`*name* indicates redirected output. For example,

```
grep -vn file1 file1 > numbered.file1
```

puts a copy of `file1`, with each line numbered, in the new file `numbered.file1`. This metacharacter causes the target file to be overwritten.

`>&`*name* directs the diagnostic output along with the standard output into the file *name*.

`>!` *name* redirects output with overwrite of target file. This is used when `noclobber` is set.

`>>`*name* redirects output by appending it to the end of *name*. If the file *name* does not exist and the variable `noclobber` is set, an error occurs.

`>>&`*name* appends diagnostic output along with the standard output to the end of *name*.

`>>!` *name* Acts like `>>` except in the case where *name* does not exist and the *noclobber* variable is set. In such a situation, `>>!` creates *name* and no error occurs.

`<<`*word* reads the shell input up to a line which is identical to *word*. *Word* is not subjected to variable, file name, or command substitution, and each input line is searched for *word* before any substitutions are performed on it. Files are processed in this manner are commonly called **here documents**.

| forms a pipeline between two processes. A pipeline causes the output of the process before the vertical bar to be the input of the process after the vertical bar.

|& forms a pipeline between two processes that sends diagnostic output as well as standard output from the first process as input to the second process.

Expansion/Substitution Metacharacters

$ indicates variable substitution. For example,

```
set M1 = /usr/man/man3
cd $M1
```

The pathname is assigned variable `M1`. To use the variable, precede the variable name with a dollar sign.

Note that you could also execute `cd M1`. C Shell then looks for a directory called "M1" and, when it cannot find it, proceeds to search for a variable of that name. When the variable is found, its value is used as an argument to *cd*.

! indicates history substitution. See discussion in Chapter 11.

: precedes substitution modifiers. See discussion in Chapter 11.

? used in special forms of history substitution indicating command substitution.

Other Metacharacters

# indicates shell comments and begins scratch file names. Must be the first character in a line to be executed by C Shell.

% prefixes job name specifications. For example:

```
[56] % cc test.c >& test &
[1] % 3265
[57] % kill %1
[58] %
```

Event 57 kills the background process with the job number 1.

Using Metacharacters as Normal Characters

Metacharacters cannot be used directly as parts of command arguments. Thus, the command

```
echo *
```

does not echo the character *. It will either echo a sorted list of file names in the current working directory or print the message `No match` if there are no files in the working directory.

To handle metacharacters as normal characters, put them between single quotes. The command:

```
echo '*'
```

will echo an asterisk to your display.

Three metacharacters cannot be "escaped" with single quotes:

- the exclamation mark (!)
- the backslash (\)
- the single-quote (')

The backslash must be used to cancel the special shell meaning of these metacharacters. Thus:

```
echo \'\!\\
```

prints

```
'!\
```

These two mechanisms, the single-quote and the backslash, let you use any printable character in a shell command. They can be combined, as in

```
echo \''*'
```

which prints

```
'*
```

The backslash (\) escapes the first single-quote (') and the asterisk (*) was enclosed between single-quotes. The result is a single-quote and asterisk.

13

Shell Variables

Built-In Shell Variables

C Shell maintains a set of variables that can be assigned values by the `set` command. Shell variables are useful for storing values for later use in commands. The most commonly referenced shell variables are, however, those which the shell itself refers to. By changing the values of these variables, you can directly affect the shell behavior. The following variables are supported by C Shell on HP-UX.

$argv

This variable contains the command line arguments from the calling shell.

$autologout

This variable is used to automatically log you off the system if you do not use the system for a specified amount of time. For example,

```
set autologout = 60
```

will automatically log you off the system if you do not use the system for an hour (60 minutes).

To disable autologout, set it to zero (0) time. For example:

```
set autologout = 0
```

or

```
unset autologout
```

$cwd

The `cwd` variable contains the path name to your current working directory. This variable is automatically changed with each `cd` (Change Directory) command. At log-on, the default for this variable is the directory in the system variable $HOME.

$home

The `home` variable contains the path name to your home directory. The default value for this variable is specified in the system file `/etc/passwd`. (See *passwd*(5).)

Boolean ignoreeof

The boolean variable `ignoreeof` determines whether CTRL-D is allowed to log you off the system. If `set`,

```
set ignoreeof
```

logout must be used to terminate a session. If `ignoreeof` is `unset`,

```
unset ignoreeof
```

you can also use CTRL-D to log off. The default is `set`.

$cdpath

Use this variable to specify alternate directories to be searched by the system when locating subdirectory arguments used with `pushd`, `cd`, and `chdir` commands.

Boolean noclobber

Suppose you use the following command sequence to send keyboard input to a file called `newfile`.

```
cat > newfile
```

If `newfile` exists before this command sequence is executed, the old copy of `newfile` will be overwritten and thus destroyed. To prevent accidental

overwriting of a file containing valuable information, set the boolean `noclobber` variable so that C Shell cannot overwrite files by including the command line:

```
set noclobber
```

in your *.login* file. To demonstrate its effectiveness, type the following C Shell commands:

```
% cat > newfile
This is a test message.
EOT
%set noclobber
cat > newfile
newfile: File Exists.
%
```

When you try to `cat` to an existing file with `noclobber` set, the system tells you the `File Exists.` and aborts the command. To override the `noclobber`, use the exclamation point metacharacter. For example:

```
%cat > newfile
newfile: File Exists.
%cat >! newfile
This is an override test.
EOT
%
```

Boolean notify

If the *notify* variable is set, you are immediately notified when a background process finishes. If `unset`, notification messages related to background process completion occur with the next presentation of the C Shell prompt. Use the `set` command to set notify.

$path

The *path* variable is one of the most important variables in C Shell operation. This variable contains a sequence of directory names C Shell searches for commands. For example:

```
set path=(/bin/posix /bin /usr/bin /usr/local/bin /usr/bin /etc .)
```

or

```
setenv PATH /bin/posix:/bin:/usr/bin:/usr/local/bin:/etc:.
```

PATH is a system variable, and `path` is a C Shell variable that serves the same purpose. The first is global, while the second is local to the running shell.

When C Shell is first executed, a hash table of command locations is created. This table is created by looking through the directories specified in $PATH (except for the current working directory) in the order specified by $PATH. Suppose you were to write one or more new commands and store them in one of the directories in your path other than your current working directory. The system has no way of knowing they are there until you notify it of their presence by using the `rehash` command. The hash table never gets built from commands in ., your current working directory, so new commands put into . don't affect the hash table. But, if your path contained `$HOME/tools/bin` and you put a new command in there, then rehash would add it to hash table. The hash table also gets re-built whenever your PATH is changed.

$prompt

This variable is used to customize your C Shell prompt. For example,

```
% set prompt = "[\!] % "
[22] % _
```

sets the prompt to indicate the command (event) number of the current command. This is very useful when using the History mechanism.

$shell

Some HP-UX commands, such as `mailx` and `vi`, spawn a new shell when they begin execution, while others may spawn one or more new shells during normal operation. If the program or command is written so that it recognizes the *$shell* variable, you can set the variable to define the type of shell to be spawned by the program. For example:

```
set shell = /bin/csh
```

selects C Shell, while

```
set shell = /bin/posix/sh
```

selects POSIX Shell.

This technique is valid only if the command or program recognizes that uses the variable when spawning new shells. Be careful when using the *shell* variable. The result may or may not be what you intended.

$status

This variable returns 0 if the most recently executed command was completed without error. A non-zero value means an error was detected.

Numeric Shell Variables

The at (@) command assigns a value to a numeric variable name, just as the `set` command assigns a string to a nonnumeric variable name. Numeric values can be decimal integers. For example:

```
[22] % @ sum=(1 + 4)
[23] % echo $sum
5
[24] % @ sum = (01 + 012)
[25] % !23
echo $sum
13
[26] %
```

Numeric Expressions

Numeric expressions evaluated by @ are very similar to those found in the C programming language. The syntax for this command is:

```
@
@ name = expression
@ name[index] = expression
```

The first form is equivalent to `set` (print csh variables).

The second form sets *name* to *expression*.

The third form sets the *index*th component of *name* to *expression* (both *name* and its *index*th components must exist).

Arithmetic Operators

In an expression of this type, the following C arithmetic operators are allowed:

`()`	Parentheses change the order of evaluation
`+`	Addition
`-`	Subtraction
`*`	Multiplication

`/`	Division
`%`	Remainder
`^`	Bitwise exclusive OR
`~`	Unary one's complement

Boolean Operators

The following boolean operators are allowed:

`==`	String comparison equal
`!=`	Boolean not equal
`!`	Exclamation point for negation

Furthermore, the following are also allowed but must be enclosed in parentheses, and their operands must be separated by white spaces, as in (*operand* `>=` *operand*).

`>`	Boolean greater than
`<`	Boolean less than
`>=`	Boolean greater than or equal
`<=`	Boolean less than or equal
`>>`	Right shift
`<<`	Left shift
`&`	Bitwise AND
|	Bitwise inclusive OR
`&&`	Logical AND
||	Logical OR

Assignment Operators

The following assignment operators are recognized:

`=`	Assignment
`+=`	As in `x += y` is the compressed form of `x = x + y`

`-=`	As in `x -= y` is the compressed form of `x = x - y`
`*=`	As in `x *= y` is the compressed form of `x = x * y`
`/=`	As in `x /= y` is the compressed form of `x = x / y`
`%=`	As in `x %= y` is the compressed form of `x = x % y`
`^=`	As in `x ^= y` is the compressed form of `x = x ^ y`

Postfix Operators

Finally, as a special case, `++` and `--` can be used as postfix operators to increment and decrement. Thus, the following statements give identical results:

```
% @ i++
% @ i = $i + 1
% @ i += 1
```

Note

The `++` and `--` operators do not require a `$` in front of the variable name.

Either of the following must appear alone on a line:

`@` *name*`++`
`@` *name*`--`

The following operators do not work:

```
&=
|=
<<=
>>=
```

File Evaluation

Expressions can also return a value based on the status of a file. If the specified file expression is *true*, the expression returns one (1). If *not true* then the expression returns a zero (0). It the file does *not* exist or is not accessible, the expression returns zero (0). The syntax for a file expression is:

-file_test filename

where `file_test` is selected from the list in Table 13-1.

Table 13-1. `file_test` Meanings

file_test	meaning
d	Is filename a directory?
e	Does filename exist?
f	Is filename a plain file?
o	Do I own filename?
r	Do I have read access to filename?
w	Do I have write access to filename?
x	Can I execute filename?
z	If filename empty (zero bytes long)?

An Example

The following example evaluates a list of filenames and returns their status. If the filename is a directory, the number of lines in it is also reported.

```
#!/bin/csh
# This script finds directories and lists the number of files
# in them and their word count.
#
foreach dir ($argv)
  set num = 0
  if ( -d $dir) then
    echo "***** $dir is a directory."
    set lsfile = `ls $dir`
    echo " number of file in $dir is $#lsfile"

    foreach file ($lsfile)
      set string = `wc -l $dir/$file`
      @ sum += $string[1]
    end

    echo "   total number of lines in $dir directory is $sum"

  else
    echo " ==> $dir is not a directory."
  endif
end
```

Now, execute the script called "find_dir":

```
[45] % find_dir src find_dir
***** src is a directory.
      number of files in src is 5.
      total number of lines in src directory is 3948
==> find_dir is not a directory.
[46] %
```

14

Commands, Jobs, and Scripts

Csh Commands

C Shell supports several "built-in" commands – commands that are normally executed within the current shell. If you invoke a command that is not a built-in C Shell command, a subshell is created (spawned) to handle its execution.

The alias Command

The `alias` command is used to assign new aliases and to show which aliases have been assigned. When executed without command-line arguments, all currently defined aliases are printed. If an argument is provided, the alias of that argument is printed. For example:

```
alias ls
```

shows the current alias, if there is one, for the directory list command `ls`.

The echo Command

The `echo` command prints its arguments to the shell's standard output file (unless redirected, the standard output is your display). `Echo` often used in shell scripts to print information about what is happening in the script. For example:

```
echo 'Your mail is sent. '
```

could be used in a mailing script to inform you that mail created by the script has been sent.

The history Command

The `history` command will show the contents of the history list. Numbers are assigned to each history event and can be used to reference previous events that may be difficult to reference using contextual mechanisms discussed previously.

The shell variable called `prompt` can be defined with an exclamation point (!) included in its definition so that the number being assigned by the history buffer is also displayed as part of normal terminal activity. This provides an easy way to reference previous commands and re-execute previous events. To set the `prompt` variable, use a command similar to the following:

```
set prompt='[\!] % '
```

Note that the '!' character had to be escaped here even though it was already enclosed between single-quote characters.

The logout Command

The `logout` command can be used to terminate a login shell which has `ignoreeof` set.

The rehash Command

The `rehash` command causes the shell to recompute a hash table of command locations. This is necessary if you add a command to a directory in the current shell's search path and want the shell to find it. Otherwise, the hashing algorithm cannot locate the command because it was not present in that directory when the hash table was originally computed.

The repeat Command

The `repeat` command can be used to repeat a command several times. For example, to make 5 copies of the file `one` in the file `five`, you could do

```
repeat 5 cat one >> five
```

The set Command

The `set` command with no arguments shows the value of all currently defined variables. For example:

```
[26] % set
argv      ()
cwd       /usr/xlf
history 15
home      /usr/xlf
cohorts   (bill john mike steve mary lars)
ignoreeof
noclobber
path      (/bin/posix /bin /usr/bin /usr/lib .)
prompt    [!] %
shell     /bin/csh
status    0
term      hp
[27] %
```

To set variables to specific values, use the `set` command with the appropriate variable names and arguments. Each of the variables shown in the preceding example were set initially by use of the `set` command.

Here is an example of how to set a variable equal to a list of string values or a set of numeric values.

```
[22] % set cohorts = (bill john mike steve mary lars)
[23] % echo $#cohorts
6
[24] % echo $?cohorts
1
[25] % echo $cohorts[3]
mike
[26] % unset cohorts
[27] % echo $?cohorts
0
[28] % set nums = (1.234 2 -3.45)
[29] % echo $nums[3]
-3.45
[30] %
```

The variable expansion sequence `$#` returns the number of elements in the variable array. The sequence `$?` returns a one (1) if the variable exists and a zero (0) if it does not.

The setenv Command

The `setenv` command is used to set environment variables whose values are global to the shell and any process it creates. For example:

```
setenv TERM hpterm
```

sets the value of the environment variable TERM to `hpterm`. See **environ**(7) in the *HP-UX Reference*.

The source Command

The `source` command can be used to force an update of the current shell environment by causing it to read commands from a file instead of standard input. For example:

```
source .cshrc
```

can be used after editing your `.cshrc` file to change any variables that you modified. Note that commands executed from the specified file are not placed in the history buffer; only "`source command_file`" is.

The time Command

The `time` command is used to determine how long execution of a specified command requires. When `time` is followed by a command name argument, the command is executed, then `time` displays user, system, and execution time for the command. If no argument is used with the `time` command, equivalent information about the current shell and any child processes it has created is printed instead. For example:

```
% time cp file1 file2
0.0u 0.1s 0:01 8%
% time wc file1 file2
      52  178  1347 file1
      52  178  1347 file2
     104  356  2694 total
0.1u 0.1s 0:00 13%
%
```

indicates that the copy command (`cp`) used a negligible amount of user time (`u`) and about 1/10th of a system second (`s`); the elapsed time was 1 second (0:01). The wordcount command (`wc`) used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage *13%* indicates that over the period when the command was active, it used an average of 13 percent of the available cpu cycles of the machine.

The unalias Command

The `unalias` command is used to remove aliases that have been assigned to the current shell. For example, if the alias command was used to cause the change directory command (`cd`) to print the working directory (`pwd`) each time it was called:

```
alias cd 'cd \!*;pwd'
```

then

```
unalias cd
```

cancels the assigned definition, and `cd` is again interpreted as the standard HP-UX command.

The unset Command

This command removes the values previously assigned to a variable by a `set` command.

The unsetenv Command

This command removes the specified variable(s) from the current environment.

Jobs

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single job is created by the shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are:

```
sort < data
ls -s|sort -n|head -5
mail harold
```

If the metacharacter '&' is typed at the end of the commands, then the job is run in the background, and C Shell returns immediately with a prompt, ready for another command. The job continues running to completion in the background while normal jobs, called foreground jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

runs the `du` program, which reports on the disk usage of your working directory (as well as any directories below it), puts the output into the file `usage` and returns immediately with a prompt for the next command without waiting for `du` to finish. The `du` program continues executing in the background until it is finished, freeing you and the terminal to execute more commands in the mean time. When a background job terminates, a message is sent to the terminal by the shell just before the next prompt telling you that the job is complete. In the following example, the `du` job finishes sometime during the execution of the mail command and its completion is reported just before the prompt that follows completion of the mail job.

```
%du > usage &
[1] 503
% mail bill
How do you know when a background job is finished?
EOT
[1] Done          du > usage
%
```

If the job did not terminate normally, the `Done` message might say something else, like `Killed`. If you want the terminations of background jobs to be

reported at the time they occur, possibly interrupting the output of other foreground jobs, you can set the `notify` variable. In the previous example, this would mean that the `Done` message might have come right in the middle of the message to Bill.

Jobs are recorded in a table inside the shell until they terminate. In this table, the shell remembers the command names, arguments and the process numbers of all commands in the job as well as the working directory where the job was started. Each job in the table is either running in the foreground with the shell waiting for it to terminate, or running in the background. Only one job can be running in the foreground at one time, but several jobs can run simultaneously in the background. Each job is assigned a job number as it starts. The job number can be used later in later references to the job, if needed. Upon completion, the job number is canceled and can be assigned by the system to another job.

When a job is started in the background using '&', its number, as well as the process numbers of all its (top level) commands, is printed by the shell before prompting you for another command. For example:

```
ls -s | sort -n > usage &
[2] 2034 2035
%
```

runs the `ls` program with the `-s` option, pipes the resulting output into the `sort` program with the `-n` option which places its output in the file `usage`. The `&` at the end of the line runs two pipelined programs as a background job. After starting the job, the shell prints the job number in brackets ([2] in this case) followed by the process number of each program included in the job. The shell then prompts for a new command as soon as the background job is underway.

To check to see what jobs are currently being run, use the `job` command. For example:

```
[42] % jobs -l    (lowercase L, not 1)
```

provides a list of current jobs and their corresponding job numbers, the commands being executed as part of each job, and the process IDs of each command. The "running" or "stopped" status of each job is also listed.

C Shell Scripts

Shell scripts are files containing a series of commands that the shell executes as a group. The files *.login*, *.cshrc* and *.logout* are all shell scripts.

When Not to Use a Script

While shell scripts are a valuable programming and operating aid, there are some situations where scripts are *not* useful. Many excellent commands and program libraries are provided with HP-UX. Before writing a script, check your *HP-UX Reference*. A solution to your problem may already exist.

Running a Script

A C Shell command script may be executed by typing in:

```
csh script_one arg_1 arg_2 ...
```

where `script_one` is the name of the shell script file to execute, and *arg_1 arg_2 ...* is a list of optional arguments that may be required by the script. C Shell places these arguments in the shell variable array `argv` as `argv[1]`, `argv[2]`, etc. There is no `argv[0]`. (C Shell uses `$0` to refer to `argv[0]` instead.) In this example, `$0` equals `script_one`. C Shell then begins to sequentially read the commands from `script_one`.

If you want to be able to execute the script file directly without beginning the command line with `csh`, edit the script file so that the first line is `#!/bin/csh`. The hash mark is also used for comment lines in the script.

Next, use the `chmod` command to make the file executable. For example:

```
chmod 755 script_one
```

makes `script_one` executable and readable for everyone and writable by you. For more information on the `chmod` command, see *chmod*(1) in the *HP-UX Reference*.

Now, when you type:

```
script_one
```

C Shell automatically executes the shell script file `script_one`. If the first line in the file is not `#!/bin/csh`, the Bourne shell will attempt to execute the shell script file instead.

Script Execution

C Shell parses each shell script line into command arguments. Each distinct command is identified, and *variable substitution* is performed. Keyed by the dollar sign character (`$`), this substitution replaces the names of variables by their values. Thus

```
echo $sum1
```

when placed in a command script, echoes the current value of the variable *sum1* to the shell script's standard output file. An error results if *sum1* has no value assigned.

To discover if a variable has a value currently assigned to it, use the notation

```
$?sum1
```

The question mark (`?`) causes the expression to return the value one (1) if the variable has a currently assigned value and zero (0) if not. This is the only available method for accessing a variable that does not have an assigned value without generating an error.

To determine how many component variables have been assigned to a variable, use the notation

```
$#sum1
```

The hash sign (`#`) notation returns the number of component variables assigned to the specified variable. For example,

```
set sum1=(a b c)
echo $?sum1
1
echo $#sum1
3
unset sum1
echo $?sum1
0
echo $#sum1
```

```
Undefined variable: sum1
%
```

You can readily access the individual components of a variable that has several assigned values. Thus

```
echo $sum1[1]
```

echoes the first component variable of *sum1*. In the example above `a` is echoed. Similarly

```
echo $sum1[$#sum1]
```

returns the component variable *sum1*, which is "c".

```
echo $sum1[1-2]
```

returns both `a` and `b`. Other notations useful in shell scripts include:

`$`*n*

(where *n* is a number), which is a shorthand equivalent of

`$argv`*n*

and returns the *n*th component variable of *argv*. Another is:

`$*`

which is a shorthand for

```
$argv
```

One minor difference between `$n` and `$argv[`*n*`]` should be noted. `$argv[`*n*`]` will yield an error if *n* is not in the range `1` through `$#argv` while `$`*n* will never yield an out-of-range subscript error. This is for compatibility with the way other shells handle parameters.

One way to avoid this type of error is to use a subrange of the form *n*`-`. If there are fewer than *n* component variables for the given variable, an empty vector is returned. A range of the form *m*`-`*n* also returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is within range.

The form

```
$$
```

expands to the process number of the current shell. Each process is unique, so the process number can be used to generate unique temporary file names.

The form

```
$<
```

is replaced by the next line of input read from the shell's standard input, instead of using the next line in the script being processed. This is useful when writing interactive shell scripts. For example,

```
echo 'yes or no?'
set a=($<)
```

would write the prompt `yes or no?` to the shells standard output device and then read the answer from the shells standard input device into the variable `a`.

Note You need the single quotes or "no?" expands to all files that start with "no" and have a single character after the "o".

Shell Script Expressions

Construction of useful shell scripts requires that it be possible to evaluate expressions in the shell based on the current values of certain variables. In fact, most C language arithmetic operations are available in the shell with the same precedence that they have in C. In particular, the operations `==` and `!=` compare strings, while the operators `&&` and `||` implement the boolean AND/OR operations. The special operators `=~` and `!~` are similar to `==` and `!=` except that the string on the right side can have pattern-matching metacharacters (like `*.?` or `[ ]`) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file inquiries of the form

`-?` *filename*

where `?` is replaced by a number of characters.

For example the expression primitive

`-r` *filename*

tells whether the file `filename` exists and is readable. The expression is *TRUE* if filename exists.

Other primitives test for read, write and execute access to the file, whether it is a directory or ordinary file, and test for non-zero length. See *test*(1) in the *HP-UX Reference* for specifications of these primitives.

You can determine whether a command terminated normally by enclosing it in braces (`{}`).

`{` *command* `}`

This notation returns a one (1) if the command terminated normally with exit status 0, or a zero (0) if the command terminated abnormally or with a non-zero exit status. If more detailed information about the execution status of a command is required, the command can be executed and the system variable `$status` examined in the next command. Remember, however, that `$status` is set by every command, so it is very transient.

As an example using the normal termination condition, consider the following command line:

```
if ({ date }) then; echo OK; endif
```

or, similarly, the shell script:

```
#!/bin/csh
if ({ date }) then        prints the date
  echo OK                 prints OK
endif
```

For a complete list of expression components available for shell scripts, see *csh*(1) in the *HP-UX Reference*.

Shell Script Control Structures

Control structures allowed by C Shell are similar to those in the C programming language.

Comments (#)

Comment your script using the hash mark (#) at the beginning of each comment line or command line that is to be ignored during execution.

The `foreach` Command

The syntax for this statement is:

```
foreach index_variable ( loop_count_value_list )

    command_1
    command_2
      .
      .
      .

 end
```

All of the commands between the foreach line and its matching end line are executed for each value in `loop_count_value_list` . The variable `index_variable` is set to the successive values of `loop_count_value_list`.

Within this loop, the `break` command can be used to stop loop execution, while the `continue` command can be used to prematurely terminate one iteration and begin the next. Upon completion of the for-each loop, the value of the iteration variable *index_counter* is the same as it was during the last loop in *loop_count_value_list*.

The if-then-endif Command

This command has the following syntax:

```
if ( expression ) then
    command_1
    command_2
      .
      .
      .
```

```
endif
```

Keyword placement is not flexible here due to current shell implementation. That means the control structure has to be *exactly* as shown. In other words, `if` and `then` must be in the same line and `endif` must be in a separate line.

You can nest these statements using the keyword `else`. For example:

```
if ( expression ) then
   command_1
   command_2
      .
      .
      .
else if ( expression ) then
   command_A
   command_B
      .
      .
      .
else
  command_X
  command_Y
               .
               .
               .
endif
```

Note that only one `endif` is used to end the entire structure.

C Shell has another form of the if statement:

```
if ( expression ) command
```

can be written

```
if ( expression ) \
      command
```

If you only need to execute one command, the `endif` statement can be omitted. In the second example, the non-printing newline character is escaped

with the backslash (\) to allow the command to appear below the expression. This is to improve visual clarity.

The while Command

The while structure is like that found in the C programming language. For example:

```
while ( expression )
    command_1
    command_2
      .
      .
      .
end
```

The switch Command

The switch structure is like that found in the C programming language. For example:

```
switch ( word )
case strl:
        commands
              .
              .
              .
        breaksw
case strn:
        commands
              .
              .
              .
        breaksw
default:
        commands
              .
              .
```

```
        .
    breaksw
endsw
```

Note C programmers should note that the `switch` command uses breaksw to exit and not break. While and foreach loops allow break.

The goto Command

C Shell allows the goto statement with labels, just like C.

```
loop:
  command_1
  command_2
        .
        .
        .
  goto loop
```

Supplying Input to Commands

By default, commands run from shell scripts use the standard input of the shell which is running the script. This is different from how other shells run under HP-UX. This allows C Shell shell scripts to fully participate in pipelines, but extra notation is required for commands which use in-line data.

Thus we need a metanotation for supplying in-line data to commands in shell scripts. For example, consider this script which runs the editor to delete leading blanks from the lines in each argument file. (The <*space*> and the <*tab*> in the example represent the space and the tab characters.)

```
#deblank - - remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/^[<space><tab>]*//
w
q
'EOF'
end
```

The notation `<< 'EOF'` means that the standard input for the `ed` command is to come from the text in the shell script file up to the next line consisting of exactly `EOF`. The fact that the `'EOF'` is enclosed in single-quote characters causes the shell to *not* perform variable substitution on the intervening lines. In general, if any part of the word following the `<<` which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form `1,$` in our editor script we needed to insure that this `$` was not variable substituted. We could also have insured this by preceding the `$` here with a `\`, that is:

```
1,\$s?[]*//
```

but quoting the `'EOF'` terminator is a more reliable way of achieving the same end.

Catching Interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. To do this, start your script with

`onintr` *label*

where `label` is a program label marking the code that handles the interrupt condition. If an interrupt is received by the shell, C Shell will do an automatic

`goto` *label*

and execute the desired code. If we wish to exit your program with a non-zero status, make

```
exit 1
```

a part of your interrupt handling code.

An Example Shell Script

This script backs up a list of C programs only if they are different from previously backed up versions. The files are stored in your home directory in the subdirectory `backup`. It makes use of the `foreach` statement to execute all of the commands between the `foreach` statement and its matching `end`. The script might be invoked with: `bkupscript *.c`.

```
#!/bin/csh
foreach i ($argv)
  if ($i !~ *.c) then
if the file is not a .c file, then
    echo $i is not a .c program
print an error message and
    continue
continue with the next file
  else
    echo $i is a .c program
  endif
  echo check file ~/backup/$i:t
```

```
# the ':t' takes the tail part of the original file name
     if(! -r ~/backup/$i:t) then
# if the file is not in the backup directory, then
        echo $i:t not in backup...not cp\'ed
# print an error message and
        continue
# continue with the next file
     endif
     echo compare two files $i and ~/backup/$i:t
     cmp -s $i ~/backup/$i:t
# compare the two files
     if ($status != 0) then
# if the file has changed,
         echo making new backup of $i
         cp $i ~/backup/$i:t
# make a new copy of it in the backup directory
     endif
   end
```

Index

Part III
POSIX and Korn Shell

- Introducing the Shells
- Starting and Stopping the Shell
- Shell Grammar
- Aliasing: Abbreviating Commands
- Substitution Capabilities
- Command-lines and Command History
- Basic Shell Programming
- Controlling Jobs
- Advanced Concepts and Commands
- Command Reference

15

Introducing the Shells

This tutorial describes the POSIX Shell (`/bin/posix/sh`) and the Korn Shell (`/bin/ksh`). POSIX Shell is based on the standard, IEEE P1003.2. Korn Shell is the command interpreter (human interface) written by David Korn at AT&T Bell Laboratories. The tutorial covers the similarities and the differences between these two shells, as well as the new features introduced in the POSIX Shell.

