Shells and Miscellaneous Tools
HP-UX Concepts and Tutorials

HP Part Number 97089-90062

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

NOTICE

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 AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

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.

September 1986...Edition 1

December 1986...Update 1. *vt* tutorial rewritten. Added new information to accurately reflect *vt* as implemented on HP-UX Series 300 Release 5.2 and Series 500 Release 5.1.

April 1987...Edition 2. Update 1 merged.

October 1987...Edition 3. Added Korn Shell Tutorial.

Table of Contents

The Bourne Shell

The Bourne Shell is most commonly known as a "command interpreter", taking your commands and interpreting them to the system. This tutorial will give you many methods which will enhance your interaction with the shell by teaching you to *program* the shell.

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 familar 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*, *condition statements*, and *variables.*

UNIX System Structure

HP-UX is a fully compatible, enhanced version of UNIX(TM) 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 what the master program, or operating system, is called. 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.

* UNIX(TM) is a trademark of AT&T Bell Laboratories, Inc.

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 Bourne shell as a default, but it is possible to run the "C" shell (csh) or "PAM" (Personal Applications Manager) 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 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).
[]	Italicized 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 will refer to screen printouts or anything you are to type (i.e., `varname=penguin` means you type the entire string).
italic	Italic font in the text refers to words not yet defined. If you see italic font in screen printouts or 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 a term used for the first time.
bold	Bold font refers to text being emphasized, text to which you need to pay particular attention.

Notes

Using Shell Commands 1

This chapter will discuss methods for combining shell commands. You should already be familiar executing single commands, like running the `date` command. In addition to simply typing a command and pressing the return key, you have many 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 preceeded with 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, the list 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
```

will accomplish the same as entering each command on a separate line. This process is called *sequential processing.* New programs or commands cannot be started until the preceeding 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 Processing

Programs can also be run *nonsequentially* (that is, each program runs without waiting for the previous program to complete). Follow the program name with **&** to specify nonsequential processing.

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

will run the three programs at the same time. It is good to remember that if you have output to the screen from more than one program, or input is needed from the terminal by more than one program, the messages may appear on the screen simultaneously. It is a good idea to not run these kinds of programs nonsequentially.

Redirecting Input and Output

Every program has at least three data *paths* associated with it: standard input, standard output, and error output. Programs use these data paths to interact with you. Standard input (**stdin**) is normally the keyboard, and standard output (**stdout**) 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 symbols for redirecting input and output allow you to change a specific data path of a program while leaving its other data paths unchanged. For example, you can specify a different device for standard output. Normally, the output would be sent to the screen. You can redirect the standard output, for example, to a file if you wish to store the output.

How to Redirect Input and Output

Symbols to redirect the input or output of a program are entered in the shell or from a shell program. The program begins executing with the data paths specified by the redirection symbols. To specify redirection of input or output for a program, each file name is preceded by a redirection symbol,

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

Spaces between the redirection symbols and the files 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 1:

Table 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

Be careful to not 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.
- If a file specified for appending output to with the >> 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:

word_a | *word_b* | *word_c...*

where *word* is a command or executable program. Pipes operate on or transform data by separate programs in stages. For example, *word_a* could have input that you type from the keyboard. This data would be passed by the first pipe to *word_b* where it would be checked for validity, and processed. The processed data could then be passed by the second pipe to *word_c* for formatting into a report, and so on.

Here are some examples:

```
ls | wc
```

Print the number of files in the current directory.

```
ls | more
```

Print a listing of each file in the directory.

```
cat file | pr | lpr
```

Send *file* to the line printer (via the print formatter *pr*).

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.

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

Takes the standard output from *test_prog1* and uses it as standard input to */usr/output_prog.*

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

Runs four programs connected with pipes and puts the output of the fourth program in *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 +1 pdir; (( pr pdir | lpr )& (sort +1 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 the section in Chapter 5). 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. The following table shows the file generation symbols you can use:

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 which end in "`.c`" or "`.a`".

Shell Scripts 2

Stringing commands together on a line with sequential processing, nonsequential 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 with the commands, tell the system you want the file to be executable (run like a program, not a data file), 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 `sh` *scriptname* and changing the permissions and 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 the "Echo" section in Chapter 3).

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 (:). When a command is executed using the above `.profile`, the shell looks in the `/docs/tools` directory first, then the `/bin` directory, and so on. Notice the last entry in the first line is a dot (.). This indicates the current directory.

  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 resonable values.

- `tabs` will set the tabs to the default format for your terminal. The `-T` option 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 2. 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.

Basic Shell Programming 3

After you have mastered simple shell scripts, you can move into the programming aspect of shell programs (shell scripts and shell programs are the same thing, except sometimes shell programs are thought to contain more of the programming constructs than just lines with commands on them). This chapter will introduce 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 there are 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 concatenate the parameter easily. 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 in the execute statement. 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 4.

Here is an example script using positional parameters:

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

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

```
get "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:

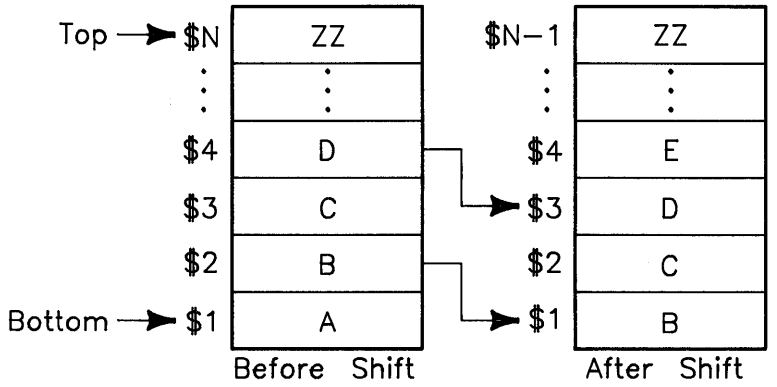


Figure 1. Shifting Positional Parameters

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 "`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`" (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, you suppress any special meaning.

The Backslash

The backslash (\) will cancel 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.

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 keyboard (or the *itf* keyboard), and the grave accent 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 shows a command substitution in an `echo` command:

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

This command will print something similar to the following:

```
People currently on the system:

tricia      console      Aug  8 09:45
jaci        tty02        Aug  8 12:02
derald      tty03        Aug  8 07:24
michael     tty04        Aug  8 12:31
davea       tty07        Aug  8 08:15
davek       tty12        Aug  8 12:31
richard     pty/ttyp0    Aug  8 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 'the 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. 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 ]
```

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`. The options with the command allow you to check if a file is a directory, is readable, and many other options which are shown in the following table:

Table 3. Test Options

Option	Description
`-r` *file*	true if *file* exists and is readable
`-w` *file*	true if *file* exists and is writable
`-x` *file*	true if *file* exists and is executable
`-f` *file*	true if *file* exists and is a regular file
`-d` *file*	true if *file* exists and is a directory
`-c` *file*	true if *file* exists and is a character special file
`-b` *file*	true if *file* exists and is a block special file
`-p` *file*	true if *file* exists and is a named pipe (fifo)
`-u` *file*	true if *file* exists and its set-user-ID bit is set
`-g` *file*	true if *file* exists and its set-group-ID bit is set
`-k` *file*	true if *file* exists and its sticky bit is set
`-s` *file*	true if *file* exists and has a size greater than zero
`-t` *[fildes]*	true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device
`-z` *s1*	true if the length of *sa1* is zero
`-n` *s1*	true if the length of *s1* is non-zero
s1 = *s2*	true if strings *s1* are identical
s1 != *s2*	true if strings *s1* and *s2* are not identical.
s1	true if *s1* is not the null string
n1 `-eq` *n2*	true if the integers *n1* and *n2* are algebraically equal. Any of the comparisons `-ne`, `-gt`, `ge`, `lt`, and `le` may be used also, and *n1* and/or *n2* can be parameter substitutions

These operators can also be used:

`!`	unary negation
`-a`	binary *and* operator
`-o`	binary *or* operator (`-a` has higher precedence)
(*expr*)	parentheses for grouping

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 4. 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. #
   ##############################################################

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

# Ask if file is to be moved to directory or file
#(2)
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
        mv $1 $dir1   #perform the move #
        echo "$1 moved to $dir1"
        exit

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

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

Explanation

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) 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) 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) 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) If `x` was instead "f", the user is prompted to enter the new filename. (5) 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.

Advanced Programming 4

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 (discussed in Chapter 2) 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 in specifying the word list. Pattern matching is discussed towards the end of this section.

While

The **while** construct will repeatedly execute 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 (which usually means a status code of 0), 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 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.

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. Finally, the last pattern uses pattern matching (which will be discussed in detail towards the end of this chapter) and in effect says, "match anything beginning with a -". Remember to end each command list with `;;` and to end the entire `case` construct with `esac`.

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 script.

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 File Name Generation in Chapter 1. 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 can use the special characters which will match certain characters.

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

Here we are generating a loop where each value is a filename from the current directory, and the filename is followed by "`.c`".

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

This case construct will match `i` on the first pattern line if `i` 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. If you need to print a character that will "beep" to alert a user, use CTRL-G in an **echo** command*. If you need to break from a "**for**" or "**while**" loop, use the **break** statement. If you want to break after a certain number of loops, add **break** *n*, where *n* in the number of loops. To continue the loop, use the **continue** statement. To continue at a certain iteration of the looping, 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.

* To add control characters to the **vi** editor you must first type CTRL-V, then type the control string.

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
fi

#(2) look for options
for i in $*
do
case $i in
-q)  query='y'
        shift ;;
-d)  dir='y'
        shift;;
-*)  echo "unknown option; available options are -q, -d"
        exit;;
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,
                        echo "copy $f? \c"      # or anything else to ignore
                        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
        # test if user wants to copy from subdirectories
        if [ $dir = y ]
        then
                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
fi
```

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) 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) 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) 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) 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) 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).

Notes

Detailed Reference 5

This chapter will attempt to 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; pr temp | lpr; rm temp)&
```

This command sequence will test if "`file`" is a file. If it is, it runs a command grouping in the background (note the terminating &)*.

Another helpful command grouping is with the 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 this command line could be simplified to read:
`test -f $file && sort -o < $file | pr | lpr`

Defining Functions

The more complicated your shell programs get, 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. Note the positional parameters are good for the function only, so you should use the **export** command to refer the values to the enclosing shell program.

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 input redirection symbol will read input from the script (the same file you have script commands in) to a line that is the same as *word.* The resulting document is then used as input. 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:

```
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 with it alone. << is particularly useful for multi-line input to commands (usually **ed**(1) commands). If you add the optional - after <<, then all leading tabs in the 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 2 for error output (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 0, 1 , or 2. 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 to 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 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 you 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 character < and >). The function will return 0 if the comparison is successful, and 1 if it is not. Here is an example how a comparison might be used:

```
if expr $a \<= $b
then
    echo The value is $a
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* (see "Regular Expressions" in Chapter 4). 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 b 3
```

returns the string "bat".

```
expr index expression expression
```

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

```
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`. Preceeding the option with - will turn the flag on. Preceeding 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 5. Options to 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.

Here is a list of the signal numbers that can be trapped and their description:

Table 6. Signals.

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

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)).

Hash

The format for **hash** is:

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

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

The purpose of the **hash** command is to make 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 occurance 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 3. You can return error codes from functions by using the `return` statement:

```
return n
```

To check the status of a return value, you can use a parameter called `$?` which is set to the one of the four values.

Parameters Set by the Shell

During the course of interaction with the shell, you can access several special parameters which 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 7. 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.

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 5 (the `set` command options).

Table 8. 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 directory in which the restricted user is placed does not give him/her access to subdirectories in which they may do damage. Also make sure they do not have commands that let them escape the restricted shell (particularly `chsh`, `csh`, and `pam`).**

Notes

Helpful Tips for Shell Programmers A

This appendix will give you some tips to use when 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 te *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:

```
proc=                                  # initialize the process list

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

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

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

echo starting process 3
process3                               # foreground process
```

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.

Notes

Glossary

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

control key
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.

edit
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.
- Network special files - contain the address of another system.
- 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

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[1] 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.

ITF

Integrated Terminal Family: the name for the standard keyboard.

[1] UNIX is a trademark of AT&T Bell Laboratories, Inc.

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/csh
/bin/rsh
/bin/pam
```

variable
See Parameter.

vi
The vi editor (visualize).

Index

a

b

c

d

e

f

g

h

i

k

l

m

n

o

p

q

r

S

t

u

v

W

Notes

Table of Contents

The C Shell (csh)

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, pronounced "C Shell", 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 abuot HP-UX shells, consult the Bourne Shell tutorial.

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 promtp = "[\!] % "
  ```

  This prompt will appear as:

  ```
  [23] % _
  ```

- Dot-matrix font is used to show what should appear on the screen. For example, to activate the C shell, type `csh`. Terminating command sequences with [Enter] or [Return] is assumed.
- **Bold** font is used for important and key words.
- *Italics* font is used to emphasize important words and refer to words illustrated in commands.

[1] 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.

Several HP-UX commands are useful in setting up and verifying shell operation. They include *chsh* (change login shell), *netunam* (used to access remote systems over a local area network), *printenv* (lists currently defined envrionment 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 both the Bourne Shell and the C Shell command interpreters. Systems are shipped with the Bourne shell as the default shell.

The normal shell prompt for the Bourne shell is the dollar sign (**$**). When C shell is active instead, the default prompt becomes the percent (%) symbol. The prompts for either or both 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 *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 in `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/C shell
status          0
term hp2622
path (/bin /usr/bin /usr/local/bin /etc/users/login_name . )
[26] % _
```

`exit` or `logout` 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 the Bourne Shell or another C 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 shell because the shell prompt changes to your Bourne 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 **ignoreof** 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 theh `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 /usr/bin)
```

the value of the environment variable PATH also becomes */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 executed 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
noclobber
prompt [!] %
shell /bin/csh
status          0
term hp2622
path (/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 on the next page.

Command	Meaning
`set ignoreeof`	Traps CTRL-D's to avoid accidental system log off. Use the logout command.
`set promtp = "[\!] %"`	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:/usr/bin:/lbin:/usr/lbin:/etc:/users/login_name:.`	This command sets the 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:

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.

Command History

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 (in this case 20) 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  vi memo
      3  pr memo > /dev/lpr&
      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 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  vi memo
     3  pr memo > /dev/lpr&
     4  mail jd < memo
     5  vi .cshrc
     6  history
     7  vi memo
     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 | col -l > /dev/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
[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 | col -l > /dev/lp &
[1] 18634 18635
[14] %
```

Note that the example generated a new C shell shell with the event number `[1]` and two process IDs `18634 18635`. This new shell is called a *background process.* The arguments `col -l` are printed on the line printer (*/dev/lp*). 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 referencee 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.

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 that 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 filenamee *file.o* leaving *file*).
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 comand number.

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

"bug.c",line 4: newline in string or char constant
"bug.c",line 5: syntax error

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

29
4s/);/"&/p
    printf("hello");
w
30
q

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

[25] % a.out

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

30
4s/lo/lo\\n/p
    printf("hello\n");
w
32
q

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

[27] % size a.out bug

a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
```

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

-rwxr-xr-x 1 jerry      3932 Feb 29 09:00 a.out
-rwxr-xr-x 1 jerry      3932 Feb 29 09:01 bug

[29] % bug

hello

[30] % pq -n !!:s/pq/pr
pq: Command not found.

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

  1 main()
  3 {
  4     printf("hello\n");
  5 }

[32] % !! > /dev/lp                Execute last executable event
pr -n -t bug.c > /dev/lp           (!!) and pipe to line printer.

[33] %
```

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 !* | col -l > /dev/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 backslach (\) in front of the exclamation point prevents the exclaimation 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

The following aliases are already provided by C shell:

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

Command Substitution

A command enclosed in single quote 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 -1 TRACE*.c'
```

runs the editor `ex`, supplying as arguments those files whose names end in `.c` and begin with 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 `/dev/lp`, type:

```
cat letter > /dev/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 file-name. 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>`, type:

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

This could produce:

```
file.o  file.p
```

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 or is not followed by an alphabetical character or a `/`, 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/C shell` is the pathname to the file `csh`. The first slash in a pathname or a lone 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 the History discussion earlier.

: precedes substitution modifiers. See the History discussion earlier.

? 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 shell script 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 pose a problem in that we cannot use them 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 prints 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 exclaim 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 astrisk (*) was enclosed between single-quotes. The result is a single-quote and astrisk.

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 /usr/bin /lbin /usr/bin /etc .)
```

or

```
setenv PATH /bin:/usr/bin:/lbin:/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 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.

$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/sh
```

selects Bourne 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
15
[26] %
```

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).

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

and the following boolean operators are allowed:

`==`	String comparison equal
`!=`	Boolean Not Equal
`!`	Exclamantion 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

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`

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 operators &= |= <<= and >>= 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 following list.

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.

```
#
# 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 one is 3948
==> find_dir is not a directory.
[46] %
```

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 shells std_out file (unless redirected, std_out 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      (.  /usr/lib  /bin /usr/bin)
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]
keith
[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 hp2627
```

sets the value of the environment variable TERM to `hp2627`. See *environ*(7) in the *HP-UX Reference Manaul.*

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 intput. 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. Ifno 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 /etc/rc/usr/login_name/file_name
0.0u 0.1s 0:01 8% 2+1k 3+2io 1pf+0w
% time wc /etc/rc/usr/login_name/file_name
      52  178  1347 /etc/rc
      52  178  1347 /usr/login_name/rc
     104  356  2694 total
0.1u 0.1s 0:00 13% 3+3k 5+3io 7pf+0w
%
```

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), there was an average memory usage of 2k bytes of program space and 1k byte of data space over the cpu time involved (2+1l); the program did three disk reads and two disk writes (3+2io), and took one page fault and was not swapped (1pf+0w). 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 wihout pipes or semicolons create the simplest jobs. Usualy, 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 cancelled 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 placess 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. Then the shell then prompts for a new command as soon as the background job is underway.

To check and 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 character is a `#` (hash mark) symbol. 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 character in the file is not a hash mark (`#`), the Bourne shell will attempts 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 a one (1) if the variable has a currently assigned value and a 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

```
$sum1[$#sum1]
```

returns the component variable *sum1*, which is "c".

```
$argv[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 **$n** and **$sum1[n]** should be noted. **$sum1[n]** will yield an error if **n** is not in the range **1** through **$#argv** while '$n' will never yield an out-of-range subscript error. This is for compatibility with the way other shells handled parameters.

One way to avoid this type of error is to use a subrange of the form **n-**. If there are less 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`.

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, all 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 your *HP-UX Reference* for specifications of these primitives.

You can determine whether a command terminated normally by

```
{ 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.

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 taken from 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.

```
#deblank - - remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1.$s/^[]*//
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 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 program 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 have not been previously backed up. 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**.

```
#

foreach i ($argv)
  if ($i \!~ *.c) continue          # is it a .c file?
  echo $i is not .c
  continue
  else
  echo $i is a .c program
  endif

  echo check file ~/backup/$i:t

  if(\! -r ~/backup/$i:t) then      # is file part of backup?

  echo $i:t not in backup...not cp\'ed
  continue
  endif

  echo compare two files $i and ~/backup/$i:t
  cmp -s $i ~/backup/$i:t           # has the file changed?

  if ($status != 0) then

     echo new backup of $i
     cp $i ~/backup/$i:t
  endif
end
```

Notes

Index

a

b

c

e

f

g

h

i

j

l

m

n

o

p

q

r

s

t

u

v

w

Notes

Table of Contents

Introducing the Korn Shell 1

This tutorial describes the **Korn Shell** (`ksh`) provided on the HP 9000 Series 300 and Series 800 computers. Korn Shell is the command interpreter (human interface) written by David Korn at AT&T[1] Bell Laboratories.

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.

Please use one of the reply cards at the back of this tutorial to tell us what was helpful, what was not, and why. Feel free to comment on depth, technical accuracy, organization, and style. Your comments are appreciated.

[1] Hewlett-Packard would like to acknowledge the following individuals and institutions in the development of Korn Shell: AT&T and David G. Korn.

Korn Shell Versus Other Shells

This program (shell) is a command interpreter and programming language that executes commands entered from a terminal or file. It is based on the Bourne Shell and C Shell. Korn Shell maintains Bourne Shell's superior programming environment while adding the unique command interpreting features of C Shell. It is 95% upward compatible with the Bourne Shell and most programs written under the Bourne Shell will run under Korn Shell without change. In developing 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

The main features of the C Shell that are built into the Korn shell are:

- history buffer and history substitution capabilities
- file name completion
- command aliasing mechanisms
- arrays
- integer arithmetic evaluation
- tilde substitution
- job control features

Differences From Bourne Shell

The Korn Shell is a superset of the Bourne Shell and contains most of the Bourne Shell constructs plus features from the C Shell. However, there are some features that Korn Shell implements above and beyond both the C Shell and Bourne shell. They are:

- the **select** and **function** statements
- new built-in commands such as **time** and **whence**
- several new shell variables such as REPLY, PPID, EDITOR, and OLDPWD
- extended **vi** and **emacs** in-line editing commands
- increased capabilities for parameter substitution

Definition of Terms

In learning this shell certain terminology is used to describe commands and arguments. Be familiar with the following terms and their definitions so you understand later descriptions.

metacharacter

one of the following characters:

```
; & ( ) | < > new-line space tab
```

blank

a **tab** or **space** often referred to as whitespace.

word or **command**

a sequence of characters separated by one or more non-quoted metacharacters or whitespace. For example:

```
date
```

The five types of command words the shell understands are: reserved word (such as `for`), built-in (such as `pwd`), alias (such as `type`), function, and others (such as a path name).

options or **flags**

a letter preceded by a dash (-) and separated from the command name by a blank. For example:

```
-v
```

argument and **parameter**

the words following a command or program name used to pass information to that command or program. In this example `file1` is the parameter and `lp` is the command:

```
lp file1
```

A parameter is also a shell variable set in the environment, such as PATH.

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
```

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 | lpr; 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 are also used for symbolic items either typed by the user or displayed by the system as discussed below under computer font.
- **Boldface** is used when a word is first defined and for **general emphasis.**
- `Computer font` indicates a literal typed to the system or displayed by the system. A typical example is:

  ```
  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 file name is symbolic (but not literal), it is shown in italics as the following example illustrates.

  `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 (dots), . . . , designate optional repetition of the word directly preceding them.
- Environment variables such as EDITOR or PATH are represented in upper-case characters, as required by HP-UX conventions.
- A keycap such as [Return] designates the pressing of that key. If the keycaps are connected by a hypen, 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* manual. If you cannot find an entry where you expect it to be, use the *HP-UX Reference* manual's "Permuted Index".

