sh Shell Command Language Tutorial

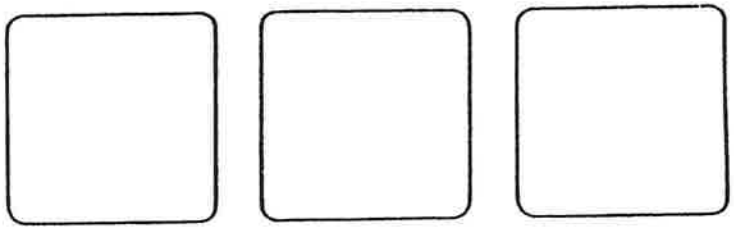

Table of Contents

1. Introduction

The commands that you give the COHERENT operating system are interpreted and acted upon by a special COHERENT program called the *shell*. The shell provides a flexible and powerful command language that handles simple and complex commands alike, and can be used to do actual programming.

If you have not used the COHERENT operating system before, the best place to start is with the *Introduction to the COHERENT System*. It will show you how to log in to the COHERENT system and introduce you to basic concepts. You should understand the material in the first five sections of the *Introduction to the COHERENT System* before you read this tutorial, since this tutorial assumes that you are familiar with the COHERENT file system structure and some basic COHERENT commands. Other useful documents are *COHERENT Command Manual* and *COHERENT System Manual* which describe the use of each command available in the COHERENT system in detail.

A related document with which you should familiarize yourself is the *ed Interactive Editor Tutorial*. This editor helps you prepare and change files containing programs, documents, or data.

2. Getting started with the shell

This section illustrates some basic COHERENT system commands and how the shell can be used to execute them. First, you must *log in* to the COHERENT system. If you do not know how, you should read the *Introduction to the COHERENT System* before proceeding with this manual.

Simple Commands

Once you have logged in, you will see a *prompt*. This prompt is often a dollar sign, '$', although it may be a different character on your system, and, as you will see below, it can be changed. The prompt indicates that the *shell* is waiting for you to enter a command. Give the computer something to do and see what happens. Type:

```
lc  <RETURN>
```

<RETURN> means the carriage return or enter key. Note that you must hit **<RETURN>** at the end of each line that you type to the shell. From here on in this tutorial, it will be assumed that each line ends with a **<RETURN>**.

lc tells the system to list the files in your current directory. You should remember this command from the *Introduction to the COHERENT System*. The shell, also known as a *command language interpreter*, first interprets the command or series of commands, and then executes them. When the shell has finished listing your files it will issue the '$' to prompt you for a new command.

Continue with this example by creating a file called **file01**, and put these lines into it:

```
This is a test
of some simple commands.
```

(If you do not remember how to create a file, reread the *Introduction to the COHERENT System*, or study the *ed Interactive Editor Tutorial*.)

Now, print the contents of the file at your screen. Type:

```
cat file01
```

cat (short for concatenate) is being used here to display a file. cat takes file01 as a *parameter* or *argument*.

The cat command can take more than one argument. To illustrate this, create a second file called file02. In it, type the sentence:

```
This is a second file.
```

Now, join (concatenate) these two files. Type:

```
cat file01 file02
```

The response at your terminal is:

```
This is a test
of some simple commands.
This is a second file.
```

cat printed both files on your terminal in the order they were listed and without breaks. cat will accept any number of files you want to give it.

Constructing shell commands

Simple commands may be combined on one line by separating them with semicolons. Rather than typing the sequence of commands:

```
who
du
mail
```

it is handier for you to use this form:

```
who; du; mail
```

In both of these examples, du will not begin execution until who is finished, and mail will not begin until du is done.

Commands in a file

Many of the commands that you use in COHERENT are programs, such as ed. Others, like the man command, are just files containing other commands to be executed by the shell.

It is often useful to put commands into a file. You might have a frequently used set of commands or a set of complex commands that you would like to abbreviate.

