HP-UX for MPE Users

By Neil P. Armstrong of Robelle Solutions Technology Inc.
E-mail: neil.armstrong@robelle.com

Robelle Solutions Technology Inc.
Suite 201, 15399 - 102A Avenue
Surrey, B.C. Canada V3R 7K1
Toll-free: 1.800.561.8311
Telephone: 604.582.1700
Fax: 604.582.1799
E-mail: support@robelle.com
WWW: http://www.robelle.com

Introduction

This paper is intended to relate MPE commands to HP-UX commands. ***HP-UX for MPE Users*** is not a debate on the merits of MPE/iX versus HP-UX. It is intended to assist the MPE user, programmer, and operator to navigate and use commands on the HP-UX operating system.

Help Me man

The UNIX equivalent to the MPE Help command is the man command. This command is used to display on-line documentation, which is sometimes referred to as the "man pages".

Syntax:

man [-] [*section* [*subsection*]] *entry_name*

Sections

On HP-UX the man pages are divided into the following sections:

1	User Commands
1M	System Maintenance
2	System Calls
3	Functions and Function Libraries
4	File Format
5	Miscellaneous Topics
7	Device (Special) Files
9	Glossary

Section 6 (not found on HP-UX) is reserved for Games. Section 8 (also not found on HP-UX) is sometimes classified as Administrative or Maintenance Commands.

Keyword Searches

To determine what section you want to read, or to get a summary of all the entries with a particular keyword, you can use the -k option of the man command. For example,

```
man -k stty
```

This keyword will only work if you have a man page index file (/usr/lib/whatis), which can be created by logging on as root (superuser) and then doing a catman -w command. This process takes a long time to run, so we suggest that you run it in the background and put an "&" at the end of the command. For example,

```
/etc/catman -w &
```

Where Does man Search?

The man command searches the directories for man page entries in the order described by the MANPATH variable. We will cover this shortly with the PATH variable description.

man Page Listing

Within the man pages, you can use the spacebar to see the next page, or you can press "Q" to stop.

Some Examples

```
$man -k stty
      stty(1)          Set the options for a terminal port
      stty(7)          Terminal interface for Version 6/PWB compatibility
      stty, gtty(2)    Control device
```

```
$man -k who
      from(1)          Who is my mail from?
      rusers(1)        Determine who is logged in on the local network
      rwho(1)          Show who is logged in on local machines
      rwhod(1M)        System status server
      users(1)         Compact list of users who are on the system
      who(1)           Who is on the system
      whoami(1)        Print effective current user id
      whodo(1M)        Which users are doing what
```

CI See a Shell

The command interpreter (CI) in MPE is built-in, but on HP-UX it's just another program. As a result, UNIX has many competing command interpreters, which are commonly referred to as shells.

These are the shells on HP-UX:

ksh	Although the Korn shell (ksh) is compatible with the Bourne shell (sh), it also has many features of csh, including command history editing, line editing, file name completion, command aliasing, and job control.
POSIX	The POSIX shell is similar to the Korn shell, but it is not as secure as the Korn shell.
sh	The Bourne shell, whose default prompt is "$", is the oldest of the shells. It is on every UNIX system. It has no job control, is the default for root, and is commonly used for writing command files or "scripts".
csh	The C shell, whose default prompt is "%", is from Berkeley and has lots of tricky features.

One of the confusing issues for MPE users is that some commands, such as printenv, are built into a specific shell, while others, such as rm and cp, are just programs in the /bin or /usr/bin directories. Built-in commands usually do not have their own man pages, so you have to use man ksh to read the description of the command in the Korn shell documentation.

Logging On and Startup Files

When you press Return to login, the getty program displays the contents of the /etc/issue file first, and then the login prompt before running the login process. Login then validates the user name and password, and places the user in his or her home directory before running the user's shell. HP-UX decides which shell to run based on the information in the /etc/passwd file.

When you login to an HP-UX system, various programs and files are involved. This is similar to the Option Logon UDC commands that can be executed every time a user logs on.

Each shell has its own series of startup files, both globally and locally. The POSIX, Bourne, and Korn shells execute the global file /etc/profile, whereas the C shell executes the commands in /etc/csh.login.

Then the local .profile or .login (C shell) is executed in the user's home directory. The POSIX and Korn shells then execute the .kshrc file in the user's home directory.