This tutorial addresses new users who are just learning shells, as well as advanced users who are Bourne and C Shell experts. New users should read all of this tutorial. However, at the beginning of each chapter advanced users will be directed to only the areas they need to concentrate on to come up to speed quickly. Both classes of users should finish reading this section.

What is a Shell?

The **kernel** is the supervisory part of the HP-UX operating system that keeps track of and assigns system resources to each user. It also coordinates I/O operation, and performs other software functions. A **shell** is a program that acts as an interface between the system and each user. The shell interprets user commands, then performs system calls to the kernel or loads and runs programs according to the commands given by the user. See Figure 15-1.

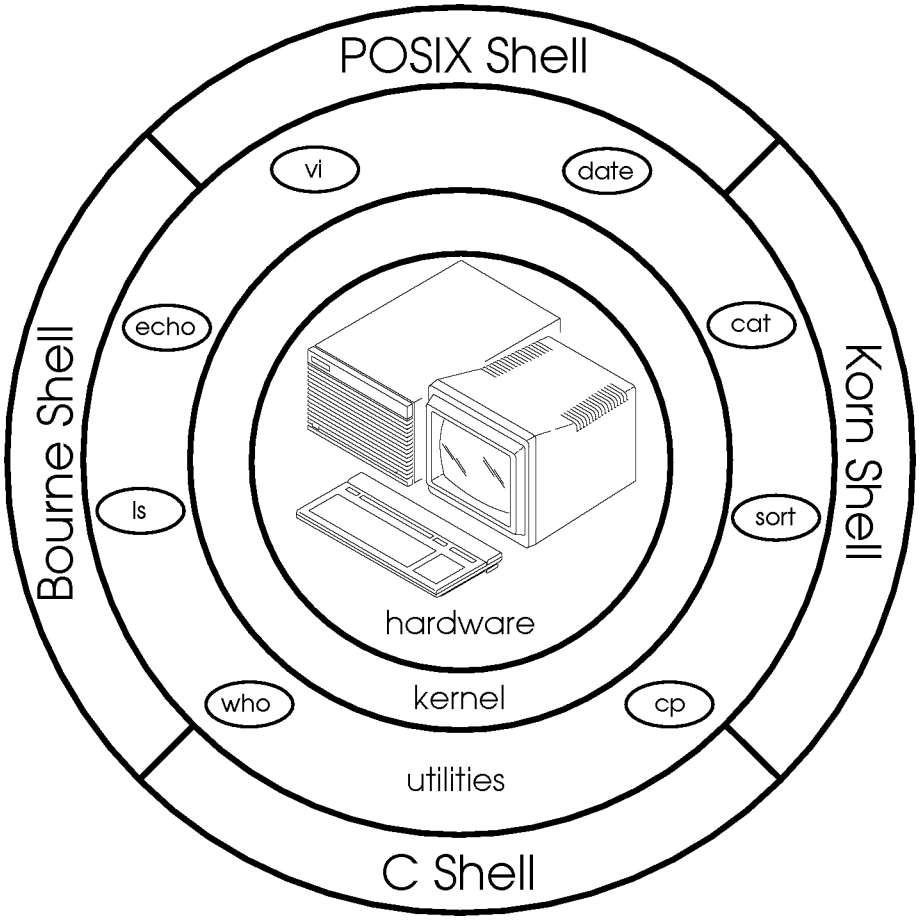

Figure 15-1. System Structure

POSIX and Korn Shell Versus Other Shells

The POSIX and Korn Shell programs are command interpreters and programming languages that execute commands entered from a terminal or file. They are based on features found in the Bourne Shell and C Shell. POSIX and Korn Shell maintain Bourne Shell's superior programming environment while adding the unique command interpreting features of C Shell. They are 95% upward compatible with the Bourne Shell, and most programs written under the Bourne Shell will run under the POSIX and Korn Shells without change. In developing POSIX and Korn Shell some of the features of the C Shell were also incorporated. This blend creates a powerful shell with improved human interface features and faster execution performance.

Features From C Shell

Some POSIX and Korn Shell features similar to C Shell features are:

- History buffer and history substitution capabilities.
- File name completion.
- Command aliasing mechanisms.
- Arrays.
- Integer arithmetic evaluation.
- Tilde substitution.
- Job control features.
- Noclobber for 8.0 ksh88.

Differences from Bourne Shell

The POSIX and Korn Shells are supersets of the Bourne Shell and contain all of the Bourne Shell's syntactic constructs, and almost all of its semantics. However, POSIX and Korn Shell implement some features above and beyond those in the Bourne shell. They are:

15

- The `select` and `function` statements.
- Built-in commands such as `alias`, `bg`, `fg`, `jobs`, `unalias`.
- Operators such as `(( ... ))` and `>|`.
- Reserved words such as `function`, `[[` and `]]`.
- Substring expansions such as `${`*name*`#`*pattern*`}` and `${`*name*`%`*pattern*`}`.
- Expansion of position parameters greater than `9`, `${`*digits*`}`.
- Command substitution syntax `$(`*command*`)`.
- Assigning values with `readonly` and `export`.
- Symbolic names for signals and traps.
- IFS variable is effective only for `read`, results of parameter expansions and command substitution. It is always initialized.
- Environment variable is passed to child process even if it is not exported.
- `for`, `while`, and `until` loops are executed in the current process environment. Assignments made within loops remain effective even after the loop completes.
- Traps defined in functions are local to the function. Errors in functions abort the functions but not the script.
- New shell variables such as REPLY, PPID, EDITOR, and OLDPWD.
- Extended `vi` and `emacs` in-line editing commands.
- Increased capabilities for parameter substitution.
- Job Control.
- Co-processors (Only in Korn Shell).

Differences between POSIX Shell and Korn Shell

Most of the POSIX Shell constructs are similar to those of the Korn Shell. The differences and the new features are highlighted wherever applicable. Some of the basic differences between the POSIX and Korn Shells are as follows:

- POSIX Shell does the function lookup before the built-in commands. Korn Shell does the built-in command search before the function lookup.
- POSIX Shell has a new built-in utility called `command` that executes commands without doing the function lookup.
- `time` keyword is removed from the POSIX Shell so as to use the POSIX-compliant `/bin/time` utility.
- `readonly` and `export` built-in commands print the variables, such that the output is suitable for re-input to the shell as commands.
- `unalias` has an option to remove all the alias definitions from the environment.

Definition of Terms

In learning this shell certain terminology is used to describe commands and arguments. If you don't know the following terms and definitions, becoming familiar with them will help you understand later descriptions.

15

argument and **parameter** — the words following a command or program name used to pass information to that command or program.

- an argument is *given* to a command. In the example, `lp file1`, `file1` is the argument and `lp` is the command.
- a parameter is something that has a value (possibly null or empty). In the example, `X=12`, `X` is a parameter. "variable" and "parameter" are usually interchangeable.

blank — a tab or space character. Sometimes called a whitespace character.

command — is a subset of **word**. A word is a command if it is the first word in a command line that is not a redirection or variable assignment.

metacharacter — One of the following characters: `; & ( ) | < >` new-line, space, and tab.

options or **flags** — a letter preceded by a dash (`-`) and separated from the command name by a blank. For example:

```
-v
```

word — a sequence of characters separated by one or more non-quoted metacharacters or whitespace. For example: `date`. The five types of words the shell understands are: reserved words (such as `for`), built-in command names (such as `pwd`), alias (such as `type`), functions, and utility names (such as a path name).

simple-command or **command-line**

a sequence of blank-separated words which may be include options and parameters. The first word specifies the name of the command to be executed. For example:

```
cat -v filename
```

A command-line may contain many simple commands. In the command-line, `cat file; rm file; who`, the three simple commands are `cat file`, `rm file`, and `who`.

identifier or **name**

a sequence of letters, digits, or underscores starting with a letter or underscore. Identifiers are used as names for aliases, functions, and named parameters. For example:

```
new_program_1
```

pipeline

a sequence of one or more commands separated by the metacharacter `|` which is called a pipe. For example:

```
ls | file_list | print_script
```

list

a sequence of one or more pipelines separated by `;`, `&`, `&&`, or `||`, and optionally terminated by `;`, `&`, or `|&`. For example:

```
(sort -o temp; pr temp | lp; rm temp)&
```

Conventions

The following conventions are used throughout this tutorial.

- *Italics* indicate manual names and references to manual pages in the *HP-UX Reference*. For example, "see *date*(1) in the *HP-UX Reference*". Italics is also used for symbolic items representing parameters or variables typed by the user. Italics is also for *general emphasis*.
- **Boldface** is used when a word is first introduced or defined.
- `Computer font` indicates a literal that must be typed exactly as shown, or text as it is displayed by the system. For example:

  ```
  findstr prog.c > prog.str
  ```

  Note, when a command or file name is a literal, it is shown in computer font and not italics. However, if the command or filename is symbolic (but not literal), it is shown in italics as shown here:

  `alias` *new_command*`=`*command_line*

  In this case you would type in your own *command_line* and *new_command*. Computer font also indicates file names, HP-UX commands, system calls, subroutines, and path names.
- In a syntax statement, brackets ([ ]) designate optional parameters; ellipses (...) designate optional repetition of a word or parameter directly preceding them.

  In the following example, *path* is an optional parameter to the `cd` utility.

  `cd [` *path* `]`

- Environment variables such as EDITOR or PATH are represented in uppercase characters, an HP-UX convention.
- A keycap such as Return designates the pressing of that key. If the keycaps are connected by a hyphen, press the first key down and hold it while pressing the second key. For example:

  CTRL-b

- Unless otherwise stated, all references such as "see the *env*(1) entry for more details" refer to entries in the *HP-UX Reference*. If you cannot find an entry where you expect it to be, use the *HP-UX Reference* index.

Supplementary Information Resources

A valuable Korn Shell resource is the book by Morris I. Bolsky and David G. Korn, *The Kornshell Command and Programming Language*. You will also find useful, related, and supporting information in other other HP-UX documentation. References to these manuals are included, where appropriate, in the text.

- The *HP-UX Reference* contains the syntactic and semantic details of all commands and application programs, system calls, subroutines, special files, file formats, miscellaneous facilities, and maintenance procedures available on the HP-UX Operating System.
- *Using HP-UX* presents new users with information on:
  - how to log in to HP-UX, work with files and the directory structure, and send and receive mail,
  - the basic elements of what a shell is, how to use it, and what shells are available on HP-UX,
  - how to create, edit, and save files using the `vi` editor. Additional tutorial information for the `vi`, `ex`, `sed` and `awk` editors and processors are contained in *The Ultimate Guide to the vi and ex Text Editors* and in *Text Processing: User's Guide*.
- *Finding HP-UX Information* provides part numbers for, and brief overviews of the contents of, manuals available for working with the HP-UX operating system. *Finding HP-UX Information* is available online, as a Helpview Help module under *HP-UX 9.0 Operating System Help*.
- *Common HP-UX Tasks* is available online, as a Helpview Help module under *HP-UX 9.0 Operating System Help*. *Common HP-UX Tasks* covers many of the topics presented on paper in *Using HP-UX*.

16

Starting and Stopping the Shell

New users should read this whole chapter. Experienced users can skip to the section on "Setting Up *.profile* and *.kshrc*".

Getting Started

Login

When you log in on a system, a program called `login` determines whether your user name and password are correct by checking the file `/etc/passwd`. The `/etc/passwd` file is a special system file that contains a listing of all the valid users and encrypted versions of their passwords on a system. See your System Administrator for more details, or *passwd*(4).

Once you type the correct user name and password, the `login` program starts (**spawns**) a shell (POSIX Shell (`/bin/posix/sh`) is the default shell on newly delivered systems) so you can begin executing system commands. HP-UX supports three other shells: Bourne (`/bin/sh`), Korn Shell (`/bin/ksh`), and C Shell (`/bin/csh`).

Command Line

When the shell is ready for your next command, it displays a prompt on the terminal display screen. Commands are given to the shell by typing the command name followed by options and/or parameters (called **command arguments**) as appropriate for that command, and pressing the Return key. The default POSIX Shell, Korn Shell and Bourne Shell prompt is a dollar sign (`$`). The default C Shell prompt is `%`. Once you have typed a command line, the shell interprets and executes it. For example:

```
$ echo Welcome to Korn Shell
Welcome to Korn Shell
$
```

This example shows the `echo` command followed by its argument, a string of text that is to be displayed on (echoed to) the terminal. Output from the echo command appears on the next line: `Welcome to Korn Shell`. (Note that `echo` is a shell built-in and so no **process** is spawned. However, `/bin/echo` is a utility and will spawn a process and assign it a **process id** (process identifier).

The software for each shell is shipped with HP-UX and resides on your HP-UX file system:

- The POSIX Shell resides in `/bin/posix/sh`.
- The Korn Shell software resides under the directory `/bin` in the file `ksh`, that is, in `/bin/ksh`.
- C Shell resides in `/bin/csh`.
- Bourne Shell resides in `/bin/sh`.

The rest of this tutorial deals mainly with the POSIX and Korn Shells. If you do not understand the login process or the file directory system, or if you have questions about shell structure or interaction with the shell, read about these topics in *Using HP-UX*.

Invoking the Shell

Initially, when you log into an HP-UX system, a default shell (POSIX) is spawned for you. To determine if `sh` is your shell, type:

```
$ echo $SHELL
/bin/posix/sh
```

The `echo` command prints out the contents or value (specified by using `$` before a parameter name) of the `SHELL` variable. `SHELL` is set to the name of the current shell at login.

The rest of this section explains how to change over to the POSIX or Korn Shell on a temporary or permanent basis.

Running POSIX or Korn Shell from the Current Shell

If you would like to experiment with the POSIX or Korn shell until you gain expertise, invoke it by simply typing:

```
/bin/posix/sh
```

or

```
/bin/ksh
```

In this manner, you are starting another shell on top of your current shell, sometimes referred to as a **subshell**. If you run this shell from the Bourne Shell the prompt does not change. However, if you are in the C Shell when you invoke the POSIX or Korn Shell, the prompt changes from `%` to `$`, unless these prompts have been redefined. It is possible to redefine prompts; see `PS1` in Table 16-1, "Shell Parameters."

If you exit the shell, by typing:

```
$ exit
```

you must reinvoke it each time with the `sh` or `ksh` command. Other methods of exiting the shell are discussed in "Terminating the Shell".

Specifying Your Login Shell

To make `ksh` your permanent or default login shell, type:

```
chsh login_name /bin/ksh
```


where *login_name* is your user name. The `chsh` command (change shell) changes your default login shell set in the `/etc/passwd` file to `/bin/ksh`. Once you have changed shells, invoke the `ksh` shell by logging out and then back in.

From now on, whenever you login, `ksh` is your shell.

To make the POSIX Shell your permanent or default login shell, type:

```
chsh login_name /bin/posix/sh
```

Setting Environment and Shell Variables

Environment variables and shell variables (*parameters*) are set in the *.profile* and *$ENV* files. (The *$ENV* file is often, but not always, `.kshrc`.) These variables create part of the **environment** in which you work, such as your prompt string (`PS1`). **Environment variables** are shell parameters that are **global** and used by your shell to create a special environment for subshells and any commands you may invoke. This environment is active until you logoff. These **global** (or **exported**) environment variables can be seen and used by subshells and other subprocesses. **Shell variables** are shell parameters that are local to your login shell and not passed on to any subprocesses or subshells.

Setting Up .profile and .kshrc

When the POSIX or Korn Shell is your login shell, it looks for these following files and executes them, if they exist:

`/etc/profile`	This default system file is executed by the shell program and sets up default environment variables.
`.profile`	If this file exists in your home directory, it is executed next at login.

At *any* time - this includes login time - the POSIX or Korn Shell is invoked, it looks for the file referenced by the following shell variable, and executes it, if it exists:

`$ENV`	When you invoke the shell, it looks for a shell variable called `ENV` which is usually set in your `.profile`. `ENV` is evaluated and if it is set to an existing file, that file is executed. By convention, `ENV` is usually set to `.kshrc` but may be set to any file name.

These files provide the means for customizing the shell environment to fit your needs.

Setting up .profile

The shell script, `.profile`, sets your environment by defining commands, variables, and parameters at login. These values, if the variables are exported, are global and available to subshells and subprocesses. Here is an example `.profile` file:

```
PATH=/bin/posix:/usr/bin:/usr/lib:/bin:/users/mary/bin:.
MAIL=/usr/mail/mary
HOME=/users/mary
EDITOR=/usr/bin/vi
ENV='${START[ (_$- = 1) + (_ = 0) - (_$- != _${-%%*i*}) ]}'
START=~/.kshrc
TERM=hp2392
export ENV START EDITOR TERM PATH MAIL HOME

stty sane susp ^Z

if mail -e
then
   echo "You have mail."
fi

PS1="$ "
```

Each line of the example `.profile` file, except the `if` statement and the `stty` command, shows a POSIX or Korn Shell variable:

- `PATH` defines the search path for the shell to look up commands (executable programs or utilities) in the system file structure. Each directory in the path is separated with a colon (:). When a command is executed, the shell looks in each of the directories specified in $PATH to find the command. When you type `ksh`, the shell checks `/usr/bin` first and then `/usr/lib` and so on down the `PATH` line until it finds the directory where the `ksh` program resides. In this instance, `ksh` is found in the third directory, `/bin`.

- `MAIL` names the file in which your mail is delivered. The `if` statement checks whether new mail has arrived and notifies you.
- `HOME` sets your home directory to the directory where the shell places you when you execute the `cd` (change directory) command with no options. This is usually set automatically by the shell at login.
- `EDITOR` sets your default editor to the `vi` editor. Then whenever you need to perform in-line command changes, you immediately enter `vi` mode. If you have never used the `vi` editor, see *Using HP-UX*, or *The Ultimate Guide to the vi and ex Text Editors*.
- `ENV` is normally assigned to be *.kshrc*, to be executed whenever a shell is spawned. For example:

  ```
  ENV=~/.kshrc
  ```

  In this example `ENV` is directly set to *.kshrc* in your home directory. The `~` specifies your `HOME` directory (see "Tilde Substitution" in Chapter 19 for more details). If your *.kshrc* is very long and involved, spawning a new shell can take awhile. The `ENV` line displayed in the screen above, although complicated, causes the *.kshrc* to not be executed, unless you are in an **interactive** shell, and therefore quickly spawns a new shell. (For a complete explanation of this command line, the `START` command line, and interactive shells, see Chapter 23.)
- `TERM` sets the terminal type for which output should be prepared. (You should set this to the terminal type you are on.)
- The `export` command puts the values of these parameters in the environment (makes them global) so that subprocesses have access to them.
- The `stty` command sets terminal characteristics to the default (i.e., `sane`) values. You should also set `susp` to ^Z - that's CTRL-Z - so you can get job control.

This is an example of just one *.profile*. When you create your own *.profile* using an editor, you can set many different shell variables depending on how you want the environment set up. See Table 16-1, "Shell Parameters," later in this chapter for a listing of possible variables.

Setting up $ENV

This shell script sets values, such as path names and aliases. These values can then be accessed by shell subprocesses. A **$ENV** file may look like:


```
lg=/abbreviation/of/long/path/name
alias who='who | sort '
set -o monitor
trap "$HOME/.logout" 0
```

The first line of the example **$ENV** file sets an abbreviation for a long path name of a directory. The variable **lg** contains the long path name, and executing:

```
cd $lg
```

moves you to the **/abbreviation/of/long/path/name** directory. Note the dollar sign ($) placed before a parameter, in this case **$lg**, designates using the value of that parameter.

The **alias** command is explained in Chapter 18 and the **set** command is explained in Chapter 23. The **trap** command-line is explained in "Terminating the Shell". (The above example just gives you an idea of the types of things to put in a *$ENV* file.)

You can create your own *$ENV* file using an editor, but you must set the **ENV** variable to the name of that file in *.profile* or the system does not read it when invoking a new shell. For example, if you use **.kshrc** as your $ENV file, the line in your **.profile** file might look like:

```
export ENV=$HOME/.kshrc
```

The set Command

There is a command that displays current environment variables, the `set` command. If you type `set`, a listing similar to this is displayed:


```
$ set
EDITOR=/usr/bin/vi
ENV='${START[ (_$- = 1) + (_ = 0) - (_$- != _${-%%*i*}) ]}'
START=~/.kshrc
FCEDIT=/bin/ed
OLDPWD=/usr/bin
HOME=/users/mary
IFS=

HISTFILE=.sh_history
HISTSIZE=30
LOGNAME=mary
MAIL=/usr/mail/mary
MAILCHECK=600
PATH=/bin/posix:/bin:/usr/lib:/usr/bin:/users/mary/bin:.
PPID=29590
PS1=$
PS2=>
PS3=#?
PWD=/users/mary/ksht/man
RANDOM=15314
SECONDS=0
SHELL=/bin/ksh
START[0]=/users/mary/.kshrc
TERM=hp2392
TMOUT=0
TZ=MST7MDT
VISUAL=vi
```

For an explanation of each of these, see Table 16-1, "Shell Parameters," and Chapter 23. At this time it is not important that you understand each of these shell variables completely. These definitions will become clearer as you become familiar with the POSIX and Korn Shell.

Table 16-1. Shell Parameters

Parameter	Definition
`#`	Represents the number, in decimal, of positional parameters supplied to a shell script.
`-` *(dash)*	Represents the flags or options supplied to the shell, on invocation, or by other commands.
`?`	Represents the decimal value (exit value) returned by the last executed command.
`$`	Represents the process number of the last invoked shell. Note that it is *not* reset for parenthesis subshells.
`!`	Represents the process number of the last background process invoked.
`_` *(underscore)*	Represents the last argument of the previous command-line (for Korn Shell only).
`CDPATH`	The search path for the `cd` command.
`COLUMNS`	This parameter, when set, defines the width of the edit window for the shell edit modes (`vi`, `emacs`, `gmacs`) and for printing lists from the `select` command.
`EDITOR`	When the `VISUAL` parameter is not set and the value of this parameter ends in `emacs`, `gmacs`, or `vi`, then the corresponding `set -o` option is turned on. (See the `set` command in Chapter 23.)
`ENV`	If this parameter is set to a script's name, when a shell is invoked the script is executed by the new shell prior to going interactive.
`FCEDIT`	Specifies the name of the editor to use when the `fc` command is executed and the `fc` command does not designate an editor.
`IFS`	Internal Field Separators (usually space, tab, and new-line), which are used to separate command words during command or parameter substitution and when using the `read` command.

Continued on next page ...

Shell Parameters (continued)

Parameter	Definition
HISTFILE	This is set to the path name of the file to be used to store the command history. The default is `.sh_history`.
HISTSIZE	This is set to the number of saved commands accessible by the shell. The default size is 128.
HOME	The default for the `cd` command, which is your home directory.
LINES	When this is set to a value, that value determines the column length for printing lists created by the `select` command.
MAIL	If this parameter is set to the name of a mail file and the `MAILPATH` parameter is not set, then the shell tells you mail has arrived in the named file.
MAILCHECK	This parameter specifies how often (in seconds) the shell checks for the arrival of new mail. The default is 600 seconds.
MAILPATH	The colon (:) separated search path for `mail_file`s. The shell informs you of mail arriving in any file in the list within the time specified by `MAILCHECK`. If you follow each `mail_file` in the search path with a question mark (?), the message immediately following the `?` appears on the screen instead of the default message.
PATH	The search path for commands.
PPID	The process number of the parent of the current shell. If you execute `ps -f`, you will see this number under the `PPID` heading. Associated with this `PPID` is a `PID`, which is the current process number.
PS1	Defines the primary prompt string for a shell. The default is `$`. If you precede the `$` with the `!` character, the primary prompt string includes the number of the current command.
PS2	Secondary prompt string, by default `"> "` used on command or script continuation lines.

Continued on next page ...

Shell Parameters (continued)

Parameter	Definition
PS3	The prompt string used with the `select` command, by default `#?`.
PWD	The present working directory set by the last `cd` command.
OLDPWD	The previous working directory set by the last `cd` command.
RANDOM	This parameter generates a random integer when referenced.
REPLY	This parameter is set by the `select` and `read` commands when no arguments are supplied on the `select` command line. Instead, the `PS3` prompt is printed and the lines read from standard input are placed in `REPLY`.
SECONDS	Returns the number of seconds since the shell was invoked.
SHELL	The path name where the shell itself lives. This refers to the user's preferred shell.
TMOUT	If this parameter is set to a value greater than zero and you do not enter another command or Return within that number of seconds, the shell terminates.
VISUAL	When this variable is set and ends in `emacs`, `gmacs`, or `vi`, then the corresponding `set -o` option is turned on. (See the `set` command in Chapter 23.)

The following variables are set automatically at login:

`#`, `-`, `?`, `$`, `HOME`,
`PPID`, `PWD`, `OLDPWD`, `RANDOM`,
`REPLY`, `SHELL`, and `SECONDS`.

These variables are given default values in the default login script:

`PATH`, `PS1`, `PS2`, `PS3`,
`MAILCHECK`, `TMOUT`, and `IFS`.

Again, use the `set` command to check these values before editing or creating a `.profile` that changes them.

Terminating the Shell

There are two ways to exit: `exit` and CTRL-D (or whatever the EOF character is set to (see *stty*(1)). The `ignoreeof` flag turns CTRL-D on/off. The value of `ignoreeof` is assigned with the `set` command (i.e., by typing `set -o ignoreeof`). (See Chapter 23 for details). To determine the current value of `ignoreeof`, type:

```
set -o
```

This lists all currently defined variables and their values. For example:

```
$ set -o
Current option settings
allexport        off
bgnice           off
emacs            off
errexit          off
gmacs            off
ignoreeof        on
interactive      off
keyword          off
markdirs         off
monitor          off
noexec           off
noglob           off
nounset          off
protected        off
restricted       off
trackall         on
verbose          off
vi               off
viraw            off
xtrace           off
```

In this example, `ignoreeof` is `on`. If `ignoreeof` is `off`, you must type `exit` to terminate the shell.

Using exit

16

Normally, you logout using `exit` or CTRL-D. If you try to logout using a CTRL-D with `ignoreeof` set, the system responds:

```
Use ‘exit‘ to logout.
```

To log out, type:

```
$ exit
```

If `ignoreeof` is not set, use CTRL-D or `exit` to logout.

Executing a `.logout` Script

If you want some special action to occur when you logout, use the `trap` command. Although not all traps are generated by a signal, for example, DEBUG, EXIT, `trap` can capture a signal, then execute a predefined command. This means you can set a trap on some condition, then the action will take place when the condition arises.

If a file named `$HOME/.logout` (a file named `.logout` in your home directory) exists, and the following `trap` statement is in your `.profile`, `.logout` is executed when you logout.

Add this to `.profile`:

```
trap "$HOME/.logout" 0
```

This `trap` statement causes the shell to execute the `.logout` script in your `HOME` directory when the shell exits. `$HOME` evaluates to the value of `HOME`.

For details on the `trap` command see Chapter 23.

Your `.logout` script might contain things like a `clear` command that clears the terminal's screen, and an echoed message, as in the following:

```
clear
echo "Have a Nice Day".
```

17

Shell Grammar

Certain characters or combination of characters in POSIX or Korn Shell have special meanings. (A small class of these special characters are called **metacharacters**. Metacharacters have meaning to the shell, other than as normal characters.)

New users should read this chapter completely; experienced users need only read the "Two-way Pipes" section for the token, `|&`, formed from two metacharacters.

Using Pipes

Pipes are connectors that join two or more programs or commands together. A pipe allows you to take the output of one program and use it as input to another program without the use of intermediate files.