For example, assume that every morning you want to: check the amount of disk space your home directory is using and the amount of disk space still available; list the user currently on the system, sorted in alphabetical order; and, finally, read your mail. It is easy to do all this with just one command. Create a file called good.am with the following commands:

```
du
df
who | sort
mail
```

To execute these commands, you need only say:

```
sh good.am
```

sh calls the shell to read commands from whatever file is given on the command line, in this case good.am. You have just created a *command file*, also known as a *shell script*. Any commands that you issue from your terminal can be incorporated into a command file.

Executable files

If you use this command frequently, you can save typing sh each time if you make the file directly executable. To make a command file executable, use the chmod (change mode) command. Type:

```
chmod +x good.am
```

Now, all you do is type:

```
good.am
```

and your mornings will be off to a good start! You can also have this (or any other command) run automatically when you log in—see the section on the .profile below. If you edit a file, ed will let you change it and the file's executable status will be maintained.

Notice that the commands used in a command file may be other command files. To see how this works, create a file called second.sh (the .sh suffix is sometimes used to indicate a shell script) as illustrated below:

```
good.am
lc
```

Then use sh to execute your latest file:

```
sh second.sh
```

sh will first execute good.am and then will list your files.

Dot—read commands

Similar to sh is the '.' (dot) command. '.' works much like sh and uses less active memory, although it will not take parameters (described below). The shell executes commands in a file directly when you use '.'; when you use sh, the system creates another shell, sometimes called a *subshell*, to execute these commands. For more information, see the *Introduction to the COHERENT System*.

Background commands

Shell commands are normally executed sequentially. If a command ends with the character '&', the shell will begin executing the command immediately, in *background*. This leaves you and your terminal free to continue with other tasks. For example, if you want to sort a large file /etc/passwd, but need to continue with other tasks while the sort is taking place, you could type:

```
sort /etc/passwd >stuff.sorted &
```


The > is a *redirection* symbol, discussed below. When you run a command with '&', the shell responds by typing the *process id* of the command, then will give you a prompt so you can issue further commands. Thus, the response from the system to the above command will be:

```
474
$
```

except that the number will almost certainly differ.

In the COHERENT system, each running command or program is assigned a process id when it begins executing. Normally, there is no need to be concerned about these numbers. But when you run background commands, the shell tells you the id of the background process so that you can keep track of its execution. This can be useful if you start a command, realize there is an error, and have to stop or kill it and start over.

The command

```
ps
```

tells you about the processes you are currently running. If you have no background commands, the response on your terminal will resemble:

```
TTY PID
30: 362 -sh
30: 399 ps
```

The TTY column shows the number that COHERENT has internally assigned to your terminal, and is the same number displayed by who. The PID column shows the process id. The third column shows the name of the program or command executing. The characters -sh in the third column means the shell. Since the process id number is sequentially assigned by the system to each process, the numbers you get will differ.

Once you have started a background command, the ps command will show you the process id for it. In this example, you can see

the sort running in background and the ps command that you issued.

```
TTY PID
30: 362 -sh
30: 474 sort /etc/passwd
30: 495 ps
```

If you want, you can wait for background commands to finish by issuing the command:

```
wait
```

The shell will then accept no further commands until all of your background commands are finished. If you have no background commands running, there will be no wait.

Note that the background processes are still connected to your terminal. If the background command produces output on standard output or standard error (see below) the output will appear on your terminal, even though you are working on another command. See the section 3 on redirection for what to do about this.

The '&' should not be overused, especially on heavily loaded systems. If users on the system send off a large number of background commands, system response will deteriorate noticably.

3. Substitution in commands

The command files described in previous sections were sent directly to the COHERENT shell without change. However, you can greatly enhance the flexibility and power of COHERENT commands through the use of parameters, special characters, and redirection.

First, for many COHERENT commands, you can give a list of one or more *file names* to be acted on.

Second, you can give a command file *parameters*, much like parameters that are passed to a Pascal, Algol, or C function or procedure. With parameters you can target the action of a command file when you call it.

Third, the output of one command can be inserted into another command line. The pipe symbol, '|' is one form of this kind of substitution, and there are several others as well.

File name substitution

File names are some of the most common command parameters. For example,

```
cat file01
```

and

```
ed file01
```

use the file name **file01** as parameters.