Starting Up the Korn Shell

/etc/profile

(#) $Revision: 70.1 $

# Default (example of) system-wide profile file (/bin/sh initialization).

```
PATH=/bin/posix:/bin:/usr/bin:/usr/contrib/bin:/usr/local/bin
                                                         # default
MANPATH=/usr/man:/usr/contrib/man:/usr/local/man     # default path

if [ -r /etc/src.sh ]   # read access to the file?
then
   .  /etc/src.sh       # set the time zone
   unset SYSTEM_NAME
else
   TZ=MST7MDT           # change this for local time zone
   export TZ
fi

if [ "$TERM" = "" ]     # if term is not set,
then                    #
   TERM=hp              # default the terminal type
fi

EDITOR="qedit -s -cvi"
export PATH MANPATH TERM EDITOR
```

# set erase to ^H

```
stty erase  "^H" intr "^Y" eof "^E" kill "^U" swtch "^Z"
stty -parity
stty -istrip
```

This is the local file that the Korn shell runs after executing the global /etc/profile file.

.profile

```
#  () $Revision: 66.1.1.1 $
# Default user .profile file (/bin/sh initialization).

# Set the Qedit variable
       export RCRTMODEL=2

# Set up my prompt
       PS1='$PWD>'
```

Variables

As in any programming language, a variable is a way of keeping track of data. On both HP-UX and MPE, you have essentially two types of variables: environment variables and shell variables.

Environment Variables

As the name implies, environment variables are those variables that tell us about your environment. Environment variables that are set in /etc/profile or your local .profile file are generally used to set information that you don't want to change on a daily basis. These variables, which can be altered in one convenient place, are also used to provide information about you and your environment to the entire system.

On the HP 3000, environment variables are generally the variables that you see when you do a Showvar HP@ command. Some of these variables are preset by MPE, and some are set when the user logs on via a command file, usually triggered by an Option Logon UDC.

Typical MPE Variables

HPPATH	A user-settable string variable used by the CI as a reference to determine where to look for programs and command files.
HPGROUP	A system string variable that contains the current group of the current user.
HPACCOUNT	A system string variable that contains the current account of the current user.

Typical HP-UX Variables

PATH	A user-settable string variable used by all of the shells as a reference to determine where to look for programs and shell scripts.
TERM	A user-settable string variable used to indicate what terminal you are using.
MANPATH	A user-settable string variable that tells the man program where and in what order to search for man pages.

Setting Variables

You can set variables in each shell by typing the following commands.

MPE

```
setvar HPPATH "cmd.sys,cmd.util,cmd.test"
```

C shell

```
setenv HPPATH /bin:/usr/bin:/usr/local::$HOME/bin
```

Bourne/Korn shell

```
PATH=/bin/posix:/bin:/usr/bin:/usr/contrib/bin:/usr/local/bin
export PATH
```

Referencing Variables

MPE

```
echo !hppath
```

C shell

```
echo $PATH
```

Bourne/Korn shell

```
echo $PATH
```

Inheritance

Processes can either inherit environment variables that are currently set, or they can set their own variables. If a process sets its own variables, however, the parent process will not be able to reference these variables once the parent process becomes active again.

Shell Variables

Over the last four years, the Robelle development environment has grown and changed significantly. Instead of doing development on one MPE machine, we now have files and source code on MPE, HP-UX, and even on NT. We move files to and from all of these servers using FTP.

Given the changes in processor technology and long-term support costs, we know that our machines will have to be replaced within three years.

When a new server comes into our environment, changing all of the jobs and scripts can be quite a daunting task. Therefore, we use a logical machine name variable instead of the actual machine name. For example, to set the NTDEV variable to the machine name of our NT development server, I put the following statements in the /etc/profile file so that the variable will be available to all Korn shell users:

```
$NTDEV="beano.robelle.com"
export NTDEV
```

Whenever we need to get a file from our NT machine, I can reference the NTDEV variable:

```
#!/bin/sh
#
#  Get the latest Suprtool WinHelp files from Beano.
#
      ftp -n -v $NTDEV <<!EOD
      user username password
      binary
      cd d:
      cd /windev/winhelp/zip
      get suprux.exe /usr/robelle/winhelp/suprhelp.exe
      quit
      !EOD
```

File System Structure

The HP-UX file system is a tree structure that consists of directories and files. The file system is often compared to a filing cabinet. For example, the filing cabinet is the entire file system, a drawer is a directory, a folder within a drawer is a subdirectory, and a report within a folder can be thought of as a file. The main difference is that HP-UX can have folders *within* folders.