The metacharacter for the pipe is the vertical bar (|). For example, suppose you want to list all the current users logged into the system and then alphabetically sort them and print them out. The command line reads:

```
who | sort
```

In the following example, a list of people logged into a system is produced by the `who` command. That output is sent as input into the `sort` command which outputs the sorted list of people on the system to the display. For example:


```
$ who
michael    tty02        Oct  4 14:49
dave       tty03        Oct  4 14:49
mary       tty00        Oct  4 13:34
george     tty04        Oct  4 14:49
keith      tty05        Oct  4 14:49
$ who | sort
dave       tty03        Oct  4 14:49
george     tty04        Oct  4 14:49
keith      tty05        Oct  4 14:49
mary       tty01        Oct  4 13:34
michael    tty02        Oct  4 14:49
```

Two-Way Pipes

Two-way pipes or **co-processes** can be established between the shell and a job. The **parent** process is the original shell and the **child** process (or subprocess) is the job, the command or shell spawned from the parent shell.

The standard input and output of the spawned command can be written to and read from the parent shell in a two-way pipe. A two-way pipe is created by placing the `|&` metacharacter after the command to be executed. See Chapter 23 for details on two-way pipes.

Command Separators and Terminators

Certain metacharacters are used by the shell to either separate or terminate commands in a line as well as perform special functions to the shell. For example:


```
date; ls &
```

where `;` is the separator and `&` is the terminator. The `date` command prints out the current date and the `ls` command lists the files in the current directory.

Table 17-1, "Separating and Terminating Characters," describes each of the special characters used by the POSIX and Korn Shells. For each special character, there is an example command-line followed by that example's output (when possible). Some output is based on the files existing in the current directory; so your output will not match exactly the output shown in the examples unless you create the files.

The examples in this chapter use the following commands:

Command	Definition
`cat`	concatenates, copies or prints files
`date`	prints the current date
`echo`	prints the arguments that follow the command
`ll`	prints a long listing of detailed information about files
`lp`	sends files to the printer
`ls`	lists the files in the current directory
`mail`	reads your mail or sends mail to another user
`more`	prints a file out for viewing on the display
`ps`	lists your current processes
`who`	lists the people logged into the system
`whoami`	prints the current user's name

Table 17-1. Separating and Terminating Characters

Character	Example	Description

;	`$ whoami; ls`
	`george`
	`file1`
	`file2`
	`file3`
		Separates commands that are executed in sequence. In this example, `ls` is executed only after the `whoami` command completes.

&	`$ lp prog.c &`
	`[1]  4094`
	`request id is lp-725`
	`$ echo hello`
	`hello`
		Indicates that the command is to be executed as a process *asynchronously*. That means you can run other commands immediately on the terminal while the previous command runs invisibly to you in the background. This sends the file `prog.c` to the line printer to be printed while freeing up your terminal for other work.

&&	`$ ls .kshrc && echo yes`
	`.kshrc`
	`yes`
		Separate commands such that the second command only runs if the first one runs successfully, that is, its exit status is 0. In this example, if the `ls` fails to find the file then the `echo` is not executed.

||	`$ mail || lsf`
	`No mail.`
	`file1`
	`file2`
	`file3`
		Separate commands such that the second command only runs if the first one fails, that is, its exit status is not 0. In this example, the `lsf` lists the files only if the `mail` command fails.

Name Completion

File Name Completion

POSIX and Korn Shell both implement the C Shell feature, file name completion. File name completion allows you to type a unique subset of letters or abbreviation for a file name or path name followed by (ESC) (ESC), and the system matches and completes the name. For example, suppose you have a file name `data_structure_3` for which you want a long listing, type:

```
$ ll data(ESC)(ESC)
```

The system responds with:

```
$ ll data_structure_3
-rw-rw-rw    1  mary   users   56  Sep 14 03:59 data_structure_3
```

It actually expands the name then executes the command upon receiving a (Return).

If you have several files starting with `data_structure_`, such as `data_structure_1`, `data_structure_2`, and `data_structure_3`, the name expands to the longest common prefix of all names that match. In this instance the change occurs as 1, 2, and 3, so the file name expands to just `data_structure_`. Now, type in 1, 2, or 3 to complete the file name. For example:

```
$ ls data(ESC)(ESC)
```

expands to:

```
$ ls data_structure_
```

If at this point you want to see all the possible expansions; type:

```
$ ls data_structure_(ESC)(=)
```

where (ESC)(=) lists three lines:

```
1) data_structure_1
2) data_structure_2
3) data_structure_3
$ ls data_structure_
```

17

This leaves you at the end of the line so you can complete the file name. To do so, type a and the appropriate letter or letters (1 in the above example) to complete the file name followed by a Return.

```
$ ls data_structure_a1Return
data_structure_1
$
```

This special mode for adding text to a command-line is explained in Chapter 20.

Another expansion character is *. The asterisk expands the current word to the entire list of file names that match. For example:

```
$ ls dataESC*
```

expands to:

```
$ ls data_structure_1 data_structure_2 data_structure_3
```

Another expansion feature allows you to interactively change your shell parameters, such as PATH. For example:

```
$ PATH=$PATHESCESC
```

expands to:

```
$ PATH=/bin/posix:/usr/bin:/usr/lib:/bin:/users/mary/bin
```

You can now edit and change the value of your PATH variable using the techniques described in Chapter 20.

Path Name Completion

This expansion process completes directory paths in a similar manner, such that:

```
$ ll /users/m(ESC)(ESC)
```

expands to:

```
$ ll /users/mary
```

This only works if you provide a unique identifier after the `/`.

File Name Substitution

File name substitution is a quick and easy way to match file names without typing the full name. File name metacharacters represent character patterns which are replaced with a matching file name pattern on execution of the command. Suppose you wanted to long list the file `data_structure_3` again; type:

```
$ ll data_structure_3
```

or use a metacharacter and type:

```
$ ll data_*
```

which matches any character or string of characters starting with `data_`. If there is more than one file starting with `data_`, they are all listed. Table 17-2, "File Name Substitution Metacharacters," lists, and gives examples using, the metacharacters used in pattern matching.

Table 17-2. File Name Substitution Metacharacters

Meta-character	Example	Description

?	`$ ls prog.?`
	`prog.a`
	`prog.c`
	`prog.o`
		Matches almost any single character. The `ls` command lists all files starting with `prog.` and ending with any other letter, such as `prog.c` and `prog.o`.

*	`$ ls p*.*`
	`p.o`
	`pattern.mat`
	`prog.a`
	`prog.c`
	`prog.a`
	`prz.2`
		Matches almost any string of characters including the null string. The `ls` command lists all files starting with a `p` and having a `.` anywhere in the middle or at the end.

[. . .]	`$ ls [a-z]rog.[co]`
	`arog.o`
	`crog.c`
	`prog.a`
	`prog.c`
	`prog.o`
	`zrog.c`
	`$ ls [a,z]rog.o`
	`arog.o`
	`zrog.o`
		Matches any of the characters enclosed in the brackets. A pair of characters separated by a minus matches any one character in the range specified by the two letters in the alphabet. In this example, `ls` lists any file starting with a lower-case letter of the alphabet, followed by `rog.`, and ending with either `c` or `o`. In the second `ls`, the comma performs the same way as the [co] does without a comma.

Note that neither `?` nor `*` match a leading period or a `/`. Also, for NLS there are additional constructs available. See *regexp*(5).		

Quoting

Each of the metacharacters discussed above can be quoted, or protected, to make it stand for itself and not be interpreted by the shell as a special character. Table 17-3, "Quoting Metacharacters," lists, and gives examples using, these metacharacters.

Table 17-3. Quoting Metacharacters

Meta-character	Example	Description
\	`$ ls prog.\*` `prog.*`	The backslash \ cancels the special meaning of the metacharacter that follows it. (Note that the backslash is not special inside single quotes.) The backslash forces `ls` to list the file actually named `prog.*`, not all files starting with `prog.`.

'	`$ echo '$PWD'`
	`$PWD`
	`$ echo '\$PWD'`
	`\$PWD`
		The single quote (') protects everything enclosed between two single quote marks except the single quote itself. That is, only the single quote can't be protected; all other metacharacters have no special meaning inside single quotes.

"	`$ echo "$PWD"`
	`/users/mary`
	`$ echo "\$PWD"`
	`$PWD`
		The double quotes allow parameter and command substitution. The \, inside double quotes, quotes the characters `\, ', "`, and `$` rather than the shell evaluating them. This example echos the path name contained in the variable `PWD`. When the \ is placed in front of the `$`, the `echo` cannot evaluate PWD.

Input and Output

Standard input (`stdin`) is the default place from which a program reads its input (the default `stdin` is the keyboard). (Note that a command can read input from anywhere it chooses, not just `stdin`.) **Standard output** (`stdout`) is the default place to which a program writes its output (the default is the terminal display). **Standard error**(`stderr`) is the default place where the system writes error messages (the default is the terminal display).

When a command is executed, its `stdin`, `stdout`, and `stderr`can be redirected using special **redirection** symbols. When you *redirect* input or output using redirection symbols, you place it somewhere other then the default areas, such as a file. To redirect `stdin` from a file, use the `<` symbol. To redirect `stdout` to a file, use the `>` symbol. To redirect `stderr` to a file, use the `2>` symbol. Table 17-4, "Input/Output Redirect Operators," lists, and gives examples using, the redirection characters.

Table 17-4. Input/Output Redirect Operators

Redirect Operator	Example	Description
< *word*	`$ mail joe < letter`	The less-than operator, <, redirects the contents of letter to input to *mail*.
> *word*	`$ ps > processes`	The greater-than operator, >, redirects output from `ps` into the *processes* file deleting any current contents.
>> *word*	`$ date >> processes`	The double greater-than operator, >>, redirects the output from `date` and appends it to the end of the *processes* file. If the file does not exist, a new one is created.

<< [-] *word*	`$ cat << eof`
	`> write`
	`> until`
	`> eof`
	`write`
	`until`
		The double less-than operator, <<, reads the shell input (typed after the `cat` command-line at the `PS2` prompts, >) up to a line which is identical to *word* (`eof`). *word* is not subjected to file name or parameter substitution. The resulting document is commonly called a **here document**. If - is appended to <<, all leading tabs are stripped from *word* and from the resulting document.

<&*digit* >&*digit*	`$ echo output 1>&2` `output`	This input redirection operator uses the file descriptor specified by the descriptor *digit*. Most programs have standard input as 0 (`stdin`), standard output as 1 (`stdout`), and standard error as 2 (`stderr`). The more commonly used redirection is >&*digit*. In the example, standard output (1) is redirected to standard error (2). The 1 is optional in this example.
<&- >&-	`$ echo no output >&-`	These operators close standard input and output, respectively.

The order in which you place redirections is significant. The shell evaluates each redirection in terms of the file descriptor associated with each file at the time of the evaluation. For example:


```
2>fname 1>&2
```

This command-line first associates the file descriptor 2 (`stderr`) with *fname*. This sends `stderr` (file descriptor 2) to the file *fname* instead of the terminal. It then associates file descriptor 1 (`stdout`) with the file associated with file descriptor 2 (which is *fname*). The `echo` command prints to `stdout`, but now `stdout` points to `stderr` which has been redirected to a file. This means both standard output and standard error are put in *fname*.

If you wanted to know what files in a large directory started with a particular letter, you might direct both `stdout`, in case files were found, and `stderr`, in case files were not found, to the same output file. In the following example, no files starting with the letter `g` were found, but there was one file starting with the letter `j`.

```
$ ls g* 2>fname 1>&2
$ ls j* 2>>fname 1>&2
$ more fname
g* not found
jg900401.hpg
$
```

Other Metacharacters

A few other metacharacters to be aware of are: `#`, `~`, and `%`.

- You can can insert comments into shell scripts by using the `#` symbol when it is the first character in a word. Any following words are treated as comments until a new-line occurs (e.g., `# This is a comment`). This is explained in "Commenting" in Chapter 21.
- The tilde substitution symbol `~` allows path name substitution and is explained in "Tilde Substitution" in Chapter 19.
- The `%` symbol allows job number substitution and is discussed in "Putting Jobs in Background/Foreground" in Chapter 22, and under `bg`, `fg`, and `kill` in Chapter 24.

18

Aliasing: Abbreviating Commands

Aliasing is a method by which you can abbreviate long command lines, or cause standard commands to perform differently by replacing the original command-line with a new command called an `alias`. The new command can be a letter or short word when typed, but will expand to the old command-line when used. Aliasing can provide easier typing by both abbreviating long command lines and automatic replacement of long path names.

Both new users and advanced users should read this chapter.

Setting an Alias

To create an alias for POSIX Shell, use this syntax:

`alias` [*word*[`=`*command*] ...]

To create an alias for Korn Shell, use this syntax:

`alias` [`-tx`][*word*[`=`*command*] ...]

Any time `alias` is followed by a word, the shell assumes you are defining a new alias or asking for the value of a defined alias. The *word* parameter specifies the new alias's name and the *command* specifies any command or long command-line. The `-x` option exports aliases and `-t` tracks aliases. These options are discussed in the following sections: "Tracking Aliases" and "Exporting Aliases".

For example:

```
$ alias who='who | sort'
```

redefines the original `who` command to the line enclosed in single quotes. Now, when you perform a `who`, you get a listing of all the users on the system sorted in alphabetical order.

If you type `alias` followed by `who` it returns the value of the new alias. For example:

```
$ alias who
who=who | sort
```

Suppose you want to use several aliased commands on one command-line. To do so, leave a space as the last letter in the alias definition. If the last letter is a blank, the word following the first alias is also checked for alias substitution. For example:

```
$ pwd
/tmp
$ alias hcd='echo hello; cd '
$ alias p=/users/george
$ hcd p
hello
$ pwd
/users/george
```

Since `cd` is followed by a space before the close quote, it can be followed by another alias, which is `p` in this case.

Now the command-line `hcd p` prints hello and changes the directory. The command-line actually executed is:

```
$ echo hello; cd /users/george
```

Tracking Aliases (for Korn Shell only)

Aliases can also be used to automatically set a command to its full path name the first time it is executed after login. This reduces the execution time needed to search for a command's location in the system directory on all later calls to the command. This ability is called **tracking**.

The value of a tracked alias is defined the first time the alias command is executed and the shell searches for the command's path.

Suppose you execute the `ls` command, yet, you never actually set an `ls` alias. The `ls` command is automatically tracked. Now list your tracked aliases; type:

```
$ alias -t
ls=/bin/ls
```

The `ls` command shows up as a tracked alias without the need of setting it with the alias command.

If you want every valid alias and trackable command name tracked, use the `set -h` command interactively or in the file specified by your ENV variable. Not all commands are trackable. Built-in commands, such as `cd` or `pwd` command are not trackable; although they are aliasable. This option is turned on automatically for non-interactive shells. See Chapter 23 for details on the `set` command.

If the PATH variable is changed while you are in the shell, either interactively or by rerunning your `.profile` or `.kshrc`, then the tracked alias definitions set are lost until you execute each command again.

Exporting Aliases (for Korn Shell only)

Exporting aliases works in much the same way as exporting variables with `export`. But, `ksh` will only export an alias to another shell that is *not* a separate invocation of `ksh` (an exported alias will survive a *fork*(2), but not an *exec*(2)). Exported aliases are available to subshells, for example, `(` *prog* `)`, and to shell scripts that do *not* start with `#!/bin/` ... .

You `export` aliases interactively or from within your `.profile` or `.kshrc`. To do so type, or add to the appropriate file:

```
``alias -x who='who | sort' ''
```

18

Then, when you type `alias` or `alias -x`, `who=who | sort` is shown.

Default Aliases

The shell provides several default aliases that are always set by the shell. To see a listing of those defaults, and any other aliases currently defined, type:

```
$ alias
```

As long as this command is typed by itself, with nothing following, it provides a list of the current shell aliases. Something similar to the following is returned:

```
$ alias
false=let 0
functions=typeset -f    (applies to Korn Shell only)
hash=alias -t
history=fc -l
integer=typeset -i      (applies to Korn Shell only)
nohup=nohup
r=fc -e -
true=:
type=whence -v
```

where `false` is the first word (the alias name) and `let 0`, on the other side of the `=` sign, is the value of the alias. Then, when the alias name `false` is used, it is replaced by the assigned value of the alias `let 0`. The `let` command is used for arithmetic evaluation and is explained in Chapter 23.

In the next example, which applies to Korn Shell only, the first word of a command-line `integer` already has an alias defined by the system (`typeset -i`, as shown in the previous example). The function of the `typeset` command is to create a value and type for a parameter. When you type:


```
$ integer val=1
$ echo $val
1
```

`typeset -i` is substituted for `integer` and `val` is created and given the value `1`. In this example:

```
typeset -i val=1
```

is what is actually executed. When you `echo` the value of `val` using the `$` metacharacter, you see it was given the value `1`. For more details on the `typeset` see Chapter 23.

When you create an alias in Korn Shell and then execute it, the shell adds it to the table of aliases. If you type `alias`, you see additions to the list:

```
hcd=echo hello; cd
false=let 0
functions=typeset -f
hash=alias -t
history=fc -l
integer=typeset -i
nohup=nohup
p=/users/george
r=fc -e -
true=:
type=whence -v
who=who | sort
```

Special Aliasing Features

Several things you should keep in mind when defining aliases are:

- Unlike the C Shell, you can alias the `alias` command. For example,

  ```
  $ alias a=alias
  ```

  In this example, whenever you use `a` an alias is created such that

  ```
  $ a who='who | sort'
  ```

  will set `who` to an alias.

- Reserved keywords, such as `while`, `do`, and `done`, cannot be changed by aliasing.

- The first character of an alias name can be any non-special printable character, but the following characters must be alphanumeric,

  ```
  $ alias @w='who | sort > user'
  ```

  Here `@w` now checks who is on the system, sorts the names and places them into a file called `user`.

- The replacement value on the right hand side of the aliasing `=` sign can contain any valid shell script name, or the script itself. For example:

  ```
  $ alias i='
  > echo Users logged in are:
  > who | sort
  > echo I am `whoami`'
  $
  ```

If you execute the alias, you will get something like the following:

```
$ i
Users logged in are:
mary       tty2        Sep 24 14:19
michael    tty4        Sep 24 09:41
nick       tty1        Sep 24 09:41
I am mary
$
```

Creating scripts is described in Chapter 21.

- Aliases take effect only after the `alias` command has been executed. If you try to run a script or command-line which references an alias before the `alias` has been executed on it, the script or command will not run. The real concern is that you define and use an alias on the same line, such as in:

```
$ unalias x             #just to make sure
$ alias x="echo XXX"; x
ksh88: x: not found
```

At the time `x` was parsed, the alias was not yet defined. But if you continue the example ...

```
$ x
XXX
$
```

This time the alias worked because it is now defined.

Unsetting an Alias

There will be times that you set a common command such as `who` with a new definition and then decide you need its old functionality back. You can quote, or protect, the alias name to temporarily override the alias, for example, `\who`. Or you can permanently regain the old functionality by unsetting the alias. To unset an alias use the `unalias` command. In one of the previous examples `who` was set to `who | sort`. To unset `who`, type:

```
unalias who
```

Then, type `alias` and notice from the listing that `who` has disappeared from the alias list and now performs it original function. The results of running `who` before and then after should look something like this:

```
$ who
mary       tty02        Sep 24 14:19
michael    tty04        Sep 24 09:41
nick       tty01        Sep 24 09:41
$ unalias who
$ who
nick       tty01        Sep 24 09:41
mary       tty02        Sep 24 14:19
michael    tty04        Sep 24 09:41
$
```

The POSIX and Korn Shells default aliases (i.e., `false, integer, ...`) can be unset or redefined, as well. The POSIX Shell also provides the `-a` option with the `unalias` built-in command. This option can be used to remove all the alias definitions by typing, at the shell command line prompt:

```
unalias -a
```

19

Substitution Capabilities

Chapter 17 discussed file name substitution and completion. This chapter discusses the other substitution concepts: tilde, parameter and command. Substitution methods are used to speed up command-line typing and execution.

New users should read this chapter completely; advanced users should see "Tilde Substitution", "Parameter Substitution" and "Command Substitution" for special POSIX and Korn Shell features.

Tilde Substitution

If a word begins with a tilde (~), tilde expansion is performed on that word. Note that tilde expansion is provided only for tildes at the beginning of a word, that is, `find /~pbm/abc` has *no* tilde expansion performed on it.

Tilde expansion is performed according to the following rules:

- A tilde by itself or in front of a `/` is replaced by the path name set in the `HOME` variable.
- A tilde followed by a `+` is replaced with the value of the `PWD` variable. `PWD` is set by `cd` to the new, current, working directory.
- A tilde followed by a `-` is replaced with the value of the `OLDPWD` variable. `OLDPWD` is set by `cd` to the previous working directory.
- If a tilde is followed by several characters and then a `/`, the shell checks to see if the characters match a user's name in the `/etc/passwd` file. If they do, then the ~characters sequence is replaced by that user's login path.

These tilde sequences are demonstrated in the following example.


```
$ echo $HOME
HOME=/users/mary
$ echo ~
/users/mary
$
$ echo $PWD
PWD=/users/mary/tmp
$ls ~+/x*
/users/mary/tmp/x_file1
/users/mary/tmp/x_file2
$
$echo $OLDPWD
/users/mary/mail
$ls ~-/f*
/users/mary/mail/from.mike  /user/mary/mail/from.nick
$
$ls ~nick/share
bitmaps    formats    templates  tools
$ls ~nick/share/bitmaps
logo1       logo2       screendump
```

In the first three applications of the tilde, the value of the respective variables were first listed with `echo`. Then the relevant tilde approaches were used to get similar output. In the last application example, Nick's files were accessed to locate a screen dump in his shared source directory `share`.

All these directory changes assume the new directories exist or the shell will send errors such as:

```
ksh: /users/michael/test: bad directory
```

Tildes can be put in aliases:

```
$ pwd
/users/mary
$ alias cdn='cd ~/bin'
$ cdn
$ pwd
/users/mary/bin
```


In this example, when `cdn` is executed it places you in the `bin` directory in your `HOME` directory.

Parameter Substitution

A parameter is an entity that holds a value.

The two types of parameters discussed in this section are:

- *named* parameters, such as `HOME IFS`.
- *positional* parameters, such as `1`, `2`, `3`.

Each of these is described in detail in the subsequent sections.

Parameter substitution is the process the shell does to a command line when it replaces the parameters with their value, for example, changing:

```
echo $HOME
```

to

```
echo /users/mary
```

Then, after parameter substitution, the line is executed. The `echo` command *never* see the `$HOME`, it is the shell that does the substitution.

The value of a named parameter can be accessed by preceding the name with a dollar sign `$`.

```
$parameter
```

where the `$` specifies substitution of the value of the parameter. For example:

```
$ x=1
$ echo $x
1
```

This is a simple example of a parameter which is *named*, (`x`), that is assigned a value (1).

Setting and Using Keyword/Named Parameters

At the risk of sounding circular, a named parameter is a parameter with a name. The name may be any word consisting only of alphanumeric characters and the `_` (underscore), and beginning with an underscore or alphabetic character. `new_prog1` is a named parameter. The value of a named parameter can be set using the syntax:

name`=`*value*

For example:

```
$ x=1
```

sets the *name* to `x` with a *value* of 1.

Attributes of a parameter may be set with the typeset command. The `typeset` command has many options or attributes (such as readonly, integer, left justify) it can assign to each *name*. See Chapter 23 for details on these.

Setting and Using Positional Parameters

Positional parameters are passed to a command or shell script or set with the `set` command. The positional parameters follow the script or command name on the command-line. Then every item on the line following the command or script name, separated by a whitespace, is given a positional parameter name `0`, `1`, `2`, `3`, and so forth. These correspond directly to the items on the command-line; `0` is the first item (script name), `1` is the second item. This assignment process continues for the rest of the parameters on the line.

For example, the following function uses the `while` construct to shift through and print the parameters and their position on the command line. The `shift` command might also help you accomplish this task (see `shift` in Chapter 24).

```
$ function print_args
> {
>    typeset -i x=0        # set x=0; declare x integer
>
>    while [  x -le $# ]   # -le is "less than or equal"
>    do
>       echo "positional parameter $x is: $(eval echo \$$x)"
>       let x=x+1
>    done
> }
$ print_args A B C
positional parameter 0 is: print_arg
positional parameter 1 is: A
positional parameter 2 is: B
positional parameter 3 is: C
$
```

In the previous example the current shell is the Korn Shell. The POSIX Shell behaves differently compared to Korn Shell with respect to the `$0` positional parameter. The same example in the POSIX Shell prints

```
-sh
```

for the `$0` parameter (if the login shell is POSIX), or

```
/bin/posix/sh
```

for the `$0` parameter (if the login shell is not POSIX). This happens because within the POSIX Shell definition, the positional parameter `$0` is either the shell filename (if the shell is interactive) or is the script name.

The set command may be used to set the positional parameters for the shell.

```
$ set -- first second third
$ echo $1 $2 $3
first second third
$
```


This sets positional parameter 1 to `first`, positional parameter 2 to `second`, and positional parameter 3 to `third`.

Parameter Substitution Conventions

This section covers special conventions used during parameter substitution:

`${`*parameter*`}`	Braces are required when *parameter* is followed by a letter, digit, or underscore that you do not want be interpreted as part of the *parameter*'s name, for example, `ls /tmp/${`*file*`}_text`. Braces are also required for enclosing multi-digit positional parameters, for example, `${17}`.

`${`*parameter*`:-`*word*`}`	If *parameter* is set and non-null, its value is substituted; otherwise *word* is substituted. For example:
	
	`$ unset x`
	`$ echo ${x:-"x is unset"}`
	`x is unset`
	`$`

`${`*parameter*`:=`*word*`}`	If *parameter* is not set or null, the value is set to *word*'s value.
`${`*parameter*`:?`*word*`}`	If *parameter* is set and non-null, substitute its value; otherwise, print *word* and exit from the shell. When *word* is omitted a standard message is printed.
`${`*parameter*`:+`*word*`}`	If *parameter* is set and non-null, substitute *word*; otherwise substitute nothing.

Note For the above four substitutions with the colon (:), if the colon is omitted, the check for a null value is omitted.

```
$ x=""
$ echo ${x:- "x is null"}
x is null
$ echo ${x- "x is null"}

$
```

19

`${`*parameter*`#`*pattern*`}` `${`*parameter*`##`*pattern*`}`	If the Shell *pattern* matches the beginning of the value of the *parameter*, substitute the value of the *parameter* with the matching *pattern* removed; otherwise substitute the value of this *parameter*. In the first form the smallest matching *pattern* is deleted and in the latter form the largest matching *pattern* is deleted.
`${`*parameter*`%`*pattern*`}` `${`*parameter*`%%`*pattern*`}`	If the Shell *pattern* matches the end of the value of *parameter*, then the value of *parameter* with the matched part deleted is substituted; otherwise substitute the value of *parameter*. In the first form the smallest matching *pattern* is deleted and in the latter form the largest matching *pattern* is deleted.

These examples show how some of the parameter substitution techniques work. To get the basename of a path:

```
$ prog=/users/mary/prog
$ basename=${prog##*/}
$ echo $basename
prog
$
```

As you can see in this example, the pattern `/users/mary` is matched and then removed from the string.

To get the corresponding path prefix you could do the following:

```
$ prog=/users/mary/prog
$ prefix=${prog%/*}
$ echo $prefix
/users/mary
$
```


In this case, the smallest substring, starting on the right, of `/users/mary/prog` that begins with a `/`, that is, matches `/*`, is removed.

Special Parameters

`$@` or `$*`	If the *parameter* is `*` or `@`, then all the positional parameters, starting with $1, are substituted.
	
	For example, the script `sc` uses `$@` to `echo` all its parameters at one time:
	
	`$ sc first second third`
	`echo $@`
	`first second third`

`${`*array*`[*]}`	If the *parameter* describes an array with elements `*` or `@`, then the value for each of the elements is substituted. For example: `echo ${array[*]}` (An **array** is a collection of contiguous elements that can be accessed by a subscript. A subscript can also be metacharacters such as `*`, `$@`, `#`, and `$*`. For more details on arrays see Chapter 21.)
`${#`*parameter*`}`	The `#` specifies the number of the characters in the *parameter* is to be substituted. If *parameter* is `*` , the number of positional parameters on the command-line is substituted.
`${#`*array*`[*]}`	If the *parameter* is an array name with elements `*` or `@`, and the array name is preceded by `#`, the number of elements in the array is substituted.

Command Substitution

This substitution method is used to replace a command with its output within the same command-line. The standard syntax for command substitution, and the one encouraged by POSIX, is `$(` *command* `)`.

For example:

```
$ echo "The people currently logged on the system are:\n$(who)"
The people currently logged on the system are:
mary     console    Sep 11 09:01
michael  tty09      Sep 11 10:35
```

In this example, the `who` enclosed in `$()` is executed and printed out within the `echo` command-line. The `\n` provides a new-line.