By using special shell characters called *wildcards*, you can substitute file names in commands using the wildcard characters in *patterns*. The character '*' will match any string of characters in file names in the current directory. Thus,

```
echo *
```

will list all the file names in the current directory, since echo takes its arguments, expands them fully, then writes them out to the terminal. The command

```
echo f*
```

will give all file names that begin with the letter f, including a file simply called f. And this command:

```
echo a*z
```

will list all names that begin with a and end with z, again, including a file called az.

Note that you can use '*' in conjunction with other characters, so that you can define your pattern more strictly.

To illustrate this, suppose you have three files called file, file01, and file02. Call the echo command as follows:

```
echo file*
```

This will produce the output:

```
file file01 file02
```

Thus, by using a single *, you can substitute several file names into a command. In other words, the command:

```
echo file*
```

is here equivalent to:

```
echo file file01 file02
```

And similarly, the usage

```
cat file*
```

is equivalent to

```
cat file file01 file02
```

assuming that these files are in the current directory.

When several file names are substituted by the shell in this manner, they will be inserted in the order of the numeric values of the ASCII characters substituted for—essentially alphabetically.

If there are no file names that match the pattern, the special characters are not translated, but passed to the command exactly as typed. The following commands should be executed with caution, as they can remove files that you may want to keep. Type:

```
rm file*
echo file*
```

The first command will remove all files whose names begin with file, and is therefore equivalent, for the example above, to:

```
rm file file01 file02
```

The echo command that follows will reply:

```
file*
```

because there are no file names beginning with file, as they were just removed.

Another wildcard character, '?', will match any single letter. Create four files called file, file1, file2, and file33; then issue the command:

```
echo file?
```

The reply will be:

```
file1 file2
```

file? will not match file33 since there are two characters after file, nor will it match file, since the pattern requires one character following the letters file.

The bracket characters '[' and ']' can be used to indicate a choice of single characters for a match. When you type this command:

```
echo file[12]
```

the reply is:

```
file1 file2
```

To match a range of characters, separate the beginning and end of the range with a hyphen. For example:

```
echo [a-m]*
```

will list all file names beginning with a lower-case letter from the first half of the alphabet. You can define a range of numbers just as easily:

```
echo *[1-3]
```

will match **file1**, **file2** and **file33**.

The character '/' defines components in file pathnames, and is not matched by '*' or '?', but must be matched explicitly. Therefore, to list all subdirectories of /usr with a .profile file, type:

```
echo /usr/*/.profile
```

The asterisk will match all the subdirectories of /usr. Or, to list all files in the subdirectory **notes**, type:

```
echo notes/*
```

The special characters discussed in this section can appear anywhere in a command or a command file where a file name can appear. A file name with a leading '.' will not be matched by '*', nor will a '?' substitute for a '.'.

The single quote or apostrophe character, ''', is used to enclose command arguments when you want to prevent the shell from matching special characters within the quotes. You should note that two single quote characters is not the same as the double quote character, '"'.

grep, from the **ed** command **g/regular expression/p**, is a program that searchs its input for lines containing a given pattern. To use **grep** to print the lines in **file01** that contain a pattern beginning with **q** and ending with **X**, type:

```
grep 'q*X' file01
```

The asterisk enclosed in single quotes so that it is passed to **grep** as a parameter and not processed by the shell.

Special characters

If you are familiar with **ed**, you know that there are certain characters that have special meaning to **ed** and must be used in certain ways. The shell also treats some characters specially. These special characters, sometimes referred to as *metacharacters*, are:

```
* ? [ ] | ; { } ( ) $
= : ' ' " < > << >>
```

Some of these special characters have already been presented; the function of the rest will be explained later in this tutorial. If you want to use one of these special characters but not have the shell treat it specially, then precede the character with a backslash, '\', or enclose it in single quotes (not double quotes). For example:

```
grep 'test*' temp
```

will pass the word **test*** to the **grep** command, not all the file names beginning with **test**.

Additionally, the shell treats the following words in a special way when they appear in a command line:

break	do	else	for	then
case	done	esac	if	until
continue	elif	fi	in	while

These are shell *keywords*. They are all commands specific to the shell, and as such are *reserved*. You could not, for example, use one of them as the name of a command file.