Mapping HP-UX to MPE

On MPE the file system organizes files within groups, and groups within accounts. With MPE/iX 4.5, HP introduced the Hierarchical File System (HFS), which maps the MPE file system to the POSIX standard and makes it more like UNIX.

I tend to associate MPE accounts and groups with HP-UX directories. Therefore, the account is the main directory, and groups are subdirectories within the main directory.

For example, you can map this MPE filename,

```
file1.group.account
```

to

```
/ACCOUNT/GROUP/file1
```

On MPE/iX 4.5 and higher, this is how the HFS file system, which allows more than three levels, maps the MPE file system so that it is similar to HP-UX.

```
/ACCOUNT/GROUP/mydir/mysubdir/myfile
```

Absolute and Relative Path Names

On both HP-UX and MPE, you can reference files in the HFS file space by using either the absolute or the relative name of the file.

I think of the absolute filename as the "full" name of the file. If you have a script file or a program that you want to run as /users/me/cmds/scriptfile, the absolute filename is, of course, /users/me/cmds/scriptfile.

The relative name of the file depends on where you are in the current directory structure. For example, if your current directory is /users/me, the relative name of the file is cmds/scriptfile.

Be careful when you have one version of a program on your current path, and there is another version of the same program (with the same filename) somewhere that is not available to your current path. For example, when we test Suprtool, I usually work from the test directory because the current production version of Suprtool is on my current path. That means if I type Suprtool from my current path, I will get the old version of Suprtool.

To run Suprtool by the absolute pathname, I would need to type in about 31 characters. I can use the ../suprtool shortcut to the previous directory because the latest version of Suprtool is one directory above the test directory. Alternately, I could change my PATH variable so that it always searches the Suprtool development directory as opposed to the production directory.

These are some other shortcuts:

`../suprtool`	Execute Suprtool program one directory above current directory
`./suprtool`	Run Suprtool program in the current directory
`../../suprtool`	Run Suprtool two directories above the current directory

File Access and Permissions

A UNIX file has permissions or modes that define who can do what to it. There are three access types (read, write, and execute) and three types of users: the user who creates it, the group that may have access to it, and all other users.

The ls command with the -l option shows the permissions, the file owner, and the group. Use the id command to see your logon user and group. Use chmod to change file permissions, the chown command to change the file owner, and chgrp to change the file group.

MPE users often forget to give execute permission to script files. You can do this in Qedit/UX by keeping the file with the Keep *filename*,Xeq command.

Permissions are coded in three sets of Read, Write, and eXecute for the owner, the group, and others, respectively. All access to everyone is rwxrwxrwx. Only read access to everyone is r--r--r--. Read/write/execute for the owner is rwx------.

Permissions can also be coded as a 3-digit value:

	owner	**group**	**other**
read = 4	rw-	r--	---
write = 2	4+2+0	4+0+0	0+0+0
execute = 1	6	4	0

```
% chmod +x jtest          {give execute access to all}
% chmod u+x g-x o-x       {add x for owner, remove from group/other}
% chmod g=rw,o= file      {group=r/w only, other=no access, owner=no change}
% chmod 640 jfile         {numeric mode 640 equals rw-r-----}
```

Use Access Control Lists (ACLs) to specify more precise security. Chmod deletes ACLs unless you do a chmod -a.

Some Simple Commands

Listf no LeSs

On MPE we commonly use the Listf or Listfile command to look at the attributes of the given MPE files. On HP-UX we can look at the various attributes of a file with the ls command.

Since the MPE and HP-UX file systems provide such different information about the files, there is no direct correlation between the different Listf modes and the options available in the ls command.

The most commonly used ls format is ls -l, which means list all of the files in the long format. This command can be shortened to ll.

```
$ll proc*
-rw-rw-r--  1 neil   users    968 Feb  1 14:46 proctime.c
-rw-rw-r--  1 neil   users    740 Feb  1 14:46 proctime.o
```

By default, the ls command does not list all of the files. To list the files that begin with a period (.), you must use the -a option.

Fcopy, MPEX, and cp

On MPE we could copy files using Fcopy, Copy and the MPEX Copy command. The MPEX Copy command allows the user to specify filesets when selecting file(s). The cp command in HP-UX also allows this by default.