The POSIX and Korn Shells implement this substitution capability using both the `$(`*command*`)` form and the back quotes form. For example:

```
$ echo "The people currently logged on the system are:\n`who`"
The people currently logged on the system are:
mary     console    Sep 11 09:01
michael  tty09      Sep 11 10:35
```

The second form of the command substitution, `` `command` ``, is the only form recognized by the Bourne Shell and should be used in scripts that may be run by POSIX, Korn and Bourne Shell.

However, using the first syntax simplifies nesting in substitution. For example:

```
$ echo $(echo $(echo hello))
hello
$ echo ‘echo \‘echo hello\‘ ‘
hello
```


Back quotes cause substitution of the output of the `echo` command and `echo` is repeated twice in the second command-line. The backslash cancels the second back quote from closing the `echo`. Therefore, the third `echo` is evaluated and outputs the `hello`. The first command-line performs the same function in a less complicated manner.

Any valid shell script may be put in command substitution. The shell scans the line and executes any command it sees after the opening quote or parenthesis until a matching, closing quote or parenthesis is found. For example:

```
$ echo "Users logged in on this date\n $(date; who)"
Fri Sep 11 16:43:34 MDT 1987
mary     console    Sep 11 09:01
michael  tty09      Sep 11 10:35
```

For POSIX and Korn Shell, there is another special command substitution for the `cat` command. Normally, you type:

```
$ echo "\n $(cat file)"
```

and the contents of `file` are displayed.

This quicker and shorter form produces the same results:

```
$ echo "\n $(< file)"
```

This shortened form outputs the entire file as a single line of text:

```
$ echo ‘< file‘
```

20

Command-lines and Command History

Editing Command-lines

Typing a long command-line, finding a mistake after executing it, re-typing the command, and finding another mistake, can be very frustrating. *Command-line editing* allows you to correct mistakes in a command line before executing the command.

If you are a new user all of these sections are of interest to you; if you are an advanced user, some of the new features of `vi` supported by the POSIX and Korn Shell and the `fc` command may be of interest.

Using In-line Editing Modes

There are two types of editing modes available in POSIX and Korn Shell: the `vi` mode, and the `emacs` mode. A discussion of each of these methods follows. These Shell editing modes emulate the corresponding editors and all common commands are the same. In-line editing is very similar to using the editor in that in-line editing uses the common editor's commands.

Using vi Line Edit Mode

The vi editing mode uses the same commands as the `vi` editor. If you are unfamiliar with the `vi` editor, see *Using HP-UX*, or *The Ultimate Guide to the vi and ex Text Editors*.

Enabling vi Line Edit Mode

There are several ways to enable the vi editing mode. One is to type:

```
set -o vi
```

For further details on the `set` command see Chapter 23.

Another is to set and export the `VISUAL` shell variable in your `.profile` or `.kshrc`, to a value ending in `vi`:

```
VISUAL=vi
export VISUAL
```

If `VISUAL` is assigned a string that ends in `vi`, `gmacs`, or `emacs`, then the corresponding editor mode is enabled.

Finally, you can set and export `EDITOR` in your `.profile` or `.kshrc`:

```
EDITOR=vi
export EDITOR
```


Now, if `VISUAL` is not set, and `EDITOR` is assigned a string containing `vi`, `gmacs`, or `emacs`, then the corresponding editor mode is enabled.

Performing In-line Edits

Now, you are ready to perform in-line editing. Enabling an editor mode places you into the editor's *command* mode, although when typing it does not appear anything has changed. This allows you to continue typing and executing command-lines as before. It also allows you to type ESC and enter *input* mode. Once you are in input mode, you can edit the specified line using most `vi` commands and then re-execute it by typing Return. For example, suppose you type:

```
$ echo surprris
```

Then, before you press the Return, press ESC. Now you can move on the line using Back space (*not* ◀) to the point where you made your mistake. Then you execute the `vi` delete command, `x`, to remove the extra `r` and the append command, `A`, to add the letter `e` to the end of the line:

```
$ echo surprris[ESC][Back space][Back space]xAe[ESC][Return]
```

The new line and output looks like this:

```
$ echo surprise
surprise
```

For a complete listing of all the `vi` commands usable within the POSIX and Korn Shell `vi` mode, see the *sh-posix*(1) and *ksh*(1) manual pages in the *HP-UX Reference*.

Using emacs and gmacs Line Edit Mode

The other editors implemented for in-line editing are `emacs` and `gmacs`. The only difference between these two editor modes is the function of the CTRL-t command (which transposes characters).

With these editors there is no command mode; you are always in input mode. To use `emacs` or `gmacs` commands, you hold the CTRL key down while pressing a character key or press ESC followed by a character key.

Enabling emacs Line Edit Mode

Again, there are different ways to enable `emacs` or `gmacs` mode. One is to type:

```
set -o emacs
```

or

```
set -o gmacs
```

The other is to set either `VISUAL` or `EDITOR` as described in the previous `vi` section.

Performing In-line Edits

Now, you are ready to perform in-line editing with `emacs`. As you know, enabling an editor mode places you into the editor's *command* mode. Although, to you it does not appear anything has changed as you continue typing in and executing command-lines. For example, suppose you type:

```
$ echo surrpri
```

Before you press the Return, press four CTRL-bs. This moves you left on the line to the point where you made your first mistake. Then, it is a simple matter of executing the CTRL-d or delete command on the extra `r`. To move forward again, use CTRL-e To move the cursor to the end of the line. Now, simply type in the rest of the line:

```
$ echo surrpri
```
CTRL-b CTRL-d CTRL-e s e Return

(Note that CTRL-b is, in fact, pressed four times rather than one.)

The new line and output looks like this:

```
$ echo surprise
surprise
```

There are also ESC sequences that move you forward and backwards by words rather than letters: ESC b moves you backwards one word and ESC f moves you forward one word.

For a complete listing of all the `emacs` commands usable within the POSIX and Korn Shell `emacs` mode, see the *sh-posix*(1) and *ksh*(1) manual entries in the *HP-UX Reference*.

Accessing the History File

Accessing the History File allows you to access a line typed in earlier with a few key strokes, easily enter an editing mode, change the line, and re-execute it. This is possible through several mechanisms provided by the POSIX and Korn Shell: the `fc` command, the `vi` line edit mode, and the `emacs` and `gmacs` line edit mode.

In Chapter 16, the two shell variables `HISTFILE` and `HISTSIZE` are discussed. The history file specified by `HISTFILE` contains the latest commands you executed at your terminal. Every time you type a command at the prompt and press Return it is stored in this history file. `HISTSIZE` specifies the maximum number of commands stored in that file. For example:

```
HISTFILE=/users/mary/.hist20
HISTSIZE=20
```

If you do not set these two variables in your `.profile`, the shell defaults to a file named `.sh_history` of 128 lines.

The history mechanism keeps continuous record of the most recent commands you have executed, even if you logout and back into the system many times or execute the commands in a subshell.

Any command contained in `HISTFILE` is accessible to you for manipulation by either the `fc` command or line editing modes.

To list the current contents of your history file, type:


```
$ history
   ⋮
20 ll -a
21 more file
22 ps
23 pwd
24 lsf
```

A listing, comparable to this, of the most recent commands you have executed is displayed with a number beside each command. These numbers are useful for accessing the history file commands by number.

The `history` command is an alias for `fc -l`. The `fc` command is explained in the next section.

Using the fc Command

There is a built-in command, `fc` (fix command), special to the POSIX and Korn Shell that allows you to list your history file or run an editor on a command-line from the file. Do not confuse this command with the `fc` (Fortran compiler) command.

The syntax of the command is:

`fc` [`-e` *editor*] [`-nlr`] [*first*] [*last*]

`fc -e -` [*old*=*new*] [*command*]

In the first line, part of the syntax indicates listing the history file. If `-l`, *first*, and *last* are indicated, the commands from the *first* string or number to the *last* string or number are listed. This example prints the lines 20 thru 23.

```
$ fc -l 20 23
20 ll -a
21 more file
22 ps
23 pwd
```

If followed by a number, as in `fc -l 23`, then command-lines from 23 on are displayed.

```
$ fc -l 23
23 pwd
24 lsf
25 echo surprise
    ⋮
```

Two other options are available: `-r` which reverses the order of the commands and `-n` which suppresses listing of the command numbers. For example:

```
$ fc -e vi -n 24 25
```

20

With this command-line, you are placed in the `vi` editor with the commands 24 thru 25, without command numbers. Edit the lines. When you write and exit the file, the commands in the file are immediately executed, as shown here.

```
lsf
echo surprise
~
~
:wq!
/tmp/sh1111.12    2 lines 20 characters
ll -a
echo surprise
adv
file1
file2
surprise
```

If you do not specify a `-l` or an editor name with `-e`, the value of the shell parameter `FCEDIT` is used, if it is set; otherwise the shell returns an error.

The `-l` option, used with no other arguments, displays the last 16 commands:

```
$ fc -l
21 more file
22 ps
23 pwd
24 lsf
    :
33 lsf /users/guest
34 pwd
35 x=file1
36 echo $x moved to new directory
$
```


In this next example, the second syntax line replaces an *old* string with a *new* string in the *command*. Here, *command* can be either a command name or line number. The shell makes this substitution possible by building into `fc` certain simple editing capabilities that are used when the `-e` editor that is specified is a dash `-`. Using that editor, `surprise` is replaced by `neat` and echoed to the screen.

```
$ echo surprise
surprise
$ fc -e -  surprise=neat echo
echo neat
neat
```

An `fc -e -` without any arguments displays and executes the last item in the history file which is also the most recent command executed:


```
$ fc -e -
echo neat
neat
$ r
echo neat
neat
```

If you type `alias` for a list of aliases, you see that `r` is set to `fc -e -` such that executing `r` executes the last command. Since the last command just happens to be `fc -e -`, this re-executes the last command, `echo`.

Accessing the History File From vi Mode

There are other ESC sequences that can be executed from `vi` mode such as ESC k, ESC -, j, +, and ESC *count* G.

The following sequences do the following:

ESC k or ESC -	Recalls previous command and steps backward. Press ESC k or ESC -, then k or - to step through.
j and +	Moves forward to next command. Once you type ESC k or ESC -, type just j to step through. After using ESC k or ESC - and k or - to step back, use j or + to step forward.
ESC *count* G	Recalls earlier command number *count*. (Note that this is upper-case G.)

So, if the set of commands looked like:

```
$ fc -l
    ⋮
21 more file
22 ps
23 pwd
24 lsf
```


and you pressed ESC k, the shell displays:

```
$       ESC
$       k
$ pwd
```

Note that the ESC seems to have no effect, and the **k** does not appear on the display.

For every k typed after that, the shell displays one line further back in the history file, but at the same prompt. That is, only one line at time appears at the current prompt. If you go too far backwards in the history file, move forward again using the j. For example:

```
$ ps    j
$ pwd
```

Note that the **ps** is replaced by the **pwd**.

If you use j or k to move from a command-line you are editing to another, all the changes are lost.

To specify a certain line number, use ESC *count* G as follows:

```
$           ESC 20 G
$ ll - a
```

Once you find the command-line you are searching for, you can re-execute it by pressing Return, or edit it using the **vi** in-line edit commands, and then re-execute it.

Accessing the History File From emacs Mode

There are other CTRL sequences that can be executed from `emacs` mode that allow you to search the history file:

20

CTRL-p	Specifies the previous command.
CTRL-n	Specifies the next command forward.
CTRL-r *string*	Specifies a search for the most recent command that contains *string*

So, if you used the same set of commands:

```
$ fc -l
    ⋮
20 ll -a
21 more file
22 ps
23 pwd
```

and you executed an CTRL-p the shell displays:

```
$       CTRL-p
$ pwd
```

For every CTRL-p typed after that, the shell displays one line further back at the same prompt.

Then, if you go too far backwards in the history file, come forward using the CTRL-n. For example:

```
$ ps    CTRL-n
$ pwd
```

If you want to specify a line with a certain *string*, use CTRL-r, such as:

```
$          CTRL-r
ll
$ ll - a
```

Once you find the command-line you are searching for, you simply re-execute it by typing a Return, or edit it using the `emacs` in-line edit commands, and then re-execute it.

21

Basic Shell Programming

The POSIX and Korn Shell are not merely command interpreters; they are also programming languages with all the standard constructs needed to write detailed shell scripts. This chapter discusses the major constructs of the shell, such as inputting and outputting data, conditional statements, and functions.

New users should read this entire chapter; advanced users should see the sections on the `print` command, `select` command, and `function` command for features unique to the POSIX and Korn Shell.

Creating and Executing Shell Scripts

Shell scripts are command-lines that the shell executes in a group. The files `.profile` and `.kshrc` are examples of shell scripts.

To create a script, edit a file using an editor such as `vi`. If you don't know how to create a file using an editor, see *The Ultimate Guide to the vi and ex Text Editors*, or *Using HP-UX*. After you create the script, the file containing your command-lines, you are ready to execute it.

First, make sure the file (script) is executable; type:

```
$ chmod +x script_name
```

This command changes file permissions on the new file so that it is executable. If you want more details on file permissions, see *Using HP-UX*. Then type the *script_name* (that is, the script filename):

```
$ script_name
```

and the script executes and prints out any output you specified.

Commenting

When writing a program, commenting the script helps someone else reading it to understand the code. Comments in the shell start with an unquoted **#** and continue through the first unquoted newline. Comments may start anywhere on the line.

21

```
# This script prints out every executable file.
  for i in 'ls'     # for all files in the current directory
```

The entire first line is a comment and in the second line everything after the **#** is a comment.

Data Input and Output

Programming inevitably requires inputting and outputting of data. The Korn Shell provides the `echo` command and the `print` command for outputting and the `read` command and positional parameter substitution for inputting.

Reading Input Data

There are several ways of passing data into a shell script. One way is by passing arguments to the script through positional parameters; the other way is by using the `read` command. A third way is for the script to run some command or program that reads `stderr` or a named file. Positional parameters have already been described in detail in Chapter 19. Therefore, the following discussion focuses mainly on the `read` command.

The `read` command provides the ability to read input during the execution of a script.

The `read` syntax for POSIX Shell is:

`read` [`-r`] *name* ...

The `read` syntax for Korn Shell is:

`read` [`-prsu`[*n*]][*name*`?`*prompt*][*name* ...]

where, in each case, the command reads a line and places each white-space-separated word into a *name*. The rest of the line goes into the last *name*. For the Korn Shell, if *name*s are not specified, the line is read into the shell `REPLY` variable (see `select` under "Conditional Statements"). If `?`*prompt* is set, the user is prompted interactively with *prompt*.

Option definitions are:

`-p`	Read from the output of the process spawned with two-way pipes, `|&`. (See Chapter 23 for two-way pipes.)
`-r`	Do not interpret the `\` at the end of a line as line continuation.
`-s`	Put the input line into the history file.
`-u`*n*	Read the input from file descriptor *n*.

In this script contained in the file `hello_script`, the first line prints a prompt and waits for input:

```
read user_name?"What is your name? "
echo "Hello, $user_name, and welcome to Korn Shell."
```

The read command prompts the user for a name and stores the name in the variable *user_name*. Running the script creates this output:

```
$ hello_script
What is your name? Stefan
Hello, Stefan, and welcome to Korn Shell!
```

When you see the question mark, type in your name (`Stefan` is typed here and underlined to indicate user input), and then press Return.

The `read` command can read and store several values at one time:

```
read field1 field2 junk
```

This reads the first whitespace-separated name from the input line into `field1`, the second into `field2`, and the rest into `junk`.

21

Printing Data

Sometimes you may wish to output data or comments from a script on the screen, such as script results, and headers to describe the results. There are two output mechanisms in the shell. The first is the `echo` command used in Bourne, C, and POSIX Shells; and the second is the `print` command, unique to Korn Shell.

Using echo

The `echo` command prints its (expanded) arguments to `stdout`. The arguments are separated by spaces.

`echo` [*arg*]

The `echo` command will do parameter expansions on unquoted arguments, and on arguments in double quotes. It will not do expansion on arguments in single quotes.

```
$ var="This is var"
$ echo $var
This is var
$ echo "The value of var is: $var"
The value of var is: This is var
$ echo 'var is $var'
var is $var
```

You can also prompt a user from a script using the `echo` command and the `\c` line-feed escape character. The escape character suppresses the linefeed and leaves the cursor after the colon (:) and blank, waiting for input. Using this idea, type:


```
$ {
> echo "Enter your user name: \c"
> read user
> echo 'User is ' $user
> }
Enter your user name: Stefan
User is Stefan
```

Certain characters can be used for formatting `echo`ed strings. These escape sequences are listed in Table 21-1.

Table 21-1. `echo` **Formatting Escape Sequences**

Escape Character	Results
`\b`	backspace
`\c`	print line without appending a new-line
`\f`	form-feed
`\n`	new-line
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
`\\`	backslash

Using print

Korn Shell provides a unique output mechanism the other shells do not: the `print` command. Its syntax is:

21

`print` [`-Rnprsu`[*n*]][*arg* ...]

The `print` command provides a superset of the `echo` command for shell output. It prints the specified *arg*s, depending on the option set. The options are:

`-R`	ignore all `echo` escape sequences except `\n`
`-n`	do not add a new-line to output
`-p`	write output to the process spawned with a two-way pipe, `|&`, instead of standard output (See Chapter 23 for two-way pipes.)
`-r`	ignore all `echo` escape sequences
`-s`	save *args* in the history file
`-u`*n*	write to the file descriptor *n*

This `print` command:

```
$ print -s "# End of the day. $(date)"
$ history
```

puts the comment `# End of the day.` , following by a date, in your history file. This makes it easier to review the current day's command-lines in the `history` file, because the end of yesterday's commands is clearly marked. (The `history` command lists the last sixteen command-lines executed.)

Conditional Statements

POSIX and Korn Shell provide constructs that allow a script to execute a designated set of command-lines only if a special condition is met. These are called **conditional statements**. Discussed in this section are the following conditional statements: `test`, `if`, `case`, `select`, `for`, and `while`.

Using the `test` Command

This command evaluates *expr*; if *expr* evaluates *true*, `test` returns a zero exit status. If *expr* evaluates *false*, `test` returns a non-zero exit status.

Syntax is:

`test` *expr*

or

`[`*expr*`]`

As shown, the `test` command can be replaced by appropriately spaced brackets ([ ]).

An extensive list of *expr*s are covered in the *HP-UX Reference* on the *test*(1) manual page. Four *expr*s limited to the POSIX and Korn Shell are:

`-L` *file*	Returns true if *file* is a symbolic link.
file1 `-nt` *file2*	Returns true if *file1* is newer than *file2*.
file1 `-ot` *file2*	Returns true if *file1* is older than *file2*.
file1 `-ef` *file2*	Returns true if *file1* has the same device and i-node number as *file2* meaning that both refer to the same physical file.

One *expr* unique to the POSIX Shell is:

`-e` *file*	Returns true if *file* exists.

The conditional command `[[`*test_expression*`]]` may also be used, where *test_expression* is a combination of the above conditional primitives combined with the **and** operator, `&&`, the **or** operator, `||`, and the **negation** operator, `!`.

Using the `if` Statement

The `if` statement allows you to execute one or several commands if a certain condition exists. The syntax is:

21

```
if command-line
then conditional_cmd_line1
else conditional_cmd_line2
fi
```

`if` checks for *command_line* true. (*true* means *command_line* returns 0.) If true, *conditional_cmd_line1* executes; if not, *conditional_cmd_line2* executes.

The following `if` statement checks whether `x` equals `hello`. If so, `Welcome` is printed; if not, `Goodbye` is printed.

```
$ x=hello
$ if [ $x = hello ]
> then echo Welcome
> else echo Goodbye
> fi
Welcome
```

In the following example, the files in the current directory are tested using brackets around the expression. Using `-x` tests for an executable file. If the test returns true the executable file's name is printed.

```
$ for file in `ls`
>    do
>       if [ -x $file ]
>          then echo $file is executable
>       fi
>    done
$
```

Using the `case` Statement

The `case` statement allows you to easily check conditions and then process a command-line if that condition evaluates to true. The syntax is:

```
case string in
pattern1 [ | pattern2 ... ] )  command-list1  ;;
pattern3 [ | pattern4 ... ] )  command-list2  ;;
 ...
esac
```

The first line receives a *string* which is checked against each of the *pattern*s to see if it matches. If the *pattern* matches, the *command-line* directly following is executed. For example:

```
$ case $i in
> -d | -r ) rmdir $dir1; echo "directory removed" ;;
> -o ) echo "option -o" ;;
> -* ) echo "not a valid option" ;;
> esac
```

The `case` statement first checks `$i` against each option for a match. If it matches `-d` or `-r`, the directory is removed (the `|` specifies logical or). If it matches `-o` or `-*` (all others), an appropriate response is printed. If the string does not begin with `-` no action is taken.

Using the `select` Statement

This is a command unique to the POSIX and Korn Shell. It prints on the screen a set of *words* each preceded by a number. It then prints the `PS3` prompt and reads into the `REPLY` variable the line typed by the user. If the contents of that line is the number of one of the listed *words*, the value of *parameter* is set to the corresponding *word*. (If the line begins with anything else, *parameter* is set to the null string.)

Then, regardless of whether the user's input matches one of the *words*, the *command_lines* execute. (Within *command_lines*, conditionals can trap the non-matches.) If the user presses Return but has input nothing, the command reprompts for input. The loop continues until it encounters a break.

The syntax is:

```
select parameter in words
do
    command_lines
done
```

In the following example all the colors in *words* are printed out with a number in front. The default `PS3` prompt, `#?`, is printed and the shell waits for a number. After Return is pressed, it `echo`s that the number's corresponding color is an RGB color, and then prompts for the next entry.

It continues prompting until Break is pressed, or it receives an interrupt. If the input is 4, or anything that is not set to a color, it returns `is an RGB color` without a preceding color name. If the user presses Return, but has input nothing, the command reprompts for input.

```
$ select color in red green blue
> do
>    echo $color is an RGB color.
> done
1) red
2) green
3) blue
#? 1
green is an RGB color.
#? 4
is an RBG color.
#? Break
```


Using the `for` Loop

The `for` loop allows you to execute a *command_line* once for every new value assigned to a *parameter* in a specified *list*. Syntax is:

`for` *parameter* [`in` *list*]
`do` *command-line*
`done`

In the following example, the first time through the loop the `for` statement sets `file` to `x` and prints it out. The second time through the loop, `y` is printed out and the last time, `z` is printed out. When the *list* is completely finished, the loop is exited.

```
$ for file in x y z
> do
>   echo The file name is $file
> done
```

Using the while/until loops

This loop continues executing *command_line* and processing through the `list` as long as the item in *list* continues to evaluate true. Once an item evaluates false, the loop is exited. The syntax is:

```
while list
do list2
done
```

Note The `until` loop is similar to the `while` loop and has the same basic syntax. However, it executes until a nonzero status is returned; the `while` command executes until a zero status is returned. Also, the `until` loop always executes at least once.

The loop in the following example initializes the variable `x`, and then increments and prints out the value until it equals 5 and you exit the loop.

```
$ x=0
$ while [ $x != 5 ]
> do
>   let x=x+1
>   echo $x
> done
1
2
3
4
5
```

Using the break Statement

This command exits loops created by the keywords `for`, `while`, `until`, or `select`.

The syntax is:

`break` [*n*]

If *n* is specified, it breaks out of *n* nested loops.

The following script checks the list of files `x, y, z, none` for executable files and prints the first executable file it encounters. If none are executable, `file` is left set to `none` but it is not printed.

```
$ for file in x y z none
> do
>   if [ -x $file ]
>   then echo $file
>     break
>   fi
> done
$
```

Using the continue Statement

This command skips any lines following it in a `for`, `while`, `until`, or `select` loop until the next iteration of the loop.

Syntax is:

`continue` [*n*]

If *n* is specified, then resume execution starting at the *n*th enclosing loop.

This next script checks for all executable files. If the file is executable the `continue` statement skips both following `echo` statements and starts another loop. If the file is not executable, the script prints that it is not executable. If the file is executable, nothing is printed.

```
$ for file in x y z
> do
>   if [ -x $file ]
>     then continue
>     echo $file is executable
>   fi
>   echo $file is not executable
> done
```

Arithmetic Evaluation Using `let`

`let`, another command unique to POSIX and Korn Shell, enables shell scripts to use arithmetic expressions. This command allows long integer arithmetic.

The syntax is:

```
let arg ...
```

where each *arg* is an arithmetic expression of shell parameters and operators to be evaluated by the shell. Table 21-2 lists the operators in decreasing order of precedence.

Table 21-2. Operator Decreasing Precedence Order

Operator	Description
`-`	unary minus
`!`	logical negation
`*  /  %`	multiplication, division, remainder
`+  -`	addition, subtraction
`<=  >=  <  >`	comparison
`==  !=`	equals, does not equal
`=`	assignment

In the next example, `x` is set to 1, and then when the `let` command executes:

- first, 1*6 is evaluated to 6,
- then 3/1 is evaluated to 3,
- then `x` is added to the 6 which equals 7, and
- finally the 3 is subtracted from the 7 to equal 4.

```
$ x=1
$ let x=x+1*6-3/1
$ echo $x
4
```

21

You can also use parenthesis to create this effect, or override the operator's precedence to produce different results, 9. (When using parenthesis, double quotes are necessary.)

```
$ let "x=x+(1*6)-(3/1)"
$ let "x=(x+1)*6-3/1"
```

This next script reads a value from the user, compares it to 14, and prints an appropriate string based on the comparison:

```
$ read x
$ y=14
$ if (( x >= y ))
>   then echo greater or equal
> else echo less than
> fi
```

Using "(( ))" around the expression, replaces using the `let`:

```
let "x >= y"
```

(which must be quoted to allow blanks and prevent the `>` from being interpreted as an I/O redirection). Also, you do not need to put `$` in front of `x` or `y`. In this situation, the `let` command is used as a condition.

Accessing Arrays

Arrays are a collection of contiguous elements that can be accessed by a subscript. Declaration of arrays in POSIX and Korn Shell is very similar to that of the C Shell. An array's syntax is:

array `[`*subscript*`]=`*value*

The first line sets the element of the named `array` at the designated *subscript* to the *value*. Unlike C Shell, POSIX and Korn Shell start placement of values at the 0 element.

In this example:

```
$ testa[0]=first
$ testa[1]=second
$ echo ${testa[1]}
second
$ echo ${testa[*]}
first second
```

the array `testa` first two elements (0 and 1) are set to `first` and `second`. The following `echo`s, display the value of the 1 element and then the value of every array element as designated by the `*`.

See "Parameter Substitution Conventions" in Chapter 19 for other possible array subscripts and uses for arrays.

Writing Functions

The `function` command is for writing modular programs. Modular programming is the concept of placing frequently used code in a certain area (module) of the shell script so you can call the module or function whenever it is needed rather then repeat the same code.

The function's syntax is:

`function` *name* { *shell_script* }

or

name `()` { *shell_script* }

where using `function` creates a module called *name* and *shell_script* is placed between curly braces, `{ }`. Just using *name* followed by parenthesis, `()`, and the `{`*shell script*`}` also creates a function.

Calling Functions

To invoke the function, type the name of the function followed by any arguments that need to be passed to the function.

Following is a function that takes a file name (`$1`) as an argument and checks whether it is executable. If it tests true, it prints out that the file is executable.

```
$ function exef
> {
>   if [ -x $1 ]
>   then echo $1 is executable
>   fi
> }
$
$ exef script
script is executable
```

where the argument, the *shell_script*, is the executable file, `script`.

In a larger program this function is easily called by specifying the function name and the argument list:

```
function exef
{
  if [ -x $1 ]
  then echo $1 is executable
  fi
}
for file in `ls`
  do
    exef $file
  done
```


where `exef $file` is the function call.

Returning from a Function

Occasionally, you need to return from a function with an exit status. The `return` command's syntax is:

`return` [*n*]

This command stops execution of the function and returns to the calling procedure with an exit status of *n*. If *n* is not specified, the returning status is that of the last command executed within the function. When `return` is invoked outside the boundaries of a function it acts as an `exit`.

The first line of the following example defines a function named `search` which checks a given file for a string, `xxx`. This function inverts the normal return value of `grep`. Therefore, if the string is found, the function returns 1. Otherwise, if the string is not found or the file is not readable, it returns 0.

```
$ search() {
> if grep xxx "$1" > /dev/null  2>&1
> then return 1
> else return 0
> fi
> }
$
$ search myfile
```

21

A **recursive function** is a function that repeatedly calls itself. It terminates when the last call to the function returns a special value the function is testing for. For example, suppose the file *fact* contained a recursive function. The the second call to `fact` within `fact` calls until the value of $1 is returned and is less than or equal to 2. Then, the recursion stops, the factorial of the number originally input is printed, and the function is exited.

```
function fact
{
  integer x
  if (( $1 <= 2 ))
  then
    echo $1
  else
    ((x=$1 - 1))
    let x=$(fact $x)
    ((x=x * $1))
    echo $x
  fi
}
fact $1
```

22

Controlling Jobs

A **job** is a pipeline. A simple job is one command typed to the shell. More complex jobs consist of one or more commands typed together as a pipeline. A complete line containing one or more HP-UX or shell commands is sometimes called the command line. Here is an example of a command line that the shell interprets as a job:

```
$ ps -ef | sort > processes
```

Creating Jobs

The shell associates each pipeline with an integer job number. Once a job is created, you can monitor it or manipulate it. The rest of this chapter covers the things you can do with jobs. Whether you are an advanced or new user, you should read this chapter.