Redirection

Most of the COHERENT system's commands and programs accept input from your keyboard and put the output onto your terminal. This is done through two files called *standard input* and *standard output* respectively. The third file *standard error* is used to report errors. These system files are automatically opened by the shell when it is first invoked. Any of these three files can be *redirected* to files or devices other than your terminal. This allows you, for example, to output a file to a printer or even append one file to another, simply by redirecting it. Much of the COHERENT system's power stems from this simple-to-use tool.

Output redirection

There are several different types of redirection, all using one or another of these symbols: '<', '>', and '|'. First look at output redirection. Type:

```
cat file01 file02 >file03
```

cat joins together the two files that have been used up till now, creating a third file named **file03**. If you already have a file named file03, its contents would be erased and overwritten by the new file.

When you are redirecting a file, the '>' and the target file name can be placed anywhere on the command line. For example, these two commands produce the same result as the command above:

```
cat >file03 file01 file02
cat file01 >file03 file02
```

It seems to be more natural if the command is written with the **>file03** at the end and this is the standard practice. Note that

```
cat file02 file01 >file03
```

is not equivalent to the last example, since it reverses the order of the input files.

The '>' tells the shell to overwrite the contents of an existing file or else create a new file. Using '>' to join two files necessarily

creates a third file. Appending a file to an existing file can be done more elegantly with another redirection command, '>>'. To see how this command works, type in the following:

```
lc >file1
who >>file1
```

The first command will overwrite **file1** with a list of the files in your current directory. The next command will append the current users to the end of **file1**. And if you type:

```
cat file1
```

you will get the names of the files in your directory, followed by a list of users.

Input redirection

The **cat** command writes a file to standard output. If you do not give it an input file, **cat** reads the standard input file. Type:

```
cat
This is a test of cat.
<ctrl-D>
```

With the **cat** command using standard input, you signal the end of the input with **<ctrl-D>**. As you recall, this means holding down the **ctrl** key while simultaneously striking the **D** key. After you hit **<ctrl-D>**, your input will be printed back onto your terminal.

You can also enter something at your terminal and redirect it to a file without using an editor. Try:

```
cat >file1
Line number one
Line number two
<ctrl-D>
```

Now use the

```
cat file1
```

command to verify that your text went into the file.

To use the cat command to write standard input to a file without having to put <ctrl-D> at the end, use the '<<' form of input redirection. Type:

```
cat >file1 << .
This is a test
of redirection symbols.
```

This command does two things: it redirects standard input into file1, and the '<<' symbol tells the shell to keep accepting input from the keyboard until it comes across a *token* —in this case the '.'.

Note that the ending token *must* be placed on a line all by itself; the period at the end of the second line did not stop the input redirection. The token can be almost anything; a period is often used because it has the same function in mail and ed. You could use, say, stop or EOF. The only things you cannot use are shell reserved words or symbols which have special meaning to the shell, such as an asterisk.

It is also possible to redirect standard input using '<'. For example, the command:

```
mail fred < gossip
```

will mail the file gossip to fred.

Standard error redirection

If you enter

```
cat file1
```

and file1 does not exist, you will get the message


```
cat: file1: no such file or directory
```

displayed on your screen. This message is not written to the standard output file, but rather to the *standard error* file. This is done so that error messages will not interfere with command output.

Just as you can redirect standard input and output, you can also redirect standard error, using the 2> symbol. The 2 is a *file descriptor*, which is used by the COHERENT system to keep track of all open files. Standard error has a file descriptor of 2, standard input has a file descriptor of 0, and standard output has a file descriptor of 1. To redirect standard error, write a command of the form:

```
cat file4 2>temp
```

If the cat command cannot find file4, it will write the error message in the file temp, instead of on your screen.

A frequently used applications of the redirection of standard error is with the file /dev/null. /dev/null is a file that is always empty, even if you write into it. Thus, the command

```
cat file1 2>/dev/null
```

will throw away all error messages.