Like most copy functions, the syntax of the cp command specifies the source file or directories followed by the destination. These are some examples of the cp command:

```
      cp source.file destination.file
      cp /development/src/*.c *.c  {copy all c source files from
/development/src}                          {                  to my current
directory}
      cp -R /users/me .             {copy all files from /users/me to my current}
                                   {                              directory}
```

Purge and rm

Beginning with MPE/iX 5.0, we can purge files on MPE with wildcard characters. On HP-UX the command for purging files is rm, which stands for remove files.

Some examples of the rm command are:

```
rm file1
rm file*
```

WARNING: ***Never*** use rm -r * because this command will remove ***all*** the files from the file system, starting from the root directory.

Rename and mv

On HP-UX the equivalent to the MPE Rename command is mv, which stands for move. This is one example of the mv command,

```
mv file1 file2
```

Overwriting Files

One problem with the cp, mv, and rm commands is that, by default, they do not prompt the user when they are going to overwrite an existing file. To prompt the user before overwriting an existing file, you must use the -i option with the commands. I make the -i option the default for some commands by putting the following alias command in my .profile file.

```
alias cp='cp -i'   # make the cp default interactive
alias mv='mv -i'   # make the mv default interactive
alias rm='rm -i'   # make the rm default interactive
```

File Command

The file command on HP-UX is not the same as on MPE. On MPE the file command is used to define the attributes of a file when it is built or on what device it is built.

On HP-UX the file command attempts to determine the file type. There are no file labels or file codes to indicate the file type on HP-UX. The file command prints out what type of file you have. For example,

```
file proctime.*
proctime.c:        c program text
proctime.o:        s800 relocatable object
```

Displaying and Searching Files

Print More cats

On MPE we typically display files using the Print command. On HP-UX we have a choice of either the more or the cat command. Cat is very basic in terms of how it displays files. It simply prints the entire file on your screen, stopping and starting whenever you press Ctrl-S and Ctrl-Q, respectively.

The more command is much more :) useful in terms of how it lists files. It can display files on a page-by-page basis and allows listing of multiple files. The simple syntax for the more command is:

`more file1`	{display file1}
`more files*`	{display all the files with the name "files"}
`more +/rm /etc/profile`	{display all the lines with the rm command in the file }

Navigating within More

The more command displays the first page and then issues another prompt. At this prompt, you can enter a number of commands (listed below) that control how and what is listed. Most of these commands are optionally preceded by an integer argument (k). The defaults are noted in brackets. Star (*) indicates that the argument becomes the new default.

<space>	Display next k lines of text [current screen size]
z	Display next k lines of text [current screen size]*
<return>	Display next k lines of text [1]*
d or Ctrl-D	Scroll k lines [current scroll size, initially 11]*
q, Q, <interrupt>	Exit from more
s	Skip forward k lines of text [1]
f	Skip forward k screenfuls of text [1]
'	Go to place where previous search started
=	Display current line number
/<regular expr>	Search for kth occurrence of regular expression [1]
n	Search for kth occurrence of last regular expression[1]
!<cmd> or :!<cmd>	Execute <cmd> in a subshell
v	Start up /usr/bin/vi at current line
h	Display this message
Ctrl-L	Redraw screen
:n	Go to kth next file [1]
:p	Go to kth previous file [1]
:f	Display current file name and line number
.	Repeat previous command

Magnet versus grep

On MPE you can search filesets for strings with tools such as Magnet, MPEX, and command file programming. I primarily use Magnet on MPE. The grep program on HP-UX is the standard utility that searches through a set of files for a selected text pattern, which can be specified through a regular expression.

By default, grep is case sensitive (use -i to ignore case), and it ignores the context of a string (use -w to match words only). The grep default also shows the lines that match (use -v to show those that don't match).

```
grep BOB tmpfile        {search the tmpfile file for "BOB" anywhere in a line}
grep -i -w blkpt *      {search files in the current directory for the word
"blkpt"}

grep run[-]time *.txt {find "run time" or "run-time" in all txt files}
who | grep root         {pipe who to grep, look for root}
```

Regular Expressions

Regular expressions are a feature of UNIX. They describe a pattern that can match text within a line. This text can be a sequence of characters, but not words.