Monitoring Jobs

The shell keeps a table of all current jobs and their numbers. To see a listing of the table type:

```
jobs
```

The screen displays something similar to this, if you have jobs running:

```
[1] + Running      lp  processes
[2] - Done         ps -ef | sort > processes
```

The job number is displayed between square brackets (`[ ]`). `+` marks a job as the current job and `-` marks a job as the previous job. `Done` or `Running` or `Stopped` indicate the current status of the job. In this example, `lp processes` is the actual command line and is telling to the system to print the `processes` file to the line printer.

Suspending Jobs

Job control is supported on Series 300 and Series 800. For HP-UX release 6.5, 7.0, 8.0 and beyond, you can **suspend** jobs. Suspending a job enables you to stop in the middle of a process and regain control of your terminal for other work. Later, you can resume the job or put it in the background.

Suppose you type in a command line and press Return, but immediately realize this process takes a long time and you need to print another job. If the current *susp* character is set to ^Z (see *stty*(1)), you can suspend the current job by pressing:

CTRL-Z

This stops the current job, and returns control of the terminal to `ksh`.

Note You can set the *susp* character to ^Z by typing:

```
stty susp Ctrl-Z Return
```

The `du` command reports the amount of disk space used by the specified directory, or the current directory if none is specified, as in the following example. This command then pipes the output into the `sort` command to be sorted and then finally redirects, `>`, the final output to a file, `diskusage`, for storage. This operation can take some time. To restart suspended processes, use the `fg` or `bg` commands as explained in the next section.


```
$ du | sort > diskusage
CTRL-Z
[1] + Stopped                    du | sort > diskusage
$
```

Putting Jobs in Background/Foreground

Fortunately, there is a way to free up your terminal and at the same time still run long processes such as `du`. You place the process in the background. A **background** process is one that runs invisibly to you at the same time a different process runs on your screen visible to you. The process visible to you is running in the **foreground**. The shell takes over the command line and places it in the background when you follow the line with an `&` metacharacter. For example, if you type:

```
$ du | sort > diskusage&
[1]  6100
```

the second line is what the system returned: a job number and a process number.

If the `set -o monitor` option is on, (type `set -o monitor` at the terminal to enable), a job sends a message to the terminal upon completion of the form:

```
[1]+ Done du | sort > diskusage&
```

identifying the job by its number and showing that it has completed, `Done`. (For details, see Chapter 23.)

Two commands enable you to manipulate jobs between background and foreground: `bg` and `fg`. `bg` places a job in background; `fg` pulls a background job into foreground (back to the terminal screen).

Suppose you had placed a job in the background using the `&`, but now have decided to return it to the screen; type:

```
$ fg %job_number
```

or type **%%** or **%+** if it is the current job. If it was the previous job, (meaning that you have typed another command after placing the command in the background), use **%-**.

The following example demonstrates how to brings a previous command (`du`) back to the foreground. The second background process (`sleep` command) suspends execution of the shell for `999` seconds.

```
$ du | sort > diskusage&
[1]  6100
$ sleep 999&
[2]  6102
$jobs
[2] +  Running                 sleep 999
[1] -  Running                 du | sort > diskusage
$ fg %-
du | sort >diskusage
```

If you later decide you want your terminal free again, first suspend the job, then type:

```
$ bg
```

to put it back into the background.

You can also use these two commands on suspended jobs to restart them in foreground or background.

Killing Jobs

Sometimes after you've started a job and placed it in the background, you realize it is an incorrect process and you do not want to run it. In this type of instance, you can destroy or **kill** a job.

Suppose, you start this process:

```
$ lsf /* | sort > filenames&
[1]   6112
```

then realize you do not want to list the full file system (that is, you really didn't want to use * in the command line), but rather, just the root directory, and decide to kill the job. To kill the process, use the job's number, ([1]), and type:

```
$ kill %1
$
```

The `kill` command kills the job and the **%** metacharacter specifies the job number `1`. As shown above, you are returned to the prompt. Recall that **%+** and **%%** perform the same function as **%1**, since it is the current job. If it was the previous job, use **%-**. To see the status of the job, type:

```
$ jobs
[1] + Terminated    lsf /* | sort > filenames&
```

The line following `jobs` shows that the current `lsf` job has been terminated.

If you log off the system while any of your processes are running, whether in background or foreground, the jobs are destroyed unless you use the `nohup` command (see *nohup*(1) entry in the *HP-UX Reference* for details).

23

Advanced Concepts and Commands

This chapter explains advanced topics and commands you will need for understanding the more difficult aspects of the Korn Shell.

The ENV Variable

In Chapter 16 the ENV variable was discussed. The `ENV` variable specifies a file, usually `.kshrc`, which is executed when ever you spawn a new, interactive POSIX or Korn Shell. An **interactive shell**, is a shell that has input and output tied directly to the terminal. Therefore, you can access standard in, standard out, and standard error. To determine whether on not your shell is interactive, type:

```
$ set -o
```

and look for:

```
interactive on
```

This `.kshrc` file normally contains commands to set up the POSIX or Korn Shell's environment. However, if this file is exceedingly long, spawning the new shell can be a long process. This complicated `ENV` variable prevents `.kshrc` from being read when not in interactive mode.

You can turn off the processing of the ENV file for non-interactive shells with the following in your ~/.profile:

```
export ENV='${FILE[(_$-=1)+(_=0)-(_$-!=_${-%%*i*})]}'
export FILE=$HOME/.envfile
```

23

The idea behind this scheme is to set up an array (FILE) whose first element is the file we want executed at startup, and whose second element is null.

```
$ export FILE=$HOME/.envfile
$ echo $FILE[0]
/users/pbm/.envfile
$ echo $FILE[1]

$
```

We then want to set ENV to FILE[0] for interactive shells and set ENV to FILE[1] for non-interactive shells. To do this we need an expression that will evaluate to "0" for an interactive shell and will evaluate to "1" for a non-interactive shell.

```
ENV='${FILE[magic_expression]}'
```

The flags variable ($-) is the key to forming our "magic_expression". If the shell is interactive, the flags will contain an 'i'. The expression used as the index consists of three parts that are combined to form the final index:

```
(_$-=1)   (_=0)   (_$- != _${-%%*i*})
```

Let's look at each part of this expression:

`(_$-=1)` This creates a parameter named _$- and assigns it a value of 1. The value of this expression is the value of the assignment, namely 1. Note: since parameter substitution is performed on this expression, the name of the variable will look something like "_ism". To see this, try:

```
$  echo $-      #  see what flags are
ism
$  echo _$-
_ism
$  echo $_ism
                # no variable _ism defined yet
$  ((_$-=1))    # create and assign to _ism
```

```
$  echo $_ism
1  # now _ism has a value
$
```

`(_=0)` — Set parameter named _ to 0 (we need this since we may reference $_ in the next step). The value of this expression is 0.

`(_$- != _${-%%*i*})` — This expression checks to see if a parameter named _$- (remember this is _ism in our example, and we have set it to 1 in the first expression) has a different value than the parameter named on the right hand side of the != operator. That parameter name will be either "_" or "_$-" (_ism) depending on the expansion of "${-%%*i*}". The latter will evaluate to null for an interactive shell and to "_$-" for a non interactive shell.

So, for an interactive shell we have (_$- != _) which is true (value arithmetically speaking is 1) and for a noninteractive shell we have (_$- != _$-_) which is false (value for the expression is 0).

Adding it all up we get the following for an interactive shell:

```
(_$-=1) + (_=0) - (_$- != _)
   1    +   0   -    1   =  0
${FILE[0]} = "$HOME/.envfile"
```

And for a non interactive shell:

```
(_$-=1) + (_=0) - (_$- != _$-)
   1    +   0   -    0   =  1
${FILE[]} = ""
```

Co-Processes

The Korn Shell provides a mechanism to spawn a child process connected by a pipe to the parent shell. The standard input and output of the spawned command can be written to and read from the parent shell. Placing the `|&` metacharacter after the command to be executed creates a special pipe where you can use the `print -p` command to write the standard input of the spawned command process and the `read -p` command to read from the output of the process. See Chapter 21 for details on the `print` and `read` commands.

These two-way pipes allow shell scripts to pass data out through a pipeline, process that data with a coprocess, and bring data back through the pipeline for further use by the script, without having to use temporary files to store the input or output data. For example, suppose you have a file, `2waypipe`, containing this script:

```
pi=3.14159
bs |&

echo "Please enter value1 and value2: \c"
read value1 value2
print -p "$value1 + $value2"               # add them.
read -p sum
print -p "$sum - $pi"
read -p result
"The answer is:  $result"
```

When you execute the script:

```
$ 2waypipe
Please enter value1 and value2: 12 12
The answer is:  20.85841
```


it immediately executes the `bs` compiler/interpreter which allows addition and subtraction. The `read` statement reads from standard input the the typed numbers 12 and 12 as `value1` and `value2`. In the `print -p` statement the numbers are piped to the spawned process `bs` and summed and the sum read back into the script using the `read -p` script. Then the values `sum` and `pi` are sent back to `bs` and `result` is read back into the script using `read -p`, again. Then the output is sent to standard output.

There are some limitations on what you can do with two-way pipes:

- They are only useful with commands that read standard input for data and write standard output with results. You cannot use commands like *vi*(1), which must talk to a terminal. Instead, use commands which read standard input and write results to standard output as soon as there is something to output.
- There is currently no way to close a two-way pipe. Therefore, you cannot use them with commands such as *sort*(1) or pipelines which require reading an EOF before emitting useful output. Instead, use commands you can tell to `quit`.

The `whence` Command

This is a command unique to the Korn Shell. When a name is provided to the `whence` command, it returns the way in which that name will be interpreted by the shell. The syntax is:

`whence` [`-v`] *name* ...

The flag, `-v`, produces a more verbose report.

When *name* is a reserved word, function or builtin command, the shell returns the command name. If the command has an alias, the alias is displayed. If neither of these is true, the full path name is printed. If *name* is not found, the shell so indicates.

```
$ whence -v type
type is an exported alias for whence -v
```

This example discovers that `type` is actually an exported alias for `whence -v`. So, just type:

```
type type
type is an exported alias for whence -v
```

The following example shows how the different commands are interpreted:

```
$ this() {
>   print that
>       }
$ whence while true alias this file fffile
while
:
alias
this
/usr/bin/file

$ whence -v while true alias this file fffile
while is a reserved word
true is an exported alias for :
alias is a shell builtin
this is a function
file is /usr/bin/file
fffile not found
```

The first part of this example defines a function `this()`, then asks `whence` to explain a series of six different words that might be used as commands.

This first five words of the set demonstrate the five types of command words the shell understands, and presents them in Korn Shell's precedence order. Notice in Table 23-1 that POSIX Shell precedence is different from Korn Shell for function and built-in.

Table 23-1.
Precedence Order for Korn and POSIX Command Words

Order	Korn Shell	POSIX Shell
1	reserved-word	reserved-word
2	alias	alias
3	built-in	function
4	function	built-in
5	other (e.g., path name)	other

The `set` Command

The `set` command is used to turn on and off shell options such as tracking or automatic exporting of commands. Its second function is to reset the values of positional parameters (`$1`).

The `set` command syntax for POSIX Shell is:

`set` [±`aCefnuvx`][±`o` *option* ...][*arg* ...]

and the `set` command syntax for Korn Shell is:

`set` [±`aefhkmnostuvx`][±`o` *option* ...][*arg* ...]

where *arg* specifies the positional parameters to be reset, and *option* can specify with a word the same meaning as the `-aCefhkmnostuvx` letters.

For example, you can turn the verbose option on in two ways:

```
$ set -v
```

or

```
$ set -o verbose
```

Here is an example showing results before and after the verbose option is set:

```
$ echo hello
hello
$ set -o verbose
$ echo hello
echo hello
hello
```

The `set -o verbose` causes the shell to print each line as it is read, and then print the output of that line.

A discussion of other options follows:

`-a`	All subsequent parameters that are defined are automatically exported.
`-C`	Prevents existing files from being overwritten by the shell's `>` redirection operator. The `>|` operator overrides this `noclobber` option.
`-e`	If the shell is non-interactive and if a command fails, execute the ERR trap, if set, and exit immediately. This mode is disabled while reading `.profile`.
`-f`	Disables file name generation.
`-h`	Each command whose name is an *identifier* becomes a tracked alias when first encountered.
`-k`	All parameter assignment arguments are placed in the environment for a command to use, not just those that follow the command name.
`-m`	Background jobs will run in a separate process group and a line will print upon completion. The exit status of background jobs is reported in a completion message.
`-n`	Read commands but do not execute them.
`-s`	Sort the positional parameters.
`-t`	Exit after reading and executing one command.
`-u`	Treat unset parameters as an error when substituting.
`-v`	Print shell input lines as they are read.
`-x`	Print commands and their arguments as they are executed.
`-`	Turns off `-x` and `-v` flags and stops examining arguments for flags.
`--`	Do not change any of the flags. This is also useful in setting $1 to a value beginning with `-` . If no arguments follow this option then the positional parameters are unset.

Using `+` rather than `-` causes these flags to be turned off.

These flags are the same ones used to invoke the shell:

```
ksh -h
```

or

```
/bin/posix/sh -h
```


which causes the shell to create a tracked alias for every command executed.

The POSIX and Korn Shell implement an option, `-o`, that turns on the specified argument or *option*. (i.e., `set -o` *option*) Many of these options correspond to the above letters that perform the same function without using `-o`. The following argument or *option* can be one of the following option names:

`allexport`	Same as `-a`.
`errexit`	Same as `-e`.
`emacs`	Puts you in an `emacs` style in-line editor for command entry.
`gmacs`	Puts you in a `gmacs` style in-line editor for command entry.
`ignoreeof`	The shell will not exit on end-of-file. The command exit must be used.
`keyword`	Same as `-k` .
`markdirs`	All directory names resulting from file name generation have a trailing `/` appended.
`monitor`	Same as `-m`.
`noexec`	Same as `-n`.
`noclobber`	Same as `-C` for POSIX Shell.
`noglob`	Same as `-f`.
`nounset`	Same as `-u`.
`verbose`	Same as `-v`.
`trackall`	Same as `-h`.
`vi`.	Puts you in insert mode of a `vi`-style in-line editor until you press the ESC key. This puts you in a mode so you can move on the line. A Return executes the line.
`viraw`	Each character is processed as it is typed in `vi` mode.
`xtrace`	Same as `-x`.

If you want a listing of all the currently set options, type:


```
$ set -o
Current option settings
allexport        off
bgnice           off
emacs            off
errexit          off
gmacs            off
ignoreeof        off
interactive      off
keyword          off
markdirs         off
monitor          off
noexec           off
noglob           off
nounset          off
protected        off
restricted       off
trackall         on
verbose          off
vi               on
viraw            off
xtrace           off
$
```

without options. This could be a very lengthy list, but should have some of these items listed.

You can use the `set` command in other ways, as in:

```
$ set third first second
$ echo $1 $2 $3
third first second
$ set -s
$ echo $1 $2 $3
first second third
```


where `set` places the three values into the appropriate positional parameters, and then sorts them and places them in the parameters in sorted order.

The `typeset` Command (for Korn Shell only)

This command creates a shell variable, assigns it a value, and specifies certain attributes for the variable, such as `integer` and `readonly`.

The syntax is:

`typeset` [`-HLRZfilprtux`[*n*][*name*[`=`*value*]] ...]

where *name* is the shell variable to be created, *value* is to be assigned according to the options set.

The following example makes `year readonly`.

```
$ typeset -r year=2000
$ echo $year
$ year=2001
ksh: year: is readonly
```

The following list of attributes may be specified by the designated option or flag:

`-F` This flag provides UNIX to host-name file mapping on non-UNIX machines.

`-L` Left justify and remove leading blanks from value. If *n* is non-zero it defines the width of the field, otherwise it is determined by the width of the value of first assignment. When the parameter receives a value, it is filled on the right with blanks or truncated to fit into the field. Leading zeros are removed if the `-Z` flag is also set. This turns the `-R` flag off.

`-R` Right justify and fill with leading blanks. If *n* is non-zero it defines the width of the field, otherwise it is determined by the width of the value of first assignment. The field is left filled with blanks or truncated from the end if the parameter is reassigned. This turns the `L` flag off.

`-Z` Right justify and fill with leading zeros if the first non-blank character is a digit and the `-L` flag has not been set. If *n* is non-zero it defines the width of the field, otherwise it is determined by the width of the value of first assignment.

`-e` Tag the parameter as having an error. This tag is currently unused by the shell and can be set or cleared by the user.

`-f` The names refer to function names rather than parameter names. No assignments can be made and the only other valid flag is `-x`.

`-i` The *name* is an integer. This makes arithmetic faster. If *n* is non-zero it defines the output arithmetic base, otherwise the first assignment determines the output base.

`-l` All uppercase characters converted to lowercase. The uppercase flag, `-u` is turned off.

`-p` The output of this command, if any, is written onto the two-way pipe.

`-r` The given names are marked `readonly` and these names cannot be changed by subsequent assignment.

`-t`	Tags the *name*. Tags are user definable and have no special meaning to the shell.
`-u`	All lowercase characters are converted to uppercase characters. This turns the lowercase flag, `-l` off.
`-x`	The given *name*s are marked for automatic export to the environment of subsequently executed commands.

Using `+` rather than `-` causes these flags to be turned off. If no *name* arguments are given but flags are specified, a list of *name*s (and optionally the *value*s) of the parameters which have these flags set is printed. (Using `+` rather than `-` keeps the values to be printed.) If no *name*s and options are given, the *name*s and attributes of all parameters are printed.

The following example covers some of the attributes set above:

```
$ typeset -i arg1=3 arg2=22
$ echo $arg1 $arg2
$ typeset
.
export PATH
readonly year
.
$ typeset -u up=letters
$ echo $up
LETTERS
```

The `trap` command

(The scope of the `trap` command is much broader than the following explanation suggests, but here is one of its uses.) Many times we execute a script and then realize a mistake was made and press the Break key to stop the process. It is possible the script created several files on the system that you would have to search for and manually delete. Fortunately, the `trap` command captures an interrupt. Now you can Break, let the `trap` command capture it and then clean up the files from within the script. The syntax is:

`trap` [*arg*] [*signal* ...]

The `trap` waits for *signal*s sent to the shell, traps it and then executes *arg*. After setting traps, typing `trap` with no *arg*s lists all commands associated with signals.

For example:

```
$ temp="/tmp/xyz$$"
$ trap "rm -f $temp; exit" 0 2 3 15
$ trap
0:rm -f /tmp/xyz18996; exit
2:rm -f /tmp/xyz18996; exit
3:rm -f /tmp/xyz18996; exit
15:rm -f /tmp/xyz18996; exit
```

in the first line a temporary file `$temp` is defined, whose name includes `xyz` and the process id number. The second line sets a trap to remove the file (without complaining if it doesn't exist yet or if the remove fails). It then exits the shell, if the shell exits (0) or receives one of a certain set of signals (2, 3, 15), which could be given by names (`INT`, `QUIT`, `TERM`). After setting the trap, `trap` with no options, lists all traps. The `exit` in the trap is necessary because otherwise the trap would be like an interrupt routine, returning to execution of the script on receipt of a signal.

If *arg* is omitted or is `-`, all trap *signal*s are reset to their original values. If *signal* is `ERR` then *arg* will be executed whenever a command has a non-zero exit code. The `ERR` trap is not inherited by functions.

If the *signal* is 0 or `EXIT` and the `trap` statement is executed inside the body of a function, then the command *arg* is executed after the function completes. If *signal* is EXIT for a trap set outside any function then the command *arg* is executed on exit from the shell.

The `ulimit` Command (for Korn Shell only)

This command sets limits on specified resources used by a spawned or child process (subprocess).

The syntax is:

```
ulimit [-f ][n]
```

where *n* is the size to be set depending on the type of limit set by the option `-f`. A list of those options follow. If no option is given, `-f` is assumed. If *n* is not given the current limit is printed.

To see the current limit, type:

```
$ ulimit
```

To change the size of file the current process or a spawned process can create, type:

```
$ ulimit -f 1000
```

where:

`-f` imposes a size limit of *n* blocks on files written by child processes (files of any size may be read)

24

Command Reference

This chapter is a command reference for the POSIX and Korn Shell commands. Commands are in alphabetical order, explained briefly, and followed by their syntax and an example. Each example is explained.

This reference is written for the intermediate or advanced user who has a firm understanding of shell concepts, whether it be Bourne Shell, C Shell, POSIX Shell or Korn Shell. It is meant as a quick reference or refresher for the basic commands used in the Korn Shell.

alias

Syntax

For POSIX Shell only:

`alias` [*word*[`=`*command*] ...]

For Korn Shell only:

`alias` [`-tx`][*word*[`=`*command*] ...]

The `alias` command defines *word* to mean `command` such that when *word* is used *command* is executed. This is useful for shortening long command lines to one or two letters.

Example

```
$ alias unpro='chmod +w'
$ unpro myfile
```

In this example, `unpro` is shorthand for (aliased to) `chmod +w`. Saying `unpro myfile` adds write permission to `myfile`.

```
$ alias cd=mycd
$ cd there
$ \cd here
```

The first statement declares `cd` to be an alias for a function (defined elsewhere) called `mycd`. The next line changes the working directory to `there` using that function. Using backslash (`\`), causes the last line to perform a real `cd`, not the function form, to the directory `here`. Note, that quoting the first word (`\`) in any way prevents it from being interpreted as an alias.

Just typing:

```
$ alias
```

lists, on the screen, all the current shells default and set aliases.

To track aliases in Korn Shell, use:

```
alias -t [ name]
```

such as

```
$ alias -t vi
```


This tracks the full path of *name* the first time it is used and sets it in a special list of tracked aliases. This speeds up the search time for commands. Using `set -h` sets automatic tracking on all commands. See the `set` command for more details. If PATH is changed interactively or in login scripts, the tracked aliases become undefined. To list tracked aliases, use the `alias -t` command without a *name*.

To export aliases in Korn Shell, use:

```
alias -x [ name]
```

such as

```
$ alias -x who='who | sort'
```

This exports *name*, or `who` in this example, for use by subshells.

bg

Syntax


```
bg [%n]
```

The `bg` command places job *n* (where *n* is the job number) in the background. If *n* is not specified, the current job is put into the background.

Example

```
$ bg %1
```

This places the command defined to the shell by the job number, 1, in the background. See `jobs` for more information.

break

Syntax

`break` [*n*]

This command exits loops created by the keywords `for`, `while`, `until`, or `select`. If *n* is specified, it breaks out of *n* nested loops.

Example

```
$ for file in x y z none
> do
>   if [ -x $file ]
>   then echo $file
>     break
>   fi
> done
$
```

This script checks the list of files, `x, y, z, none`, for executable files and prints the first executable it encounters. If none are executable, `$file` is left set to to `none`, but it is not printed.

case

Syntax

24

```
case string in
pattern1 [ | pattern2 ... ] ) command-list1;;
pattern3 [ | pattern4 ... ] ) command-list2;;
 ...
esac
```

The `case` statement allows you to easily check several conditions and then process a command_line if that condition evaluates to true. The first line receives a *string* which is checked against each of the *pattern*s to see if it matches. If the *pattern* matches, the *command_line* directly following is executed.

Example

```
$ case $i in
> -d | -r ) rmdir $dir1; echo "directory removed" ;;
> -o ) echo "option -o" ;;
> -* ) echo "not a valid option" ;;
> esac
```

the `case` statement first checks `$i` against each option for a match. If it matches `-d` or `-r`, the directory is removed (the `|` specifies logical or). If it matches –o or `-*` (all others), an appropriate response is printed. If the string does not begin with a `-`, no action is taken.

cd

Syntax


```
cd
cd [path]
cd old new
```

Change directory from your current (or `old`) directory to your `new` directory.

Example

```
$ cd
```

This command transports you from your present working directory, `PWD`, to your home directory, `HOME`, which becomes the new `PWD`.

```
$ cd -
```

The `-` transports you to the previous `PWD`, which is contained in `OLDPWD`.

```
$ cd ../otherdir
```

This use of *../otherdir* changes the working directory from the present working directly to another directory contained underneath the same parent. In other words, change to parent, then to *otherdir* contained in the parent of the current directory.

```
$ cd /bin/
$ cd / /usr/
```

This example changes your present working directory to `/bin/`. Then, it replaces the old directory (`/`) with the new directory (`usr`) and transports you to `/usr/bin`.

```
$ CDPATH="$HOME/work:$HOME/src"
$ cd aardvark
```

In this example, the present working directory changes to `$HOME/src/aardvark`, unless there is a directory named `$PWD/aardvark` or `$HOME/work/aardvark`. The first line sets `CDPATH` to a list of directories to be searched if a full path name is not given to `cd`. So, when you type the second line in the example, the shell first checks for `$PWD/aardvark` then for `$HOME/work/aardvark`.

continue

Syntax

`continue` [*n*]

This command skips any lines following it in a `for`, `while`, `until`, or `select` loop and restarts the loop at the top. If *n* is set, resume execution at the *n*th enclosing loop.

Example

```
$ for file in x y z
> do
>   if [ -x $file ]
>   then continue
>   echo $file is executable
>   fi
>   echo $file is not executable
> done
```

This script checks for all executable files. If the file is executable the `continue` statement skips both following `echo` statements and starts another loop. If the file is not executable, the script prints that it is not executable. If the file is executable, nothing is printed.

echo

Syntax

`echo` [*arg* ...]

This command writes to standard output all arguments, *args*, separated by blanks, or a blank line if no arguments are specified.

Example

```
$ var='short'
$ echo 'This is a' $var 'example.'
```

In this example, `echo` prints the line `This is a short example.`.

```
echo "\n\nusage: $0 arg1 arg2" >&2
```

This example is a line that might appear in a shell script. It prints to standard error: two blank lines (`\n`), and a usage message including the invocation name of the script, (designated by `$0`). Using double quotes rather than single quotes, causes `$0` to be interpreted. Certain characters can be used for formatting `echo`ed strings. These escape characters must be preceded by a backslash and enclosed in double quotes for interpretation such as the `\n`. See Chapter 21 for a list of these characters.

eval

Syntax

`eval` [*arg* ...]

This command is unique because the command_line is scanned twice by the shell. First, the shell interprets the command_line when it passes the *arg*s to the `eval` command and then interprets it a second time as a result of executing the `eval` command. Consequently, you can execute command_lines that normally would not be possible, as shown next.

Example

```
$ cmd='ps -ef > ps.out'
$ eval $cmd
```

When `eval` is executed the shell has already expanded `cmd`, so it runs `ps -ef` and redirects the output to file `ps.out`. If `eval` was not used, redirection or pipes would not be interpreted by the shell after parameter substitution.

exec

Syntax

`exec` [*arg* ...]

The `exec` command replaces the current shell with the new shell or program specified by *arg*s without spawning a new process or subshell.

Example

```
$ exec 2>/dev/null
```

This example redirects the the shell's standard error to `/dev/null` where it is ignored by the shell.

```
:!exec ps -ef
```

From `vi` it is possible to run `ps -ef` without wasting time spawning another process. Using `:!` causes `vi` to pass the command `exec ps -ef` to the shell for interpretation, and then `exec` causes the shell to execute `ps` in place of itself.

exit

Syntax


```
exit [n]
```

Use this command to exit a shell. The *n* parameter, if set, specifies the exit status. If *n* is not specified, the exit status is the same status as that of the most recently executed command.

Example

```
$ if grep xxx myfile > /dev/null  2>&1
> then :
> else exit
> fi
$
```

This script searches `myfile`, using `grep`, for the string "xxx". If `grep` finds the string it returns a 0 and writes the string to `/dev/null`, so the shell executes the null command (":"). If the string is not found, or `myfile` isn't readable, the shell script exits with the same return value as from `grep`. Notice that both standard output and standard error from `grep` are ignored by sending them to `/dev/null`. If the third line instead read:

```
> else exit 15
```

the shell script would exit with a value of 15.

export

Syntax

`export` [*name* ...]

This command marks *name* parameters to be passed to the environment for use by other commands and subshells. The `export` command by itself lists all currently exported values.