Pipes

Up until now, all the forms of redirection discussed have involved files. However, a very useful feature of COHERENT is the linking of input and output of commands in process. This type of redirection is done with a *pipe:* '|'. The pipe directs the output of one command to be the input of another. The string of commands is called a *pipeline*. For example, if you want to list the COHERENT commands in /bin and /usr/bin, sort them, and look at the output one screenful at a time; you can write:

```
ls /bin /usr/bin >temp1
sort temp1 >temp2
cat temp2
```

You will be left with the files **temp1** and **temp2** cluttering up your directory, and you will have to remove them:

```
rm temp[12]
```

On the other hand, if you use pipes, you can simplify this to:

```
ls /bin /usr/bin | sort | cat
```

The list of files will be sent as input to the **sort** command, giving an alphabetic list of the files. When you *pipe* that list to **cat** —a command that writes output to the terminal, a screenful at a time— the sorted list will be printed on your screen.

As you can see, the pipe symbol elegantly avoids the complications of having to create and then remove temporary files created during intervening steps in a series of commands.

It is useful to note that redirection is based on two simple concepts. First, that the COHERENT system sees all files as simply a string of bytes; and second, that most programs act like **filter** in that they take input, do an operation on it, and produce output. Thus, program input can come from a terminal, a file, or from another program; and the output can go to the terminal, another file, or another program; without the program itself being concerned with the details of where its input came from or where the output is going.

When you combine this flexibility with effective commands and utilities, the result is power in a friendly environment.

4. Parameter substitution

Each shell script can have up to nine *positional parameters* labeled **$1** through **$9** with **$0** being the command name itself.

Recall that parameters to a command follow the command itself and are separated by spaces or tabs. An example of a command reference with two parameters is:

```
show first second
```

where **first** and **second** are the positional parameters.

Create the executable shell script **show** containing the following commands and parameters:

```
cat $1
cat $2
diff $1 $2
```

Recall that you must type **chmod +x show** to make **show** executable. The **$1** and **$2** refer to the first and second parameters, respectively. Create a file and call it **first**:

```
line1
line two
line 3
```

Then, create another file, called **second**:

```
line 1
line 2
line 3
```

Then issue the **show** command:

```
show first second
```

Because the shell substitutes **first** for **$1** and **second** for **$2**, this has the same effect as typing:

```
cat first
cat second
diff first second
```

If you issue a command with fewer than the number of parameters referenced in the command, the shell will substitute an empty, or *null*, string in place of each missing parameter. For example, if you only give the parameter first to the show command, like this:

```
show first
```

then the shell will only substitute the first parameter:

```
cat first
cat
diff first
```

The empty string has been substituted for $2.

The example above shows the parameter references separated from each other by a space. Note that for positional parameters it is not necessary to distinguish the parameters, since a positional parameter can only be one digit. To illustrate, build an executable command file and name it pos:

```
echo $167
```

Then call the command file with:

```
pos five
```

And the result will be:

```
five67
```

If pos is

```
echo $1 67
```


then the output from the command

```
pos five
```

would be:

```
five 67
```

Note the space.

There are also two special variables that you can use to let a shell script use all the positional parameters. The first is '$*'. For example, build the file prt to contain:

```
cat $*
```

A script containing this command will cat every file given in the positional parameters. If you type:

```
prt file01 file02 file03 file04 filezz
```

the script will cat all of the files onto your screen.

The variable '$@' resembles '$*' except that it passes all the parameters as one variable. If the shell script seek contained:

```
grep "$@" addr.list
```

then you could type:

```
seek Fred Flintstone
```

and the seek command would use grep to search for the entire string "Fred Flintstone" in the file addr.list. This is because the first argument to grep

```
Fred Flintstone
```

is the pattern that grep looks for in the files listed in the remaining arguments, in this case addr.list.

If seek was written as

```
grep "$*" addr.list
```

using '$*' instead of '$@', then the command

```
seek Fred Flintstone
```

would look for the string "Fred" in the files **Flintstone** and **addr.list**. This is because **grep** has three arguments **Fred**, **Flintstone** and **addr.list**. Since **Flintstone** is a separate (second) argument, **grep** treats it as a filename.