Here is a quick summary of the special characters used in the grep tool:

^ (caret)	Match expression at the start of a line, as in ^A
$ (question)	Match expression at the end of a line, as in A$
\ (back slash)	Turn off the special meaning of the next character, as in \^
[ ] (brackets)	Match any one of the enclosed characters, as in [aeiou] Use a hyphen (-) for a range, as in [0-9]
[^]	Match any one character except those enclosed in [ ],as in [^0-9]
. (period)	Match a single character of any value, except end of line
* (asterisk)	Match zero or more of the preceding characters or expression
\{x,y\}	Match x to y occurrences of the preceding characters or expression
\{x\}	Match exactly x occurrences of the preceding characters or expression
\{x,\}	Match x or more occurrences of the preceding characters or expression

As an MPE user, you may find regular expressions difficult to use at first. Please persevere because they are used in many UNIX tools, from more to Perl. Unfortunately, it looks as if every tool has its own syntax because some tools use simple regular expressions, others use extended regular expressions, and some extended features have been merged into simple tools. In addition, regular expressions use the same characters as shell wildcarding, but they are not used in exactly the same way.

If you usually type regular expressions within shell commands, it is good practice to enclose the regular expression in single quotes (') to stop the shell from expanding the expression before passing the argument to your search tool. Here are some grep examples:

```
grep smug files          {search files for lines with "smug"}
grep '^smug' files       {"smug" at the start of a line}
grep 'smug$' files       {"smug" at the end of a line}
grep '^smug$' files      {lines containing only "smug"}
grep '\^s' files         {lines starting with "^s", "\" escapes the ^}
grep '[Ss]mug' files     {search for "Smug" or "smug"}
grep 'B[oO][bB]' files {search for "BOB", "Bob", "BOb", or "BoB" }
grep '^$' files          {search for blank lines}
grep '[0-9][0-9]' file {search for pairs of digits}
```

The back slash (\) is used to escape the next symbol (i.e., turn off the special meaning of the symbol). To look for a caret (^) at the start of a line, the expression is ^\^. A period (.) matches any single character. For example, "b.b" will match "bob", "bib", "b-b", etc. An asterisk (*) does not mean the same thing in regular expressions as in wildcarding. It is a modifier that applies to the preceding single character or expression, such as [0-9]. An asterisk matches zero or more of what precedes it. Thus [A-Z]* matches any number of upper-case letters, including none, while [A-Z][A-Z]* matches one or more upper-case letters.

The vi editor uses \<\> to match characters at the beginning or end of a word boundary. A word boundary is either the end of the line or any character, except a letter, digit, or underscore (_). To look for the word "if", but to skip "stiff", the expression is \<if\>. For the same logic in grep, invoke it with the -w option and remember that regular expressions are case sensitive. If you don't care about the case, the expression to match "if" would be [Ii][Ff], where the characters in square brackets define a character set from which the pattern must match one character. Alternately, you could invoke grep with the -i option to ignore case.

Here are a few more examples of grep to show you what can be done:

```
grep '^FROM: ' /usr/spool/mail/$USER     {list your mail}
grep '[a-zA-Z]'                          {any line with at least one letter}
grep '[^a-zA-Z0-9]                       {everything except a letter or a number}
grep '[0-9]{3}-[0-9]{4}'                 {7 digit phone numbers, such as 999-9999}
grep '^.$'                               {lines with exactly one character}
grep '"smug"'                            {"smug" within double quotes}
grep '"*smug"*'                          {"smug", with or without quotes}
grep '^.'                                {any line that starts with a period}
grep '^.[a-z][a-z]'                      {lines starting with "." and 2 lower case letters}
```

I/O Redirection

I/O redirection allows input (stdin) or output (stdout or stdlist) to be directed to a file instead of your terminal. On MPE, redirected output goes to a temporary file, unless a file command is in effect.

Stdin, Stdout, and Stdlist

```
prog> filename        {send all output from program to a new file}
prog>>filename        {append all output from program to a file }
prog< filename        {receive all input from a file to give to program}
```

Stderr

On HP-UX we have an additional system file called stderr. This is the file that all error messages are written to. Error messages are normally sent to your terminal (same as stdout), but they can be redirected to a file in the Korn shell by using:

```
prog >> file 2>&1     {redirect both stdout and stderr to file}
```

This list shows the associated file numbers for the various system files that the Korn shell recognizes:

0	stdin
1	stdout and stdlist
2	stderr

MPE/iX 5.5 will also be using these three file numbers, although final implementation has not been decided at the time this paper was written.

Pipes and Hereis

Pipes

Pipes allow the user to transfer the output of one command directly into the input of another command. The syntax for piping two commands together is the "|" symbol. The command to the left of the pipe sends its output directly to the stdin of the command on the right.