Example

```
PS1='hello: '
export PS1
```

In this example, the shell prompt is set to the string `hello:` (followed by a space character which causes the same string to be used by subshells.

For another example, in your `.profile`, which is read only at login time, add:

```
SHDEPTH='-1'  # initial depth; incremented in .kshrc.
export SHDEPTH
```

Then in your `.kshrc` file, which is read whenever a shell starts up (depending on how you configure things), add:

```
((SHDEPTH = SHDEPTH + 1))

if [ $SHDEPTH = 0 ]
then PS1=":; "      # useful with ENTER key on HP terminals.
else PS1=": $SHDEPTH; "
fi
```

Now, in your login shell, your prompt will be `:;` , and in subshells it will be `: 1;` , `: 2;` , etc. where the number indicates the nested depth of the shell.

Note The POSIX Shell also provides the `-p` option with the `export` command. The `-p` option formats the output so that it is suitable for re-input to the shell as commands that achieve the same exporting results.

fc

Syntax

`fc` [`-e` *editor*] [`-nlr`] [*first*] [*last*]

`fc -e -` [*old*=*new*] [*command*]

The `fc` command is one of the three methods used for listing and editing command_lines. In the first form, `fc` searches the history file for the command lines that contain the commands specified by strings *first* through *last* and acts on them according to the option specified (`-nlr`). The second line invokes the editor to replace the *old* string with the *new* string in command_line specified by *command* and then execute the new version.

Example

```
$ fc -l
$ fc -e -
```

The first line lists the last 16 commands you have executed. The second line executes the previous command which just happens to be `fc -l` so the last 16 commands are displayed again.

```
$ fc -l ps
```

This lists all the commands in the history file that have been executed since the last `ps`.

```
$ fc -e - cd=ls cd
```

This command_line causes the replacement of `cd` with `ls` in the most recently executed command in the history file, which contains a `cd`. After the replacement the new command_line is executed.

See Chapter 20 for a detailed explanation of `fc`.

fg

Syntax

`fg` [*%n*]

The `fg` command places job *n* (the job's number), currently running in the background or suspended, in the foreground. The current job is put into the foreground, if *n* is not specified.

Example

```
$ fg %1
```

This places the command defined to the shell by the job number, 1, in the foreground. See `jobs` for more information.

for

The `for` loop allows you to execute a *command_line* once for every new value assigned to a *parameter* in a specified *list*.

Syntax

`for` *parameter* [`in` *list*]
`do` *command_line*
`done`

Example

```
$ for file in x y z
> do
>   echo The file name is $file
> done
```

the first time through the loop the `for` statement takes the file `x` and prints it out. The second time through the loop, `y` is printed out and the last time, `z` is printed out. When the *list* is completely finished, the loop is exited.

function

Syntax

`function name` { *shell_script* }

or

`name ()` { *shell_script* }

The `function` command is used to modularize programs. To create a function, use `function` followed by the *name* and a *shell script* enclosed between curly braces, `{}`, or use just the name followed by parenthesis `()` and then `{` *shell script*`}`. Nothing is required or allowed inside the parenthesis. To invoke the function, type the *name* followed by any positional parameters that need to be passed in as arguments. Recursion is possible by using the `typeset` command (listed later in this chapter). See Chapter 21 for details on functions.

Example

```
$ function exef
> {
>     if [ -x $1 ]
>     then echo $1 is executable
>     fi
> }
$
$ exef script
```

This simple function takes a file name (`$1`) as an argument and checks whether it is executable. If it tests true, it prints out that the file is executable.

if

The `if` statement allows you to execute one or several commands if a certain condition exists.

Syntax

```
if command_line
then conditional_cmd_line1
else conditional_cmd_line2
fi
```

First, `if` checks whether *command_line* is true. If it is then, *conditional_cmd_line1* is executed; if it is not, *conditional_cmd_line2* is executed.

Example

```
if [ $x = passwd ]
then echo "Welcome to Korn Shell"
else echo "Please log off"
fi
```

`if` checks whether the value of `x` is equal to `passwd`. If so, the first `echo` line is printed; otherwise the second line is printed.

jobs

Syntax


```
jobs [-l][-p][job_id ... ]
```

To list all the jobs currently running in your shell, including job number and status, use the `jobs` command. Using the `-l` option lists the process ID directly after the job number, as well. Using the `-p` option displays the process IDs of the selected jobs.

Example

```
$ (sleep 20; date) &
$ jobs
$ jobs -l
```

This example puts a `date` program in the background to execute in 20 seconds, and then looks at the waiting job using the two different command versions.

kill

Syntax

For POSIX Shell only:

`kill -s` *signal_name process_id ...*

`kill -l` [*exit_status*]

For Korn Shell only:

`kill` [`-`*signal*]*process_id*

This command cancels (kills) the designated *process_id* using *signal* if specified. *signal*s are specified by number or name as explained in the *signal*(2) entry in the *HP-UX Reference*. If *signal* is not specified, `kill` uses a default signal 15 (`SIGTERM`) which causes software termination. Process IDs can be displayed using the `ps` command; see *ps*(1).

Example

```
$ sleep 20 &
$ kill 1235
```

This example executes a `sleep 20`, which happens to be process 1235, then sends it a `SIGTERM` (terminate).

```
$ sleep 20 &
$ kill -9 %1
```

This starts `sleep 20`, which happens to be job 1, then sends it a kill (`SIGKILL`).

```
$ kill -l
$ kill -HUP 3140
```

Using the `-l` option with `kill` lists the signal names. The second line sends a `SIGHUP` to process 3140.


```
$ kill -l 9   (applies to POSIX Shell only)
```

This will display the signal name corresponding to the exit status. In this case `KILL`. This is usually used with the `$?` (exit status) after returning from a `wait` command.

let

Syntax

For Korn Shell only:

`let` *arg ...*

This Korn Shell command allows for long integer arithmetic normally performed by the `expr` command. Each *arg* is an arithmetic expression of shell parameters and operators that are evaluated by the shell. Table 24-1 lists operators in decreasing order of precedence.

Table 24-1. Operator Decreasing Precedence Order

Operator	Description
`-`	unary minus
`!`	logical negation
`*`, `/`, `%`	multiplication, division, remainder
`+`, `-`	addition, subtraction
`<=`, `>=`, `<`, `>`	comparison
`==`, `!=`	equals, not equals
`=`	assignment

Example

```
$ x=1
$ let x=x+1
```


In this example, `$x` is set to 1, then incremented to 2 using the `let` command. If the `expr` command had been used a new process would have been created. Also, with `let` the `$` is not needed to obtain the value of `x`.

```
read x
y=14
if (( x >= y ))
then echo greater or equal
else echo less
fi
```

This script reads a value from the user, compares it to 14, and prints an appropriate string based on the comparison. Using `(( ))` around the expression replaces:

```
let "x >= y"
```

(which must be quoted to allow blanks and prevent the `>` from being interpreted as an I/O redirection). Again, the `$` is not needed in front of `x` or `y` to obtain their values.

print

Syntax


```
print [-Rnprsu[n]][arg ... ]
```

The Korn Shell `print` command provides the same functionality as the `echo` command for shell output. It prints the specified *arg*s dependent upon the option set. A description of the options follows:

`-R`	Ignore all `echo` escape sequences except `-n`.
`-n`	Do not add a new-line to output. Similar to including `\c` in *arg*.
`-p`	Write output to the process spawned with `|&` instead of standard output.
`-r`	Ignore all `echo` escape sequences.
`-s`	Write *args* into the history file.
`-u`*n*	Write to file descriptor *n*.

Example

```
$ print -s "# End of the day."
$ history
```

This `print` puts the comment `# End of the day.` in your history file. Then, you can easily determine the current day's commands when looking at your history file.

pwd

Syntax


```
pwd
```

This command prints the current working directory.

Example

```
$ cd
$ pwd
/users/guest
```

The first line places you in your `$HOME` directory and `pwd` prints where it is.

read

Syntax

For POSIX Shell only (note that *name* must be specified):

`read` [`-r`] *name ...*

For Korn Shell only:

`read` [`-prsu`[*n*]][*name*`?`*prompt*][*name ...*]

This shell input mechanism reads a line from standard input and places each word into the parameter *name* using the separator specified by the `IFS` shell parameter. If *name*s are not specified, the line is read into the Korn Shell `REPLY` variable (see `select`). If the `?`*prompt* is included the user is prompted interactively with *prompt*. The definitions of the options are:

`-p`	Read from the output of the process spawned with `|&`.
`-r`	Do not interpret the `\` at the end of a line as line continuation.
`-s`	Put the input line into the history file.
`-u`*n*	Read the input from file descriptor *n*.

Example

```
$ echo 'What is your name? \c'
$ read name
$ echo "Hello, $name ... "
```

The first line prints a prompt and leaves the cursor one blank to the right of the `?`. The next line reads in text from the user and saves it in `$name`. Last, a line is printed which includes the value of `$name` (since the string is in double, not single, quotes).

```
read field1 field2 junk
```

This reads the first whitespace-separated word from an input line into `$field1`, the second into `$field2`, and the rest into `$junk`, which is presumably ignored.

readonly

Syntax

For POSIX Shell only:

```
readonly name[=word] ...

readonly -p
```

For Korn Shell only:

```
readonly [ name ... ]
```

This command marks the parameter *name*s as readonly, such that they cannot be assigned values. The shell issues an error if you try to overwrite a *name*s value. A subshell does not inherit a variable's readonly setting. If you give no *name*s, all the readonly parameters are listed.

Example

```
$ who='who am i'
$ readonly who
```

This example sets `$who` to the output of the command_line `who am i`, and then marks `$who` so it can't be changed.

Note The POSIX Shell provides the `-p` option with the `readonly` command. The `-p` option lists all the `readonly` parameters, and formats the output so it is suitable for re-input to the shell as commands that achieve the same attribute-setting results.

return

Syntax

`return` [n]

The `return` command stops execution of a function and returns to the calling shell script with an exit status of n. If n is not specified, the returning status is that of the last command executed within the function. When `return` is invoked outside the boundaries of a function it acts as an `exit`.

Example

```
$ if grep xxx myfile > /dev/null  2>&1
> then :
> else return 4
> fi
$
```

This is the same example used in the `exit` section. The only difference in the scripts response is that it returns with the status of 4.

```
$ search() {
> if grep xxx "$1" > /dev/null  2>&1
> then return 1
> else return 0
> fi
>         }
$
$ search myfile
```

The first line defines a function called `search` which checks a given file for a string, "xxx". This function inverts the normal return value of `grep`. Therefore, if the string is found, the function returns 1, else if the string is not found or the file is not readable, it returns 0.

select

Syntax


```
select parameter in words
do
  command_lines
done
```

This command prints on the screen a set of *word*s each preceded by a number. Then the PS3 prompt is printed and the line typed by the user is read into the REPLY variable. If this line consists of the number of one of the listed *word*s, then the value of the *parameter* is set to the corresponding *word* and REPLY is set to the input line (i.e., the number). If this line begins with anything else, *parameter* is set to the null. If you input nothing, type Return, it reprompts for input. No matter which way it evaluates, the *command_line*s are executed. The loop continues until a break is encountered.

Example

```
$ select char in a e i o u
> do
>   echo $char is a vowel.
> done
1) a
2) e
3) i
4) o
5) u
#? 4          Return
o is a vowel.
#?            Break
```

all the vowels in *words* are printed out with a number in front. The default PS3 prompt, **#?**, is printed and waits for a number and Return to be typed in. When it receives the number, it echos that the corresponding letter is a vowel then prompts for the next entry. It continues prompting until you press Break. If you designate 6, which is not set, a null (`is a vowel`) is returned.

set

Syntax

For POSIX Shell only:

`set` [±`aCefnuvx`] [±`o` *option* ...] [*arg* ...]

For Korn Shell only:

`set` [±`aefhkmnostuvx`] [±`o` *option* ...] [*arg* ...]

This command is used to set shell options as well as reset the values of positional parameters (*arg*). See Chapter 23 for a detailed explanation of the various options available with `set`.

Example

```
$ set
```

If you just type `set`, it lists all your currently set shell variables.

```
$ set -f
$ echo x*y
$ set +f
```

In this example, you `echo x*y` without expanding it against all the filenames in the current directory. This is a result of the `-f` option which disables file name substitution. Using the `+` turns the previously set `f` option off.

```
$ set -o vi
```

This enables `vi`-mode history editing.

`set` is also used to set the arguments of an array:

```
$ set third first second
$ echo $1 $2 $3
third first second
```

shift

Syntax


```
shift [n]
```

The `shift` command moves the contents of positional parameters (`$1`, `$2`, `$3`, etc.) left one position such that `$1` now contains the former value of `$2` and `$2` contains the former value of `$3`, etc.

Example

```
yflag=0
zopt="

for arg in "$@"
  do
     if [ "x$arg" = x-y ]
     then  yflag=1;    shift
     else  zopt="$2";  shift 2
     fi
  done
```

In this shell script, `$yflag` is initialized to 0 and `$zopt` to the null string. It checks all the parameters (`$@`) passed to the script. If any one of them matches `-y`, `$yflag` is set to 1. Using `x$arg` avoids asking `test` (which is invoked by the brackets, `[ ]`) to interpret `-y` as an option. If any shell argument doesn't match `-y`, it saves the *next* argument (`$`) in `$zopt`. Using quotes preserves any whitespace embedded in `$2`. Also, note the shifting of arguments such that `$2` has the correct value when it is needed. `$@` is evaluated only once: before the first shift takes place.

test

Syntax

`test` *expr*

or

[*expr*]

or

`[[`*test_expression*`]]`

This command tests or evaluates the *expr* and if it evaluates *true* returns a zero exit status. If it evaluates *false* it returns a nonzero exit status. The `test` command can be replaced by appropriately spaced brackets ([ ]).

A extensive list of *expr* forms are covered in the *HP-UX Reference* in the *test*(1) entry. Four *expr*s limited to the POSIX and Korn Shell are:

`-L` *file*	Returns true if *file* is a symbolic link
file1 `-nt` *file2*	Returns true if *file1* is newer than *file2*
file1 `-ot` *file2*	Returns true if *file1* is older than *file2*
file1 `-ef` *file2*	Returns true if *file1* has the same device and i-node number as *file2*

One *expr* unique to the POSIX Shell is:

`-e` *file*	Returns true if file exists.

Example

```
$ for file in 'ls'
>    do
>     if [ -x $file ]
>     then echo $file is executable
>     fi
>    done
$
```


This script tests a file for executability, using brackets around the expression, and then prints that the file is executable if the `expr` returns true.

```
$ if [ $file -nt $oldfile  -a  $file -ot $newfile ]
> then echo $file is newer
> fi
```

This example, echos the filename only if it is newer then the filename in `$oldfile` and older than `$newfile`.

time

Syntax

`time` *command_line*

This keyword executes the *command_line* and then displays the execution time of the user, the system, and *command_line*.

Example

```
$ time ls
```

This line lists out the files in the current directory followed by three lines, `real, user, sys`, showing execution times.

Note `time` is not a keyword in the POSIX Shell.

times

Syntax


```
times
```

This command simply prints the accumulated user and system times, to the nearest hundredth of a second, for the shell and for processes run from the shell.

Example

```
$ times
```

trap

Syntax

`trap` [*arg*][*signal* ...]

This command waits for *signal*s sent to the shell and traps it. Then it executes, *arg*, a *command_line*. If *signal* is 0, *arg* is executed only once after the shell is exited. After setting traps, typing `trap` with no *arg*s lists all commands associated with signals.

Example

```
$ trap 'echo "Command failed."' ERR
```

This sets a trap which prints `Command failed.` on the terminal screen whenever a command run by the shell returns a non-zero value. See Chapter 23 for a detailed explanation of signals and traps.

typeset

Syntax

For Korn Shell only:

```
typeset [-HLRZfilprtux[n][name[=value]] ... ]
```

The `typeset` command sets the shell variable *name* equal to *value* whose type depends on the options used. When invoked inside a function, the *value* of the *name* is only temporary (i.e., local) until the function is exited; then the original *value* is restored.

If instead of the `-` in front of the options, a `+` is used, the type is turned off. If no options or specific options and no *name*s are given the parameters with those options are displayed.

Example

```
$ typeset -i num1 num2 total
$ typeset
$ typeset -r
```

This example defines the variables `num1, num2`, and `total` as integers. Then all the attributes of all the parameters are listed followed by the `-r` or readonly parameters. See Chapter 23 for a detailed explanation of all the options.

ulimit

Syntax

For Korn Shell only:

`ulimit` [`-f`][*n*]

This command sets limit *n* on certain resources a spawned process uses such as time, stack area, files sizes, etc. See Chapter 23 for a detailed explanation of all the various options.

Example

```
$ ulimit -f 1000
```

This line limits the size of files written by the shell or a spawned process to 1000 disk blocks.

umask

Syntax

For POSIX Shell only:

`umask` [`-S`] [*nnn*]

For Korn Shell only:

`umask` [*nnn*]

This command sets the user's file-creation mask to the string, *nnn*, unless *nnn* is omitted, then the current value of the mask is displayed.

Example

```
$ umask 022
```

If this line was in your `.profile`, it would set your process `umask` value to 022, which means a file created later will be 644 (`rw-r--r--`) rather than 666 (`rw-rw-rw-`), or 755 (`rwxr-xr-x`) instead of 777 (`rwxrwxrwx`). Actually, saying `umask 022` does not cause the execute (`x`) bits to be turned off, because they are normally not turned on at create time but later by `chmod` calls. `chmod` is discussed at greater length in the book *Introducing UNIX System V* by Morgan and McGilton.

Note

The POSIX Shell provides the `-S` [uppercase] option to the `umask` command. This option prints the mask in the symbolic mode. If this mode is specified (as mask), the permissions `+` and `-` will be interpreted relative to the current file mode creation mask. The `+` causes the bits to be cleared in the mask, and the `-` causes the bits to be set.

unalias

Syntax

`unalias` *name* ...

This command reverses the affect of the `alias` command on *name* and removes it from the alias list.

Example

```
$ alias cd='cd; ls'
$ unalias  cd
```

This creates an alias then removes it.

Note The POSIX Shell provides the `-a` option to the `unalias` command. This option removes all the aliases from the current shell environment.

unset

Syntax

24

`unset` [`-fv`] *name* ...

The `unset` command removes the specified *name* (or function) that has been set by the shell. You must use the `-f` option to unset a function, and the `-v` option to unset a variable. Variables with `readonly` set cannot be `unset`.

Example

```
$ param=6
$ echo $param
6
$ unset param
$ echo $param
```

The variable `param` is set to 0 and the `unset` unsets the variable.

wait

Syntax

`wait` [*process_id*]

The `wait` command suspends the shell until the spawned process *process_id* terminates and then reports the processes termination status. If *process_id* is not specified, currently active processes are waited for. The shell resumes after all processes terminate or when it receives a signal such as Break.

Example

```
$ contemplate gravity &
$ mailx
$ wait
$ rm gravity
```

In this example you run a very slow program (`contemplate`) in the background, then read your mail, and when done, `wait` for the background job to finish (if it hasn't already) before removing the file it used.

Note For POSIX Shell, the exit status of `wait` is the same as the exit status of the process requested by the last *process_id* operand in the `wait` command.

whence

Syntax

For Korn Shell only:

`whence` [`-v`] *name* ...

This command, limited to the POSIX and Korn Shell, indicates for each *name* how it would be interpreted if used as a command name. If the `-v` option is set, the results are more verbose.

Example

```
whence history
```

This example discovers that `history` is actually an exported alias for `fc -l`.

See Chapter 23 for a detailed explanation.

while/until

Syntax


```
while list1
do list2
done
```

This loop cycles through *list1* and executes all the items in *list2* if *list1* evaluates to true. Once *list1* evaluates false, the loop is exited.

Example

```
$ x=0
$ while [ $x != 10 ]
> do
>    let x=x+1
>    echo $x
> done
1
2
3
4
5
6
7
8
9
10
```

This loop initializes the variable `x`, increments and prints the value until it reaches 10 then exits the loop. The `until` loop has the same syntax as `while`. However, it executes until a non-zero is returned and always executes the loop at least once.

Index

Part IV
Key Shell

Korn Shell Softkey Interface

- Description of Key Shell
- Using Key Shell
- Visible Softkeys
- Invisible Softkeys
- Configuring Key Shell
- Softkey Navigation

25

Introducing the Key Shell

This chapter introduces the Key Shell (`keysh`), a menu-based softkey interface to the Korn Shell (`ksh`). It covers the following topics:

- Introduction to Key Shell.
- Who should use Key Shell.
- Conventions.

Introduction to Key Shell

The Key Shell was developed by Hewlett-Packard to provide a friendly user interface to the power of the Korn Shell. The Key Shell provides softkey menus and online help to let you build softkey commands for such tasks as printing files, viewing files, and listing the contents of directories. These softkey commands, which resemble normal English, are automatically translated into HP-UX syntax before being executed.

The Key Shell gives you all the capabilities of the Korn Shell, including the following:

- Command history buffer and history substitution.
- Extended in-line command editing.
- File name completion.
- Fast response time.
- Command aliasing mechanism.
- Bourne Shell programming environment.
- Integer arithmetic evaluation.
- Tilde substitution.
- Job control features.
- Arrays.

For more information on the Korn Shell, refer to the part in this manual titled "Korn Shell (ksh)."

In addition to the above Korn Shell capabilities, Key Shell provides the following features (listed in Table 25-1), which extend Korn Shell and make it easier to use:

Table 25-1. Key Shell Features

Feature	Description
Flexible user interface.	You can enter HP-UX commands by using the softkey menus or by typing on the command line.
Online help.	You can get online help for all softkey commands and their options, plus general help on shell topics and `keysh`'s features.
Softkey menus.	The top-level softkey menu displays softkey commands. Sub-menus list options for each command. You can use these softkeys to build a complete command without having to remember and type cryptic HP-UX commands or options.
Visible softkey commands.	Twenty-two pre-configured softkey commands appear on the top-level softkey menu. These softkeys allow you to perform HP-UX functions such as changing directories, listing files, and printing files.
Invisible softkey commands.	`keysh` also "recognizes" approximately 70 common HP-UX commands as you type them, and provides softkey sub-menus showing options for those commands.
Softkey command translation.	After building a softkey command, you can see it translated into HP-UX syntax before it is executed.
Status line.	A configurable status line near the bottom of the screen can display your hostname, username, current directory, mail status, time, date, and any other text you specify. The status line also displays prompts or messages from `keysh`.
User configurability.	You can customize `keysh`'s appearance and behavior. You can also add, move, and delete softkeys, and create custom softkeys and online help.
Support for keyboard editing keys.	You can use the standard editing keys, such as Delete line and Insert char, to edit the command line. You can edit the softkey command line or the translated command line.
Softkey error detection and correction.	For softkey commands, `keysh` immediately notifies you of syntax errors and prompts you to correct them before executing the command.

Who Should Use Key Shell

Key Shell provides features for both the novice and the experienced user:

- If you do not have much experience with HP-UX, you can use the pre-configured softkeys to enter basic HP-UX commands. You can also learn HP-UX syntax as you go: Key Shell shows you how your readable softkey commands translate into standard HP-UX.

  Read Chapter 26 to learn about starting, using, and configuring Key Shell.

- If you are a more experienced user but use certain HP-UX commands infrequently, the softkey options provided for those commands can serve as "memory joggers." You can also customize Key Shell to meet your needs. For example, you can create your own custom softkeys and online help.

  Read Chapter 26 to learn the basics of Key Shell and how to configure it, then refer to Chapter 27 for information on advanced use and customization.

Conventions

This part of the manual uses the following typographic conventions:

Boldface — Words defined for the first time appear in boldface. For example, **enabled** softkeys are softkeys that appear on the softkey menu.

`Computer` — Computer font indicates literal items either typed by the user or displayed by the system. For example:

```
/usr/keysh/C
```

Italics — Manual titles and emphasized words appear in italics, as do values that you supply.

For example, in the command below you would substitute the actual name of a file for *file_name*.

`more` *file_name*

Return — Words or letters in boxes refer to keys on the keyboard.

`Mail` — Words in shaded boxes refer to softkeys. Each softkey corresponds to a function key on your keyboard, f1 through f8.

Refer to *cp*(1). — References like this refer to entries in the *HP-UX Reference*. If you cannot find an entry in the manual where you expect it to be, use the *HP-UX Reference* index.

These entries can also be found online with the command `man` *entryname*. For example, `man cp`.

26

Getting Started With Key Shell

This chapter explains the basics of using Key Shell. All users should read this chapter. More advanced users should also read Chapter 27 for information on customizing Key Shell.

This chapter discusses the following topics:

- Starting Key Shell
- Using Key Shell.
- Configuring Key Shell.
- Setting shell variables.
- Using Key Shell with Terminal Session Manager.

Starting Key Shell

When you first log on to an HP-UX system, the Bourne Shell is the default shell. To use the Key Shell instead, first ensure that the following shell variables are set and exported in your `.profile` file. Depending on your terminal type, you may be able to use the *identify*(1) command to check these variables and set them correctly.

- `$TERM` must be set to the terminal type you are using.
- If your terminal is a non-standard size (such as an X-window), the `$LINES` and `$COLUMNS` variables must be set to the correct values for the terminal.

To start Key Shell, do *one* of the following:

1. *If you want to use* `keysh` *temporarily*, type this command:

   `keysh` (Return)

   Return to your regular shell by typing `exit` (Return).

2. *If you want to use* `keysh` *permanently*, type this command:

   `chsh` *your_username* `/usr/bin/keysh` (Return)

   Then log out and log back on again. From now on, whenever you log on, `keysh` is your shell.

26

The Default Key Shell Environment

When you first start `keysh`, you will see a display like that in Figure 26-1:

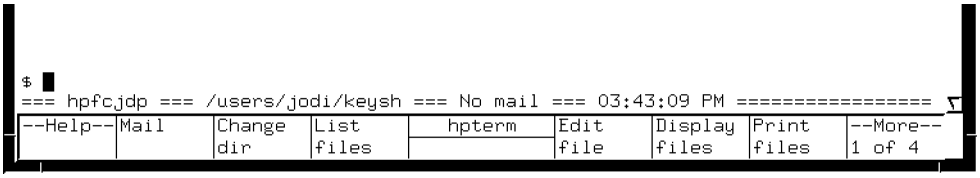

Figure 26-1. Key Shell Softkey Display

The parts of this display are as follows:

1. The first line shows the standard Korn Shell prompt, `$`.
2. The second line is the status line. By default, it displays the host name, current directory, mail status, and time.
3. The third line shows the top-level softkey menu. Each softkey corresponds to a function key on your keyboard, (f1) through (f8). The `hpterm` at the center separates the keys into groups of four.

   Note that there are four banks of available softkeys in the top-level menu. Select `--More--` several times to familiarize yourself with the softkeys on each bank.

Key Shell Initialization

This section describes what happens when you start Key Shell. For information on the `login` program and on setting up your `.profile` and `.kshrc` files, refer to the chapter titled "Starting and Stopping the Shell" in the "The Korn Shell (ksh)" part of this manual.

When you start `keysh`, it performs the following tasks:

- Customizes your shell environment by executing these files: `/etc/profile`, `.profile`, and the file indicated by the `$ENV` variable in `.profile` (typically `.kshrc`).

  `keysh` then resets the initial value of the `$PS1`, `$PS2`, and `$PS3` shell variables. Do not subsequently change these variables from the command line. Instead, you can use the `$KEYSH` variable to display status information (described in the section titled "Setting Shell Variables").

- Determines characteristics of your terminal, as specified by the `$TERM` shell variable. Also checks `$LINES` and `$COLUMNS`, if set.

- Configures itself, based on your local `.keyshrc` file. (If this file does not exist, `/usr/keysh/C/keyshrc` is used instead.)

  The `.keyshrc` file contains configuration information for the following:

  - Visible softkeys and their labels.
  - `keysh` global option settings.
  - Status line contents.

  Your `.keyshrc` is automatically updated every time you change Key Shell's configuration. You should not need to edit this file manually.

Using Key Shell

This section explains guidelines for using Key Shell, and describes how to get online help, enter commands, and edit commands. It includes examples for you to try.

Guidelines for Using Key Shell

The following list provides general guidelines for using Key Shell:

- Select softkeys by pressing the corresponding function key on your keyboard, or by clicking on the softkey with your mouse.
- Always select softkeys from left to right.

  Do not attempt to insert words or options out of order on the command line.
- Follow the prompt messages describing required actions.
- To see additional softkey commands or options, use the `--More--` softkey. This softkey toggles through banks of softkeys, showing your position.
- Option softkeys insert the corresponding command or option into the command line.