Using Other HP-UX Manuals

This tutorial may be used with 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.
- *A Beginner's Guide to HP-UX* teaches new users how to login to HP-UX, work with files and the directory structure, and send and receive mail.
- *A Beginner's Guide to Using Shells* teaches new users the basic elements of what a shell is, how to use it, and what shells are available on HP-UX.
- *A Beginner's Guide to Text Editing* teaches new users how to create, edit, and save files using the `vi` editor.
- The *HP-UX Documentation Roadmap* provides part numbers for the HP 9000 Series 200, 300, and 500 manuals.
- The *Documentation Guide* provides part numbers for the Series 800 manuals.

Starting and Stopping the Shell 2

New users should read this whole chapter; advanced users should jump to the section on "Setting Up .profile and .kshrc".

The **kernel** is the HP-UX operating system. A **shell** is a program that the kernel runs so each user can interact with the computer using commands in the **utilities** area. See the figure below.

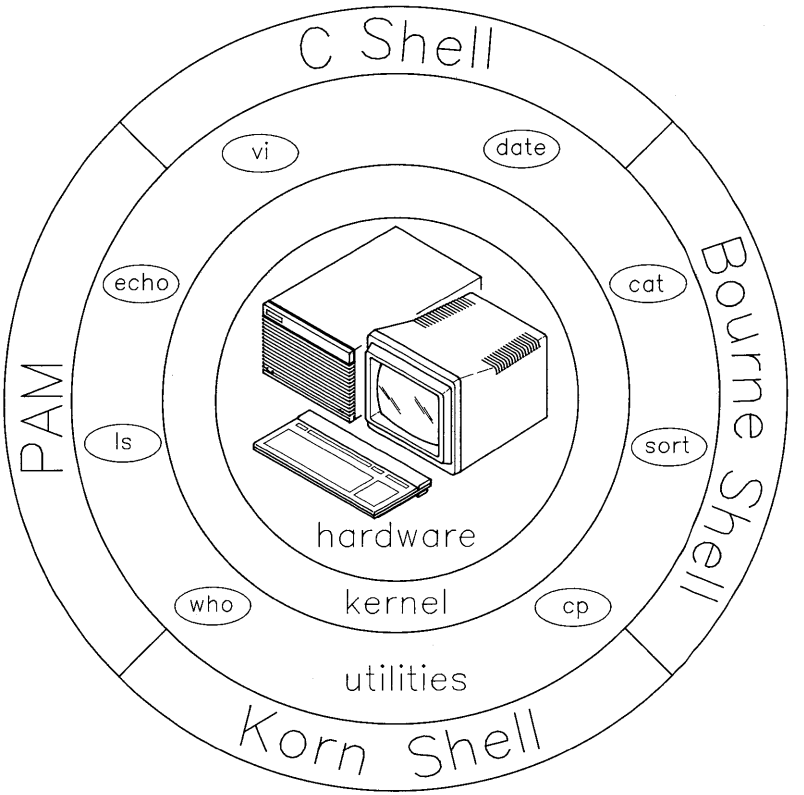


Figure 1-1. System Structure

When you log into a system a special program called **login** determines if 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 passwords on a system. See your System Administrator for more details.

Once you type in the correct user and password the **login** program gives you (**spawns**) a shell (usually the Bourne Shell). HP-UX operating systems are shipped with Bourne Shell as the default shell. There are three other shells, the Korn Shell, the C Shell, and PAM. When you communicate with your shell you type commands at the **prompt**. The default prompt for the Korn Shell and the Bourne Shell is a dollar sign (**$**). The default prompt for the C Shell is **%** . A prompt tells you the shell is ready to accept typed input from the terminal. After you type in a command-line the shell interprets and executes it. For example:

```
$ echo Welcome to Korn Shell
Welcome to Korn Shell
$
```

The command is **echo** which outputs to the screen the line directly following "Welcome to Korn Shell". When the shell executes the **echo** command, it spawns a **process** for the **echo** command and assigns 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 Korn Shell software resides under the directory **/bin** in the file **ksh**.
- Likewise C Shell resides in **/bin/csh**.
- Bourne Shell resides in **/bin/sh**.
- PAM resides in **/bin/pam**.

The rest of this tutorial deals mainly with the Korn Shell. If you do not understand the login process or the file directory system, read these topics in *A Beginner's Guide to HP-UX*. If you do not understand the shell structure or interaction with the shell, read *A Beginner's Guide to Using Shells.*

Invoking Ksh

Normally, when you log into an HP-UX system, a default shell (Bourne) is spawned for you. To determine if **sh** is your shell, type:

```
$ echo $SHELL
/bin/sh
```

The **echo** command prints out the **contents** or **value** (specified by using **$** before a parameter) 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 Korn Shell on a temporary or permanent basis.

Running Korn Shell From the Current Shell

If you would like to experiment with the shell until you gain expertise, invoke it by simply typing:

```
/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 Korn Shell, the prompt changes from **%** to **$**, unless these prompts have been redefined. It is possible to redefine prompts; see PS1 in the "Shell Parameters" table.

If you exit the shell, by typing:

```
$ exit
```

you must reinvoke it each time with the **ksh** command. Other methods of exiting the shell are discussed in the "Terminating Ksh" section.

Making Korn Shell 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 **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.

Setting Environment and Shell Variables

Environment variables and shell variables (**parameters**) are set in the `.profile` and `.kshrc` files. These variables create part of the **environment** in which you work, such as your prompt string (`$ `). **Environment variables** are shell parameters that are **global** and used by your shell to create a special environment when you log into or spawn a subshell. 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 Korn Shell is invoked it looks for these following files and executes them, if they exist:

`/etc/profile`	This default system file is executed by the `login` program and sets up default environment variables.
`.profile`	If Korn Shell is your login shell, and this file exists in your home directory, it is executed next at login.
`.kshrc`	When you invoke `ksh`, 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 are global and available to subshells and subprocesses. This is an example of a possible `.profile`:

```
PATH=/usr/bin:/usr/lib:/bin:/users/tricia/bin:.
MAIL=/usr/mail/tricia
HOME=/users/tricia
EDITOR=/usr/bin/vi
ENV='${START[ (_$- = 1) + (_ = 0) - (_$- != _${-%%*i*}) ]}'
START=~/.kshrc
TERM=hp2392
export ENV START EDITOR TERM PATH MAIL HOME

stty sane

if mail -e
then
     echo "You have mail."
fi

PS1="$ "
```

Each line above shows an example of Korn Shell variables:

- 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 on the line 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 later `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 *A Beginner's Guide to Text Editing.*

- 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" section in the "Substitution Capabilities" chapter for more details). If your `.kshrc` is very long and involved, spawning a new shell can take awhile. The ENV line displayed in the crt 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 the "Advanced Concepts and Commands" chapter.)

- TERM sets the terminal type for which output should be prepared.
- 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.

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 the "Shell Parameters" table later in this chapter for a listing of possible variables.

Setting up .kshrc

This shell script sets values, such as path names and aliases. These values can then be accessed by shell subprocesses. A `.kshrc` may look like:

```
lg=/abbreviation/of/long/path/name
alias who='who | sort '
set -o monitor
trap "$HOME/.logout" 0
```

where the command-line:

```
cd $lg
```

places you in the `/abbreviation/of/long/path/name` directory. The variable `lg` contains the long path name and the `cd` moves you to that directory. Again, the dollar sign (`$`) placed before a parameter designates using the value of `lg`.

The `alias` command is explained in the "Aliasing: Abbreviating Commands" chapter and the `set` command is explained in the "Advanced Concepts and Commands" chapter. The `trap` command-line is explained in the "Terminating Ksh" section. This just gives you an example of the types of things to put in a `.kshrc` file.

Once again, you can create your own `.kshrc` using an editor, but you must set the ENV variable to it in `.profile` or the system does not read it when invoking a new shell.

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/tricia
IFS=
HISTFILE=.sh_history
HISTSIZE=30
LOGNAME=tricia
MAIL=/usr/mail/tricia
MAILCHECK=600
PATH=/bin:/usr/lib:/usr/bin:/users/tricia/bin:.
PPID=29590
PS1=$
PS2=>
PS3=#?
PWD=/users/tricia/ksht/man
RANDOM=15314
SECONDS=0
SHELL=/bin/ksh
START[0]=/users/tricia/.kshrc
TERM=hp2392
TMOUT=0
TZ=MST7MDT
VISUAL=vi
```

For an explanation of each of these, see the "Shell Parameters" table. 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 Korn Shell.

Table 2-1. Shell Parameters

Parameter	Definition
#	Represents the number, in decimal, of positional parameters supplied to a shell script.
-	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 this shell.
!	Represents the process number of the last background process invoked.
–	Represents the last argument of the previous command-line.
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 the "Advanced Concepts and Commands" chapter.)
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 (i.e., space, tab, and new-line), which are used to separate command words during command or parameter substitution and when using the **read** command.
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.

Table 2-1. Shell Parameters (continued)

Parameter	Definition
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_files*. 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. You can type in **ps -f** to see the number under the PPID heading and the PID which is the current process number.
PS1	Defines the primary prompt string for a shell. The default is **"$ "**. If you use the **!** character the primary prompt string includes the current command's number (i.e., **"!$ "**).
PS2	Secondary prompt string, by default **"> "** used on command or script continuation lines.
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.

Table 2-1. Shell Parameters (continued)

Parameter	Definition
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. (See the **select** command in the "Advanced Concepts and Commands" chapter.)
SECONDS	Returns the number of seconds since the shell was invoked.
SHELL	The path name where the shell itself lives. This is also the shell used when starting a subshell. If an **r** is specified in the basename (i.e., **/bin/rksh**), then the shell is **restricted** (has fewer capabilities).
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 the "Advanced Concepts and Commands" chapter.)

Some of these variables are set automatically at login. They are:
#, -, ?, $, –, HOME, PPID, PWD, OLDPWD, RANDOM, REPLY, SHELL, and SECONDS.

Others are also given default values: PATH, PS1, PS2, PS3, MAILCHECK, TIMEOUT, and IFS. Again, use the **set** command to check these values before editing or creating a **.profile** that changes them.

Terminating Ksh

There are several ways to exit the shell depending on the current value of the Boolean flag `ignoreeof`. The value of `ignoreeof` is assigned with the `set` command (i.e., by typing `set -o ignoreeof`). (See the "Advanced Concepts and Commands" chapter 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              <====ignoreeof is set
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
```

If `ignoreeof` is set to `off`, it is not set.

Using exit

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.
```

Then, when you type:

```
$ exit
```

you are logged out. If `ignoreeof` is not set, use CTRL-D or `exit` to logout.

Using logout

If you want some special action to occur when you logout, use the `trap` command. The **trap** captures a signal and then executes a defined command-line.

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 it successfully traps a 0 signal. The 0 signal is sent on exiting your current shell. `$HOME` evaluates to the value of HOME.

For details on the `trap` command see the "Advanced Concepts and Commands" chapter.

Your `.logout` script might contain things like an echoed message:

```
echo "Have a Nice Day".
clear
```

or a `clear` statement that clears the terminal's screen.

Notes

Using Metacharacters 3

Certain characters or combination of characters in Korn Shell have special syntatic and semantic meanings. These characters are called **metacharacters**. Metacharacters affect the operation of the Korn Shell when read by the shell and at no other time.

New users should read this chapter completely; experienced users need only read the "Two-way Pipes" section for the new metacharacter `|&`.

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 symbol 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 this 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
tricia     tty00        Oct  4 13:34
stefan     tty04        Oct  4 14:49
keith      tty05        Oct  4 14:49
$ who | sort
dave       tty03        Oct  4 14:49
keith      tty05        Oct  4 14:49
michael    tty02        Oct  4 14:49
stefan     tty04        Oct  4 14:49
tricia     tty01        Oct  4 13:34
```

Two-Way Pipes

Two-way pipes or **coprocessing** can be established between a parent and child process. The **parent** process is the original shell and the **child** process (or subprocess) is the command or shell spawned from the parent shell. Typing `ps -f` displays the parent process id (PPID) and the child process id (PID).

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 the "Advanced Concepts and Commands" chapter 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.

Following is a table, "Separating and Terminating Metacharacters", that describes each of the metacharacters used by the Korn Shell. For each metacharacter, 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 3-1. Separating and Terminating Metacharacters

Meta-character	Example	Description

;	`$ whoami; ls`
	`stefan`
	`file1`
	`file2`
	`file3`
		Separates commands that are executed in sequence. In this example, **ls** is executed only after the **date** command completes.

&	`$ lp prog.c &`
	`[1]  4094`
	`$ echo hello`
	`hello`
	`request id is lp-725`
		Indicates that the command is to be executed as a process in the **background**. 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. 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. In this example, the **lsf** lists the files only if the **mail** command fails.

File Name Completion

Korn Shell implements 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  tricia users   56  Sep 14 03:59 data_structure_3
```

It actually expands the name and 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 up to the point of change. 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 might want to see all the possible expansions; type:

```
$ ls data_structure_[ESC]=
```

where [ESC]= lists:

```
1) data_structure_1
2) data_structure_2
3) data_structure_3
$ ls data_structure_[a][1][Return]
data_structure_1
$
```

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]. This special mode for adding text to a command-line is explained in the "Editing Command-lines" chapter.

Another expansion character is *. The star expands the current word to the entire list of file names that match. For example:

```
$ ls data[ESC]*
```

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=$PATH[ESC][ESC]
```

expands to:

```
$ PATH=/usr/bin:/usr/lib:/bin:/users/tricia/bin
```

You can now edit and change the value of your PATH variable using the techniques described in the "Editing Command-lines" chapter.

Path Name Completion

This expansion process completes directory paths in a similar manner, such that:

```
$ ll /users/t[ESC][ESC]
```

expands to:

```
$ ll /users/tricia
```

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. See the table "File Name Substitution Metacharacters" for the metacharacters used in pattern matching.

Table 3-2. File Name Substitution Metacharacters

Meta-character	Example	Description

?	`$ ls prog.?`
	`prog.a`
	`prog.c`
	`prog.o`
		Matches 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 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.

Quoting

Each of the metacharacters discussed above can be quoted to make it stand for itself and not be interpreted by the shell as a special character. The table "Quoting Metacharacters" covers these metacharacters.

Table 3-3. Quoting Metacharacters

Meta-character	Example	Description
\	`$ ls prog.\*` `prog.*`	The backslash \ cancels the special meaning of the following metacharacter. 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 (') quotes everything enclosed in the two single quotes except the single quote itself.

"	`$ echo "$PWD"`
	`/users/tricia`
	`$ 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.

`	`$ echo I am: 'whoami'` `I am: tricia`	The back quote causes the enclosed command-line to insert its output at that point on the current command-line.

Input and Output

Standard input (stdin) is the place from which a program reads its input (the default is the keyboard). **Standard output (stdout)** is the place to which a program writes its output (the default is the terminal display). **Standard error (stderr)** is the place where the system writes error messages (the default is the terminal display).

When a command is executed, its input, output, and error 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 input from a file rather than the keyboard, use the < symbol. To redirect output to a file rather than the display, use the > symbol. See the table "Input/Output Metacharacters".

Table 3-4. Input/Output Metacharacters

Meta-character	Example	Description
`< word`	`$ mail joe < letter`	This symbol redirects the contents of letter to input to `mail`.
`> word`	`$ ps > processes`	This symbol redirects output from **ps** into the **processes** file deleting any current contents.
`>> word`	`$ date >> processes`	This symbol redirects the output from **date** to the end of the **processes** file, unless it does not exist; then it creates it. (It appends output to the file.)

`<< [-] word`	`$ cat << eof`
	`> write`
	`> until`
	`> eof`
	`write`
	`until`
		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 << then all leading tabs are stripped from *word* and from the resulting document.

`<&digit` `>&digit`	`$ echo output 1>&2` `output`	This input redirection symbol 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 >&-`	This closes standard input or 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 disassociates 2 from standard error so that output is sent to the file *fname* rather than the display. It then associates file descriptor 1 (**stdout**) with the file associated with file descriptor 2 (which is *fname*). The **echo** command normally 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*. For example:

```
$ echo hello 2>fname 1>&2
$ more fname
hello
```

Other Metacharacters

A few other metacharacters to be aware of are: **#**, **~**, and **%**.

- Shell comments are indicated by the **#** symbol when it is the first character in a word followed by other words until a new-line (e.g., **# This is a comment**). This is explained in the "Basic Ksh Programming" chapter.
- The tilde substitution symbol **~** which allows path name substitution is explained in the "Substitution Capabilities" chapter.
- The **%** symbol which allows job number substitution is explained in the "Substitution Capabilities" and "Controlling Jobs" chapters.

Aliasing: Abbreviating Commands 4

Aliasing is a method by which you can abbreviate long command lines, create new commands 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

Using aliases, you can define commands to shorten long command-lines to simple letters or redefine commands that are pre-defined by the system. To create an alias, use this syntax:

```
alias [ -tx ] new_word='old_command_line'
```

Anytime `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 *new_word* parameter specifies the new alias's name and the *old_command_line* 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 the `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/stefan
$ hcd p
hello
$ pwd
/users/stefan
```

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/stefan
```

Tracking Aliases

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 the "Advanced Concepts and Commands" chapter 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

Exporting aliases works in much the same way as exporting variables with `export`. When you export an alias it becomes accessible to subshells such that when you execute a script or new shell the alias remains defined. 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'
```

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, 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
hash=alias -t
history=fc -l
integer=typeset -i
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 the "Advanced Concepts and Commands" chapter. The aliases shown are all the default aliases set by the Korn Shell upon invocation.

In this example, 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 the "Advanced Concepts and Commands" chapter.

When you create an alias and then execute it, the shell adds it to the table of aliases. After creating 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/stefan                 <—
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 up the **who** command as 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 as well as metacharacters. Suppose a shell script, **scrip**, exists on your system. **scrip** is a file containing a set of command-lines, which are executed when you type the file name:

```
echo Users logged in are:
who | sort
echo I am `whoami`
echo Current directory is `pwd`
```

  These are the command-lines contained in the script. Now, you can set the alias:

```
$ alias i='scrip'
$ i
Users logged in are:
michael    tty4        Sep 24 09:41
nick       tty1        Sep 24 09:41
tricia     tty2        Sep 24 14:19
I am tricia
Current directory is /users/tricia
```

  Creating scripts is described in the "Basic Ksh Programming" chapter.

- Aliases only take effect 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.

```
$ i
ksh: i: not found
$ alias i='scrip'
$ i
Users logged in are:
michael    tty4        Sep 24 09:41
nick       tty1        Sep 24 09:41
tricia     tty2        Sep 24 14:19
I am tricia
Current directory is /users/tricia
```

Here trying to execute the `i` alias before setting it causes the system to not recognize the new command. Once it is set, as in the second line, it runs and returns the output.

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. To gain the old functionality, you can unset aliases. Unsetting an alias is simple, just 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 depending on the directory you select:

```
$ who
michael     tty04        Sep 24 09:41
nick        tty01        Sep 24 09:41
tricia      tty02        Sep 24 14:19
$ unalias who
$ who
nick        tty01        Sep 24 09:41
tricia      tty02        Sep 24 14:19
michael     tty04        Sep 24 09:41
$
```

The Korn Shells default aliases (i.e., `false`, `integer`, ...) can be unset or redefined, as well.

Substitution Capabilities 5

Earlier in the chapter "Using Metacharacters", file name substitution and completion is discussed. In this chapter, the other substitution concepts: tilde, parameter, command, and process, are discussed. 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 "Process Substitution" for special Korn Shell features.

Tilde Substitution

This type of substitution replaces a single character, the tilde (~), with a full path name.

- 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 + statement is replaced with the path name in the PWD variable. PWD is set when `cd` is executed.
- A tilde followed by a - statement is replaced with the path name in the OLDPWD variable. OLDPWD is also set when `cd` is executed.
- 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 the user's login path.

These tilde sequences are demonstrated next.

To see the current values of HOME, PWD, and OLDPWD, use the **set** command with no options. For example:

```
$ set
HOME=/users/tricia
PWD=/users/tricia
...
$ cd bin
$ pwd
/users/tricia/bin
$ set
HOME=/users/tricia
OLDPWD=/users/tricia
PWD=/users/tricia/bin
...
$ cd ~/newdir
$ pwd
/users/tricia/newdir
$ cd ~
$ pwd
$ /users/tricia
$ cd ~-
$ pwd
/users/tricia/newdir
$ cd ~richard/anothernewdir
$ pwd
/users/richard/anothernewdir
```

In the first line, **set** lists the shell variables. The lines directly after the **set** show the values of HOME and PWD. OLDPWD may not be set if you have not executed a **cd** since login. The next line, changes the directory to **bin**. Typing **pwd**, displays the new directory's path. Typing **set** again, displays the changed shell variables and OLDPWD. Now, when **cd ~/newdir** is typed the shell replaces the tilde with **/users/tricia** and moves to that directory. When **cd ~** is typed, the shell moves to HOME and when **cd ~-** is typed, the shell moves to OLDPWD. The last line checks for **richard** in the login password file and then replaces the tilde sequence with **richard**'s login path.

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/tricia
$ alias cdn='cd ~/bin'
$ alias cdn
$ pwd
/users/tricia/bin
```

and when `cdn` is executed it places you in the `bin` directory in your HOME directory.

Parameter Substitution

A parameter is simply a shell variable (or argument) that is passed to or manipulated by a command or function. A **function** is a group of command-lines placed together is a certain area that can easily be accessed and used again.

The two types of parameters discussed in this section are:

- **keyword** or **named** parameters, and
- **positional** parameters.

Each of these is described in detail in the subsequent sections.

Parameter substitution allows a parameter to be named, assigned a value and then accessed by a command or function. The simplest form of this has been shown in earlier chapters:

```
$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

Keyword or named parameters are defined as identifiers or names containing alphanumerics and underscores (e.g., `new_prog1`). The value of a named parameter can be set using the syntax:

name=value

or

`typeset [ -HLRZfilprtux[`*n* `] [` *name*`[ =`*value* `] ] ... ]`

The first line shows the most common method of assigning values to names while the second shows the implementation unique to Korn Shell. For example:

```
$ x=1
$ typeset -i x=1
```

both set the *name* to `x` with a *value* of 1. However, the `typeset` command specifies `x` is an integer.

The `typeset` command has many options or attributes (such as readonly, case definition, and automatic exporting) it can assign to each *name*. See the "Advanced Concepts and Commands Chapter" 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`, up to `$9`. These correspond directly to the command-line such that `$0` is the first item and script name, and `$1` thru `$9` are the rest of the parameters on the line. For example:

```
cp file1 file2
```

The value of `$0 is cp`; the value of `$1` is `file1`, and the value of `$2` is `file2`. If you have more than 9 arguments, use the `shift` command to bring them in from a default buffer or access them using the syntax, `${17}`. A `shift` moves the value in `$2` into `$1` continuing down the line until it shifts a new value from the buffer into `$9`. The `${17}` accesses the seventeenth element on the line.