These are some examples of the "|" functionality:

```
ps -ef | grep suprtool     {who is running Suprtool}
man who | man -o2 lp       {print out the man page for the who command}
```

Hereis

UNIX has a nice feature called the hereis document for specifying input to a program in a shell script. You could use the I/O redirection character (<), but then you would have to find some way to put the input into a file and remember to remove it later. With hereis, you include the input data as in-line text after the program name, eliminating the separate file and making the meaning more obvious. To do this, use the << operator followed by a terminating string that will not appear in your text.

```
sort >file <<EndSortData
bob
sam
bill
EndSortData
```

Another advantage of this syntax is that variable and command substitution is done on the input data. There may be times, however, when you do not want variable substitution to occur. To suppress substitution, put a back slash in front of the terminator string. Otherwise, anything in your hereis document that starts with "$" may be screwed up. In one case, a user did the following commands for an RJE job and the "$$" kept getting replaced with what appeared to be random numbers, which were actually the shell's process id (PID):

```
cat > file <<EOF
$$ADD ID=foo PASSWORD=bar
EOF
```

Unfortunately, the C shell (csh) expects you to escape the terminator at the end of your list with back slash, while the Bourne and Korn shells (sh and ksh) do not.

```
! /bin/csh
cat << \eof
Smug book example of a three-line hereis document in csh.
eof
'e'of
\eof
```

Optionally, you can precede the EOF string with a "-" to strip leading tabs.

```
cat >filelist <<-[]
qedit.*
man/qedit.*
cat/qedit.*
[]
```

Shell Programming

Shell programs are regular ASCII files that contain a series of commands, and have at least read and execute access. You can run a shell program by typing the filename at the shell prompt.

Data can be passed into a shell script through environment variables, command line arguments, and user input. Comments in a shell script are preceded with a "#".

HP-UX Shell Programming Tips and Tricks

Command substitution syntax:

'*command*'

This will execute the command, then replace the contents of the single quote with the output from the command. One useful application of this feature is to simulate MPE's indirect files.

`tar cv 'cat filelist'` {analogous to MPE's store ^filelist}

This is not a general solution to having indirect files because the result of the command substitution is still limited to the maximum length of the command line (usually several thousand characters).

Another use of command substitution is to simulate exporting variables from a program. In MPE, we call setjcw() to set an environment variable (JCW) from a program. While a process cannot change its parent's environment in HP-UX, it can create files. For example, to simulate passing a JCW called "LinesChanged" with a value of 5 back to the shell, simply have the process create a file called "LinesChanged" that contains the line

```
echo '5'
```

The shell can access this "variable" by using this syntax,

```
'LinesChanged'
```

Expression Evaluator

Syntax:

```
expr expression
```

Evaluate the expression and print the result on stdout. Shell special characters (e.g., *) must be in double quotes. The expr command gives you basic math and string operations, and when combined with command substitution, it gives you a convenient way to work with variables.

```
expr length "Qedit"                # 5
expr substr "Suprtool" 5 3         # too
expr index "Suprtool.docnew" "."   # 9
count='expr $count + 1'      # add 1 to count; note the cmd substitution
```

Testing Conditions

```
test condition
[ condition ]       {alternative; note the space around [ and ]}
command
```

If condition returns error code 0, then the test has succeeded. Otherwise, the test has failed.

```
[ -f file ]         {file exists and is a regular file}
[ -d file ]         {file is directory}
[ -s file ]         {file exists and size > 0}
[ str = str ]       {note the space around "="}
[ str != str ]
[ num  -eq -ne -gt -lt -ge -lt  num ]
[ ! condition ]     {not equal}
[ cond -a cond ]    {and}
[ cond -o cond ]    {or}
```

Example

```
if [ -s file1 ]
then
echo "file1 exist"
elif [ 'CompareOutCount' -ne 0 ]; then
   echo "two files not in sync"
else
   echo "the end"
fi
```

Loops

The while loop is the most commonly used looping construct in shell scripts. The following example creates file0 to file99:

```
while [ $count -le $limit ]
do
   create an empty file; can also use touch
   > file$count
   count='expr $count + 1'
done
```

The for loop iterates over a given set of values. The following example creates Qedit.List, Suprtool.List, etc. Each of the .List files contains the contents of the corresponding directory.

```
for var in Qedit Suprtool Xpress Howmessy.*
do
   ls -R /users/robdev/$var > ${var}List
done
```