  For example, selecting `Change dir` inserts `Change_dir` into the command line.
- Enter your own text for string softkeys. **String softkeys** (or parameter softkeys) are enclosed in angle brackets, and indicate that you need to type text on the command line for that parameter or string, such as a user name or file name.

  For example, for the softkey `<file>` you would type a file name.
- If you make a mistake, use the Back space key to back up past the error and fix it. You can also edit the command line with the keyboard editing keys and arrow keys. For more information, refer to the section titled "Editing the Command Line."
- To see a softkey command translated into HP-UX syntax before being executed, use the Insert line key. (If you do not have an Insert line key, press Tab instead.)

- To execute the command directly, use the (Return) key. After you execute a command, you will remain on the current key bank, unless you set the `$KEYMORE` variable (refer to the section titled "Setting Shell Variables.")
- To cancel a command, use the (Delete line) key.
- If you encounter errors while using Key Shell, refer to the online help topic `keysh errors`.

To determine if a softkey command you have entered is complete and can be executed, look for the following cues:

1. Check that Key Shell is not displaying a prompt message that describes a required action, such as `Enter the name of the file.`
2. (Optional.) Press (Insert line) to translate the command to HP-UX syntax. Key Shell will display a prompt and refuse to translate the command if the command is incomplete.
3. Press (Return) to execute the command. Key Shell will display a prompt and refuse to execute the command if the command is incomplete.

   Note that you can press (Return) or (Insert line) regardless of the cursor's position on the command line.

Using Online Help

Online help is available for all pre-configured softkeys (listed in Table 26-3 and Table 26-4), and their options. In addition, the following help topics are available:

`using help`	How to use the online help.
`using keysh`	How to use Key Shell.
`editing`	Editing the command line.
`visibles`	Visible softkeys.
`invisibles`	Invisible softkeys.
`keysh errors`	Key Shell error messages.
`regexp_patterns`	Regular expressions and pattern matching.
`redirect_pipe`	Command input/output redirection and piping.

`--More--` at the bottom of a help message indicates that the help is more than one screen long. To continue reading, press the space bar.

To exit a help screen, press q.

Table 26-1 explains the methods for accessing help.

Table 26-1. Using the Online Help

To get help on ...	Do this:
A general topic.	Select `--Help--` `--Help-- (topics)`, then select a topic softkey.
A visible softkey command or softkey option, *before* you have selected the softkey.	Select `--Help--`, then select the softkey for which you want help.
A visible softkey command or softkey option, *after* you have selected the softkey.	Select `--Help--`, then press (Return).
An invisible softkey command.	Type the command. Select `--Help--`, then press (Return).

If you have entered more than one softkey on the command line and you want to see help for an earlier softkey, backspace to that softkey. Then select `--Help` and press (Return).

For example, to see the online help for `Print files` and its options, follow these steps:

1. Select `--Help--`.
2. Select `Print files`. The help for this command is displayed.
3. Select `Print files` again, to actually select the command.
4. Then select `--Help--` again, and select the `double spaced` option. The help for this option is displayed.

Note that simply typing `help` at the `keysh` prompt gives you a prompt from the *help*(1) command. This "help" has nothing to do with Key Shell online help.

Entering Commands

Key Shell provides three methods of entering commands:

- Using visible softkey commands.

  The softkey commands shown on the top-level softkey menu banks are called the **visible** softkeys (listed in Table 26-3).

- Using invisible softkey commands.

  `keysh` recognizes approximately 70 common HP-UX commands for which softkey options are available (listed in Table 26-4). These commands are called **invisible** softkey commands because they do not appear on the top-level softkey menus. However, when you type the commands, softkey options appear.

- Using standard HP-UX commands.

Each of these methods is described below. Figure 26-2 shows how you can use any of these methods to perform the same task.

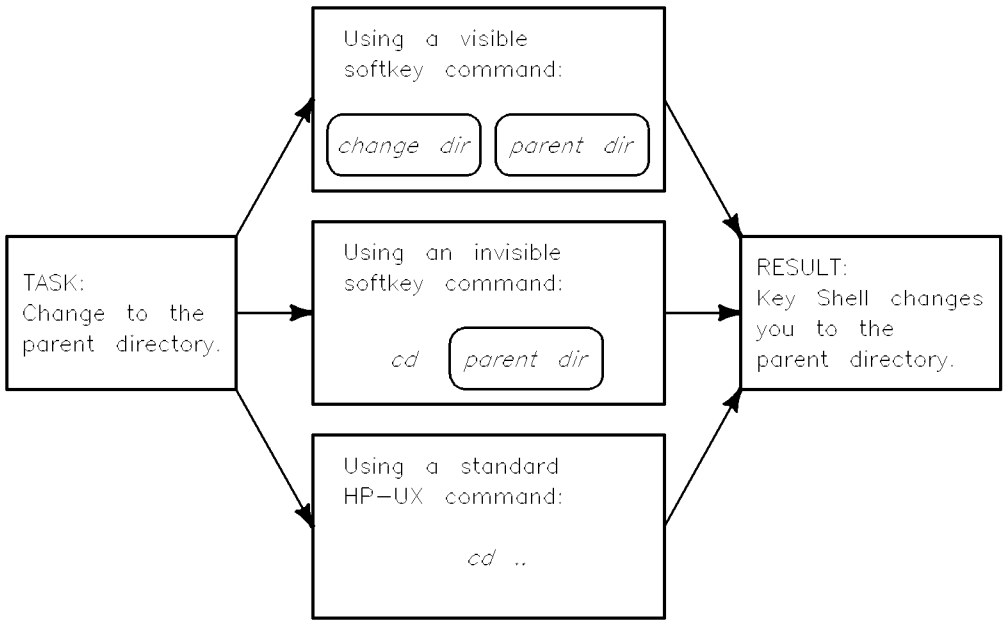

Figure 26-2. Entering Commands

Using Visible Softkey Commands

To use the visible softkey commands, select the softkey command, then select any options. Press (Return) to execute the command, or (Insert line) to see the command line translated into HP-UX syntax. `keysh` notifies you if you forget to supply any required information.

For example, to see how basic softkey command entry works, follow these steps:

1. After starting `keysh`, select the `Change dir` softkey.

   Your screen should look like that in Figure 26-3. Notice that the softkeys have now changed to show the options available for the `Change Dir` command.

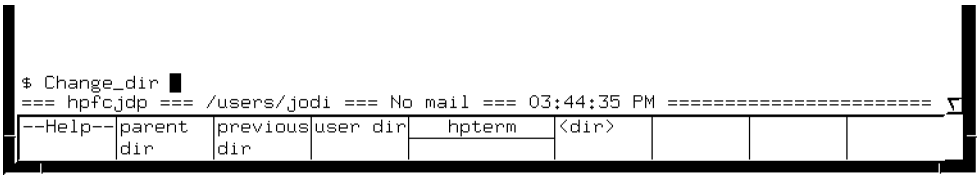

Figure 26-3. After Selecting the Change Dir Softkey

2. Select the `parent dir` option from the softkey options.

3. Press (Insert line). The readable softkey command you created is now translated into HP-UX syntax, but the command has not been executed yet. Your screen should look like that in Figure 26-4.

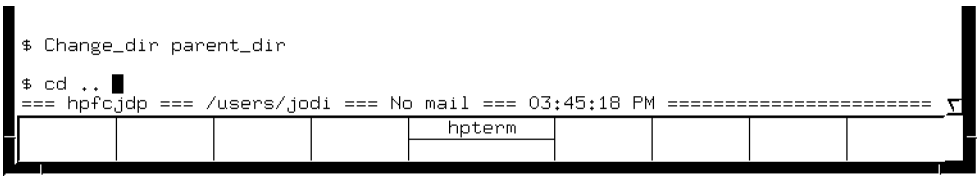

Figure 26-4. After Selecting the Parent Dir Option

4. Look at the directory shown on the status line, then press Return to execute the command.

   The `Change dir` `parent dir` command changes your current directory to its parent (the next higher directory). In the above example, the current directory is `/users/jodi`. The parent directory and new current directory would be `/users`.

   Notice also that the status line has changed to reflect the new current directory.

26

Using Invisible Softkey Commands

To use the invisible softkey commands, type a recognized HP-UX command (one of those listed in Table 26-4). `keysh` will display the softkey options for the command. Select any softkey options. Then press Return to execute the command, or press Insert line to see the command translated into HP-UX syntax. `keysh` notifies you if you forget to supply any required information.

For example, to see how the invisible softkey commands work, follow these steps:

1. At the `keysh` prompt (`$`), type the following. Do not press Return yet.

   ```
   cal
   ```

2. Because `cal` is a standard HP-UX command that `keysh` recognizes, it displays the available softkey options for the `cal` command.

3. Select the `for month` option softkey.

   Your screen should look like that in Figure 26-5. The softkey menu now shows the months of the year. Notice that two banks of `for_month` softkey options are available.

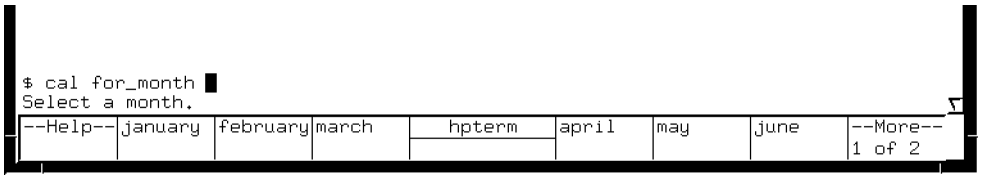

Figure 26-5. Using Invisible Softkeys

4. Select the `--More--` softkey to see the second bank of options. (Selecting `--More--` again returns you to the first bank of options.)
5. Select any month, then press Return.

   `keysh` displays the HP-UX equivalent, then executes the command.

Using Standard HP-UX Commands

You can also use `keysh` as you would any shell, by simply typing HP-UX commands and options. You do not have to use the softkeys at all.

For example, to see how the standard commands work, follow these steps:

1. At the `keysh` prompt (`$`), type `uname`. Do not press Return yet.
2. Ignore the softkey options that appear. Type a `-n`, then press Return.
3. Your screen should look like that in Figure 26-6. The `uname -n` command displays your node name.

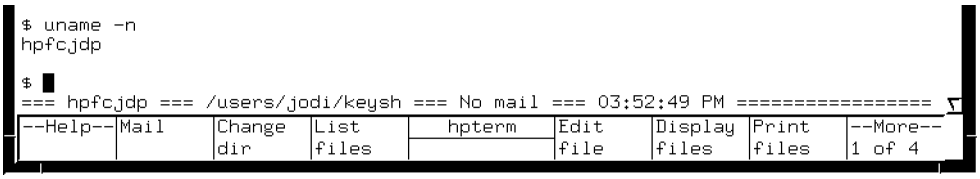

Figure 26-6. Using Standard HP-UX Commands

Editing the Command Line

Key Shell allows extensive command line editing to correct mistakes. You can edit either of two command lines:

- The readable command line that you built by using the softkeys.

  You can edit this command line even after you have pressed (Insert line) to translate it to HP-UX syntax. Simply press the (▲) key to retrieve your softkey command from the command buffer.

26

- The HP-UX command line that you typed, or that appeared when you pressed (Insert line) after building a softkey command line.

`keysh` supports the Korn Shell command line editing modes, such as `vi` mode. Refer to the chapter titled "Editing Command Lines" in the "The Korn Shell (ksh)" part.

You can also edit the command line by using the cursor movement and editing keys found on most terminals. These keys are listed in Table 26-2.

Use caution when mixing `vi` and key editing on the same command line. Key editing does not affect the `vi` editing mode.

Table 26-2. Editing Keys

Key	Function
Back space	Backs the cursor up on the command line, deleting one character at a time. The available softkey options will change to reflect your position on the command line.
Clear display	Clears the screen and command line. If the screen is scrolled, it clears from the cursor position to the end of the command line.
Clear line	Clears from the cursor position to the end of the command line.
Delete line	Cancels the command.
Insert line	Translates the readable softkey command line to the equivalent HP-UX command line.
Delete char	Deletes the character under the cursor.
Insert char	Toggles between insert and overwrite modes.
▲	Recalls the previous command from the command history buffer.
▼	Recalls the next command from the command history buffer.
◀	Moves the cursor left.
▶	Moves the cursor right.
Home	Moves the cursor to the beginning of the command line.
Shift-Home	Moves the cursor to the end of the command line.
Tab	If no Insert line key is present, performs the `insert line` function. Otherwise, if no `--Help--` softkey is present, performs the `help` function. Otherwise, performs the normal tab function.
Shift-Tab	Moves the cursor to the beginning of the previous word.
CTRL-L	Redraws the lower lines of the screen and restores any necessary terminal modes.

The following example shows how to edit a command line.

1. Select the `Print status` softkey (on the third bank of top-level softkeys).
2. Select the `all info` option.
3. Suppose you now decide to select the `scheduler info` option instead.

   Press ◀ or Shift-Tab to move to the beginning of the word `all_info`, then press Clear line to clear from the cursor position to the end of the command line.

   Alternately, press Back space to back up past `all_info`.
4. Select `scheduler info`.
5. Now suppose you decide to cancel the command altogether. To do this, simply press Delete line.

Configuring Key Shell

You can configure Key Shell's appearance and behavior through several options. These options are accessed with the `Keysh config` softkey or the special `kc` command.

You can make the following changes to `keysh`:

- Adding, moving, and deleting softkeys.
- Changing global options.
- Changing the status line.
- Saving configuration changes.
- Restarting `keysh`.
- Undoing any configuration changes.

The following sections describe how to do each of the above.

Adding, Moving, and Deleting Softkeys

This section explains how to add visible and invisible softkeys, move softkeys, and delete softkeys.

If you encounter errors while adding softkeys, refer to the online help topic `Errors`.

Softkey Names and Labels

All softkeys have names, which are the HP-UX commands they correspond to. Visible softkeys (the softkeys that appear on the top-level softkey menu) also have labels, which are the words that appear on the softkey itself. Labels allow you to give a less cryptic name to a command. For example, `Search_lines` is the less cryptic label given to the `grep` command. A label can have a maximum of 16 characters.

Many of the following sections ask you to type a softkey's name or label. If you refer to a softkey by its label, you must replace any blank space in the label with an underscore "`_`". For example, type `Edit_file` or `Create_dir`.

Adding Visible Softkeys

The pre-configured visible softkeys are listed in Table 26-3. These softkeys appear in the top-level softkey menu. You can also create your own visible softkeys or add any of the invisible softkeys listed in Table 26-4 and make them visible. Creating your own softkeys is described in Chapter 27.

To add a visible softkey, follow these steps:

1. Select `Keysh config softkey add`.

2. Type the name of an invisible softkey as listed in Table 26-4, or the name of a softkey you have created.
3. By default, the softkey label that will appear on the menu is the same as the softkey's name.

   To specify a label that is easier for you to remember, select `with label`, then type the label.
4. By default, the softkey is added from `/usr/keysh/C/softkeys`. If you want to add softkeys from another file, do either of the following:

   a. Select `from file`, then type the name of the file containing the softkey you want to add.

   b. Select `from user`, then type the name of the user whose `$HOME/.softkeys` file contains the softkey.

   Note that when you add a softkey, the remaining softkeys from that file are automatically loaded for use as invisible softkey commands.
5. To place the softkey in the default position (after all the other softkeys), press Return.

   Otherwise, select `and place`, select where the softkey should be placed in the menu, and press Return.

   If you select `and place before softkey`, you will be prompted to type the name or label of the existing softkey before which you want the new softkey to be placed.

Following are some examples that illustrate adding visible softkeys:

- To add the `od` softkey to the end of the top-level softkey menu and label it `Octal dump`, use this command:

  ```
  Keysh config softkey add od with label Octal_dump Return
  ```

- To add the `paste` softkey to the beginning of the top-level softkey menu, use this command. The softkey will be labelled `Paste` by default.

  ```
  Keysh config softkey add paste and place
  as first softkey Return
  ```

- To add the custom `emacs` softkey from the file `/users/rpt/.softkeys` to the top-level softkey menu immediately before the `ls` softkey, use this command:

  ```
  Keysh config softkey add emacs from user rpt
  and place before softkey ls Return
  ```

- To change the definition of the `Mail` softkey so that it calls the `elm` mailer instead of the `mailx` mailer, use the following command.

  ```
  Keysh config softkey add elm with label Mail
  and place as first softkey Return
  ```

Table 26-3 shows the visible softkey commands that are configured in `keysh`:

Table 26-3. Visible Softkey Commands

Softkey	HP-UX Equivalent	Function
`Mail`	`mailx`	Processes electronic mail interactively.
`Change dir`	`cd`	Changes the current directory.
`List files`	`ls`	Lists the contents of a directory.
`Edit file`	`vi`	Edits files on a screen-oriented display.
`Display files`	`more`	Displays the contents of a file one screen at a time.
`Print files`	`pr | lp`	Formats a file and sends it to the line printer.
`Search lines`	`grep`	Searches for lines matching a pattern.
`Sort lines`	`sort`	Sorts the lines of a file.
`Find files`	`find`	Locates files within a directory.
`Copy files`	`cp`	Copies file to another location.
`Move files`	`mv`	Moves or renames a file.
`Set file attribs`	`chmod, chown, chgrp`	Changes permissions, owner, or group of a file.
`Remove files`	`rm`	Deletes a file.
`Remove dirs`	`rmdir`	Deletes a directory.
`Create dirs`	`mkdir`	Creates a new directory.
`Shell archive`	`shar`	Bundles one or more files into a shell archive package for mailing or moving.
`Print status`	`lpstat`	Shows current status of all printers.

Table 26-3. Visible Softkey Commands (continued)

Softkey	HP-UX Equivalent	Function
`Cancel print`	`cancel`	Cancels a print request.
`Process info`	`ps`	Shows status of active processes.
`Kill process`	`kill`	Terminates a process.
`Manual page`	`man`	Accesses the online manual pages.
`Keysh config`	`kc`	Configures the appearance and behavior of `keysh`.

Adding Invisible Softkeys

The pre-configured invisible softkeys are listed in Table 26-4. These commands do not show up on the softkey menu, but if you type one of them, `keysh` will recognize it and display the appropriate softkey options.

You can also create your own invisible softkeys, as described in Chapter 27.

To add additional invisible softkeys, follow these steps:

1. Select `Keysh config` `softkey` `add` `invisibles`.
2. By default, invisible softkeys are added from `/usr/keysh/C/softkeys`. If you want to add invisible softkeys from another file, do either of the following:
   a. Select `from file`, then type the name of the file containing the softkeys you want to add. Then press Return.
   b. Select `from user`, then type the name of the user whose `$HOME/.softkeys` file contains the softkeys. Then press Return.

For example, to add all invisible softkeys from user `rpt`, use this command:

`Keysh config` `softkey` `add` `invisibles`
`from user` `rpt` Return

Table 26-4. Invisible Softkey Commands

Softkey	Function
`adjust`	Performs simple text formatting.
`ar`	Creates and maintains library archives.
`bdf`	Displays free disk space.
`cal`	Displays a calendar.
`cancel`	Cancels a print request.
`cat`	Concatenates and displays files.
`cd`	Changes the current directory.
`cdb`	C programming language symbolic debugger.
`chatr`	Changes a program's internal attributes.
`chgrp`	Changes the group of a file.
`chmod`	Changes the permissions of a file.
`chown`	Changes the owner of a file.
`cmp`	Compares two files and notifies you of any differences.
`col`	Used with *nroff*(1) to filter reverse linefeeds and backspaces.
`comm`	Prints all the lines common to two sorted files.
`cpio`	Copies file archives.
`cut`	Cuts selected fields or columns from a file.
`dd`	Copies a tape or file.
`df`	Displays the number of free 512-byte blocks and free inodes on a file system.
`diff`	Compares two files and notifies you of any differences.
`dircmp`	Compares two directories and notifies you of any differences.
`disable`	Disables `lp` printers.
`du`	Displays disk usage for files or directories.

Table 26-4. Invisible Softkey Commands (continued)

Softkey	Function
`elm`	Processes electronic mail interactively.
`enable`	Enables `lp` printers.
`exit`	Terminates the shell.
`find`	Locates files within a directory.
`fold`	Wraps text lines that exceed maximum width.
`grep`	Searches for lines matching a pattern.
`head`	Displays the first ten lines of a file.
`jobs`	Displays all active jobs.
`kill`	Terminates a process.
`lp`	Sends files to an `lp` printer or plotter.
`lpstat`	Shows current status of all `lp` printers.
`ls`	Lists the contents of a directory.
`mailx`	Processes electronic mail interactively.
`make`	Maintains, updates, and regenerates groups of programs.
`man`	Accesses the online manual pages.
`mkdir`	Creates a new directory.
`more`	Displays the contents of a file one screen at a time.
`nroff`	Formats text for printing.
`od`	Creates an octal dump of a file.
`paste`	Merges the same line in several files or subsequent lines of one file.
`pg`	Displays the contents of a file one screen at a time.
`pr`	Formats text for printing.

Table 26-4. Invisible Softkey Commands (continued)

Softkey	Function
`ps`	Shows the status of active processes.
`remsh`	Executes a command on a remote host.
`rlogin`	Connects your terminal to a remote host.
`rm`	Deletes files or directories.
`rmdir`	Deletes directories.
`sdiff`	Compares two files and displays a side-by-side listing of any differences.
`set`	Sets shell options.
`shar`	Bundles one or more files into a shell archive package.
`sort`	Sorts the lines of a file.
`tail`	Displays the last ten lines of a file.
`tar`	Creates, maintains, and accesses a file archive on tape.
`tcio`	Improves data transfer rate to cartridge tape. Commonly used with *cpio*(1).
`tee`	In a command pipeline, copies data passing between commands to a file.
`touch`	Updates the access, modification, and change times of a file.
`tr`	Translates characters.
`umask`	Sets file-creation mode mask.
`uname`	Displays the name of the current HP-UX version.
`vi`	Edits files on a screen-oriented display.
`wc`	Counts lines, words, and characters in a file.
`who`	Lists who is logged on to the system.
`write`	Interactively writes to another user.
`xd`	Creates a hexadecimal dump of a file.
`xdb`	C, FORTRAN, and Pascal symbolic debugger.

Moving Softkeys

To change the placement of a visible softkey, follow these steps:

1. Select `Keysh config softkey move`.
2. Type the name or label of the softkey you want to move.
3. Select where you want the softkey to be moved, then press Return.

   If you select `before softkey`, you will need to type the name or label of the existing softkey before which you want the other softkey to be moved.

For example, to move the `Mail` softkey to immediately before the `Keysh config` softkey, use this command:

```
Keysh config softkey move Mail before softkey
Keysh_config Return
```

Deleting Softkeys

To delete a visible softkey, follow these steps:

1. Select `Keysh config softkey delete`.
2. Type the name or label of the softkey you want to delete.
3. Press Return.

For example, to delete the `Edit file` softkey from the top-level softkey menu, use this command:

```
Keysh config softkey delete Edit_file Return
```

Note that after deleting a softkey from the softkey menu, you can still access the softkey invisibly.

Changing Global Options

This section explains the global configuration options found under the `Keysh config options` softkey menu. Global options allow you to control such things as which type of softkeys are available for use, whether HP-UX translations are displayed, and whether prompts are given.

In the `Keysh config options` menu, an asterisk next to an option name means that the option is on. Global configuration options can be turned on and off with the following commands:

`Keysh config options` *option_softkey* `on`

`Keysh config options` *option_softkey* `off`

For example, to turn off the prompts, use this command:

`Keysh config options prompts off` Return

Table 26-5 describes the global options:

Table 26-5. Global Options

Option	Default	Description
`help`	Enabled.	If you set `help` to off, the `--Help--` softkey disappears, but online help is generally still available by using the Tab key.
`invisibles`	Enabled.	If you set `invisibles` to off, `keysh` will not recognize invisible softkey commands.
`prompts`	Enabled.	If you set `prompts` to off, `keysh` will not display prompt messages describing actions that are required to complete the current softkey command.
`selectors`	Disabled.	If you set `selectors` to on, an uppercase selector character appears in each softkey label. Typing this character (unquoted) selects the softkey. `keysh` automatically sets `selectors` to on if you are using a terminal that does not support a sufficient number of softkeys.
`translations`	Enabled.	If you set `translations` to off, `keysh` will not display the HP-UX translations of softkey commands before executing them.
`visibles`	Enabled.	If you set `visibles` to off, `keysh` will not display softkey commands on the top-level softkey menu. If you are familiar with HP-UX commands (but not necessarily with the options), you may wish to set `visibles` to off. You can then decrease `keysh` start-up time by editing `.keyshrc` and removing the lines that add visible softkeys.

Changing the Status Line

In the `Keysh config status line` menu, an asterisk next to a status line indicator means that the indicator is on. You can turn the various status line indicators on and off with the following commands:

`Keysh config status line` *indicator_softkey* `on`

`Keysh config status line` *indicator_softkey* `off`

For example, to add your user name to the status line, use this command:

`Keysh config status line user name on` Return

Table 26-6 describes status line indicators:

Table 26-6. Status Line Indicators

Indicator	Default	Description
`user name`	Disabled.	Your user name.
`host name`	Enabled.	Your host name.
`current_dir`	Enabled.	The current directory.
`mail_status`	Enabled.	The mail status ("You have mail", "No mail", or "You have new mail").
`date`	Disabled.	The date.
`time`	Enabled.	The time of day.

In addition, you can use the `$KEYSH` shell variable to add any arbitrary text to the status line. This text is always displayed first. You can include the same type of information in `$KEYSH` that you normally would in the `$PS1` variable. For more information, refer to the section titled "Setting Shell Variables."

Saving Configuration Changes

If you want to manually write the configuration changes you have made to your `.keyshrc` file, select `Keysh config write`. Note that `keysh` automatically writes configuration changes as you make them.

For example, if you have configured `keysh` differently in two windows and you do not want to keep one of the configurations, go to the window with the configuration you want and select `Keysh config write` to write that configuration to `.keyshrc`. Then go to the other window and restart `keysh`.

Restarting Key Shell

After changing `keysh`'s configuration in one window, you can update the configuration of any other windows by selecting `Keysh config restart` in those windows.

If you want `keysh` to restart from the original default configuration, select `Keysh config restart default`. Any changes you have made to `.keyshrc` will be lost.

Undoing Configuration Changes

If you want to undo the configuration changes you made, select `Keysh config undo`. This command undoes all configuration changes made since `keysh` was last invoked or since the last `Keysh config undo` command. It then rewrites your `.keyshrc` file to reflect the undone changes.

Selecting `Keysh config undo` a second time will restore your configuration changes.

Setting Shell Variables

You can set any of the shell variables listed in Table 26-7. However, the default values will normally be sufficient. For more information, refer to the section titled "Setting Environment and Shell Variables" in the chapter titled "Starting and Stopping the Shell" in the "The Korn Shell (ksh)" part of this manual.

You can set shell variables in either of two ways:

- *Temporarily, on the command line.* For example, to force `keysh` to simulate softkeys rather than using the built-in labels, type:

  ```
  KEYSIM=true Return
  ```

  Use the `unset` command to unset a variable. For example, to return `keysh` to using the built-in labels, type:

  ```
  unset KEYSIM Return
  ```

- *Permanently, in your* `.profile` *or* `.keyshrc` *file.* For example, to make the phrase "Hi there" a permanent part of your status line, add the line below to your `.profile` file.

  ```
  export KEYSH="Hi there" Return
  ```

Table 26-7. Key Shell Variables

Variable	Description
`$COLUMNS`	Defines the width of the edit window (if different than the *terminfo*(4) default).
`$KEYBEL`	Specifies the character sequence to be used for **keysh**'s bell. The default is CTRL-G (ASCII 007). To disable the bell, set this variable to an empty string.
`$KEYENV`	Specifies an alternate **keysh** configuration file. The default is `$HOME/.keyshrc`.
`$KEYESC`	Specifies the maximum number of milliseconds allowed between characters that are part of a terminal escape sequence. The default is 350.
`$KEYKSH`	If set, causes **keysh** to mimic the behavior of the Korn Shell. The softkeys and status line are not displayed. This mode is useful over slow modem lines.
`$KEYLOC`	If set, specifies that **keysh** should leave the terminal keypad in local mode while commands are being entered. This mimics the behavior of the Korn Shell.
`$KEYMORE`	If set, specifies that **keysh** should return to the first bank of softkeys after executing a command, rather than remaining on the bank where the command was located.
`$KEYPS1`	If set, specifies that **keysh** should not reset the initial values of `$PS1`, `$PS2`, and `$PS3`.
`$KEYSH`	Lets you specify arbitrary text to include in the **keysh** status line.
`$KEYSIM`	If set, specifies that **keysh** should simulate softkey labels rather than using the built-in labels on HP terminals. When you use simulated softkey labels, **keysh** also shows an input mode indicator between the two groups of four softkey labels. This indicator shows your current command line editing mode. For example, `vi insrt` means that *vi*(1) is the editor and it is in insert mode.