This function uses positional parameters and shifting: See the "Basic Ksh Programming" chapter for details on functions.

```
$ sf()
> { for arg in $1 $2
>       do
>        echo This is the function $0
>        echo $1 $2
>        shift
>        echo $1 $2
>       done       }
$ sf first second third
This is the function sf
first second
second third
$ sf first second
This is the function sf
first second
second
```

This function, **sf**, reads in three positional parameters on the command-line, **first**, **second**, and **third**. It then prints out the contents substituting the value for the **$1**, and **$2**. When **sf** is executed with only two parameters, a null value is shifted in and echoed. Explanations of functions and the **for** statement are provided in the "Basic Ksh Programming" chapter. For more details on the **shift** command, see the "Command Reference" chapter.

A second way to set positional parameters is by the **set** command. If **set** is the first item on a command-line, then the parameters immediately following are assigned to **$1**, **$2**, **$3**, up to **$9** just as before. So, typing:

```
$ set first second third
$ echo $1 $2 $3
first second third
$
```

assigns **first** to **$1**, **second** to **$2**, and **third** to **$3**.

Parameter Substitution Conventions

An **array** is collection of contiguous elements that can be accessed by a subscript. For example, suppose `arr` is the name of the array and `0` is the subscript, then `arr[0]` represents the first element of the array. For more details on arrays see the "Basic Ksh Programming" chapter. A subscript can also be metacharacters such as `*`, `$@`, `#`, and `$*`.

The following section covers special conventions used during parameter substitution:

${*parameter*}

Whenever characters following a **$*parameter*** conflict, curly braces can be used to prevent incorrect substitution. The 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.

When the *parameter* is an integer then it is a positional parameter, (.e.g., `$1` or `${1}`).

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
```

If the *parameter* describes an array with elements `*` or `@`, then the value for each of the elements is substituted. For example:

```
echo ${array[*]}
```

${#*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*[*]}

The number of elements in the array is substituted.

${*parameter*:-*word*}**	If *parameter* is set and non-null, the value is substituted; otherwise *word* is substituted. For example: `${arg:-third}`
${*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.
${***parameter*#*pattern*}** **${*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:

```
$ x="aaaabbbbcccc"
$ echo ${x#a*b}
bbbccc
$ echo ${x#c}
aaaabbbbcccc
```

As you can see in this example, the pattern **aaaab** is matched and then removed from the string. In the second **echo**, the **c** does not match the beginning value of the string, therefore, the full string is substituted.

Using the same value for **x**:

```
$ echo ${x##a*b}
cccc
```

matches the largest matching pattern (i.e., all the **b**'s) and leaves just **cccc**.

Using **x** again:

```
$ echo ${x%c*}
aaaabbbbccc
$ echo ${x%%c*}
aaaabbbb
```

the **c*** matches the end of the string and its smallest form is deleted, one **c**. The second **echo** deletes the largest form, **cccc**.

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 is placing the command to be executed within single back quotes (` `). For example:

```
$ echo "The people currently logged on the system are:\n `who`"
The people currently logged on the system are:
 tricia   console    Sep 11 09:01
 michael  tty09      Sep 11 10:35
```

In this example, the `who` enclosed in back quotes is executed and printed out within the `echo` command-line. The `\n` provides a new-line. This escape character outputs a new-line to the terminal. Escape characters are explained in the "Printing Out Data" section of the "Basic Ksh Programming" chapter.

The Korn Shell implements this substitution capability using both back quotes and the syntax `$(command)`. For example:

```
$ echo "The people currently logged on the system are:\n $(who)"
The people currently logged on the system are:
 tricia   console    Sep 11 09:01
 michael  tty09      Sep 11 10:35
```

This performs the same function as the back quotes. The `$` evaluates to the value or output of the commands within the parenthesis (`()`) and replaces the command with the command's output in the original command-line. Using this alternate 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 first command-line. The backslash cancels the second back quote from closing the `echo`. Therefore, the third `echo` is evaluated and outputs the `hello`. The second command-line performs the same function in a far less complicated manner.

There is no limit on the number of commands that can be placed within quotes marks or parenthesis. 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
tricia   console    Sep 11 09:01
michael  tty09      Sep 11 10:35
```

For 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.

A quicker and shorter form of the above command is:

```
$ echo "\n $(< file)"
```

Again, the contents of **file** are displayed. An even faster syntax is:

```
$ echo `< file`
```

Although, this outputs the file on one continuous line.

Notes

Editing Command-lines 6

Typing a long command-line, finding a mistake after executing it, re-typing the command, and finding another mistake, etc. can be very frustrating. **Command-line editing** 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 Korn Shell: the `fc` command, the `vi` line edit mode, and the `emacs` and `gmacs` line edit mode.

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 Korn Shell and the `fc` command may be of interest.

Accessing the History File

In the earlier chapter, "Starting and Stopping the Shell", 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 hit Return it is stored in this history file. HISTSIZE specifies the maximum number of commands stored in that file. For example:

```
HISTFILE=/users/tricia/.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 In-line Editing Modes

There are three types of editing modes available in Korn Shell: the `fc` command, the `vi` line edit mode, and the `emacs` and `gmacs` line edit mode. A discussion of each of these methods follows. These Korn 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 the fc Command

There is a built-in command, `fc` (fix command), special to the 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
```

```
$ fc -l 24
23 pwd
24 lsf
25 echo surprise
...
```

If followed by a number, as in `fc -l 24`, then command-lines from 24 on are displayed. Two other options are available: `-r` which reverses the order of the commands and `-n` which suppresses the command numbers from being listed. For example:

```
$ fc -e vi -n 24 25
```

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 an `-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
20 11 -a
21 more file
22 ps
23 pwd
24 lsf
25 echo surprise
26 cd /users/guest
27 pwd
28 cp /users/guest/file1 /users/stefan/file2
29 more file2
30 ll file2
31 chmod +x file2
32 rm /users/guest/file1
33 lsf /users/guest
34 pwd
35 x=file1
36 echo $x moved to new directory
```

In this next example, the second syntax line, allows immediate replacement of an *old* string with a *new* string in the *command*. In this instance, *command* can either be a command name or line number. Korn 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 `-`. For example:

```
$ echo surprise
surprise
$ fc -e -  surprise=neat echo
echo neat
neat
```

where the `-e -` calls on the special editor built into `fc`. Then, `surprise` is replaced by `neat` and echoed to the screen.

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`.

Using vi Line Edit Mode

Korn Shell implements a builtin **vi** screen editor that works on single lines. If you are unfamiliar with the **vi** editor, see *HP-UX Concepts and Tutorials: Text Editors and Processors* or the *A Beginner's Guide to Text Editing.*

To execute a command, you normally type in the command-line followed by a [Return] or [Enter]. You enter the **vi** edit mode by pressing the [ESC] key instead. After editing, you execute the changed line by pressing the [Return] or [Enter]. Since the shell is not normally in **vi** edit mode, you enable that mode by either executing the **set** command or setting the shell parameters EDITOR or VISUAL to **vi**.

Enabling vi Line Edit Mode

There are several ways to enable an editor mode. One is to type:

```
set -o vi
```

which makes the shell's default editor **vi**. For further details on the **set** command see the "Advanced Concepts and Commands" chapter.

Another is to set and export the VISUAL shell variable in your **.profile** or **.kshrc**:

```
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 in and executing command-lines just as you did 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 surrpri
```

Then, before you hit the [Return], hit [ESC]. Now you can move on the line using the [Back space] and space bar (not the arrow keys) to the point where you made your first mistake. Then it's a simple matter of executing the `vi` delete command, `x`, on the extra `r` and then the append command `a` on the end of the line:

```
$ echo surrpri[ESC][Back space][x]space[a][s][e][ESC][Return]
```

The above [Back space] is actually hit three times and the space bar is hit two times. The new line and output looks like this:

```
$ echo surprise
surprise
```

Granted, for this example the command-line is pretty short, but on long commands in-line editing can be very useful.

For a complete listing of all the `vi` commands usable within the Korn Shell `vi` mode, see the *ksh(1)* manual page in the *HP-UX Reference.*

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]. These sequences allow you to search the history file:

[ESC] [k] and [ESC] [-]	Specifies the previous command. Once you type [ESC] [k], type just [k] to step through. This applies to [-] also.
[j] and [+]	Specifies the next command forward. Once you type [ESC] [k] or [ESC] [-], type just [j] to step through. This applies to [+] also.
[ESC] *count* [G]	Specifies the command with the number *count.*

So, if the set of commands looked like:

```
$ fc -l
...
20 ll -a
21 more file
22 ps
23 pwd
```

and you executed an [ESC] [k] the shell displays:

```
$[ESC] [k]
$ pwd
```

For every [k] typed after that, the shell displays one line further back at the same prompt.

If you go too far backwards in the history file, move forward again using the [j]. For example:

```
$ ps[j]
$ pwd
```

If you use [j] or [k] to roll off a command-line you are editing, all the changes are lost.

If you want to specify a certain line number, use [ESC] *count* [G], such as:

```
$ [ESC] 20 [G]
$ ll - a
```

Once you find the command-line you are searching for, you can simply re-execute it by typing a [Return], or edit it using the `vi` in-line edit commands, and then re-execute it.

Using emacs and gmacs Line Edit Mode

The other editors Korn Shell implements 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 hold the [ESC] key down while pressing 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 **vi** section above.

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. To perform the in-line edits type [CTRL] and while holding it down type another character or type [ESC] and while holding it down type another character. For example, suppose you type:

```
$ echo surrpri
```

before you hit the [Return], hit four [CTRL]-[b]s. 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]
```

The [CTRL]-[b] is actually hit four times. 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 Korn Shell **emacs** mode, see the *ksh(1)* manual page in the *HP-UX Reference.*

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:

[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.

Basic Ksh Programming 7

The Korn Shell is not merely a command interpreter; it is also a programming language with all the standard constructs needed to write detailed shell scripts. The major constructs of the Korn Shell, such as inputting and outputting data, conditional statements, and functions, are discussed in subsequent sections.

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 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. A script can be created two ways: interactively, or by creating and editing a file.

Certain shell programming constructs allow you to interactively create a shell script, such as curly braces (`{ }`) the `for` loop, or the `if` statement. (These are both explained in subsequent sections.) When you type the first command-line containing the construct, it tells the shell that the script continues on on the following lines until a closing command-line tells the shell the construct is complete. After the first line, the shell prompts you with the value of the secondary prompt, PS2, to input the rest of the script. The default value of PS2 is `>`. When the shell reads the closing statement of a construct, it automatically exits the input mode and executes the script. For example:

```
$ {
> echo Hello, welcome to Korn Shell
> }
Hello, welcome to Korn Shell
```

This example demonstrates the secondary prompts' appearance after the bracket, `{`, and acceptance of input until the closing bracket, `}`, is typed.

The second method of creating a script is to create and edit a file using an editor, such as `vi`. If you are unfamiliar with how to create a file using an editor, see the *HP-UX Concepts and Tutorials: Text Editors and Processors* or the *A Beginner's Guide to Text Editing*. Once you create the file (i.e., the script) containing your command-lines, you are ready to execute the script.

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 the *Introducing UNIX System V* book. Then type the *script_name* (i.e., file name):

```
$ 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. To comment a line in a script place a # at the beginning of the line. Everything after the # up to a new-line is ignored by the system. For example:

```
# This script prints out every executable file.
  for i in 'ls'              # for all files in the current directory
```

The whole first line is ignored by the system and in the second line everything after the # is ignored.

Inputting and Outputting Data

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 in 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 the "Substitution Capabilities" chapter. 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. Its syntax is:

read [**-prsu**[*n*]][*name*?*prompt*][*name*...]

where 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*. If *name*s are not specified, the line is read into the Korn Shell REPLY variable (see **select** under "Conditional Statements"). If the ?*prompt* is set the user is prompted interactively with *prompt*. The definitions of the options are:

-p	Read from the output of the process spawned with two-way pipes, **|&**. (See the "Advanced Concepts and Commands" chapter 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 leaves the cursor one blank to the right of the ? waiting for input from you:

```
echo 'What is your name? '
read name
echo "Hello, $name, and welcome to Korn Shell"
```

The second line reads in text from the user and saves it in **$name**. Finally, a line is printed which includes the value of **$name** (since the string is in double, not single, quotes). 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), followed by a [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**, which is presumably ignored.

Printing Out 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 Korn Shell. The first is the **echo** command used in Bourne Shell and C Shell; and the second is the **print** command, unique to the Korn Shell.

Using echo

The **echo** command can print out comments, data, or the values of positional parameters to the display. The syntax is:

```
echo [ arg... ]
```

where **echo** writes to standard out any *arg*s (arguments) separated by blanks, or a blank line, if no arguments are specified.

```
var='short'
echo 'This is a' $var 'example.'
echo
```

In this script, `var_script`, the value of `var` is set to `short` and `echo` prints the line `This is a short example.`

```
$ var_script
This is a short example.
```

The quotes keep the `$var` from being part of the text, so that the parameter substitution is performed and it evaluates to `short`.

You can also prompt a user from a script using the `echo` command and the `\c` **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 characters** must be preceded by a backslash and enclosed in quotes for interpretation, such as the `\c` shown above. They are:

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

The Korn Shell provides a unique output mechanism the other shells do not, the **print** command. Its syntax is:

```
print [ -Rnprsu[n ] ] [ arg... ]
```

The **print** command provides a superset of the **echo** command for shell output. It prints the specified *args* 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
`-p`	write output to the process spawned with a two-way pipe, **|&**, instead of standard output (See the "Advanced Concepts and Commands" chapter for two-way pipes.)
`-r`	ignore all **echo** escape sequences
`-s`	save *args* in the history file
`-un`	write to the file descriptor *n*

This **print** command:

```
$ print -s "# End of the day."
$ history
```

puts the comment **# End of the day.** 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.

Conditional Statements

The Korn Shell provides 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: `if`, `case`, `select`, `for`, and `while`.

Using the if Condition

The `if` statement allows you to execute one or several commands if a certain condition exists. The syntax is:

```
if command-line
then diff_command-line
else even_diff_command-line
fi
```

First, `if` checks if *command-line* is true. **True** means it returns 0. If it is then, *diff_command-line* is executed; if it is not, *even_diff_command-line* is executed. Such that:

```
$ x=hello
$ if [ $x = hello ]
> then echo Welcome
> else echo Goodbye
> fi
Welcome
```

This `if` statement checks whether `x` equals `hello`. If it is, `Welcome` is printed; if it is not, `Goodbye` is printed.

Using the test Command

This command tests or evaluates the *expr* and if it evaluates **true**, it returns a zero exit status. If it evaluates **false**, it returns a nonzero exit status.

Its 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 unique to the 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*. (i.e., they are identical files)

In this example:

```
$ for file in 'ls'
>    do
>        if [ -x $file ]
>        then echo $file is executable
>        fi
>    done
$
```

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.

Using the case Statement

The `case` statement eliminates the need for several `if then else then else fi` statements to be strung together. It 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 *patterns* 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 Korn Shell that prints on the screen a set of *words* 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 *words*, 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, just type [Return], it reprompts for input. No matter which way it evaluates, the *command_lines* are executed. The loop continues until a break is encountered. The syntax is:

```
select parameter in words
do
  command_lines
done
```

For 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
#? 1
a is a vowel.
#? 4
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 the shell waits for a number and [Return] to be typed. When it receives the number, it `echo`s that the corresponding letter is a vowel and then prompts for the next entry. It continues prompting until [Break] is pressed. If you enter 6, which is not set, a null (`is a vowel`) is returned. If you enter nothing, just type `Return`, it reprompts.

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*. The syntax is:

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

In the following example:

```
$ for file in x y z
> do
>   echo The file name is $file
> done
```

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.

Using the while/until loops

This loop continues executing *command-line* and processing through the `list` as long as the item an *list* continues to evaluate true. Once an item evaluates false, the loop is exited. The syntax is:

```
while list
do list2
done
```

```
$ 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`, and then increments and prints out the value until it equals 10 and you exit the loop.

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.

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.

```
$ 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 file it encounters. If none are executable, `$file` is left set to to `none` but it is not printed.

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.

The syntax is:

```
continue [ n ]
```

If *n* is specified, then resume execution starting at the *n*th enclosing loop.

```
$ 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.

Arithmetic Evaluation Using let

In the Korn Shell, there is a unique command, `let`, which allows arithmetic expressions to be used in Korn Shell scripts. This command allows for 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. A list of operators, in decreasing order of precedence, follows:

Operator	Description
-	unary minus
!	logical negation
* / %	multiplication, division, remainder
+ -	addition, subtraction
<= >= < >	comparison
== !=	equals, not equals
=	assignment

In this example, `x` is set to 1.

```
$ x=1
$ let x=x+1*6-3/1
$ echo $x
4
```

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.

You can also use parenthesis to create this effect:

```
$ let "x=x+(1*6)-(3/1)"
$ let "x=(x+1)*6-3/1"
```

or override the operator's precedence to produce different results, 9. When using parenthesis, double quotes are necessary.

This 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
> 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 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, the Korn Shell starts 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 the "Parameter Substitution Conventions" section of the "Substitution Capabilities" chapter for other possible array subscripts and uses for arrays.

Writing Functions

The `function` command is used to modularize programs. **Modularization** is the concept of placing often used code in a certain area (module) of the shell script. Then you call the module or function whenever it is needed rather then re-enter 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 the *shell script* is inclosed within curly brackets, {}. Just using `name` followed by parenthesis, `()`, and the `{ shell script }`, also creates a function.

To invoke the function, type the *name* followed by any positional parameters that need to be passed in as arguments.

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
```

where the argument being passed in is `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                    < -call to function
done
```

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 a function and then 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**.

For example:

```
$ 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.

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 this recursive function:

```
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
```

Then 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 inputted number is printed, and the function exited.

Controlling Jobs 8

A simple **job** is one command typed to the shell. More complex jobs consist of one or more commands typed together as a pipeline or as a sequence of commands separated by semicolons (sometimes called a command-line). This is an example of a command-line that the shell interprets as a job:

```
$ ps -ef | sort > processes
```

Creating Jobs

The shell associates each command-line you type with an integer job number. Therefore, every time a command-line is typed the shell creates a job for that command-line and gives it a unique job number. Once a job is created, you can monitor it or manipulate it in other ways. 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 also 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's number is displayed inside the brackets ([ ]). The + marks the job as the current job and the - marks the job as the previous job. **Done** or **Running** specifies the status of the actual job. The **lp processes** is the actual command-line and is telling to the system to print the **processes** file to the line printer.

Suspending Jobs

On the Series 800, you can **suspend** jobs. Suspending a job is stopping it from completion at some midway point, but not destroying it. Suspending a job allows you to stop in the middle of the process and have control of your terminal returned to you for other work.

Suppose you type in command-line and hit [Return], realize this process takes a long time and you need to print another job. To suspend the current job type:

[CTRL]-[Z]

This suspends the previous command-line you typed in and returns you to the prompt (**$**). For example:

```
$ du | sort > diskusage
CTRL-Z
[1] + Stopped du | sort > diskusage
$
```

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 above 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.

Putting Jobs in Background/Foreground

Fortunately, there is way to free up your terminal and at the same time still run long processes such **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 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 indicates what the system returns, a job number and a process number.

If the **set -o monitor** option is on, (i.e., you type **set -o monitor** at the terminal), when a job completes it sends a message to the terminal of the form:

```
[1] + Done  du | sort > diskusage&
```

signifying the job by its number and that it has completed, **Done**. (See the **set** command in the *Advanced Concepts and Commands* chapter for details.)

Both the Series 300 and Series 800 system shells create and number jobs, as well as allow background processes. However, only the Series 800 allows you to manipulate the jobs from the shell by pulling background jobs into the foreground and suspending foreground jobs. The Series 300 cannot reaccess background jobs once they are placed in the background. It can only monitor their progress using the **jobs** command or wait for the processes completion response by having using the **set -o monitor** command.

On the Series 800, two commands allow you to manipulate jobs between the background and foreground. They are **bg** and **fg**. The **bg** command allows you to place a job in the background while the **fg** command allows you to pull a background job into the foreground or back to the terminal screen.

Suppose, you placed a job in the background using the **&** and then wanted to pull it back to the screen; type:

```
$ fg %job_number
```

or type **%%** or **%+** if it is the current job. If it was the previous job, that is you have typed another command after placing the command in the background, use **%-**. For example:

```
$ du | sort > diskusage&
[1]  6100
$ sleep 999&
[2]  6102
$ fg %-
du | sort >diskusage
```

this brings the previous command, which is **du**, back to the foreground. The second background process, **sleep** command, suspends execution of the shell for **999** seconds. If you later decide you want your terminal free again, type:

```
$ bg %1
```

and put it back into the background.

You can also use these two commands on suspended jobs to restart them in the 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
```

and then realize you do not want to list the full file system (i.e., you do not use * in the command-line), 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 you the current `lsf` job has been terminated.

If you log off the system while any of your processes are running, background or otherwise, the jobs are destroyed unless you use the **nohup** command (see the *HP-UX Reference* for details).

Advanced Concepts and Commands 9

This chapter explains advanced topics and commands you will need to understand the more difficult aspects of the Korn Shell.

The ENV Variable

In the earlier chapter, "Starting and Stopping the Shell" the ENV variable was discussed. The ENV variable specifies a file, usually `.kshrc`, which is executed when ever you spawn a new interactive 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 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.

Consequently, a very complicated but effective ENV variable had been developed. This line, when placed in your `.profile` allows new Korn Shells to be created very quickly. That ENV variable is:

```
ENV='${START[ (_$-= 1) + (_ = 0) - (_$- != _${-%%*i*}) ]}'
START=~/.kshrc
export ENV START
```

This complicated line is broken up and each part is described in the the rest of this section.

In the first part:

```
ENV='...'
```

even though ENV is evaluated for parameter substitution at every use, the quotes hide it from immediate evaluation.

```
${START[...]}
```

This parameter substitution, after the rest of line is evaluated, sets the element `[0]` of array START, to the name of the `.kshrc` file.

```
(...) + (...) - (...)
```

Within the brackets is this arithmetic expression, which is the array parameter indices.

```
_$- = 1
```

In this assignment statement the parameter named _<*current flags*> and is assigned the value 1. The expression also evaluates to 1. The leading underscore is just a way to insure the left side is never null even if *$-* evaluates to null.

```
_ = 0
```

This parameter named _ is assigned the value 0.

```
_$- != _${...}
```

At this point, `$-` and `${...}` are being expanded, and then the resulting parameter named _<*result*> is being expanded too. The parameter _<*current flags*> is compared with `_${...}` and the result is either 1 (true) or 0 (false). If the result is true, they are not equal, so the element is 0 and ENV gets set to something. If the result is false, they are equal and ENV evaluates to null.

```
-%%*i*
```

If the shell's options or current flags (`$-`) pattern matches `*i*`, (that is it contains an `i` for interactive), then the parameter evaluates to null.

If the shell is interactive and `$-` includes `i`, the left side `_$` evaluates to a non null, which has the value 1. Since the right side is null and which has value, the two are not equal and return true (1). Therefore the array element 0 is used and ENV evaluates to non-null.