Table 26-7. Key Shell Variables (continued)

Variable	Description
`$KEYTSM`	If set and Terminal Session Manager is running, specifies that `keysh` should not display TSM softkeys.
`$LANG`	Defines the language in which softkeys and Key Shell messages are displayed. The default is `C`.
`$LC_TIME`	Defines the format to use for time and date display on the status line. The default is `american`.
`$LINES`	Defines the number of lines in the terminal screen (if different than the *terminfo*(4) default).
`$PAGER`	Defines the pager to use for displaying online help. The default is *more*(1).
`$PATH`	Defines the search path for commands.
`$TERM`	Defines the terminal type.
`$TZ`	Defines the time zone to use for time and date display on the status line.

Using Key Shell with Terminal Session Manager

If you are using Key Shell with the Terminal Session Manager (*tsm*(1)), note the following considerations:

- `keysh` shows the `tsm` session number in the status line.
- If you want to use the softkey session switching capability in TSM, turn off `keysh`'s visible softkeys with this command:

  ```
  Keysh config options visibles off
  ```

- If you want session switching capability and also want to use the visible softkeys, do one of the following:
  - Use CTRL-W # to switch sessions, where # is the number of the session to which you want to switch.
  - Add the pre-configured `Switch` softkey as a visible softkey with this command:

    ```
    Keysh config softkey add switch
    ```

TSM is described in detail in the *Terminal Session Manager User's Guide*.

27

Customizing the Key Shell

Chapter 26 explained how to configure Key Shell and described the pre-configured softkeys available. This chapter explains how you can customize Key Shell to suit your needs. If you are a more advanced user and are familiar with HP-UX, the information in this chapter can help you make Key Shell even more powerful.

This chapter discusses the following topics:

- Understanding Key Shell.
- Adding text to softkeys.
- Creating custom softkeys.

Understanding Key Shell

This section explains the background and concepts that are necessary for customizing Key Shell. You can apply these concepts to create your own custom softkeys and online help.

The following list gives some example uses for custom softkeys:

- If you frequently change to certain directories, you could create your own `Change dir` command and add the directories you most often move to as options.
- If you often invoke an application, you could create a softkey command for the application and add the application's options in a sub-menu.
- If you frequently use an HP-UX command with the same set of options, you could create a softkey command for the entire HP-UX command line (the

command plus its options). This softkey command acts as a macro, so you do not need a sub-menu for options.

- If you often send electronic mail to the same people, you could add the names of those people as options to your own version of the `Mail` softkey.

How Key Shell Stores Softkey Information

`keysh` stores softkey information in softkey files. Softkeys are defined as a hierarchy of **softkey nodes**. The top level of this hierarchy represents the softkey command itself. The lower levels represent command options and parameters.

Figure 27-1 illustrates the softkey node hierarchy, using the `Keysh config` softkey as an example:

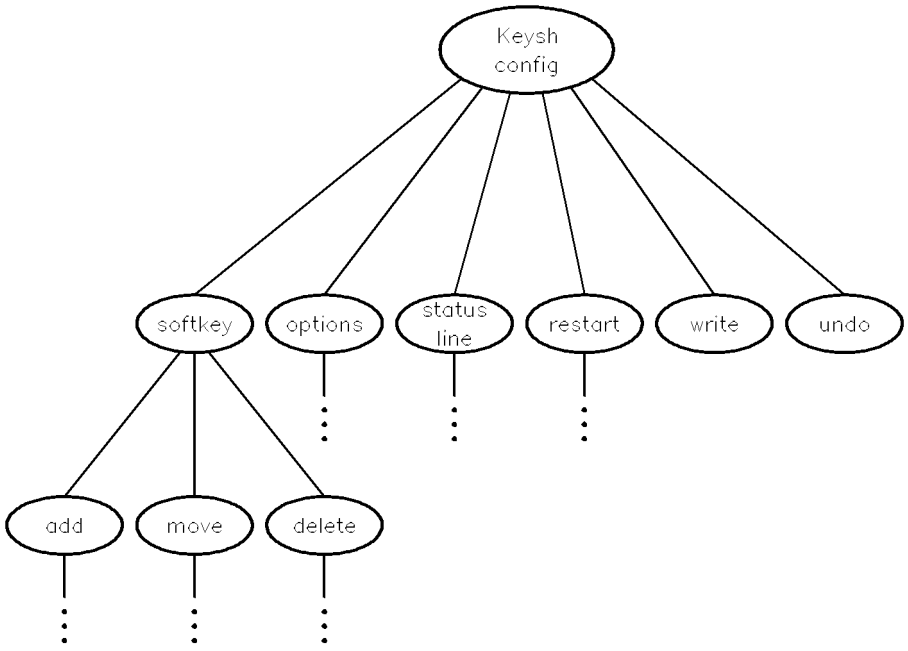

Figure 27-1. Example Key Shell Node Hierarchy

Softkey Navigation

As you build a softkey command, you navigate through the hierarchy of softkey nodes. At any one time, the softkey menu displays only the nodes that you can select. These are the **enabled** nodes; **disabled** nodes are not displayed. When you select a node, other nodes on the same level may be enabled or disabled as appropriate.

`keysh` initially displays a top-level softkey menu. When you select a softkey with a sub-menu, `keysh` displays the appropriate softkeys for the sub-menu. When you have finished selecting sub-menu options, `keysh` re-displays any remaining softkeys in the parent menu. For example, the `Keysh config` options in Figure 27-1 are mutually exclusive. Selecting the `status line` node disables all other nodes at the same level so that they are not subsequently re-displayed.

A node is displayed and can be selected if either of the following is true:

- The node was enabled by default and was not subsequently disabled by the selection of another node.
- The node was disabled by default, was not subsequently disabled by the selection of another node, and was enabled by the selection of another node.

How Key Shell Defines Softkeys

Key Shell's pre-defined softkeys are in the `/usr/keysh/C/softkeys` file. Each softkey node has the following major components:

name	(Required.) The *name* is the command string that you type to access an invisible softkey. It is also the default label for the visible softkey.
type	(Required for sub-menu nodes.) The *type* defines whether sub-menu softkeys are options or strings.
attributes	(Optional.) *Attributes* define the behavior of the softkey and how the softkey is interpreted.
editrules	(Optional.) *Editrules* are part of the softkey attributes, and define how the softkey is translated into HP-UX syntax.

The basic top-level softkey node definition has the following format:

```
softkey name
    attributes
    editrules
;
```

This example shows the top-level softkey definition for `Copy files`:

```
softkey Copy_files command
editrule { append("cp"); }
```


If a softkey has associated sub-menus, the trailing ; is replaced with a list of child softkey nodes enclosed in curly braces ("{" and "}"). These sub-menus correspond to the lower levels of the hierarchy.

```
{
    type name
    attributes
    editrules
    ;
    .
    .
    .
}
```

Child nodes can be of two types:

`option` Options appear on softkey labels and insert literal text into the command line when selected. Examples include the `Print files` command and the `double spaced` option.

`string` Strings (or parameters) appear on softkey labels but do not insert text into the command line when selected. Instead, they display a hint message in the status line. The hint message prompts you to type your own text into the command line. Examples of string softkeys include `<files>` and `<user>`.

Note that the name of a string softkey must be enclosed in angle brackets.

This example shows the sub-menu options for the `Copy files` softkey:

```
{
  string <files> disable -1 enable all
  editrule { append(argument); }
  required "Enter the name of the file(s) to copy."
  ;
  option to disabled
  required "Enter the name of the file(s) to copy;
            then select \"to\"."
  {
    string <dest>
    editrule { append(argument); }
    required
      "Then, enter the name of the file or directory to
       copy the file(s) to."
    ;
  }
}
```

Softkey Attributes

Softkey attributes define the behavior of softkey nodes. For example, attributes specify prompts and online help, indicate if selecting the softkey enables or disables other softkeys, and specify how the command should be translated into HP-UX syntax.

Softkey nodes can have the following attributes, as shown in Table 27-1. All attributes are optional.

Table 27-1. Softkey Attributes

Attribute	Meaning	Example
`disable` *count*	Selecting this softkey node disables *count* softkey nodes to the right of this one and all the nodes to its left. *count* can be a positive or negative integer, or the word "all." The default is 0.	The line `option owner disable all` in a softkey definition means that when the `owner` option is selected, all other nodes on the same level are disabled.
`enable` *count*	Selecting this softkey node enables *count* softkey nodes to the right of this one and all the nodes to its left. *count* can be a positive or negative integer, or the word "all." The default is 0.	The line `option search enable 12` in a softkey definition means that when the `search` option is selected, the next 12 nodes on the same level are enabled.
`{filter | command}`	This softkey node is only active for filters or commands. A filter is a command used after a pipe "|". A command is used by itself or as the first command in a pipeline. By default, a softkey node can be used in either place.	The line `softkey Move_files command` in a softkey definition means that the `Move_files` softkey can only be used as a command by itself or as the first command in a pipeline.
`disabled`	This softkey node is initially disabled and must be enabled to be used. By default, softkeys are initially enabled.	The line `option to disabled` in a softkey definition means that the `to` option cannot be selected unless you first select another softkey that enables it.

Table 27-1. Softkey Attributes (continued)

Attribute	Meaning	Example
`automatic`	Selecting this softkey node will automatically execute the command, without the user having to press Return. By default, you must press Return to execute a command.	The line `option 1 automatic` in a softkey definition means that when you select `1`, a Return is included and the command executes automatically.
`editrule` *editrule*	An editrule for this softkey node, which defines how it is translated into an HP-UX command.	(Editrules are discussed in the section titled "How Key Shell Translates a Softkey Command.")
`hint` *string*	The one line hint for this softkey node, displayed when the softkey is selected. Hints are only valid for string type softkey nodes.	(Hint messages are discussed in the section titled "Adding Text to Softkeys.")
`required` *string*	The one line error message for this softkey, displayed if you do not select a required softkey or type a required string.	(Required messages are discussed in the section titled "Adding Text to Softkeys.")
`help` *helptext*	The help message for this softkey. It can be more than one line long.	(Help is discussed in the section titled "Adding Text to Softkeys.")

How Key Shell Translates a Softkey Command

When you have entered a complete softkey command and pressed Insert line or Return, `keysh` translates the command into HP-UX syntax by applying the *editrules* associated with each of the selected softkey nodes. Editrules are part of the attributes of a softkey node. They tell `keysh` what HP-UX command or option is associated with each part of the softkey command, where the HP-UX command or option belongs in the translated command line, and where any user-typed text belongs.

The HP-UX command that `keysh` constructs can be thought of as a list of words. Each word corresponds to a part of the HP-UX command, and has an index according to its position in the command line. As `keysh` constructs the HP-UX command, it uses the position of other words in the command line to determine where to place options or text. The completed list of words, each separated by a space, is then passed to the Korn Shell for execution.

The following example shows a softkey command, its translation to HP-UX, and the corresponding word list:

`Remove files`	`interactively`	`*.c`	*Softkey command.*
`rm`	`-i`	`*.c`	*HP-UX translation.*
word[0]	*word[1]*	*word[2]*	*Corresponding word list elements.*

Editrules

The editrules associated with softkey nodes contain instructions for manipulating the words in the word list. Editrules can be used to add, delete, or modify words.

When an editrule is invoked, the following constants are set:

- `last` is defined to be the index of the last word in the word list ("2" in the preceding example).
- `next` is defined to be the index of the word that would be next in the word list ("3" in the preceding example).
- `argument` is defined to be the user input for the softkey ("`*.c`" in the preceding example).

Syntactically, an editrule is a list of edit statements enclosed in curly braces ("{" and "}"). Edit statements can be any of the following:

- An expression followed by a semicolon (";").
- An `append` statement.
- A `dash` statement.
- An `if` statement.

These four types of edit statements are explained in the following sections.

Expressions. Table 27-2 shows the simple expressions you can use in `keysh`. Key Shell stores all expressions as strings. The expressions then take on string or numeric values as appropriate.

Simple expressions can be combined to form more complex expressions, as shown in Table 27-3. Note that `keysh` expression combinations are like those in C, with the following two exceptions:

- C does not have a multiple character substring.
- The "binary and" symbol in C (`&`) is the "concatenation" symbol for `keysh`.

Simple expressions can also be assigned values, as shown in Table 27-4. Again, value assignment is the same as in C.

Table 27-2. Simple Expressions

Expression	Meaning	Example
variable	A single letter from a to z.	`x`
number	An unsigned integer.	`31`
string	Any literal text, enclosed in double quotes.	`"egrep"`
char	Any literal character, enclosed in double quotes.	`"v"`
`word[`*index*`]`	Indicates the word located at that index in the softkey command being built.	`word[next]`
`last`	Indicates the last position in the softkey command being built.	
`next`	Indicates the position that would be next in the softkey command being built.	
`argument`	Indicates the user input for the softkey.	

You can combine simple expressions in any of the ways shown in Table 27-3:

Table 27-3. Combining Expressions

Combination	Meaning	Example
string`[`*number*`]`	Single character substring.	`word[last][0]`
string`[`*number*`,`*number*`]`	Multiple character substring.	`a[0,1]`
number`+`*number*	Addition.	`argument+1`
number`-`*number*	Subtraction.	`argument-1`
string`&`*string*	Concatenation.	`"-w"&argument`
`-`*number*	Negation.	`-1`
string`==`*string*	Equality.	`word[last][0]=="+"`
string`!=`*string*	Inequality.	`x!=last`
number`&&`*number*	Logical and.	`x&&y`
number`||`*number*	Logical or.	`x||y`
`!`*number*	Logical not.	`!x`
`(`*string*`)`	Grouping.	`(argument+1)`

Table 27-4 shows how you can assign values to variables or indexed words.

Table 27-4. Assigning Values

Assignment	Meaning	Example
variable=*string*	Simple assignment.	`x=argument`
variable`+=`*number*	Add and assign.	`x+=1`
variable`-=`*number*	Subtract and assign.	`x-=1`
variable`&=`*string*	Concatenate and assign.	`x&=argument`
`word[`*index*`]=`*string*	Simple assignment.	`word[x]="egrep"`
`word[`*index*`]+=`*number*	Add and assign.	`word[last]+=1`
`word[`*index*`]-=`*number*	Subtract and assign.	`word[last]-=1`
`word[`*index*`]&=`*string*	Concatenate and assign.	`word[last]&="v"`

Append Statement. The `append` statement is used to add a string to the command being built. It has this format:

```
append("string");
```

The `append` statement causes *string* to be appended as a new word in the word list immediately after the last word.

For example, this editrule appends `cp` to the command line when you select `Copy files`:

```
softkey Copy_files command
editrule { append("cp"); }
```

In the following example, the `required` line prompts the user to type the *dest*, the name of the destination file or directory. This user-supplied argument is then appended to the command line.

```
string <dest>
editrule { append(argument); }
required "Then, enter the name of the file or
          directory to copy the file(s) to."
```

Dash Statement. The `dash` statement is used to add options to a command. Many HP-UX commands allow you to specify multiple options with one dash, rather than using a separate dash for each option. For example, you can type `ls -Fla` rather than `ls -F -l -a`.

The `dash` statement looks like this:

```
dash("string");
```

If the last word in the word list begins with a dash, *string* is appended to that word. Otherwise, a dash is appended as a new word in the word list immediately after the last word, and *string* is appended to that dash.

For example, if you had already selected an option that appended `-c` to the command, this editrule would add only the `b` to form `-cb`.

```
option without_tabs
  editrule { dash("b"); }
```

If Statement. The `if` edit statement allows you to specify a condition for executing an edit statement. The general syntax of the `if` statement is as follows. Note that the `else` is optional.

```
if (condition) {
    edit statement
      .
      .
}  else {
    edit statement
      .
      .
}
```

If *condition* is true (evaluates non-zero), the first block of *edit statement*s is executed. Otherwise, the second block of *edit statement*s (if it exists) is executed.

In the following example from the `ls` command, the editrule checks for an `l` in the command line. If `l` is not already in the command line, a `-l` is added. Then a `-d` is added.

```
option dir_info_only disable all
editrule { if (! l) { dash("l"); } dash("d"); }
```

Blanks. To strip leading and trailing blanks from a user-typed string, use the `trim` function as follows:

```
trim(string);
```


In the following example, you are required to enter a number or select another option (`thru`). If you enter a number followed by a blank space, the `trim` function strips the blank space when it adds the number to the command line.

```
string number enable all
    editrule { word[last] &= trim(argument); }
    required "Enter the number of the first page to
              print or select \"thru\"."
```

Examples of Editrule Use

This section contains annotated examples showing how editrules are used in softkey definitions. Refer to the `/usr/keysh/C/softkeys` file for other examples.

"Remove Files" Command Line Example. The following example explains the editrules that tell `keysh` how to construct an HP-UX command from the softkey command `Remove files interactively *.c`. (Recall that this command was used as an example in the section titled "How Key Shell Translates a Softkey Command.")

The example shows the lines from the softkey file that define the parts of the above softkey command. The annotations describe the effect of those lines. `rm` is assigned the label `Remove files` in the `.keyshrc` file.

```
softkey rm command
editrule { append("rm"); }
```

When you type **rm** *or select* **Remove files**, **keysh** *appends* **rm** *to the HP-UX command line.*

```
option interactively disable 1
editrule { dash("i"); }
```

When you select the **interactively** *option,* **keysh** *appends a* **-i** *to the HP-UX command line.*

```
string <files> disable -1
editrule { append(argument); }
required "Enter the name of the file(s)
          to remove."
```

When you type in the name of the file(s) (`*.c` *in this case),* **keysh** *appends the argument to the HP-UX command line verbatim.*

"Man" Softkey Example. This example shows the complete softkey definition for the `man` command, which lets you access the online HP-UX Reference manual pages. The `man` command is assigned the label `Manual page` in the `.keyshrc` file.

```
softkey man command
editrule { append("man"); }
{
 option keyword_search disable all
 editrule { append("-k"); }
 {


  string <keys> disable -1
  editrule { append(argument); }
  required "Enter the keyword(s) to search
            the manual page entries for."
  ;
 }

 option file_search disable all
 editrule { append("-f"); }
 {


  string <files> disable -1
  editrule { append(argument); }
  required "Enter the file name(s) to search
            the manual page entries for."
  ;
 }

 option from_section
 {



  option user_commands
  disable all
  editrule { append("1"); }
  required "Select a manual section."
  ;
  .
  .
  .
 }
 string <topic>
 editrule { append(argument); }
 required "Enter the name of the manual topic."
 ;
}
```

When you type **man** *or select* **Manual page**, **keysh** *appends* **man** *to the HP-UX command line.*

If you select the **keyword search** *option,* **keysh** *appends a* **-k** *to the command line. All other options at this level are disabled, since you cannot perform more than one type of search at a time.*

If you selected the **keyword search** *option, the* **required** *line then prompts you for the required information: the keyword(s) to search for. Since multiple key words are allowed, this node is not disabled after it is selected.* **keysh** *then appends the keyword argument(s) to the command line.*

If you select the **file search** *option,* **keysh** *appends a* **-f** *to the command line. All other options are disabled, since you cannot perform more than one type of search at a time.*

If you selected the **file search** *option, the* **required** *line then prompts you for the required information: the file name(s) to search for. Since multiple file names are allowed, this node is not disabled after it is selected.* **keysh** *then appends the argument(s) to the command line.*

If you select the **from section** *option, you must select an option for the manual section to search.* **keysh** *then appends the corresponding HP-UX option to the command line.*

If you select the **user_commands** *option, all other options at this level are disabled, since you can only choose one section to search. (Other section options are omitted here to save space.)*

Unless you disabled the **<topic>** *node by selecting either the* **keyword search** *or* **file search** *nodes, you must also specify the topic to search for.* **keysh** *then appends this argument to the command line.*

"Cat" Softkey Example. This example shows the complete softkey definition for the invisible softkey `cat`, which lets you concatenate and display files.

```
softkey cat
editrule { append("cat"); }
{
  option visible enable all
  editrule { dash("v"); }
  ;

  option show_tabs enable all disabled
  editrule { dash("t"); }
  ;

  option show_newlines disabled
  editrule { dash("e"); }
  ;

  option un+buffered
  editrule { dash("u"); }
  ;

  option ignore_errors
  editrule { dash("s"); }
  ;

  string <files> command disable -1
  editrule { append(argument); }
  required "Enter the name of the file(s)
            to concatenate."
  ;
}
```

When you type **cat**, **keysh** *appends* **cat** *to the beginning of the HP-UX command line.*

If you select the **visible** *option,* **keysh** *appends a* **-v** *to the command line. All other options are enabled, in particular the options that can only be used to modify the* **visible** *option.*

If you select the **show_tabs** *option,* **keysh** *appends a* **-t** *to the command line and keeps all the other options enabled. This option is initially disabled, but is enabled by selecting the* **visible** *option.*

If you select the **show_newlines** *option,* **keysh** *appends a* **-e** *to the command line. This option is initially disabled, but is enabled by selecting the* **visible** *option.*

If you select the **unbuffered** *option,* **keysh** *appends a* **-u** *to the command line.*

If you select the **ignore_errors** *option,* **keysh** *appends a* **-s** *to the command line.*

The **required** *line then prompts you for the required information: the file name(s) to concatenate. Since multiple file names are allowed, this node is not disabled after it is selected. However, all options preceding this one are disabled.* **keysh** *then appends the argument(s) to the command line.*

Adding Text to Softkeys

When you create a softkey command, you can add your own text to it, to prompt the user for information or to provide help in using your command. This text can be any of the following:

- **Required** text is used to inform users that they must select an option or supply information before proceeding.
- **Hint** text is used to prompt users for optional information. Hints can only be used for string-type softkey nodes.
- **Help** text is used to explain custom softkeys and options.

The following sections explain each of these types of text.

Adding Required and Hint Text

To add `required` or `hint` text to a softkey, follow these guidelines:

- Start the text with the keyword `required` or `hint`.
- Enclose the text in quotes.
- Do not make the text longer than 79 characters.
- Use the backslash (\) to escape special characters within the text such as quotation marks.

The following example of `required` text is from the `Copy files` softkey. Because the `Copy files` command does not make sense unless you specify the name of the file(s) to copy, the file name is required. `keysh` will not continue until you supply this information.

```
string <files> disable -1 enable all
  editrule { append(argument); }
  required "Enter the name of the file(s) to copy."
```

The following example of `hint` text is from the `Change dir` softkey. If you do not specify a directory to move to, you move to your home directory by default. Hence, the directory name is not required and can be prompted for with a `hint`.

```
string <dir>
  editrule { append(argument); }
  hint "Enter the name of the directory to move to."
```

27

Adding Help Text

To add `help` text to a softkey, follow these guidelines:

- Start the text with the keyword `help`.
- Enclose the text in quotes.
- You can make the text any length.
- Use the backslash (\) to escape special characters within the text such as quotation marks.

Online help that you create is accessed in the same way as the pre-configured online help. You determine how the text will look with a few simple formatting commands, described in Table 27-5. These commands are a subset of the *man*(5) macro commands used by *nroff*(1).

Table 27-5. Formatting Commands

Command	Meaning
`.br`	Force a break in the current output line. Display subsequent text on the next line.
`.sp`	Force a break and then display a single blank line.
`.p`	Force a break, display a single blank line, and then begin a new paragraph with no indentation.
`.ip` *tag indent*	Force a break and display a single blank line. Then begin a new paragraph with the specified tag and indentation. For example, `.ip * 5`.
`.il` *tag indent*	Force a break, then immediately begin a new paragraph with the specified tag and indentation. No blank line is displayed. For example, `.il * 2`.
`.ti` *indent*	Indent just the next line by the specified number of characters. For example, `.ti 3`.
`.in` *indent*	Indent all following text by the specified number of characters. To stop indenting, repeat the macro with a negative value (for example, `.in -5`).
`.nf`	Begin no-fill mode. Display text as-is, preserving new-lines and spacing, until an `.fi` is encountered.
`.fi`	Resume fill mode after using an `.nf`. Display text with words filled to 90% of the screen width. "Fill" is the default mode.

Note that the formatting commands can appear anywhere in the text, and can be upper- or lower-case.

Following are two examples of help text. The first example shows the help text with formatting commands as it appears in the softkey definition file:

```
help "You can use the Copy_files command to copy a file to
      a new or existing file, or to an existing directory.
      .ip * 5
      If you copy a file to an existing file,
      the existing file is overwritten.
      .ip * 5
      If you want to copy more than one file, you must copy
      the files to an existing directory rather
      than another file.
      .p
      For more information, refer to cp(1)."
```


The second example shows the above help text as it is displayed to the user:

```
You can use the Copy_files command to copy a file to a new or
existing file, or to an existing directory.

*     If you copy a file to an existing file, the existing file
      is overwritten.

*     If you want to copy more than one file, you must copy the
      files to an existing directory rather than another file.

For more information, refer to cp(1).
```

Creating Custom Softkeys

This section helps you apply the concepts described in "Understanding Key Shell" to create your own softkeys. Also refer to *keysh*(1) and *softkeys*(4).

To create a custom softkey, the following steps are suggested:

1. Decide what you want your softkey to do. If the softkey is complex, you may wish to sketch a node hierarchy like that in Figure 27-1 showing the top-level command and any options or sub-options.
2. Create a softkey file in your home directory, using *vi*(1) or another editor.
3. Look at the `/usr/keysh/C/softkeys` file for examples of commands similar to the one you want to add.

   You can copy a similar command into your own softkey file and use it as a template.
4. Create the softkey, using the appropriate attributes and editrules as described in the previous section titled "Understanding Key Shell."

   Also add any text or online help, as explained in the previous section titled "Adding Text to Softkeys."
5. Add the softkey to the list of softkeys recognized by `keysh`, as explained in the section in Chapter 26 titled "Adding, Moving, and Deleting Softkeys."

Backup Softkeys

You can also use the above steps to create backup softkeys. Backup softkeys are the softkeys that Key Shell can display when it cannot display its own softkeys (for example, when the `vi` editor is running). Backup softkeys program the function keys to provide the static softkey control that you may have used before.

Key Shell does not contain pre-configured backup softkeys. If you wish to use backup softkeys, you must create them. For more information on backup softkeys, refer to *keysh*(1) and *softkeys*(4), and to the online help for `Keysh config options backups`.

Examples

The following example shows the definition for a custom *cd*(1) command. Frequently-used directories are added as options to the `cd` command. Note that these options have the attribute `disable all` because they are mutually exclusive (you can move to only one directory at a time).

```
softkey cd
editrule { append("cd"); }
{
  softkey keysh-src disable all
  editrule { append("~/keysh/src"); }
  ;
  softkey keysh-test disable all
  editrule { append("~/keysh/test"); }
  ;
  softkey keysh-doc disable all
  editrule { append("~/keysh/doc"); }
  ;
  softkey demo disable all
  editrule { append("~/demo"); }
  ;
  softkey tmp disable all
  editrule { append("/tmp"); }
  ;
  string <dir> disable all
  editrule { append(argument); }
  hint "Enter the name of the directory to move to."
  ;
}
```

This example shows a custom softkey for RCS source control. The options allow you to perform such tasks as checking files into and out of RCS control, reviewing different versions of a source-controlled file, and deleting locks on files. (For more information about RCS source control, refer to *rcs*(1).)

```
softkey Source_control
{
  option check_in disable 5 enable all
  editrule { append("ci -u"); }
  required "Select a source control function."
  ;
  option check_out disable 4 enable all
  editrule { append("co -l"); }
  ;
  option review_revision disable 3 enable all
  {
    string <rev>
    editrule { append("co -p" & argument); }
    required "Enter the number of the revision to review."
    ;
  }
  option show_log disable 2 enable all
  editrule { append("rlog"); }
  ;
  option show_changes disable 1 enable all
  editrule { append("rcsdiff"); }
  ;
  option admin disable 0 enable all
  {
    option set_lock disable all
    editrule { append("rcs -l"); }
    required "Select an administration function."
    ;
    option delete_lock disable all
    editrule { append("rcs -u"); }
    ;
```

```
    option set_tag disable all
    {
      string <tag>
      editrule { append("rcs -N" & trim(argument) & ":"); }
      required "Enter the name of the symbolic tag to set."
      ;
    }
    option delete_tag disable all
    {
      string <tag>
      editrule { append("rcs -N" & argument); }
      required "Enter the name of the symbolic tag to delete."
      ;
    }
  }
  string <files> disabled disable -1
  editrule { append(argument); }
  required "Enter the name of the RCS or working file(s)."
  ;
}
```

Index

S

T

Master Index

Index

B

C

F

K

L

M

N

O

P

Q

R

S

T

U

V

W