If the shell is not interactive, `$-` may be null or not. If `$-` is not null, `$_<result>` (1) is equal to `$_` (1) and the expression yields zero. Then element 1 is used which does not have a value and ENV evaluates to null.

Two-way Pipes

Two-way pipes or **coprocessing** can be established between a parent and child process. 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 the "Basic Ksh Programming" for details on the `print` and `read` commands.

Two-way pipes allow shell scripts to pass data through pipelines and bring it back for further use by the script again, without using temporary files. This allows a shell script to interact with a pipeline in real time. 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 a command 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.

```
$ 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
while
:
alias
this
/usr/bin/file
$ whence -v while true alias this file
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
```

The first part of this example defines a function `this()`, then asks `whence` to explain a series of five different words that might be used as commands.

This set was chosen to demonstrate the five types of command words the shell understands, in the order in which they take precedence:

- reserved-word,
- alias,
- built-in,
- function,
- and other, such as the path name.

The set Command

The **set** command is used to turn on and off shell options in the environment such as tracking or automatic exporting of commands. Its second function is to reset the values of positional parameters (**$1**).

Its syntax is:

set [-aefhkmnostuvx] [-o *option...*][*value...*]

where *value* specifies the positional parameters to be reset. The *option* can specify with a word the same meaning as the the **-aefhkmnostuvx** letters. For example:

```
$ set -v
$ echo hello
echo hello
hello
$ set +v
$ echo hello
hello
$ set -o verbose
echo hello
hello
$ set +o verbose
$ echo hello
hello
```

the **set -v** and **set -o verbose** perform the same task; print each line as it is read followed by the output. The **+** in front of the **v** and **o**, turns the verbose option off so that it can be turned back on in the next line.

A discussion of other options follows:

`-a`	All subsequent parameters that are defined are automatically exported.
`-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
```

which causes the shell to create a tracked alias for every command executed.

The Korn Shell implements 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`.
`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

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.

For example:

```
$ typeset -r year=2000
$ echo $year
$ year=2001
ksh: year: is readonly
```

makes **year 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 upper-case characters converted to lower-case. The upper-case 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 lower-case characters are converted to upper-case characters. This turns the lower-case flag, `-l` off.

`-x` The given *names* 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 *names* (and optionally the *values*) of the parameters which have these flags set is printed. (Using + rather than - keeps the values to be printed.) If no *names* and options are given, the *names* 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

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 *signals* sent to the shell, traps it and then executes *arg*. If *signal* is 0, then *arg* is only executed once the shell is exited. After setting traps, typing **trap** with no *args* lists all commands associated with signals.

For example:

```
$ temp="/tmp/xyz$$"
$ trap "rm -f $temp; exit" 0 1 2 3 15
$ trap
0:rm -f /tmp/xyz18996; exit
1: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 (1, 2, 3, 15), which could be given by names (HUP, 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 -, then all trap *signals* 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 0 or EXIT for a trap set outside any function then the command *arg* is executed on exit from the shell.

The ulimit command

This command sets limits on specified resources used by a spawned or child process (subprocess).

The syntax is:

```
unlimit [ -fp ] [ n ]
```

where *n* is the size to be set depending on the type of limit set by the options `-fp`. 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
```

`-f`	imposes a size limit of *n* blocks on files written by child processes (files of any size may be read)
`-p`	changes the pipe size to *n*

Notes

Command Reference 10

This chapter is a command reference for the 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, C, or Korn. It is meant as a quick reference or refresher for the basic commands used in the Korn Shell.

alias

Syntax

```
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, 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, 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 ]
```

On the Series 800, the **bg** command places job *n* (the job's number) in the background. The current job is put into the background, if *n* is not specified.

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

```
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 *patterns* 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 present working directory to the directory, *otherdir*, which is directly above the one you are in currently.

```
$ 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` $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, first skipping two lines (**\n**), a usage message including the invocation name of the script, which is 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 the chapter "Basic Ksh Programming" 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 *args* 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 *args* 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: " 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.

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 the chapter on "Editing Command-lines" for a detailed explanation of `fc`.

fg

Syntax

```
fg [ %n ]
```

On the Series 800, 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* inclosed in curly brackets, `{}`, 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 (see below). See the chapter "Basic Ksh Programming" 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 diff_command-line
else even_diff_command-line
fi
```

First, `if` checks if *command-line* is true. If it is then, *diff_command-line* is executed; if it is not, *even_diff_command-line* is executed.

Example

```
if [ $x = passwd ]
then echo "Welcome to Korn"
else echo "Please log off"
fi
```

The `if` checks whether the value of `x` is equal to `passwd`. If it does equal, the first `echo` line is printed; otherwise the second line is printed

jobs

Syntax

```
jobs [ -l ]
```

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.

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

`kill [ -`*signal* `]` *process id*

This command cancels (kills) the designated *process id* using *signal* if specified. The *signals* are specified by number or name; see the manual page *signal(2)* of the *HP-UX Reference* for a list of signals. If *signal* is not specified, the `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, and then sends it a kill (SIGKILL).

```
$ kill -l
$ kill -HUP 3140
```

Using the `-l` option with `kill` lists the signal names. Then the second line sends a SIGHUP to process 3140.

let

Syntax

`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. A list of operators in decreasing order of precedence follows:

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.

newgrp

Syntax

```
newgrp [ arg... ]
```

This command changes your group identity to the new group specified by *arg*. If *arg* is not set, it defaults to your login group. You must be a legal member of a group to change to that group.

Example

```
$ newgrp other
```

This changes you from the current group identity to group `other`.

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 *args* 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
`-un`	write to the 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

`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

`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.

return

Syntax

```
return [ n ]
```

The **return** command stops execution of a function and then 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 *words* 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 *words*, 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_lines* 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
#? 1
a is a vowel.
#? 4
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 `echo`s that the corresponding letter is a vowel and then prompts for the next entry. It continues prompting until a [Break] is hit. If you designate 6, which is not set, a null (`is a vowel`) is returned.

set

Syntax

```
set [ -aefhkmnostuvx ] [ -o option...][ arg... ]
```

This command is used to set shell options as well as reset the values of positional parameters (*arg*). See the chapter "Advanced Concepts and Commands" 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 turns on the vi-mode history editing.

Also, **set** is 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 positional parameters ($1, $2, $3, etc.) left one position such that $1 now contains the value of $2 and $2 contains the 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 ($2) in **$zopt**. Using quotes preserves any whitespace embedded in **$2**. Also, note the shifting of arguments such that **$2** has the correct value when it's needed. **$@** is only evaluated once, before the first shift takes place.

test

Syntax

test *expr*

or

[*expr*]

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* on the *test(1)* manual page. The four unique to the 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*

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 the 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.

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, then *arg* is only executed once 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 says `Command failed.` any time a command run by the shell returns a non-zero value. See the "Advanced Concepts and Commands" chapter for a detailed explanation of signals and traps.

typeset

Syntax

```
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 *names* 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 the "Advanced Concepts and Commands" chapter for a detailed explanation of all the options.

ulimit

Syntax

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

This command sets limits, *n*, on certain resources a spawned process uses such as time, stack area, files sizes, etc. See the "Advanced Concepts and Commands" chapter 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

```
umask [ nnn ]
```

This command sets the user's file-creation mask to *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. See the *Introduction to UNIX System V* chapter for a detailed explanation.

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 and then removes it.

unset

Syntax

unset [-f] *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. 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 [ n ]
```

The `wait` command suspends the shell until the spawned process *n* terminates and then reports the processes termination status. If *n* is not specified, currently active processes are waited for. The shell resumes after all processes terminate or when it receives a signal (e.g., Break).

Example

```
$ cogitate gravity &
$ mailx
$ wait
$ rm gravity
```

In this example you run a very slow program (`cogitate`) 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.

whence

Syntax

```
whence [ -v ] name...
```

This unique Korn Shell command 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 the "Advanced Concepts and Commands" chapter for a detailed explanation.

while/until

Syntax

```
while list
do list2
done
```

This loop continues executing *command-line* and processing through the `list` as long as the item an *list* continues to evaluate true. Once an item 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`, and then increments and prints out the value until it equals 10 and you exit 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.

Notes

Index

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

Notes

Table of Contents

BC: An Arbitrary-Precision Desk-Calculator Language

BC is a language and a compiler for doing arbitrary-precision arithmetic on your HP-UX system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers. These routines are based on a dynamic storage allocator. Overflow does not occur until all available internal memory is exhausted.

The BC language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution. A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

BC and BS are similiar in capabilities, with BS being a more complete language supporting strings and I/O, but limited to "ordinary" double-precision floating point numbers. BC is limited in operating on numeric data, but operates on arbitrary percision numbers and arbitrary bases. The selection of one or the other is primarily based on the need for large value or high percision calculations. If these are not needed, BS may be the better choice. There is no significant advantage of one over the other for activities such as balancing your checkbook, unless you are the federal government.

Some of the uses of this compiler are to:

- perform computations on large integers
- perform computation accurate to many decimal places
- convert numbers from one base to another.

BC supports a scaling provision that permits the use of decimal point notation. Provision is also made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base te equal eight.

The maximum number of digits that can be handled depends on the amount of internal memory in the machine. Manipulation of numbers with many hundreds of digits is possible.

The syntax of BC is very similiar to the C programming language. Thus, users who are familiar with C can easily use BC.

Running BC

To use *bc*, type in:

```
bc
```

Your prompt is no longer displayed, and the BC calculator is ready for use.

To exit *bc* and return to your shell, type in:

CTRL-D

Your shell's prompt is displayed showing that you are no longer using *bc*.

Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The following operators may be used:

Operator	Meaning
+	addition
-	subtraction
/	division
%	modulo (remaindering)
^	exponentiiation

Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the **unary** minus sign). The expression

```
7+ -3
```

is interpreted to mean that -3 is to be added to 7 for a result of

```
4
```

More complex expressions with several operators and with parentheses are interpreted using the following mathematical precedence hierarchy:

Precedence	Operator
Highest	Parentheses (may be used to force any order of operations)
	Functions, user-defined and machine-resident
	Exponentiation (right to left)
	- (unary minus), + (unary plus)
	Multiplication and Division (left to right)
	Addition and Subtraction (left to right)
Lowest	All relational operators (=, <, >, ...)

The following expressions are equivalent:

`a^b^c` and `a^(b^c)`

`a*b*c` and `(a*b)*c`

`a/b*c` and `(a/b)*c`

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named x.

When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
x = sqrt(191)          execute assignment
x                      request current value of x
```

produce the printed result

```
13                     current value of x
```

Bases

There are special internal quantities, called 'ibase' and 'obase'. The contents of 'ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
ibase = 8
11
```

will produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. Beware, however, of trying to change the input base back to decimal by typing

```
ibase = 10
```

because the number `10` is interpreted as octal, so statement will have no effect.

If you use hexadecimal notation, the characters `A` through `F` are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values `10` through `15`, respectively. The statement

```
ibase = A
```

always restores decimal input base no matter what the current input base is.

Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of *obase*, initially set to 10, are used as the base for output numbers. The lines

```
obase = 16
1000
```

will produce the output line

```
3E8
```

which is to be interpreted as a **3-digit hexadecimal number**. Very large output bases are permitted.

An obase Gotcha

Certain difficulties arise whenever formatting systems are developed to accommodate arbitrary number systems. The problems are obvious to users having extensive mathemtical experience. For those who have no formal contact with the problem, ignorance may well be bliss. A brief synopsis is provided here. For more information, consult a suitable mathemtical text.

Consider the following instructions. `obase` is set to 2, then 10 and 12 are added. The result is 22 decimal, displayed in binary format.

```
obase = 2
12 + 10
10110
```

Suppose that you want to know what the current value of `obase` is. Type `obase` Return. The result is as follows:

```
obase
10
```

Printing the base always yields a variation of the digit **one** followed by the digit **zero**, formatted according to the current value of `obase`. Typing

```
obase - 1
```

produces information that is more useful.

Use of Bases

Large numbers can be output in groups of five digits by setting `obase` to 100000. Strange (i.e., 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with a slash (\). Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that `ibase` and `obase` have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

Scaling

A third special internal quantity called `scale` is used to determine the scale of calculated quantities. We refer to the number of digits after the decimal point of a number as its `scale`. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules:

- For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result.
- For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity `scale` and always less than 100.
- The scale of a quotient is the contents of the internal quantity `scale`. The scale of a remainder is the sum of the scales of the quotient and the divisor.
- The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.
- The scale of a square root is set to the maximum of the scale of the argument and the contents of `scale`.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case, truncation is used when digits are discarded. No rounding is ever performed.

The contents of `scale` must be no greater than 99 and no less than 0. It is initially set to 0.

The internal quantities `scale`, `ibase`, and `obase` can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of `scale` by one, and the line

```
scale
```

causes the current value of `scale` to be printed.

The value of `scale` retains its meaning as a number of decimal digits to be retained in internal computation even when `ibase` or obase are not equal to `10`. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

Functions

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
```

or

```
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one `auto` statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
 auto z
 z = x*y
 return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them. For example:

```
b()
```

If the function

```
a
```

above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of x to become 60.

Subscripted Variables (Arrays)

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript.

Only one-dimensional arrays are permitted.

The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use.

Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

Control Statements

The `if`, `while`, and `for` statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators:

Operator	Meaning
<	less than
>	greater than
<=	less than or equal
>=	greater than
==	equals
!=	not equal

Beware of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = will do an assignment, not a comparison.

The `if` statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The `while` statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The `for` statement begins by executing `expression1`. Then the relation is tested and, if true, the statements in the range of the `for` are executed. Then `expression2` is executed. The relation is tested, and so on. The typical use of the `for` statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

```
f(a)
```

prints a factorial if a is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n\-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1==1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c==d) return(c)
                d = c
        }
}
```

Some Details

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to `x` and also increments `i` before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manual for their exact workings.

```
x=y=z  is the same as        x=(y=z)
x += y       x = x+y
x -= y       x = -+y
x =* y       x = x*y
x =/ y       x = x/y
x =% y       x = x%y
x =^ y       x = x^y
```

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

Note

In some of these constructions, spaces are significant. There is a real difference between x=-y and x= -y. The first replaces x by x-y and the second by -y.

Three Important Things

- To exit a BC program, type `quit` or CTRL-D.
- There is a comment convention identical to that of C. Comments begin with `/*` and end with `*/`.
- There is a library of math functions which may be obtained by typing at command level

  ```
  bc -l
  ```

  This command will load the following library functions:
- sin (named `s`)
- cos (named `c`)
- arctangent (named `a`)
- natural logarithm (named `l`)
- exponential (named `e`) and
- Bessel functions of integer order (named `j(n,x)`).

The library sets the scale to 20.

If you type

```
bc file ...
```

`bc` will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

Notation

Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs, or comments. Newline characters or semicolons separate statements.

Comments

Comments are introduced by the characters `/*` and terminated by `*/`.

Identifiers

There are three kinds of identifiers:

- **Ordinary identifiers** include the characters a through z.
- **Array identifiers** are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers.
- **Function identifiers** are followed by parentheses, possibly enclosing arguments.

All three types consist of single lower-case letters. The three types of identifiers do not conflict; a program can have a variable named **x**, an array named **x[ ]** and a function named **x()** , all of which are separate and distinct.

Keywords

The following are reserved keywords:

- ibase
- obase
- while
- length
- if
- for
- auto
- sqrt
- break
- quit
- scale
- define
- return

Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits `A` through `F` are also recognized as digits with values `10` through `15`, respectively.

Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

Named Expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

Simple Identifiers

Simple identifiers are named expressions. They have an initial value of zero.

Array Elements

Array elements are named expressions. They have an initial value of zero.

Scale, Ibase and Obase

The internal registers **scale**, **ibase** and **obase** are all named expressions.

- scale is the number of digits after the decimal point to be retained in arithmetic operations. Scale has an initial value of zero and a maximum possible value of 99.
- ibase and obase are the input and output number radix respectively. Both ibase and obase have initial values of 10.

Function calls

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value.

As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

```
sqrt(expression)
```

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of scale, whichever is larger.

```
length(expression)
```

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

```
scale(expression)
```

The result is the scale of the expression. The scale of the result is zero.

Constants

Constants are primitive expressions.

Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

Unary operators

The unary operators bind right to left.

```
-(expression)
```

The result is the negative of the expression.

```
++(named-expression)
```

The named expression is incremented by one. The result is the value of the named expression after incrementing.

```
-(named-expression
```

The named expression is decremented by one. The result is the value of the named expression after decrementing.

```
(named-expression)++
```

The named expression is incremented by one. The result is the value of the named expression before incrementing.

```
(named-expression)--
```

The named expression is decremented by one. The result is the value of the named expression before decrementing.

Exponentiation Operator

The exponentiation operator binds right to left.

```
(expression)^(integer_expression)
```

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If **a** is the scale of the left expression and **b** is the absolute value of the right expression, then the scale of the result is:

$$\min(a \times b, \max(scale, a))$$

Multiplicative Operators

The operators *, /, % bind left to right.

*(expression) * (expression)*

The result is the product of the two expressions. If *a* and *b* are the scales of the two expressions, then the scale of the result is:

min(a+b,max(scale,a,b))

(expression) / (expression)

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

(expression) % (expression)

The % operator produces the remainder of the division of the two expressions. More precisely, a%b is a-a/b*b.

The scale of the result is the sum of the scale of the divisor and the value of scale.

Additive Operators

The additive operators bind left to right.

(expression) + (expression)

The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

(expression) - (expression)

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

Assignment Operators

The assignment operators bind right to left.

named_expression = *expression*

The above expression results in assigning the value of the expression on the right to the named expression on the left.

named_expression =+ *expression*

named_expression =− *expression*

named_expression =* *expression*

named_expression =/ *expression*

named_expression =% *expression*

named_expression =^ *expression*

The result of the above expressions is equivalent to **named expression = named expression OP expression**, where OP is the operator after the = sign.

Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

expression < *expression*

expression > *expression*

expression <= *expression*

expression >= *expression*

expression == *expression*

expression != *expression*

Storage classes

There are only two storage classes in BC, global and automatic (local).

- Only identifiers that are to be local to a function need be declared with the *auto* command.
- The arguments to a function are local to the function.
- All other identifiers are assumed to be global and available to all functions.
- All identifiers, global and local, have initial values of zero.

Identifiers declared as *auto* are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. *auto* arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in C. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

Quoted string statements

```
"any string"
```

This statement prints the string inside the quotes.

If Statements

`if`(relation)statement

The statement is executed if the relation is true.

While Statements

`while`(relation)statement

The statement is executed while the relation is true. The test occurs before each execution of the statement.

For Statements

```
for(expression; relation; expression)statement
```

The for statement is the same as

```
first-expression
while(relation) {
     statement
     last-expression
}
```

All three expressions must be present.

Break Statements

`break`

Break causes termination of a *for* or `while` statement.

Auto Statements

```
auto ordinary_identifier,array_identifier[]
```

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

Define statements

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

Return Statements

```
return

return(expression)
```

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

Quit

The quit statement stops execution of a BC program and returns control to HP-UX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an *if*, *for*, or *while* statement.

Index

r

s

t

v

Table of Contents

DC: An Interactive Desk Calculator

Dc is an interactive desk calculator program implemented in the HP-UX operating system to do arbitrary-precision integer arithmetic. It has provision for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated is limited only by available memory. HP-UX can handle number sizes varying from several hundred digits on the smallest systems to several thousand on the largest.

Dc is an arbitrary-precision arithmetic package implemented in the HP-UX operating system in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily, *dc* operates on decimal integers, but you can optionally specify an input base, output base, and the number of fractional digits to be maintained.

A language called BC(1) has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by *dc*. Some of the commands described here were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into *dc* are put on a push-down stack. *Dc* commands then take the top number or two off the stack, perform the desired operation, then push the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

Synoptic Description

This section describes the *dc* commands that are intended for use by people. The additional commands intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number	The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A thru F which are treated as digits with values 10 thru 15 respectively. Negative numbers should be preceded by an underscore (_). Numbers can contain decimal points.
+ - * % ^	The top two values on the stack are added (+), subtracted (-), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack, combined, then the result is pushed back on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.
s<x>	The top of the main stack is popped and stored into a register named <x>, where <x> may be any character. If *s* is uppercase (*L*), <x> is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.
l<*x*>	The value in register <*x*> is pushed onto the stack without being altered. If `l` is uppercase (`L`), register <*x*> is treated as a stack and its top value is popped onto the main stack. All registers start with empty value which is treated as a zero by the command *l* and is treated as an error by the command *L*
d	The top value on the stack is duplicated.
p	The top value on the stack is printed. The top value remains unchanged.
f	All values on the stack and in registers are printed.
x	Treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of *dc* commands.

[...] Puts the bracketed character string onto the top of the stack.

q Exits the program. If executing a string, the recursion level is popped by two. If *q* is capitalized (*Q*), the top value on the stack is popped and the string execution level is popped by that value.

<x >x =x !<x !>x !=x The top two elements of the stack are popped and compared. Register <x> is executed if they obey the stated relation. Exclamation point is negation.

v Replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.

! Interprets the rest of the line as an HP-UX command. Control returns to *dc* when the HP-UX command terminates.

c All values on the stack are popped; the stack becomes empty.

i The top value on the stack is popped and used as the number radix for further input. If *i* is uppercase (*I*), the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o The top value on the stack is popped and used as the number radix for further output. If *o* is capitalized (*O*), the value of the output base is pushed onto the stack.

k The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If *k* is uppercase (*K*), the value of the scale factor is pushed onto the stack.

z The value of the stack level is pushed onto the stack.

? A line of input is taken from the input source (usually the console) and executed.

Detailed Description

Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0 thru 99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always -1 and all other digits are in the range 0 thru 99. The digit preceding the high order -1 digit is never a 99. The representation of -157 is 43,98,-1. We shall call this the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high order digit. The value of this extra byte is called the **scale factor** of the number.

The Allocator

Dc uses a dynamic string storage allocator for all of its internal storage. All internal reading and writing of numbers is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and *dc* is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that resides next to it in memory and, if free, can be combined with it to make a string twice as long. This is an implementation of the "buddy system" of allocation described in [2].

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in *dc*. If at any time in the process of trying to allocate a string, the allocator runs out of headers, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

Internal Arithmetic

All arithmetic operations are done on integers. The operands (or operand) used in the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called <scale> plays a part in the results of most arithmetic operations. <scale> is the bound on the number of decimal places retained in arithmetic computations. <scale> can be set to the number on the top of the stack truncated to an integer with the *k* command. *K* can be used to push the value of <scale> on the stack. <scale> must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of <scale> on the computations.

Addition and Subtraction

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit-by-digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration 99,−1 by the digit −1. In any case, digits which are not in the range 0 thru 99 must be brought into that range, propagating any carries or borrows that result.

Multiplication

The scales are removed from the two operands and saved. The operands are both made positive, then multiplication is performed in a digit-by-digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register <scale> and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

Division

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity <scale>. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

Remainder

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

Square Root

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity <scale> and the scale of the operand.

The method used to compute the square root of (Y) is Newton's method of successive approximations by the rule:

$$Xsub(n+1) = \frac{1}{2} \times [Xsub(n) + (Y/Xsub(n)]$$

The initial guess is found by taking the integer square root of the top two digits.

Exponentiation

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

Input Conversion and Base

Numbers are converted to internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with an underscore (_). The hexadecimal digits A thru F correspond to the numbers 10 thru 15 regardless of input base. The *i* command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, using it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal- or hexadecimal-to-decimal conversions. The command *I* will push the value of the input base on the stack.

Output Commands

The command *p* causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command *f*. The *o* command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It will work correctly for any base. The command *O* pushes the value of the output base on the stack.

Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a backslash (\) indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-to-octal or decimal-to-hexadecimal conversions.

Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands *s* and *l*. The command *s*<x> pops the top of the stack and stores the result in register <x> where <x> can be any character. *l*<x> puts the contents of register <x> on the top of the stack. The *l* command has no effect on the contents of register <x>. The *s* command, however, is destructive.

Stack Commands

c clears the stack.

d pushes a duplicate of the number on the top of the stack on the stack.

z pushes the stack size on the stack.

X replaces the number on the top of the stack with its scale factor.

Z replaces the top of the stack with its length.

Subroutine Definitions and Calls

When the following commands are creatively used in conjunction with strings, the effect is roughly equivalent ot subroutine calls and operation.

Enclosing a string in brackets ([...]) pushes the ASCII string on the stack. The *q* command quits or, in executing a string, pops the recursion levels by two.

Internal Registers – Programming dc

The load and store commands together with *[]* to store strings, *x* to execute and the testing commands `<`, `>`, `=`, `!<`, `!>`, and `!=` can be used to program *dc*. The *x* command assumes the top of the stack is a string of *dc* commands, and executes the string. Testing commands remove the top two elements on the stack, test them, then, if the relation holds, execute the register following the elements tested. For example, to print the numbers 0-9, use the following commands:

```
[lip1+ si li10>a]sa
0si lax
```

Push-Down Registers and Arrays

These commands involve push-down registers and arrays, and were designed for use by a compiler, not by people. In addition to the stack that commands work on, *dc* can be thought of as having individual stacks for each register. These registers are operated on by the commands *S* and *L*. *S*<x> pushes the top value of the main stack onto the stack for register <x>. *L*<x> pops the stack for register <x> and puts the result on the main stack. The commands *s* and *l* also work on registers, but not as push-down stacks. *l* doesn't affect the top of the register stack; *s* destroys what was there before.

The commands that work on arrays are colon (:) and semicolon (;). (:<x>) pops the stack and uses the value obtained as an index into the array <x>. The next element on the stack is stored at the indexed location in <x>. The index value must be greater than or equal to 0 and less than 2048. (;<x>) loads the main stack from the array <x>. The value on the top of the stack is popped and used as the index into the array <x>. The indexed value is then loaded from the array onto the stack.

Miscellaneous Commands

If an exclamation point (!) appears in a line, *dc* treats the rest of the line as an HP-UX command and passes it to HP-UX for execution.

Another compiler command is *Q*. This command uses the top of the stack as the number of levels of recursion to skip.

Design Choices

The real reason for using dynamic storage allocation was that a general purpose program is useful for a variety of other tasks. The allocator has some value for input and for compiling (i.e., the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

The choice of 100 as a base for internal arithmetic seems to have no compelling advantage, but with a hardware limit of 127 and only 5% additional space required, debugging was made a great deal easier and decimal output was made much faster.

Stack-type arithmetic design permitted all *dc* commands from addition to subroutine execution to be implemented in essentially the same way, resulting in a considerable degree of logical separation of the final program into modules with very little communication required between modules.

Eliminating interaction between the scale and the bases provided an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. For example, if the value of <scale>. were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results.

The scheme implemented has the advantage that the value of the input and output bases are only used for input and output, respectively, and are ignored in all other operations. The scale value is not used during program operation, serving only to reasonably limit the number of decimal places resulting from arithmetic operations.

The design rationale for scaling arithmetic results was that no significant digits should be discarded if there was any indication that the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for <scale>. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a <scale> to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

Table of Contents

Mailx
Mail Handler

Introduction

Mailx provides a simple and friendly environment for sending and receiving mail. It divides incoming mail into its constituent messages and allows the user to deal with them in any order. In addition, it provides a set of *ed*-like commands for manipulating messages and sending mail. *Mailx* offers the user simple editing capabilities to ease the composition of outgoing messages, as well as providing the ability to define and send to names which address groups of users. Finally, *mailx* is able to send and receive messages across HP-UX-supported networks such as UUCP.

This document describes how to use the *mailx* program to send and receive messages. You do not need to be be familiar with other message handling systems, but you should be familiar with the HP-UX shell, the text editor, and some of the common HP-UX commands. The *HP-UX Reference* and *HP-UX Concepts and Tutorials* manuals covering text editors and *csh* can be consulted for more information on these topics.

Here is how messages are handled: the mail system accepts incoming *messages* for you from other people and collects them in a file, called your *system mailbox.* When you login, the system notifies you if there are any messages waiting in your system mailbox. If you are a *csh* user, you will be notified when new mail arrives if you inform the shell of the location of your mailbox. Your system mailbox is located in the directory /usr/mail in a file with your login name. For example, if your login name is *sam*, you can make *csh* notify you of new mail by including the following line in your *.cshrc* file:

```
set mail=/usr/mail/sam
```

When you read your mail using *mailx*, it reads your system mailbox and separates that file into the individual messages that have been sent to you. You can then read, reply to, delete, or save these messages. Each message is marked with its author and the date sent.

Common Usage

The *mailx* command has two distinct usages, depending on whether you want to send or receive mail. Sending mail is simple: to send a message to a user whose login name is, say, *root*, use the shell command:

```
% mailx root
```

then type your message. When you reach the end of the message, type an EOT ([CTRL]-[D]) at the beginning of a line, causing *mailx* to echo an EOT and return you to the Shell. The next time the person you sent mail to next logs in, he will receive the message:

```
You have mail.
```

indicating the availability of your message.

If, while you are composing the message you decide that you do not wish to send it after all, you can abort the letter by pressing [DEL] (or **RUBOUT**). Pressing **RUBOUT** once causes *mailx* to print (or display):

```
(Interrupt -- one more to kill letter)
```

Pressing [DEL] a second time causes *mailx* to save your partial letter on the file *dead.letter* in your home directory and abort the letter. Once you have sent mail to someone, there is no way to undo the act, so be careful.

The message your recipient reads will consist of the message you typed, preceded by one or more lines telling who sent the message (your login name), the date and time it was sent, and other information about the letter.

If you want to send the same message to several other people, you can list their login names on the command line. Thus,

```
% Mail sam bob john
Tuition fees are due next Friday.  Don't forget!!
[CTRL]-[D]
EOT
%
```

sends the reminder to sam, bob, and john.

If, when you log in, you see the message,

```
You have mail.
```

you can read the mail by typing:

```
% mailx
```

Mailx responds by typing (displaying) its version number and date, then listing the messages you have waiting. It then sends a prompt and awaits your next command. Messages are assigned numbers starting with `1--`, and each message is accessed by using the assigned message number.

Mailx keeps track of which messages are **new** (have been sent since you last read your mail) and **read** (have been read by you). New messages have an `N` next to them in the header listing, while old, but unread, messages have a `U` next to them. *mailx* keeps track of new/old and read/unread messages by putting a header field called **Status** into your messages.

To look at a specific message, use the *type* command (abbreviated *t*). For example, if you had the following messages:

```
N 1 root      Wed Sep 21 09:21  "Tuition fees"
N 2 sam       Tue Sep 20 22:55
```

you could examine the first message by giving the command:

```
type 1
```

causing *mailx* to respond with, for example:

```
Message  1:
>From root  Wed Sep 21 09:21:45 1978
Subject: Tuition fees
Status: R
Tuition fees are due next Wednesday.  Don't forget!!
```

Many *mailx* commands, such as *type*, that operate on messages take a message number as an argument. For these commands, there is a notion of a current message. When you enter the *mailx* program, the current message is initially the first one. Thus, you can often omit the message number and use, for example,

```
t
```

to type (display) the current message. As a further shorthand, you can type a message by simply giving its message number. Hence,

```
1
```

would display the first message.

Frequently, it is useful to read the messages in your mailbox in order, one after another. You can read the next message in *mailx* by simply typing a newline (press Return). As a special case, you can type a newline (Return) as your first command to *mailx* to type (display) the first message.

After the message has been typed or displayed, if you wish to send an immediate reply, you can do so with the *reply* command. *Reply*, like *type*, takes a message number as its argument. *Mailx* then begins a message addressed to the user who sent you the message. You can then type in your letter of reply, followed by a CTRL-D (EOT) at the beginning of a line, as before. *Mailx* then sends EOT followed by the ampersand prompt to indicate it is ready for another command. In our example, if, after reading the first message, you wished to reply to it, you might give the command:

```
reply
```

Mailx responds with:

```
To: root
Subject: Re: Tuition fees
```

and waits for you to enter your letter. You are now in the message-collection mode described at the beginning of this section so *mailx* gathers your message up to an EOT (CTRL-D).

Note that *mailx* copies the subject header from the original message because correspondence about a particular matter tends to retain the same subject heading, making it easy to recognize. If there are other header fields in the message, that information is also used. For example, if the letter had a `To:` header listing several recipients, *mailx* would arrange to send your reply to the each of them. Similarly, if the original message contained a `Cc:` (carbon copies to) field, *mailx* would send your reply to **those** users. However, *mailx* does not send the message to you, even if you appear in the `To:` or `Cc:` field, unless you explicitly ask to be included. See Tilde Escapes and Special Recipients sections of this article for more details.

After typing in your letter, the dialog with *mailx* might look like the following:

```
reply
To: root
Subject: Re: Tuition fees
Thanks for the reminder
EOT
&
```

The *reply* command is especially useful for sustaining extended conversations over the message system, with other "listening" users receiving copies of the conversation. The *reply* command can be abbreviated to *r*.

Sometimes you will receive a message that has been sent to several people and wish to reply **only** to the person who sent it. *Reply* with an uppercase *R* replies to a message, but sends a copy to the sender only.

If, while reading your mail, you wish to send a message to someone, but not as a reply to one of your messages, you can send the message directly using the *mail* command, which takes as arguments the names of the recipients you want to send to. For example, to send a message to "frank",

```
mail frank
This is to confirm our meeting next Friday at 4.
EOT
&
```

The *mail* command can be abbreviated to *m*.

Normally, each message you receive is saved in the file *mbox* in your login directory at the time you leave *mailx*. Often, however, you will not want to save a particular message you have received because it is only of passing interest. To avoid saving a message in *mbox*, delete it using the *delete* command. In our example,

```
delete 1
```

prevents *mailx* from saving message 1 (from root) in *mbox*. In addition to not saving deleted messages, *mailx* does not let you type (display) them either. The effect is to make the message disappear altogether, along with its number. The *delete* command can be abbreviated to *d*.

Many features of *mail* can be tailored to your liking with the *set* command. *Set* has two forms, depending on whether you are setting a **binary** or **valued** option. Binary options are either on or off. For example, the *ask* option informs *mailx* that each time you send a message, you want it to prompt you for a subject header, to be included in the message. To set the *ask* option, type:

```
set ask
```

Another useful *mailx* option is *hold*. Unless told otherwise, *mailx* moves the messages from your system mailbox to the file *mbox* in your home directory when you leave *mailx*. If you want *mailx* to keep your letters in the system mailbox instead, set the *hold* option:

```
set hold
```

Valued options tailor *mailx* to match your needs. For example, the *shell* option tells *mailx* which shell you like to use. For example to select the shell */bin/csh*, type:

```
set SHELL=/bin/csh
```

Note that no spaces are allowed in `SHELL=/bin/csh`. A complete list of the *mailx* options appears at the end of this article.

Another important valued option is *crt*. If you use a fast video terminal to print long messages, they fly by too quickly for you to read them. You can use the *crt* option to force *mailx* to send messages longer than a given number of lines through the paging program *more*. For most CRT displays, use the following command:

```
set crt=24
```

to paginate messages that will not fit on a 25-line screen. *More* prints a screenful of information, then displays `--MORE--` on the remaining line. Type a space to see the next screenful.

Mailx also provides an *alias* option where the specified alias is a name that stands stands for one or more real user names. Mail sent to an alias is then sent to the list of real users associated with the alias. For example, an alias can be defined for the members of a project, so that you can send mail to the whole project by sending mail to just a single name. The *alias* command in *mailx* defines an alias. Suppose that the users in a project are named Sam, Sally, Steve, and Susan. To define an alias called *project* for them, use:

```
alias project sam sally steve susan
```

Alias can also be used to provide a convenient name for someone whose user name is inconvenient. For example, if a user named "Bob Anderson" had the login name "anderson", you might want to use:

```
alias bob anderson
```

so that you could send mail to the shorter name, "bob".

While *alias* and *set* commands enable you to customize *mailx*, they must be retyped each time you enter *mailx*. To make them more convenient to use, *mailx* always looks for two files when it is invoked. It first reads a system-wide file */usr/lib/mailx/mailx.rc*, then a user-specific file, *.mailrc* which is found in the user's home directory. The system-wide file is maintained by the system administrator and contains *set* commands that are applicable to all system users. The *.mailrc* file is usually set up by each user to seelect options to fit his preference and to define individual aliases. Here is an example *.mailrc* file:

```
set ask nosave SHELL=/bin/csh
```

As you can see, it is possible to set many options in the same *set* command. The *nosave* option is described in the Additional Features section of this article.

Mail aliasing is implemented at the system-wide level by the mail delivery system *sendmail.* These aliases are stored in the file */usr/lib/aliases* and are accessible to all system users. The lines in */usr/lib/aliases* are of the form:

```
alias: <alias>, <name 1>, <name 2>, <name 3>, ...
```

where <alias> is the mailing list name and the <names> are the members of the list. Long lists can be continued onto the next line by starting the next line with a space or tab. Remember that you must execute the shell command *newaliases* after editing */usr/lib/aliases* because the delivery system uses an indexed file created by *newaliases.*

Note

Mailx supports *alias* **only** for mail originators on the system as defined here. System-wide aliasing requires the *sendmail* facility which is not presently available on HP-UX.

We have seen that *mailx* can be invoked with command line arguments (people to send the message to), or with no arguments (to read mail). Specifying the *−f* flag on the command line causes *mailx* to read messages from a file other than your system mailbox. For example, if you have a collection of messages in the file *letters* you can use *mailx* to read them with:

```
% mailx -f letters
```

You can use all the *mailx* commands described in this article to examine, modify, or delete messages from your *letters* file which will be rewritten when you leave *mailx* with the *quit* command described below.

Since mail that you read is saved in the file *mbox* in your home directory by default, you can read the file from your home directory by typing:

```
% mailx -f
```

Normally, messages that you examine using the *type* command are saved in *mbox* in your home directory if you leave *mailx* with the *quit* command described below. If you wish to retain a message in your system mailbox you can use the *preserve* command to tell *mailx* to leave it there. *Preserve* accepts a list of message numbers, just like *type* and can be abbreviated to *pre.*

Messages in your system mailbox that you do not examine are normally retained in your system mailbox automatically. To save such a message saved in *mbox* without reading it, use the *mbox* command. For example,

```
mbox 2
```

in our example would cause the second message (from sam) to be saved in *mbox* when the *quit* command is executed. *Mbox* can also be used to direct messages to your *mbox* file if you have set the *hold* option described previously. *Mbox* can be abbreviated to *mb.*

When you have perused all messages of interest, use the *quit* command to leave *mailx.* Any messages you have typed but not deleted are saved in the file *mbox* in your login directory. Deleted messages are discarded irretrievably, and messages left untouched are preserved in your system mailbox so that you will see them the next time you type:

```
% mailx
```

Quit can be abbreviated to *q.*

If, for some reason, you want to leave *mailx* quickly without altering either your system mailbox or *mbox*, type *x* (short for *exit*), which immediately returns you to the Shell without changing anything.

If, instead, you want to execute a Shell command without leaving *mailx*, type the command preceded by an exclamation point, just as in the text editor. For instance:

```
!date
```

prints the current date without leaving *mail.*

The *help* command prints out a brief summary of the *mailx* commands, using only single-character command abbreviations.

Maintaining Folders

This section describes a simple *mailx* facility for maintaining groups of messages together in folders.

To use the folder facility, you must tell *mailx* where you want to keep your folders. Each folder of messages will be a single file. For convenience, all of your folders are kept in a single directory of your choosing. To tell *mailx* where your folder directory is, put a line of the form:

```
set folder=letters
```

in your *.mailrc* file. If, as in the example above, your folder directory does not begin with a "/", *mailx* assumes that your folder directory is to be found starting from your home directory. Thus, if your home directory is */usr/person* the above example told *mailx* to find your folder directory in */usr/person/letters.*

Anywhere a file name is expected, you can use a folder name, preceded by a "+" with no intervening spaces. For example, to put a message into a folder with the *save* command, use:

```
save +classwork
```

to save the current message in the *classwork* folder. If the *classwork* folder does not yet exist, it will be created. Note that messages saved by use of the *save* command are automatically removed from your system mailbox.

In order to make a copy of a message in a folder without causing that message to be removed from your system mailbox, use the *copy* command, which is identical in all other respects to the *save* command. For example,

```
copy +classwork
```

copies the current message into the *classwork* folder and leaves a copy in your system mailbox.

The *folder* command can be used to direct *mailx* to the contents of a different folder. For example,

```
folder +classwork
```

directs *mail* to read the contents of the *classwork* folder. All of the commands that you can use on your system mailbox are also applicable to folders, including *type*, *delete*, and *reply*. To inquire which folder you are currently editing, type:

```
folder
```

To list your current set of folders, use the *folders* command.

To start reading one of your folders, use the $-f$ option described in earlier in this section. For example,

```
% mailx -f +classwork
```

causes *mailx* to read your *classwork* folder without looking at your system mailbox.

More About Sending Mail

Tilde Escapes

While typing in a message to be sent to others, it is often useful to be able to invoke the text editor on the partial message, print the message, execute a shell command, or do some other auxiliary function. *mailx* provides these capabilities through **"tilde escapes"** which consist of a tilde (~) at the beginning of a line, followed by a single character indicating the function to be performed. For example, to print the text of the message so far, use:

```
~p
```

which will print a line of dashes, the recipients of your message, and the text of the message so far. Since *mailx* requires two consecutive DEL s (or **RUBOUT**s) to abort a letter, you can use a single DEL to abort the output of *~p* or any other ~ escape without killing your letter.

If you are dissatisfied with the message as it stands, you can invoke the text editor on it by using the escape:

```
~e
```

which causes the message to be copied into a temporary file and an instance of the editor to be spawned. After modifying the message to your satisfaction, write it out and quit the editor. *mailx* then responds by typing (or displaying):

```
(continue)
```

after which you can continue typing text to be appended to your message, or you can type CTRL-D to end the message. A standard text editor is provided by *mailx*.

To override the default editor, set the valued option `EDITOR` to specify a different shell file such as:

```
set EDITOR=/usr/bin/ex
```

or

```
set EDITOR=/usr/bin/vi
```

To use the screen or *visual* editor as an alternative to the standard text editor on your current message, you can use the escape,

```
~v
```

~v works like *~e*, except that the screen editor is invoked instead. A default screen editor is defined by *mailx*. To select a different visual (screen) editor, set the valued option **VISUAL** to the path name of a different editor.

It is sometimes useful to be able to include the contents of some file in your message. The escape,

```
~r filename
```

is provided for this purpose, and causes the named file to be appended to your current message. *Mailx* complains if the file doesn't exist or can't be read. If the read is successful, the number of lines and characters appended to your message is printed, after which you can continue appending text. The filename may contain shell metacharacters like * and ? which are expanded according to the conventions of your shell.

As a special case of ~r, the escape,

```
~d
```

reads in the file *dead.letter* from your home directory. This is often useful because *mailx* copies the text of your message there when you abort a message with [DEL].

To save the current text of your message on a file, use the escape:

```
~w filename
```

Mailx then prints out the number of lines and characters written to the file, after which you can continue appending text to your message. Shell metacharacters can be used in the filename, as in *~r* and are expanded with the conventions of your shell.

If you are sending mail from within *mailx*'s command mode, you can read a message sent to you into the message you are constructing with the escape:

```
~m 4
```

which reads message 4 into the current message, shifted right by one tab stop. You can name any non-deleted message, or list of messages. To forward messages without shifting by a tab stop, use *~f* (this is the usual way to forward a message).

If, in the process of composing a message, you decide to add more people to the list of message recipients, you can do so with the escape:

```
~t <name1> <name2> ...
```

You can name as few or many additional recipients as you wish. Note that the users originally on the recipient list will still receive the message because you cannot remove someone from the recipient list with *~t*.

To associate a subject with your message, use the escape:

```
~s <arbitrary string of text>
```

which replaces any previous subject with <arbitrary string of text>. The subject, if given, is sent near the top of the message prefixed with `Subject:`. To see what the message will look like, use *~p*.

For political reasons, one occasionally prefers to list certain people as recipients of carbon copies of a message rather than direct recipients. The escape:

```
~c <name1> <name2>...
```

adds the named people to the **Cc:** list as when using *~t*. Again, you can execute *~p* to see what the message will look like.

The recipients of the message together constitute the `To:` field, the subject the `Subject:` field, and the carbon copies the `Cc:` field. If you wish to edit these in ways impossible with the ~t, ~s, and ~c escapes, you can use the escape:

```
~h
```

which prints `To:` followed by the current list of recipients and leaves the cursor (or printhead) at the end of the line. If you type ordinary characters, they are appended to the end of the current list of recipients. You can also use your erase character to erase back into the list of recipients, or your kill character to erase them altogether. Thus, for example, if your erase and kill characters are the standard # and @ symbols,

```
~h
To: root kurt####bill
```

changes the initial recipients `root kurt` to `root bill`. When you type a newline (Return or Enter), *mailx* advances to the `Subject:` field, where the same rules apply. Another newline brings you to the `Cc:` field, which can be edited in the same fashion. Another newline leaves you appending text to the end of your message. You can use ~*p* to print the current text of the header fields and the body of the message.

To temporarily escape to the shell, use the sequence:

```
~!<command>
```

is used, which executes <*command*> and returns you to mailing mode without altering the text of your message. If you wish, instead, to filter the body of your message through a shell command, use:

```
~|<command>
```

which pipes your message through the command and uses the output as the new text of your message. If the command produces no output, *mailx* assumes that something is amiss and retains the old version of your message. A frequently-used filter is the command *fmt*, designed to format outgoing mail.

To effect a temporary escape to *mailx* command mode instead, use:

```
~:mailx <command>
```

This is especially useful for retyping the message you are replying to, using, for example:

```
~:t
```

It is also useful for setting options and modifying aliases.

If you wish (for some reason) to send a message that contains a line beginning with a tilde, a double tilde must be used. For example,

```
~~This line begins with a tilde.
```

sends the line:

```
~This line begins with a tilde.
```

Finally, the escape,

```
~?
```

prints out a brief summary of the available tilde escapes.

On some terminals (particularly those with no lower case) tildes are difficult to type. *Mailx* enables you to change the escape character by using the **escape option**. For example, to use a right bracket, type:

```
set escape=]
```

As with the tilde, if you need to send a line starting with the escape character, type a pair of adjacent escape characters as when using tilde. Redefining the escape character removes the special significance of ~.

Network Access

This section describes how to send mail to people on other machines. Recall that sending to a plain login name sends mail to that person on your machine only. If your recipient logs in on a different machine connected to yours by UUCP, You must know the list of machines through which your message must travel to arrive at his site. If his machine has a continuous (modem or direct-connect) datacomm link to yours, you can send mail to him using the syntax:

```
host!name
```

where *host* is the name of his machine and *name* is his login name. If your message must go through an intermediate machine first, you must use the syntax:

```
intermediate!host!name
```

and so on. It is actually a feature of UUCP that the map of all the systems in the network is not known anywhere (except where people decide to write it down for convenience). Talk to your system administrator about the machines connected to your site.

If you need to use an HP-UX-supported network to access recipients on other networks, contact the System Administrator of the system providing the link between networks for procedures.

When you use the *reply* command to respond to a letter, there is a problem of figuring out the names of the users in the `To:` and `Cc:` lists **relative to the current machine**. If the original letter was sent to you by someone on the local machine, then this problem does not exist, but if the message came from a remote machine, the problem must be dealt with. *mailx* uses a heuristic to build the correct name for each user relative to the local machine. So, when you *reply* to remote mail, the names in the `To:` and `Cc:` lists may change somewhat.

Special Recipients

As described previously, you can send mail to either user names or *alias* names. It is also possible to send messages directly to files or to programs, using special conventions. If a recipient name has a "/" in it or begins with a "+", it is assumed to be the path name of a file into which to send the message. If the file already exists, the message is appended to the end of the file. If you want to name a file in your current directory (i.e., one for which a "/" would not usually be needed) you can precede the name with "./". For example, to send mail to the file *memo* in the current directory, use the command:

```
% mailx ./memo
```

If the name begins with a "+", it is expanded into the full path name of the folder name in your folder directory. This ability to send mail to files can be used for a variety of purposes, such as maintaining a journal and keeping a record of mail sent to a certain group of users. The second example can be done automatically by including the full pathname of the record file in the *alias* command for the group. Using our previous *alias* example, you could use the command:

```
alias project sam sally steve susan /usr/project/mail_record
```

to save all mail sent to `project` would be saved on the file */usr/project/mail_record* as well as being sent to the members of the project. This file can be examined using `mailx -f`.

Sometimes it is useful to send mail directly to a program (such as a project billboard program). To use *mailx* to send messages to the program, use a vertical bar followed by the program file name: `|billboard`, for example.

Mailx treats recipient names that begin with a "|" as a program to send the mail to. An `alias` can be set up to reference a "|"-prefaced name if desired. **Caveats:** the *mailx* shell treats "|" specially, so it must be quoted on the command line. Also, the `"| program"` must be presented as a single argument to *mailx*. The safest course is to surround the entire name with double quotes. This also applies to usage in the *alias* command. For example, to alias *rmsgs* to *rmsgs −s* type:

```
alias rmsgs "| rmsgs -s"
```

Additional Features

This section describes some additional commands of use for reading your mail, setting options, and handling lists of messages.

Message Lists

Several *mailx* commands accept a list of messages as an argument. Along with *type* and *delete*, described earlier, the *from* command prints the message headers associated with the message list passed to it. *From* is particularly useful in conjunction with some of the message list features described below.

A <message list> consists of a list of message numbers, ranges, and names, separated by spaces or tabs. Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters <up arrow>, <.>, or <$> to specify the first relevant, current, or last relevant message, respectively. For most commands, **relevant** here means **not deleted**, (or **deleted** for the *undelete* command).

A range of messages consists of two message numbers (of the form described in the previous paragraph) separated by a hyphen (dash). Thus, to print the first four messages, use:

```
type 1-4
```

and to print all the messages from the current message to the last message, use:

```
type .-$
```

A <name> is a user name. The user names given in the message list are collected together and each message selected by other means is checked to make sure it was sent by one of the named users. If the message consists entirely of user names, then every message sent by one those users that is **relevant** (in the sense described earlier) is selected. Thus, to print every message sent to you by *root*, use the command:

```
type root
```

As a shorthand notation, you can specify * to get every **relevant** (same sense) message. Thus,

```
type *
```

prints all undeleted messages,

```
delete *
```

deletes all undeleted messages, and

```
undelete *
```

undeletes all deleted messages.

You can search for the presence of a word in subject lines with /. For example, to print the headers of all messages that contain the word `Pascal`, use the command:

```
from /pascal
```

Note that subject searching ignores upper/lowercase differences.

Noninteractive Mail

Mailx can be used to transmit files to recipients noninteractively, and as a background process, if desired. Only the mailing address of the recipient and the file(s) to be transmitted are needed as follows:

`mailx` *address* < *filename(s)*

Address is the mailing address or alias that identifies the recipient(s) to which the file is being sent. *Filename* is the file(s) that are to be used as input to the mail handler. As shown, the command operates as a foreground process, and the user terminal hangs until the process is complete. To operate in the background, add an ampersand (&) at the end of the *mailx* command line. This frees your terminal for other activities while the file transfer is processed.

List of Commands

This section describes all the *mailx* commands available when receiving mail.

! Used to preface a command to be executed by the shell.

– The – command goes to the previous message and prints it. The – command may be given a decimal number <n> as an argument, in which case the <n>th previous message is gone to and printed.

Print (abbr: *P*) Like *print*, but also print out ignored header fields. See also *print* and *ignore*.

Reply (abbr: *R*) Note the capital R in the name. Frame a reply to a one or more messages. The reply (or replies if you are using this on multiple messages) will be sent ONLY to the person who sent you the message (respectively, the set of people who sent the messages you are replying to). You can add people using the ~*t* and ~*c* tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with "Re:" unless it already began thus. If the original message included a "reply-to" header field, the reply will go **only** to the recipient named by "reply-to." Type in your message using the same conventions available through the *mail* command.

The *Reply* command is especially useful for replying to messages that were sent to distribution groups when you really just want to send a message to the originator. Use it often.

Type (abbr: *T*) Identical to the *Print* command.

alias (*abbr: a*) Define a name to stand for a set of other names. This is used when you want to send messages to a certain group of people and want to avoid retyping their names. For example:

```
alias project john sue willie kathryn
```

creates an alias *project* which expands to the four people John, Sue, Willie, and Kathryn.

If no argument is given, all aliases are printed; one argument prints only the alias specified.

alternates (abbr: *alt*) If you have accounts on several machines, you may find it convenient to use the */usr/lib/aliases* on all the machines except one to direct your mail to a single account.

The *alternates* command informs *mailx* that each of these other addresses is really **you**, so that when you *reply* with messages to one of these alternate names, *mailx* will not bother to send a copy of the message to this other address (which would simply be directed back to you by the alias mechanism).

If *alternates* is given no argument, it lists the current set of alternate names. *Alternates* is usually used in the *.mailrc* file.

chdir (abbr:*cd*) The *chdir* command allows you to change your current directory. *Chdir* takes a single argument, which is taken to be the pathname of the directory to change to. If no argument is given, *chdir* changes to your home directory.

copy (abbr: *co*) The *copy* command is identical to *save* except that copied messages are not marked for deletion when you quit.

delete (abbr: *d*) Deletes a list of messages. Deleted messages can be reclaimed with the *undelete* command.

dp or *dt* *dpt* or *dt* deletes the current message and prints the next message. It is useful for quickly reading and disposing of mail. Displays `At EOF` if no messages remaining.

edit (abbr: *e*) The *edit* command provides editing capabilities for individual messages. It takes a list of messages as described under the *type* command and processes each by writing the message into a file *message*<x> (where <x> is the message number being edited) then executing the text editor on the file. When you have edited the message to your satisfaction, write the message out and quit the editor. *Mailx* then reads the message back and removes the file. *Edit can be abbreviated to e.*

else Marks the end of the **then** part of an *if* statement and the beginning of the part to take effect if the condition of the *if* statement is false.

endif Marks the end of an *if* statement.

exit (abbr: *ex* or *x*) Leaves *mailx* without updating the system mailbox or the file your were reading. Thus, if you accidentally delete several messages, you can use *exit* to avoid scrambling your mailbox.

file (abbr: *fi*) Identical to *folder.*

folders List the names of the folders in your folder directory.

folder (abbr: *fo*) The *folder* command switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. If you give it an argument, it will write out changes (such as deletions) you have made in the current file and read the new file. Some special conventions are recognized for the file/folder name:

Name	Meaning
#	Previous file read
%	Your system mailbox
%<name>	*Name*'s **system mailbox**
&	Your *~/mbox* **file**
+<folder>	A file in your folder directory

from (abbr: *f*) The *from* command takes a list of messages and prints out the header lines for each one; hence

```
from joe
```

is the easy way to display all the message headers from **joe**.

headers (abbr: *h*) When you start up *mailx* to read your mail, it lists the headers from each message in your mailbox. These headers tell you who each message is from, when they were sent, how many lines and characters each message is, and the `Subject:` header field of each message, if present. In addition, *mailx* tags the message header of each message that has been the object of the *preserve* command with a `P`.

Messages that have been *saved* or *written* are flagged with a `*`. *Deleted* messages are not printed at all. To reprint the current list of message headers, use the *headers* command.

Headers (and thus the initial header listing) only lists the first so many message headers. The number of headers listed depends on the speed of your terminal. This can be overridden by specifying the number of headers you want with the `window` option. *mailx* maintains a notion of the current `window` into your messages for the purposes of printing headers.

Use the *z* command to move forward or back one window. You can move *mailx*'s notion of the current window directly to a particular message by using, for example,

```
headers 40
```

to move *mailx*'s attention to the messages around message 40. The *headers* command can be abbreviated to *h*.

help Print a brief (and usually out-of-date) help message about the commands in *mailx*. Refer to this manual instead.

hold (abbr: *ho*; also *preserve*) Arrange to hold a list of messages in the system mailbox, instead of moving them to the file *mbox* in your home directory. If you set the binary option *hold*, this will happen by default.

if The *if* command is used to conditionally execute commands in your *.mailrc* file, depending on whether you are sending or receiving mail. Here is an example of the general structure used:

```
if receive
                    commands...
endif
```

An *else* form is also available:

```
if send
                    commands...
else
                    commands...
endif
```

Note that the only allowed conditions are `receive` and `send`.

ignore Add the list of header fields named to the `ignore list`. Header fields in the ignore list are not printed on your terminal when you print a message. This allows you to suppress printing of certain machine-generated header fields, such as `Via` which are not usually of interest. The *Type* and *Print* commands can be used to print a message in its entirety, including ignored fields. If *ignore* is executed with no arguments, it lists the current set of ignored fields.

list List the valid *mailx* commands.

mailx (abbr: *m*) Send mail to one or more people. If you have the *ask* option set, *mailx* will prompt you for a subject to your message. Type in your message, using tilde escapes described earlier to edit, print, or modify your message. To signal your satisfaction with the message and send it, type control-d at the beginning of a line, or a "." alone on a line if you set the option *dot*.

To abort the message, type two interrupt characters ([DEL] or **RUBOUT** by default) in a row or use the ~*q* escape.

mbox Indicate that a list of messages be sent to *mbox* in your home directory when you quit. This is the default action for messages if you do **not** have the **hold** option set.

next (abbr: *n*; also + or Return) The *next* command goes to the next message and types it. If given a message list, *next* goes to the first such message and types it. Thus,

```
next root
```

goes to the next message sent by *root* and types it. *Next* can be abbreviated to simply a newline, which means that one can go to and type a message by simply giving its message number or one of the magic characters: <up arrow>, < . >, or < $ >. Thus,

. (period)

prints the current message and

```
4
```

prints message 4, as described previously.

preserve Same as *hold.* Preserves listed messages in your system mailbox when you quit.

print (abbr: *p*; similar to *type*) Prints all messages specified by the message list, but does not print *ignored* header fields.

quit (abbr: *q*) Leave *mailx* and update the file, folder, or system mailbox your were reading. Messages that you have examined are marked as `read` and messages that existed when you started are marked as `old`. If you were editing your system mailbox and the binary option `hold` is **set**, all messages which have not been deleted, saved, or mboxed will be retained in your system mailbox. If you were editing your system mailbox and `hold` is **not set**, all messages which have not been deleted, saved, or preserved are moved to the file *mbox* in your home directory.

reply (abbr: *r*) Frame a reply to the originator of a single message and send it to the originator plus all the people who received the original message, except you. You can add people using the ~*t* and ~*c* tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with `Re:` unless it already began thus.

If the original message included a `reply-to` header field, the reply will go **only** to the recipient named by `reply-to`. Type in your message using the same conventions as with the *mail* command.

respond Same as *reply.*

save (abbr: *s*) It is often useful to be able to save messages on related topics in a file. The *save* command gives you ability to do this. The *save* command takes as its argument a list of message numbers, followed by the name of the file on which to save the messages. The messages are appended to the named file, thus allowing one to keep several messages in the file, stored in the order they were put there. *Save* can be abbreviated *s*. Here is how *save* can be used relative to our running example:

```
s 1 2 tuitionmail
```

Saved messages are not automatically saved in *mbox* at quit time, nor are they selected by the *next* command described above, unless explicitly specified.

If the filename is preceded by a vertical bar (|), *mailx* treats that file as a pipe. For example,

```
s 2 | lp
```

submits message 2 to the printer.

set (abbr: *se*) Set an option or give an option a value. Used to customize *mailx*. Options are listed near the end of this article. Binary options are *on* or *off*; valued options require an accompanying <value> parameter. To set a binary option, type:

```
set <option>
```

For valued options:

```
set <option>=<value>
```

Several options can be specified in a single *set* command. Use *unset* to disable options.

shell (abbr: *sh*) The *shell* command enables you to escape to the shell. *Shell* invokes an interactive shell and allows you to type commands to it. When you leave the shell, return is to *mailx*. The shell used is a default assumed by *mailx* which can be overridden by setting the valued option *SHELL*. For example,

```
set SHELL=/bin/csh
```

source (abbr: *so*) The *source* command reads *mailx* commands from a file. It is useful when you are trying to fix your *.mailrc* file and you need to re-read it.

top The *top* command takes a message list and prints the first five lines of each addressed message. It can be abbreviated to *to*. If you wish, you can change the number of lines that *top* prints out by setting the valued option *toplines*. On a CRT terminal,

```
set toplines=10
```

might be preferred.

type (abbr: *t*; similar to *print*) Print a list of messages on your terminal. If the *crt* option is **set** to a given value, and the total number of lines in the messages to be printed exceeds the *crt* value, the messages are printed by a terminal paging program such as *more*.

unalias Deletes the specified *alias*(es) from the alias list.

undelete (abbr: *u*) The *undelete* command causes a message that had been deleted previously to regain its initial status. Only messages that have been deleted can be undeleted. This command can be abbreviated to *u*.

unset Reverse the action of setting a binary or valued option.

visual (abbr: *v*) It is sometimes useful to be able to select between two editors, based on the type of terminal being used. To invoke a display-oriented editor, use the *visual* command. Except for the type of editor being used, *visual* is identical to *edit*.

Edit and *visual* commands both assume some default text editor. The default for each can be overridden by the valued options `EDITOR` and `VISUAL` for the standard and screen editors. For example, you could use:

```
set EDITOR=/usr/bin/ex VISUAL=/usr/bin/vi
```

write (abbr: *w*) The *save* command always writes the entire message, including the headers, into the file. If you want to write just the message itself, you can use the *write* command. *Write* has the same syntax as *save*, and can be abbreviated to *w*. For example, to write the second message in *file.c*, type:

```
w 2 file.c
```

As suggested by this example, *write* is useful for such tasks as sending and receiving source program text over the message system.

xit (abbr: *x*) Same as *exit*.

z *mailx* presents message headers in windowfuls as described under the *headers* command. You can move *mailx*'s attention forward to the next window by typing:

```
z+
```

command. Analogously, you can move to the previous window with:

```
z-
```

Custom Options

Throughout this article, we have seen examples of binary and valued options. This section describes each of the options in alphabetical order, including some that you have not seen yet. Options should be typed as all uppercase or all lowercase letters as listed; don't mix letter case within an option.

EDITOR The valued option **EDITOR** defines the pathname of the text editor to be used in the *edit* command and *~e*. If not defined, a standard default editor is used.

SHELL The valued option **SHELL** gives the path name of your shell. This shell is used for the *!* command and *~!* escape. In addition, this shell expands file names with shell metacharacters like * and ? in them.

VISUAL The valued option **VISUAL** defines the pathname of your screen editor for use in the *visual* command and *~v* escape. A standard screen editor is used if you do not define one.

append The **append** option is binary and causes messages saved in *mbox* to be appended to the end rather than prepended. Normally, *mailx* will put messages in *mbox* in the same order that the system puts messages in your system mailbox. By setting **append**, you are requesting that *mbox* be appended to, regardless. It is in any event quicker to append.

ask **Ask** is a binary option which causes *mailx* to prompt you for the subject of each message you send. If you respond with simply a newline, no subject field will be sent.

askcc **Askcc** is a binary option which causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline shows your satisfaction with the current list.

autoprint `Autoprint` is a binary option which causes the *delete* command to behave like *dp*. Thus, after deleting a message, the next one will be typed automatically. This is useful for quickly scanning and deleting messages in your mailbox.

debug The binary option **debug** causes debugging information to be displayed. Use of this option is the same as using the $-d$ command line flag.

dot `Dot` is a binary option which, if set, causes *mailx* to interpret a period alone on a line as the terminator of a message you are sending.

escape To change the escape character used when sending mail, use the valued option **escape**. Only the first character of the **escape** option is used, and it must be doubled if it is to appear as the first character of a line of your message. If you change your escape character, then ~ loses all its special meaning, and need no longer be doubled at the beginning of a line.

<folder> The name of the directory to use for storing folders of messages. If this name begins with a "/", *mailx* considers it to be an absolute pathname; otherwise, the folder directory is found relative to your home directory.

hold The binary option **hold** causes messages that have been read but not manually dealt with to be held in the system mailbox. This prevents such messages from being automatically swept into your mbox.

ignore The binary option **ignore** causes **RUBOUT** characters from your terminal to be ignored and echoed as @'s while you are sending mail. **RUBOUT** characters retain their original meaning in *mailx* command mode. Setting the **ignore** option is equivalent to supplying the $-i$ flag on the command line as described under Command Line Options which follows.

ignoreeof This option is related to **dot**, and causes *mailx* to refuse to accept a CTRL-D as the end of a message. **Ignoreeof** also applies to *mailx* command mode.

keep The **keep** option causes *mailx* to truncate your system mailbox instead of deleting it when it is empty. This is useful if you elect to protect your mailbox, which you would do with the shell command:

```
chmod 600 /usr/mail/<yourname>
```

where <yourname> is your login name. If you do not do this, anyone can probably read your mail, although people usually don't.

keepsave When you *save* a message, *mailx* usually discards it when you *quit*. To retain all saved messages, set the **keepsave** option.

metoo When sending mail to an alias, *mailx* makes sure that if you are included in the alias, that mail will not be sent to you. This is useful if a single alias is being used by all members of the group. If however, you wish to receive a copy of all the messages you send to the alias, you can set the binary option `metoo`.

noheader The binary option `noheader` suppresses the printing of the version and headers when *mailx* is first invoked. Setting this option is the same as using $-N$ on the command line.

nosave Normally, when you abort a message with two DEL s (or **RUBOUT**s), *mailx* copies the partial letter to the file *dead.letter* in your home directory. Setting the binary option `nosave` prevents this.

quiet The binary option `quiet` suppresses the printing of the version when *mailx* is first invoked, as well as printing the *type* command message number with each message.

record If you love to keep records, then the valued option `record` can be set to the name of a file to save your outgoing mail. Each new message you send is appended to the end of the file.

screen When *mailx* initially prints the message headers, it determines how many headers to print by looking at the speed of your terminal; the faster your terminal, the more it prints. The valued option `screen` overrides this calculation and specifies how many message headers you want printed. This number is also used for scrolling with the *z* command.

sendmail To select an alternate delivery system, set the `sendmail` option to the full pathname of the program to use. **Note: this is not for everyone! Most people should use the default delivery system.**

toplines The valued option `toplines` defines the number of lines that the *top* command will print out instead of the default five lines.

verbose The binary option "verbose" causes *mailx* to invoke sendmail with the $-v$ flag, which causes it to go into versbose mode and announce expansion of aliases, etc. Setting the "verbose" option is equivalent to invoking *mailx* with the $-v$ flag as described earlier.

Command Line Options

This section describes command line options for *mailx* and what they are used for.

$-N$	Suppress the initial printing of headers.
$-d$	Turn on debugging information. Not of general interest.
$-f$ <file>	Show the messages in <file> instead of your system mailbox. If <file> is omitted, *mailx* reads *mbox* in your home directory.
$-i$	Ignore tty interrupt signals. Useful on noisy phone lines, which generate spurious RUBOUT or DELETE characters. It's usually more effective to change your interrupt character to CTRL-C (see the *stty* shell command for more information).
$-n$	Inhibit reading of */usr/lib/Mail.rc*. Not generally useful, since */usr/lib/Mail.rc* is usually empty.
$-s$ <string>	Used for sending mail. <String> is used as the subject of the message being composed. If <string> contains blanks, you must surround the string with quote marks.
$-u$ <name>	Read <name>'s mail instead of your own. Other unwitting systems users often neglect to protect their mailboxes, but discretion is advised. Essentially, `-u kathy` is a shorthand way of doing `-f /usr/mail/kathy`.
$-v$	Use the $-v$ flag when invoking sendmail. This feature can also be enabled by setting the the option "verbose".
	The following command line flags are also recognized, but are intended for use by programs invoking *mailx* and not for people.
$-T$ <file>.	Arrange to print on <file> the contents of the `article-id` fields of all messages that were either read or deleted. $-T$ is for the *readnews* program and should NOT be used for reading your mail.
$-h$ <number>	Pass on hop count information. *Mailx* takes <number>, increments it, and passes it with $-h$ to the mail delivery system. $-h$ has effect only when sending mail and is used for network mail forwarding.

$-r$ <name>
: Used for network mail forwarding where <name> is the sender of the message. <name> and $-r$ are simply sent along to the mail delivery system. *Mailx* waits for the message to be sent and the exit status returned. Also restricts formatting of message.

  Note that $-h$ and $-r$ (which are for network mail forwarding) are not used in practice since mail forwarding is now handled separately. They may disappear in future HP-UX versions.

Message Format

This section describes message formats. Messages begin with a `From` line, which consists of the word `From` followed by a user name, followed by anything, followed by a date in the format returned by the *ctime* library routine described in section 3 of the *HP-UX Reference Manual.* A possible *ctime* format date is:

```
Tue Dec  1 10:58:23 1981
```

The *ctime* date may be optionally followed by a single space and a time zone indication, which should be three capital letters, such as MDT.

Following the `From` line are zero or more **`header field`** lines. Each header field line is of the form:

```
name: information
```

`Name` can be anything, but only certain header fields are recognized as having any meaning. The recognized header fields are: **`article-id`**, **`bcc`**, **`cc`**, **`from`**, **`reply-to`**, **`sender`**, **`subject`**, and **`to`**.

Other header fields may be significant to various networks. Refer to the message standards documentation for the network being used for more information. A header field can be continued onto following lines by making the first character on the following line a space or tab character.

If any headers are present, they must be followed by a blank line. The part that follows is called the `body` of the message, and must be ASCII text containing no null characters. Each line in the message body must be terminated with an ASCII newline character and no line can be longer than 512 characters. If binary data must be passed through the mail system, it is suggested that this data be encoded in a format that encodes six bits into a printable character.

For example, one could use the upper- and lowercase letters, the digits, comma and period to make up a set of 64 characters. Thus, a 16-bit binary number could be sent as three characters. These characters should be packed into lines, preferably lines about 70 characters long because long lines are transmitted more efficiently.

The message delivery system always adds a blank line to the end of each message. This blank line must not be deleted.

The UUCP message delivery system sometimes adds a blank line to the end of a message each time it is forwarded through a machine.

Note that some network transport protocols enforce message length limits.

Glossary

This section contains the definitions of a few phrases peculiar to *mailx*.

alias	An alternative name for a person or list of people.
flag	An option, given on the command line of *mailx*, prefaced with a −. For example, *−f* is a flag.
header field	At the beginning of a message, a line containing information that is part of the structure of the message. Popular header fields include `to`, `cc`, and `subject`.
mail	A collection of messages. Often used as in the phrase, "Have you read your mail?"
mailbox	The place where your mail is stored, typically in the directory */usr/mail*.
message	A single letter from someone, initially stored in your *mailbox*.
message list	A string used in *mailx* command mode to describe a sequence of messages.
option	A piece of special purpose information used to tailor *mailx* to your taste. Options are specified with the *set* command.

Summary of Commands, Options, and Escapes

Command Summary

The following tables provide a quick summary of the *mailx* commands, binary and valued options, and tilde escapes. Command abbreviations, where applicable, are shown in bold type in parentheses at the beginning of the description.

Command	Description
!	Single command escape to shell
-	Back up to previous message
Print	(**P**) Type message with ignored fields
Reply	(**R**) Reply to author of message only
Type	(**T**) Type message with ignored fields
alias	(**a**) Define an alias as a set of user names
alternates	(**alt**) List other names you are known by
chdir	(**cd**) Change working directory, home by default
copy	(**co**) Copy a message to a file or folder
delete	(**d**) Delete a list of messages
dp or *dt*	Delete current message, type next message
endif	End of conditional statement; see *if*
edit	(**e**) Edit a list of messages
else	Start of else part of conditional; see *if*
exit	(**ex** or **x**) Leave mail without changing anything
file	(**fi**) Interrogate/change current mail file
folder	(**fo**) Same as *file*
folders	List the folders in your folder directory
from	(**f**) List headers of a list of messages
headers	(**h**) List current window of messages
help	Print brief summary of *mailx* commands

hold	(**ho**) Same as *preserve*
if	Conditional execution of *mailx* commands
ignore	Set/examine list of ignored header fields
list	List valid *mailx* commands
local	List other names for the local host
mailx	(**m**) Send mail to specified names
mbox	Arrange to save a list of messages in *mbox*
next	(**n**, **+**, or Return) Go to next message and type it
preserve	Arrange to leave list of messages in system mailbox
print	Print specified messages without *ignored* headers
quit	(**q**) Leave *mailx*; update system mailbox and *mbox* as appropriate
reply	(**r**) Compose a reply to a message
save	(**s**) Append messages, headers included, on a file
set	(**se**) Set binary or valued options
shell	(**sh**) Invoke an interactive shell
source	(**so**) Reads *mailx* commands from a file.
top	Print first so many (5 by default) lines of list of messages
type	(**t**) Print messages
unalias	Remove one or more *alias* groups
undelete	(**u**) Undelete list of messages
unset	Undo the operation of a *set*
visual	(**v**) Invoke visual editor on a list of messages
write	(**w**) Append messages to a file, don't include headers
xit	(**x**) Synonym for *exit*
z	Scroll to next/previous screenful of headers

Options Summary

The following table describes the options. Each option is shown as being either a binary or valued option.

Option	Type	Description
`EDITOR`	valued	Pathname of editor for ~e and *edit*
`SHELL`	valued	Pathname of shell for *shell*, ~!, and *!*
`VISUAL`	valued	Pathname of screen editor for ~*v* and *visual*
`append`	binary	Always append messages to end of *mbox*
`ask`	binary	Prompt user for Subject: field when sending
`askcc`	binary	Prompt user for additional Cc's at end of message
`autoprint`	binary	Print next message after *delete*
`crt`	valued	Minimum number of lines before using *more*
`debug`	binary	Print out debugging information
`dot`	binary	Accept . alone on line to terminate message input
`escape`	valued	Escape character to be used instead of ~
`folder`	valued	Directory to store folders in
`hold`	binary	Hold messages in system mailbox by default
`ignore`	binary	Ignore DEL (or **RUBOUT**) while sending mail
`ignoreeof`	binary	Don't terminate letters/command input with `EOF`
`keep`	binary	Don't unlink system mailbox when empty
`keepsave`	binary	Don't delete *saved* messages by default
`metoo`	binary	Include sending user in aliases
`noheader`	binary	Suppress initial printing of version and headers
`nosave`	binary	Don't save partial letter in *dead.letter*
`quiet`	binary	Suppress printing of *mailx* version and message numbers
`record`	valued	File to save all outgoing mail in
`screen`	valued	Size of window of message headers for *z*, etc.
`sendmail`	valued	Choose alternate mail delivery system

`toplines`	valued	Number of lines to print in *top*
`verbose`	binary	Invoke sendmail with the *-v* flag

Tilde Escapes Summary

The following table summarizes the tilde escapes available while sending mail.

Escape and Arguments	Description
~*!* command	Execute shell command
~*c* name ...	Add names to Cc: field
~*d*	Read *dead.letter* into message
~*e*	Invoke text editor on partial message
~*f* messages	Read named messages
~*h*	Edit the header fields
~*m* messages	Read named messages, right shift by tab
~*p*	Print message entered so far
~*q*	Abort entry of letter; like DEL (or **RUBOUT**)
~*r* filename	Read file into message
~*s* string	Set Subject: field to *string*
~*t* name ...	Add names to To: field
~*v*	Invoke screen editor on message
~*w* filename	Write message on file
~| command	Pipe message through *command*
~~ string	Quote a ~ in front of *string*

Command Line Flags

The following table shows the command line flags that *mailx* accepts:

Flag	Description
−N	Suppress the initial printing of headers
−T <file>	Article-id's of read/deleted messages to <file>
−d	Turn on debugging
−f <file>	Show messages in <file> or <~/mbox>
−h <number>	Pass on hop count for mail forwarding
−i	Ignore tty interrupt signals
−n	Inhibit reading of */usr/lib/Mail.rc*
−r <name>	Pass on <name> for mail forwarding
−s <string>	Use <string> as subject in outgoing mail
−u <name>	Read <name's> mail instead of your own
−v	Invoke sendmail with the *-v* flag

Note that *−T*, *−d*, *−h*, and *−r* are not for human use.

Index

a

b

c

d

e

f

h

i

l

m

n

o

p

q

r

s

t

u

v

w

x

z

Table of Contents

Using vt

1

Vt is an HP-UX command that provides a means for you to use the HP-UX system where you are currently logged in as a virtual terminal for logging in on other HP-UX systems that are connected to your system through a Local Area Network (LAN). *Vt* is currently implemented on HP 9000 Series 300 and 500 computers running the HP-UX operating system.

Before the *vt* command can be used, *vt* software must be installed on your system. If *vt* is already installed, you can immediately start using it by following the instructions in this chapter. If you are a system administrator and *vt* has not been installed, you can install *vt* by following the procedures documented in Chapter 2. For additional information about the HP-UX virtual terminal command, refer to the *vt*(1) and *vtdaemon*(1M) pages in the *HP-UX Reference.*

Vt can be used essentially three ways:

- **−p** option to poll the LAN and identify all systems in the network that currently have *vtdaemon* running. These are the systems you can currently access through *vt.*
- Log in directly on another (remote) system and interact with it in the same manner as if you were using a local terminal on that system by using *Ovt*'s **remote mode.**
- Use the *vt* escape character to change to *vt*'s **command mode.** *Vt* commands are used to transfer files between the local and remote system and to perform other tasks related to *vt* operation such as directory changes and shell escapes.

Identifying Available Systems

You may or may not know which systems are currently connected to the LAN and have active *vtdaemon* programs running. These are the systems that can currently be accessed through *vt*. If you already know which system you want to access, skip to the next topic.

To determine what systems are currently ready to communicate with your system by means of *vt*, use the **−p** option as follows:

```
vt -p [Return]
```

Vt polls all active devices on the network and lists those systems that responded with messages indicating that they have a currently running *vtdaemon*. Depending on the size of the network, the list may be short or long. Here are some helps in interpreting the list of systems:

- If a given system name is followed by an asterisk (*), that system serves as a gateway into the network for *vt*.
- If a listed system name is followed by one or more plus signs (+), the number of plus signs indicates how many gateways must be traversed by *vt* in order to access that system.

In most cases, you will have little interest in the number of gateways between you and the target remote system. If you are handling large amounts of data between systems, the number of gateways involved may affect overall transfer times. For most situations, the effects are usually insignificant.

Accessing Remote Systems

Accessing a remote system consists of:

1. Establishing a communication path (network connection) to the remote system.
2. Logging in on the remote system.
3. Setting up user identification and password clearance prior to any file transfers.
4. Performing tasks on the remote computer using *vt*'s **remote mode**.
5. Performing file transfers and local system shell escapes using *vt*'s **command mode**.
6. Terminate the connection by logging off of the remote system or using *vt*'s **quit** command.

Connecting to the Remote System

To connect to a remote system, use the *vt* command as follows:

```
vt system_name
```

where *system_name* is the name of the remote system as listed using the *vt* $-p$ command discussed previously (do not include the plus signs or asterisk after the name). This command accesses the LAN through the device special file */dev/ieee* (for most users, the name of the device special file is of little interest).

If, for some reason, you need to access the LAN through a different device special file, use the command:

```
vt system_name lan_device
```

where *lan_device* is the name of the device special file to be used instead of */dev/ieee*.

Logging In on the Remote System

When *vt* has completed its connection to the remote system, *vt* and the remote system respond with a series of messages similar to the following (the first line is the original local shell command telling *vt* to connect to a system named *hpss3c*):

```
vt hpss3c
Connected to hpss3c.
Escape character is '^]'.

Welcome to hpss3c.
Login:
```

The first two lines are produced by *vt*, acknowledging that the connection to *hpss3c* is complete and telling you that the *vt* command escape character (discussed later) is CTRL-]. This escape character must be typed as the first character on the line whenever you need to send a command to *vt* instead of to the remote system.

The blank line and the next two lines following the acknowledgement messages from *vt* are sent to you by the remote system acknowledging that it has recognized your connection and is ready for you to log in as a regular system user. To successfully log in on the remote system, you must have a valid login name and password assigned to you by the system administrator for that remote system or permission to use a login name and password assigned to another user on the same system.

To complete your login on the remote system, use the same procedure as on your local system except for login name and password. You can then use the remote HP-UX system just as you would the one to which your terminal is physically connected.

Aborting the Login Sequence

If, for some reason, you want to terminate the connection without logging in, press CTRL-D.

If you have already started the login then change your mind, send an invalid login message, then when you receive a new login message (if you get a password prompt, simply press Return to get the next login message), type CTRL-D as the first character on the new login line. *Vt* will terminate and then acknowledge the disconnection.

Disconnecting from the Remote System

The easiest way to terminate a remote connection from *vt* is usually to send a [CTRL]-[D] just as on a local computer. The remote shell terminates gracefully, and the *vt* connection is dropped. *Vt* then exits gracefully back to the shell from which it was invoked.

You may sometimes want to terminate *vt* while in command mode, which is discussed in the topics that follow. When in command mode, it may be easier to use the **quit** command discussed later.

Should you experience difficulty when using [CTRL]-[D] to log off, it may be because the remote system's *ptydaemon* has been terminated or terminated and restarted since you logged in. Should this occur, enter the *vt* command mode as described in the following topics and use the **quit** command mentioned in the preceding paragraph.

Sending Shell Commands to the Remote System

When you log in on the remote system, *vt* operates in **remote mode**. Anything you type on your terminal keyboard is transmitted directly to the remote system by *vt* (unless the first character on the line is [CTRL]-[ ] which switches *vt* to command mode and suspends forwarding of commands until command mode is terminated as discussed in the next topic). *Vt* operates as a separate program, so anything you type is ignored by the shell that started the *vt* program (when *vt* terminates, you are returned to the shell that started it). Commands from your terminal that are sent to the remote system by *vt* are interpreted by the remote system and processed accordingly. Error messages and other interaction with the remote system are handled much like they would be if you were logged in on the remote system through a direct line from a local terminal.

File Transfers and Other vt Commands

Vt also operates in **command mode**. Command mode is used on those occasions when you need to use the commands that have been built into *vt* for such tasks as transferring files between the remote and local system or performing other (usually related) tasks such as changing current directories or escaping to a new subshell on the local system. *Vt* commands are interpreted by the *vt* program itself and do not interact directly with HP-UX or user shells on the local or remote system. Rather, *vt* interprets the commands, then issues its own commands to the local or remote HP-UX system according to its own requirements based on the command it received.

Entering Command Mode

To enter *vt* command mode after logging in on the remote system, type the *vt* escape character (CTRL-] keys pressed simultaneoulsy unless defined otherwise). *Vt* then responds with a prompt preceded by the local system name[1] that is similar to the following:

```
[hpss3a] vt>
```

Returning to Remote Mode

Except for the **escape**, **quit**, and **!** (shell escape) commands, *vt* remains in command mode until you press Return without typing a command (this condition is called a null command). When *vt* returns to remote mode following a shell escape, your terminal should display the following prompt:

```
[return to remote]
```

When *vt* returns to remote mode following an **escape** or **quit** command or after a null command, no prompt is provided. You can verify that you are in remote mode by pressing Return again which should produce a new prompt from the remote system.

This missing prompt condition occurs because *vt* does not forward your *vt* command keystrokes to the remote computer. This means that the remote computer has no way of knowing that *vt* has been handling keyboard input and therefore cannot respond with a new prompt. For more information about exit conditions after the **escape**, **quit**, and **!** commands, refer to the discussion about commands in the next section of this chapter.

[1] Having the name of the "local" system can be important if you are using *vt*, *rlogin*, *telent*, or some other facility to log in on a remote system which then becomes your "local" system, from which you use *vt* to access a second system, and so forth. With the availability of shell layers on Series 300 (see *shl*(1) in the *HP-UX Reference* for details), the possibilities for moving from system to system become even greater.

Using Command Mode

Here is a summary of available commands when operating in Command Mode: *Vt* needs only enough characters to uniquely identify the command. Thus only the first letter of the command is required, but you can type as many characters in the command name as you prefer.

Table 1. *vt* Command Summary

Command Syntax	Description
`cd` *remote_directory*	Change file-transfer directory on remote system. Remains in command mode.
`escape` *[escape_char]*	Set the escape character. Returns to remote mode.
`help` `?`	Print *vt* command summary. Remains in command mode.
`lcd` *[directory]*	Change the file-transfer directory on the local system. Remains in command mode.
`get` *remote_file local_file* `receive` *remote_file local_file*	Transfer file from remote system to local system. Remains in command mode.
`put` *local_file remote_file* `send` *local_file remote_file*	Transfer file from local system to remote system. Remains in command mode.
`quit`	Terminate the connection and exit *vt*. Returns to user shell on local computer.
`user` *user_name[:password]*	Identify yourself to the remote *vt* server. Remains in command mode.
`!` *[shellcommand]*	Shell escape. Returns to remote mode.
Return	Null command. Returns to remote mode.

As indicated, some *vt* commands execute and immediately return *vt* to remote mode. Others execute and leave *vt* in command mode. To return *vt* from command mode to remote mode, press Return without typing a command.

Description of Commands

This section describes each *vt* command in detail.

Change Directory on Remote System (cd)

The **cd** command specifies the target file transfer directory on the **remote** system. The **user** command must be executed before **cd** can be used. Syntax is as follows:

`cd` *remote_directory*

For example, to transfer files from directory */users/proj1/jme/work*, type the *cd* command as follows:

`cd /users/proj1/jme/work` [Return]

Vt remains in command mode after completion or failure due to error. If an error occurs, an appropriate message is displayed before the new prompt.

Note

Directory and file transfer operations cannot be started until the necessary user name and password has been provided for the remote system by the **user** command.

Redefining the Escape Character (escape)

The **escape** command is used to define a new escape character or determine the current escape character. To define a new character, use:

escape ⟨escape_char⟩

where ⟨escape_char⟩ is the new escape character. If no new escape character is specified (**escape** followed immediately by [Return]), *vt* will prompt you to enter one (type the new character and press [Return]). If you press [Return] in response to the prompt, *vt* prints the current escape character on the terminal display.

The default escape character is ^] which is obtained by pressing the [CTRL] and [ ]] keys simultaneously.

On-line help summary (help or ?)

The **help** or **?** command tells *vt* to print a summary of available *vt* commands on the terminal display. The syntax is very simple:

```
help
```

or

```
?
```

followed by Return. *Vt* remains in command mode after displaying the command summary.

Change Directory on Local System (lcd)

The **lcd** command specifies the file transfer directory on the **local** system. Syntax is as follows:

`lcd` *directory*

where *directory* is an existing directory on the local system. *Vt* forwards the command to the local system for interpretation, so the rules are the same as for the normal *cd* shell command (see *cd*(1) in the HP-UX Reference). If no directory name is provided, *vt* uses your home directory. If no **lcd** command is used prior to a file transfer, the current working directory that was in effect before *vt* was invoked is used for all *vt* file transfers and shell escapes. Use of **lcd** affects only *vt*. It does not change the current working directory for the shell from which *vt* was invoked.

For example, to transfer files from directory */users/proj3/jsm/work* on the local computer, use the *lcd* command as follows:

```
lcd /users/proj3/jsm/work Return
```

Vt remains in command mode when finished.

File Transfers from Remote to Local (get or receive)

The **get** and **receive** commands are synonymous. They are used to copy *remote_file* from the defined current directory (see **cd** previously discussed) on the remote system to file *local_file* in the defined or default (see **lcd**) current directory on the local system. Syntax is as follows:

```
get remote_file local_file
```

or

```
receive remote_file local_file
```

vt prompts for missing file names if you do not supply them on the command line.

For example, to copy file *vt.xfer* from the remote directory defined by a previous **cd** command to file *vt.receive* in the local (default or specified by a previous **lcd** command), type:

```
get vt.xfer vt.receive [Return]
```

If no **lcd** command is executed prior to **get** or **receive**, the current working directory that was in effect before *vt* was invoked is used. *Vt* remains in command mode upon completion.

To abort a file transfer in progress, press the [Break] key or type the interrupt character (defined by the *stty* command and usually executed during login – see *stty*(1) in the *HP-UX Reference.*

Note

Full directory pathnames can be used when specifying *remote_file* and *local_file* instead of using the **cd** and **lcd** commands. However, user name and password must be provided to the remote system by the **user** command before any file transfers can be undertaken.

File Transfers from Local to Remote (put or send)

The **put** and **send** commands are also synonymous. They are used to copy *local_file* from the default or defined current directory (see **lcd** previously discussed) on the remote system to file *remote_file* in the defined (see **cd**) current directory on the remote system. Syntax is as follows:

```
put local_file remote_file
```

or

```
send local_file remote_file
```

vt prompts for missing file names if they are not specified.

For example, to copy file *hpux.vt* from the local directory (default or defined by a previous **lcd** command) to file *vt.xfer* in the remote directory defined by a previous **cd** command, type:

```
put hpux.vt hpux.vt.xfer [Return]
```

If no **lcd** command is executed prior to **put** or **send**, the current working directory that was in effect before *vt* was invoked is used. *Vt* remains in command mode upon completion.

To abort a file transfer in progress, press the [Break] key or type the interrupt character (defined by the *stty* command and usually executed during login – see *stty*(1) in the *HP-UX Reference*).

Note

Full directory pathnames can be used when specifying *remote_file* and *local_file* instead of using the **cd** and **lcd** commands. However, user name and password must be provided to the remote system by the **user** command before any file transfers can be undertaken.

Terminating the Connection (quit)

The **quit** command logs off of the remote system, terminates the remote connection, exits *vt* and returns you to the shell you were using when you executed the *vt* command. Syntax is as follows:

```
quit
```

This command can be used to conveniently terminate *vt* after a file transfer if you have no need to return to remote mode. It is also useful for terminating *vt* when a network malfunction or other condition makes it impossible to log off of the remote computer (such as the remote *ptydaemon* being terminated and restarted for some reason). Refer to the earlier discussion entitled "Disconnecting from the Remote System" for more information.

Establish File Access Permission (user)

The **user** command is used to set up file access permission on the remote system. Unless you provide a valid name and password to the remote system, internal security will not allow you to access files or change directories on the remote system. This safeguard is in addition to the normal login name and password sequence. Syntax is:

user *user_name[:password]*

vt will prompt for a password (after disabling local echo) if a colon (:) is appended to *user_name*. This command must be executed successfully before any file transfers can be performed.

For example, to identify yourself on a remote system as user "guest" with password "guestin", type:

user guest: Return

When you receive a password prompt in response, enter the password:

guestin Return

Terminal echo is disabled during password requests, so the password you type does not appear on your terminal display screen. *Vt* remains in command mode when finished with the **user** command.

Shell escape (!)

The shell escape provides a means for executing an HP-UX command without terminating *vt*. To execute a command, type an exclamation point (!) followed by the HP-UX command as follows:

! [*shell_command*]

Vt spawns a new sub-shell to execute *shell_command* on the local system. If you do not specify a *shell_command* after the **!**, the new shell is spawned and you receive a shell prompt in response. Use CTRL-D to exit the sub-shell and return to *vt*. Upon completion of *shell_command* or return from the subshell by use of CTRL-D, *vt* resumes operation in remote mode.

For example, to list the local transfer directory, type:

`!ls` Return

Example File Transfer

The most common use for the *vt* package is in transferring files between the local system and a remote system. The basic steps used in transferring files are:

1. Log in on the remote system.
2. Establish file system access permission by using the *vt* **user** command.
3. Use the *vt* **lcd** command to define the local file-transfer target directory.
4. Use the *vt* **cd** command to define the remote file-transfer target directory.
5. Use the **get** or **put** command (or their synonyms, **receive** and **send**) to transfer the file.

The following example is a segment from a typical session and includes login, file access permissions, and a file transfer. Other details such as sending shell commands to the remote computer and logout procedures were discussed previously in this chapter.

The steps shown in this example include logging in on the remote system named "hpss3c" using the "guest" login name, then transferring file */users/guest/xfer/testing1* from "hpss3c" to */users/jme/transfer1* on the local system named "hpss1a". The prompt issued by the local system is "`==>`", the remote system prompt is "`$`", and the *vt* command mode prompt is "`[hpss1a] vt>` ":

System Prompt	Your Response:	Description
`==>`	`vt hpss3c` Return	Connect *vt* to the remote system.
`Welcome to hpss3c.` `login:`	`guest` Return	Send login name to remote system.
`Password:`	`guestpassword` Return	Local echo is disabled, so password does not appear on the screen.
`$`	`^]` Return	Enter *vt* command mode by pressing the CTRL and] keys simultaneously.
`[hpss1a] vt>`	`user guest:` Return	Provide user name and password for file system access permission.
`Password:`	`guestpsswd` Return	Local echo is again disabled, so password does not appear on the screen.
`[hpss1a] vt>`	`cd /users/guest/xfer` Return	Set target transfer directory on the remote system (hpss3c).
`[hpss1a] vt>`	`lcd /users/jme` Return	Set target transfer directory on the local system.
`[hpss1a] vt>`	`get testing1 transfer1` Return	Transfer file from remote system to local system. Still in command mode when finished.
`[hpss1a] vt>`	Return	Type Return to get out of *vt* command mode. Remote system does not send new prompt.
	Return	Type a second Return to get a prompt.
`$`		Prompt indicates remote system is ready for command.

Installing vt 2

Configuration Steps

Before system users can access *vt* capabilities, *vt* and its supporting software and devices must be correctly installed and configured. The necessary and configuration steps are described in this chapter in their usual order of execution:

1. Modify the */etc/rc* script file to automatically start *ptydaemon* and *vtdaemon* at boot-up.
2. Create the LAN device special file.
3. Add LAN segments in the boot area of the root disk.
4. Create pseudo-terminals (*ptys*)
5. Start *vt.*

In addition to these steps, you may want to set up optional UUCP connections or a *vt* gateway. Appropriate procedures are explained later in this chapter.

Background Daemons

When a *vt* command is executed on the local system, *vt* sends a request to the target remote system to log in on that system. The request is received by the *vtdaemon* program running on that remote system which, in turn, sends a request to the *ptydaemon* on the same system to assign a pty (pseudo-terminal) for use by the *vtdaemon. Vtdaemon* then creates a server process to handle communication with *vt* on the local system over LAN, with the pty acting as an interface between the server process spawned by *vtdaemon* and the login shell on the remote system, much like a tty driver interfaces a user terminal and shell on the local system.

Thus it is clear that *vt* requires two demons operating in the background on any accessible remote system to handle incoming connection and login operations. They are *vtdaemon* and *ptydaemon.*

vtdaemon

Vtdaemon is a background program that administers all incoming *vt* connections. *Vtdaemon* should be started automatically by the */etc/rc* script file each time the system boots up. The procedure for modifying */etc/rc* is discussed in the next section of this chapter.

Vtdaemon performs the following functions:

- Responds to *vt* login requests from other systems.
- For each login request received, *vtdaemon* spawns a server to service commands and system requests from the remote user.
- Determines which systems in the network are to be allowed access to the system on which it is running. For example, the *−n* option tells *vtdaemon* to ignore all requests that have come through a gateway, accepting only requests from systems that can access the host system without using a gateway access.
- Creates portals and services portal requests on those systems that use *uucp* in combination with *vt*.

  A **portal** is a device that can be used by *uucico(1M)* to call out to another system via LAN. Portals are created by *vtdaemon* according to the configuration information found in the file */usr/lib/uucp/L-vtdevices*. Up to 48 uucp portals can be configured.

Vt allows up to 16 simultaneous in/out *vt* connections. Up to 48 uucp portals can be configured.

Vtdaemon maintains an activity log in file */etc/vtdaemonlog*. Examine this log file when you encounter problems involving *vt*.

ptydaemon

Ptydaemon is another background program that should be started automatically by the */etc/rc* script file each time the system boots up. The procedure for modifying */etc/rc* is discussed in the next section of this chapter.

Ptydaemon processes requests from *vtdaemon* (as well as other local or network processes) and allocates ptys on the remote system as they are needed, based on availability. Ptys are allocated in master/slave pairs. The slave pty interacts with the shell being used by the assigned process. The master pty interacts with the process controlling the shell. The interaction between program and shell are handled by the pty pair just as their counterparts handle user terminals and shells in tty drivers.

Once a pty has been assigned to the server spawned by *vtdaemon* after receving it from *ptydaemon*, *ptydaemon* remains dormant until the *vt* connection is terminated and the pty pair is returned to the pool of available ptys.

Ptydaemon maintains an activity log in file */etc/ptydaemonlog.* Examine this log file when you encounter problems involving *ptydaemon* startup or pty allocations.

Modifying /etc/rc

The shell script contained in file */etc/rc* is automatically executed by HP-UX at system boot-up. This script should be modified to start *ptydaemon* and *vtdaemon* as background processes whenever it is executed (*cron* is also a background process that serves a comparable role for other aspects of HP-UX system operation).

Use *vi* or any other suitable editor to modify the */etc/rc* file as follows (these changes apply to both Series 300 and Series 500):

1. Examine the line that defines the PATH variable and be sure that it includes */etc* in the list of pathnames.
2. Insert the following lines immediately before the call to *npowerup*.

```
# Start ptydaemon:
                ptydaemon
                echo ptydaemon started
```

3. The *vtdaemon* must start **after** the network is powered up. To accomplish this, modify the following lines in */etc/rc* from:

```
# Start network:
          npowerup <npowerup arguments> &
          echo network powerup started in background
```

   to:

```
# Start network:
          ( npowerup <npowerup arguments> ; vtdaemon ) &
          echo network powerup and vtdaemon started in background
```

   This enforces correct *vtdaemon* start-up sequencing.

Setting up the LAN Device File

The *vt* program uses a character-mode device special file whose major number maps to an IEEE 802 device. A new IEEE 802 special file must be created if one does not already exist on the system.

Vt uses device special file */dev/ieee* by default. If possible, use the same name when creating the new file so that system users do not have to specify a different device special file name in the *vt* command line described in Chapter 1.

Creating LAN Device Files on Series 300 Systems

On Series 300 systems, the major number for the HP 98643A IEEE 802 interface is 18.

To determine whether any LAN device special files already exist on your system, pipe the long listing (*ll*) command through *grep* as follows:

```
ll /dev | egrep "18|19" Return
```

Any existing device special files having a major number of 18 or 19 will be listing in a format similar to:

```
crw-r-----    1 root      other      18 0x030000 Jun 3 19:25 ieee
crw-r-----    1 root      other      19 0x030000 Jun 3 14:39 lan
```

There are 3 possible scenarios:

1. If your system has a device file with `18` in the major number field as shown above, that device file is configured to communicate with the LAN interface. If the name of that device special file is *ieee*, skip to the next topic in this section: Setting Device Special File Permissions.

   If the existing device special file is **not** named *ieee*, create a link from the existing file to a new file of that name in the */dev* directory as follows:

   ```
   ln /dev/existing_file /dev/ieee Return
   ```

   where *existing_file* is the name of the existing device special file previously identified.

2. If the listing shows a device file with major number 19 (ethernet protocol) but no special file with major number 18 is listed, a new device file (*/dev/ieee*) must be created with with major number 18 and the same minor number as the ethernet special file with major number 19.

For example, suppose your system has an ethernet special file named *lan* with major number 19 and minor number 0x030000. To create */dev/ieee* type:

```
/etc/mknod /dev/ieee c 38 0x030000 Return
```

3. If no device file exists in directory */dev* for major number 18 or 19, determine the correct minor number then execute a *mknod* command similar to the command shown in Step 2 to create the required device special file.

   Refer to the *Network Services/9000 LAN Node Manager's Guide* for more details on setting up LAN device special files.

Creating LAN Device Files on Series 500 Systems

On Series 500 systems, the major number for the HP 27125A IEEE 802 interface is 39 (the Series 500 implementation of *vt* does not support the HP 2285A LAN interface).

To determine whether any LAN device special files already exist on your system, pipe the long listing (*ll*) command through *grep* as follows:

```
ll /dev | egrep "38|39" Return
```

Any existing device special files having a major number of 38 or 39 will be listing in a format similar to:

```
crw-r-----    1 root      other     39 0x030000 Jun 3 14:39 ieee
crw-r-----    1 root      other     38 0x030000 Jun 3 19:25 lan
```

There are 3 possible scenarios:

1. If your system has a device file with **39** in the major number field as shown above, that device file is configured to communicate with the LAN interface. If the name of that device special file is *ieee*, skip to the next topic in this section: Setting Device Special File Permissions.

   If the existing device special file is **not** named *ieee*, create a link from the existing file to a new file of that name in the */dev* directory as follows:

   ```
   ln /dev/existing_file /dev/ieee Return
   ```

   where *existing_file* is the name of the existing device special file previously identified.

2. If the listing shows a device file with major number 38 (ethernet protocol) but no special file with major number 39 is listed, a new device file (*/dev/ieee*) must be created with with major number 39 and the same minor number as the ethernet special file with major number 38.

For example, suppose your system has an ethernet special file named *lan* with major number 39 and minor number 0x030000. To create */dev/ieee* type:

```
/etc/mknod /dev/ieee c 39 0x030000 Return
```

3. If no device file exists in directory */dev* for major number 38 or 39, determine the correct minor number then execute a *mknod* command similar to the command shown in Step 2 to create the required device special file.

   Refer to the *Network Services/9000 LAN Node Manager's Guide* for more details on setting up LAN device special files.

Setting Device Special File Permissions

After creating */dev/ieee*, you must configure ownership and access permissions for correct security controls. You have two options:

- If there is no need to restrict access to */dev/ieee*, set the file access permission mode to 0666, file ownership to *root*, and the owner's group to *other*. Three commands are required to accomplish this:

  ```
  chmod 666 /dev/ieee Return
  chown root /dev/ieee Return
  chgrp other /dev/ieee Return
  ```

- If access to */dev/ieee* needs to be restricted for proper system security, *vt(1)* must be able to do a *setuid(getuid())* and a *setgid(getgid())* after it opens */dev/ieee*. Set the access permission mode for */dev/ieee* to 0600, owner to match the owner of */usr/bin/vt*, and change the mode of */usr/bin/vt* to 04555 (setuid bit on).

  For example, if */usr/bin/vt*'s is owned by *root*, execute the following commands:

  ```
  chmod 0600 /dev/ieee Return
  chown root /dev/ieee Return
  chmod 4555 /usr/bin/vt Return
  ```

Setting Up LAN Kernel Support

Before *vt* can use LAN capabilities, the system must be set up to correctly provide kernel support for *vt* LAN operations.

Configuring LAN Device Drivers on Series 300 Systems

On Series 300 systems, the LAN device drivers must be configured as part of kernel customization. Refer to the kernel customization chapter of the Series 300 *System Administrator Manual* for more information.

Adding LAN Segment to Series 500 System Boot Area

On Series 500 systems the boot area on the root disk must contain an optional segment named *HP27125A.opt*. To determine whether the segment already exists, execute the command:

```
osck -v /dev/rhd Return
```

The *osck* command produces a listing of segments in the system boot area. If the listing includes a line containing *HP27125A* as in the following example line:

```
33  669728  15536  HP-UX OPTION  HP27125A  05.05
```

skip the rest of this section because the segment is already in place and does not need to be added. If the output listing does not include a line containing *HP27125A*, execute the following commands to install the segment and reboot the system:

1. Use the *oscp* command to add file *HP27125A.opt* to the sytem boot area:

   ```
   oscp -a /system/xxxxx/HP27125A.opt /dev/root_volume Return
   ```

   where *xxxxx* is the LAN software product number (such as 50954A for multi-user LAN or 50953A for single-user LAN), and *root_volume* is the name of the root volume device character special file.

2. Execute the system shutdown command using the automatic reboot option as follows:

   ```
   shutdown -r 0 Return
   ```

   This command shuts down the system with a zero grace period (meaning that no advance warning is given to any currently active sytem users), then automatically reboots the system.

Creating ptys

One or more master/slave pseudo-terminal (pty) pairs must be present in directories */dev/pty* and */dev/ptym*. These master/slave pty pairs may already exist if HP Windows/9000 is installed on your system.

Series 300

To create a master/slave pty pair on a Series 300 system, use the commands:

```
/etc/mknod /dev/pty/tty[p-w][0-f] c 17 0x0000nn Return
/etc/mknod /dev/ptym/pty[p-w][0-f] c 16 0x0000nn Return
```

Series 500

To create a master/slave pty pair on a Series 500 system, use the commands:

```
/etc/mknod /dev/pty/tty[p-w][0-f] c 29 0xfennn00 Return
/etc/mknod /dev/ptym/pty[p-w][0-f] c 45 0xfennn00 Return
```

where the value range for *nn* is 00 through 0f for p0 through pf, 10 through 1f for q0 through qf, etc. Refer to the section on "Pseudo Terminals" in the "Toolbox" chapter of the *System Administrator Manual* for more details on naming conventions and pty setup.

The number of pty pairs that need to be present (or created) depends on the number of applications on your system that use them. If *vt* is the only application on your system that uses ptys, create only enough pty pairs for the expected number of incoming *vt* connections plus one pty pair for each UUCP portal listed in */usr/lib/uucp/L-vtdevices*.

If you are using both *vt* and HP Windows/9000 you may want to change the number of ptys that are available to HP Windows/9000 because the ptydaemon and HP Windows/9000 both allocate ptys from the same default directories.

Vt allows you to place pty pairs in directories other than */dev/pty* and */dev/ptym*, provided you specify the correct directory on the *ptydaemon* command line. However, other HP-UX commands do not allow specifying an alternate directory, so the practice is not recommended. To access pty pairs in non-default directories, start the *ptydaemon* with the following options:

`ptydaemon` *[master pty directory] [slave pty directory]*

Starting vt

Now that the *vt* support structure is installed and configured, the supporting processes must be started before *vt* can be used. You can either reboot the system or directly execute the start-up commands. If you reboot the system, *vt* is started from the */etc/rc* entries. If you prefer to avoid rebooting the system, execute the following commands:

```
ptydaemon Return
vtdaemon Return
```

Setting Up uucp Connections (optional)

Another feature of the *vt* package is its ability to allow *uucico* to communicate with other systems running *vtdaemon(1M)* over the LAN. This capability speeds up file transfer programs such as *mail* that use a *uucp* connection. To provide this capability on your system, you must create a file called */usr/lib/uucp/L-vtdevices*. You will also need to modify */usr/lib/uucp/L-devices* and file */usr/lib/uucp/L.sys*.

Creating the L-vtdevices File

For each remote system that is to be accessed, add a line of the following form to file */usr/lib/uucp/L-vtdevices*:

`calldev`*[,lan device]* `nodename` *[<reserved for future use>]* `# Comment`

where:

`calldev` is the name of the device file that will be created by *vtdaemon(1M)* when it starts up. By convention, use `culp`*n* where *n* is a hexadecimal digit in the range **0** through **f**.

IMPORTANT

Do not use a name for `calldev` that already exists in the */dev* directory.

[,lan device] This optional field is used to specify an alternate device special file when the LAN connection request is not being sent through the default network special file */dev/ieee*. Otherwise, it can be omitted. When an alternate LAN device special file is specified, its name must match one of the device file names provided as arguments to *vtdaemon* when it was started.

The optional *lan device* field is separated from *calldev* by a comma if it is present in the line. Note that there are no spaces before or after the comma. The square brackets ([and]) only indicate that the field is optional. They should not be typed as part of the line. Use of this field is clearly illustrated in the example which follows.

nodename The node name of the remote system being connected to when *uucico* opens *calldev*.

*/usr/lib/uucp/L-vtdevices* can contain arbitrary white space, blank lines and comments after a '**#**'.

For example, suppose you are setting up three *vt* connection paths to remote HP-UX systems using uucp connections where two systems are accessed through device file */dev/ieee* but the third system is accessed through device file */dev/network*. For such an arrangement, file *L-vtdevices* would resemble:

```
culp0 hpuxa1
culp1,/dev/network hpuxb1
culp2 hpuxz1
```

Note that the *vtdaemon* start-up command must include the LAN device special file argument */dev/network* or the **culp1** entry cannot be used.

Note

Any changes that you make to *L-vtdevices* while the system is running cannot be used until you *kill*, then restart *vtdaemon*.

Modifying the L-devices File

For each line in */usr/lib/uucp/L-vtdevices*, add a line similar to the following to */usr/lib/uucp/L-devices*:

```
DIR calldev 0 9600
```

where *calldev* is the same as the *calldev* in the corresponding line in */usr/lib/uucp/L-vtdevices* (discussed in the previous topic).

Using the previous example, the following lines would be required:

```
DIR culp0 0 9600
DIR culp1 0 9600
DIR culp2 0 9600
```

Modifying the L.sys File

Add a line to */usr/lib/uucp/L.sys* for each system that was included in the list of *callout* devices in the previous two topics. For maximum throughput efficiency, use "f" protocol.

For example:

```
hpuxa1  Any,5 culp0        9600 f/culp0     in: uucp word: XXXX
```

IMPORTANT

Do not use the "@" convention in the login sequence because the *callout* devices are not using *tty* discipline.

Because of a bug in *uucico*, if *L.sys* contains alternate lines for the same system following the above line, the protocol to be used must be explicitly specified each time an entry is set up for that system.

For example, if you have a line like:

```
hpuxa1  Any,5 ACUVENTEL212 1200 2284 in:-@-in: uucp word: XXXX
```

you should change it to:

```
hpuxa1  Any,5 ACUVENTEL212 1200 g/2284 in:-@-in: uucp word: XXXX
```

Setting up a vt Gateway (optional)

A *vt* **gateway** is a system that transfers data between two or more LANs. If your system is connected to more than one LAN and is set up as a gateway, then systems on one LAN can communicate with systems on another LAN through the inter-LAN link provided by your system.

To become a gateway you must use the **−g** option on the *vtdaemon* command line in */etc/rc*.

Being a gateway is not simply being connected to more than one LAN. If you specify multiple LAN device files when invoking *vtdaemon* but do not specify the *-g* option, your system will not be a gateway, meaning other systems on each of those LANs can communicate with your system, but cannot communicate with systems on another LAN through a link provided by your system. In such a situation, the only workaround would be to first *vt* onto your system and log in, then *vt* to the desired destination system.

The *-g* option has an optional numerical argument to limit the number of concurrent *vt* gateway servers your system can run concurrently. This gives you control over how much CPU and other resources (IEEE 802.3 SAPs) you wish to allow other systems to use. There will be no limit (other limits such as the number of IEEE 802.3 SAPs and LAN memory will still apply) if the **−g** option is specified without this optional numerical argument (SAP is an acronym for Service Access Point. Refer to the *LAN User's Guide* for more information about this and other network details.

You can ignore requests that have come through a *vt* gateway by using the **−n** option to *vtdaemon*. One scenario where this option would be useful is:

- System A is a fairly unsecure test system attached to a test LAN (LAN A). The administrator of System A trusts everyone on LAN A.
- However, System B (also connected to LAN A) is a *vt* gateway to a larger facility LAN (LAN B) serving users that should not have any access to System A.

As long as System B is secure[1], System A can use the **−n** option to prevent nodes on LAN B from accessing it.

[1] A secure system in this context is simply a system that enforces passwords. You cannot prevent someone from LAN B from logging in on System B then using System B to access System A through nested *vt*s. However, only users that have logins on the gateway system can login and use a nested *vt*.

Changing Network Memory (optional)

Vt uses lots of network memory, so you may need to increase the network memory limit if your system is supporting several connections simultaneously. On Series 300 systems, the network memory limit is specified by an argument to the *npowerup* command. On Series 500 systems, the network memory limit is specified in */etc/netdir*. *Vt* produces an error message when the local or remote system does not have sufficient memory available to support the requested connection.

Shutting Down vt

All programs to *vt* operation can be shut down gracfully by sending a SIGTERM signal to them (SIGTERM is equivalent to the *kill* command without the **−9** option). If a SIGTERM signal is sent to *ptydaemon*, the System V IPC message queues used by *ptydaemon* for requests/responses are removed. *Ptydaemon* cannot be started if the queues already exist (thus preventing it from being started twice). If *ptydaemon* is killed with a SIGKILL (SIGKILL is equivalent to *kill -9*), *ipcrm* (see *ipcrm*(1)) just be used to remove the message queues before the *ptydaemon* can be restarted.

Index

a

b

c

d

e

f

g

h

i

k

l

m

n

p

q

r

s

u

v

Notes

MANUAL COMMENT CARD

Device I/O and User Interfacing
HP-UX Concepts and Tutorials

HP Part Number 97089-90062 10/87

Please help us improve this manual. Circle the numbers in the following statement that best indicate how useful you found this manual. Then add any further comments in the spaces below. **In appreciation of your time, we will enter your name in a quarterly drawing for an HP calculator.** Thank you.

The information in this manual:

Is poorly organized	1	2	3	4	5	Is well organized
Is hard to find	1	2	3	4	5	Is easy to find
Doesn't cover enough	1	2	3	4	5	Covers everything
Has too many errors	1	2	3	4	5	Is very accurate

fold

Particular pages with errors? _____

_____

Comments: _____

_____

_____

_____

_____

_____

_____

Name: _____

Job Title: _____

Company: _____

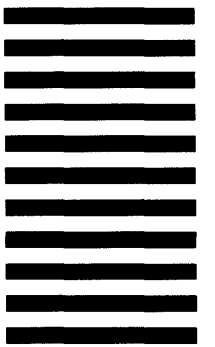
Address: _____

_____

☐ Check here if you wish a reply.

Please Tape Here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



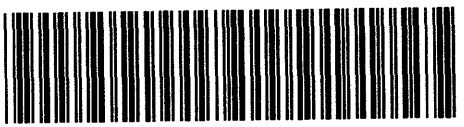
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HP Part Number
97089-90062
Microfiche No. 97089-99062
Printed in U.S.A. 10/87



For Internal Use Only