Another important special variable is '$#', which gives the number of positional parameters passed to the command. To see how '$#' is used, see the next example. There are several other predefined special variables, discussed later.

Shell variable substitution

In addition to positional parameters, the shell provides *variables*. The variable name can be constructed from letters, numbers, and the underscore character '_'. Here are some sample names:

```
high_tension
a
directory
DATE1
```

Variable names cannot be single digits, and cannot start with digits, because the shell will treat the leading digit as a positional parameter. Upper case letters and lower case letters are treated as being different in shell variable names.

Values are given to variables by an assignment statement:

```
a=welcome
```

Note that there can be no space before or after the equals sign. You can inspect parameter values with the **echo** command:


```
echo $a
```

The '$' is a special character that signals the shell that you want the *value* of the variable indicated. You must not forget the '$' when referring to the value; this is true for shell variables and positional parameters.

To avoid problems with special characters when you are assigning values, enclose the value to be assigned in single quotes. For example:

```
phrase='several words long'
```

allows the use of embedded blanks in the string.

A handy use for variables is to hold a long string that you expect to type repeatedly in a command. If you are editing files in a directory called **/usr/wisdom/source/widget**, you can abbreviate the pathname by assigning it to the variable **pw**. Type:

```
pw='/usr/wisdom/source/widget'
```

Then you can simply replace the complete pathname with:

```
$pw
```

You can also use a shell variable as a parameter to a command. These can be used instead of positional parameters and are called *keyword parameters*. Create an executable command file, **show2** resembling **show** above:

```
cat $one
cat $two
diff $one $two
```

Then, to use **show2**, issue the following command:

```
one=first two=second show2
```

It is important to note that there are no semicolons separating the parts of this command.

The above command is equivalent to:

```
cat first
cat second
diff first second
```

In this case, the assignment of keyword parameters does not affect the variable after the command is executed. For example, if you type:

```
one=ordinal
one=first two=second show2
echo 'value of one is ' $one
```

the **echo** command will produce:

```
value of one is ordinal
```

Variable names or keyword parameters immediately followed by other text will not be properly recognized. To illustrate, make a shell script, **show3**, containing:

```
echo $onetwo three
```

Calling **show3** by entering:

```
one=careful
. show3
```

will not give you

```
carefultwo three
```

but rather

```
three
```


This is because the variable **$onetwo** has not been defined. The shell does not recognize that you want the variable **one** because it is joined to the word **two**. To prevent this, enclose the name of the variable in braces, as follows:

```
echo ${one}two three
```

and you will now get

```
carefultwo three
```

The braces delimit the variable name from a following string.

Variables not set on a command line are not normally accessible to the command. To illustrate, build an executable parameter display command file named **pars**:

```
echo 1 $1
echo 2 $2
echo p1 $p1
echo p2 $p2
```

pars can be used to show the behavior of parameters. First the name of the parameter is echoed, and then the value of the parameter is echoed (as indicated by the '$' sign). To pass positional parameters, type:

```
pars ay bee
```

and the output will be:

```
1 ay
2 bee
p1
p2
```

To pass keyword parameters, type:

```
p1=start p2=begin pars
```

and the result will be:

```
1
2
p1 start
p2 begin
```

However, the values of **p1** and **p2** have not been kept in your shell. To illustrate, type:

```
echo $p1 $p2 'to show'
```

To which **echo** will reply:

```
to show
```

thus indicating that **p1** and **p2** are not set.

To show that variables set separately from a command are not seen by the command, type:

```
p1=outside1 p2=outside2
pars
```

And you will get:

```
1
2
p1
p2
```

This may come as quite a surprise to you, but if you now type:

```
echo $p1 $p2
```

you will get

```
outside1 outside2
```

Why does **sh** know the value of **p1** and **p2** for the **echo** command, but not for the command file **pars**? The reason is that **p1** and **p2**

are set for your current shell, but not for any other shells. When you use the **echo** command, the variables are read from your current shell, but when you use the **pars** command, the shell starts up another shell, frequently called a *subshell*.

This new shell does not know about the values for **p1** or **p2** or any other local shell variables. The subshell can its own variables and environment—you can, for example, change directories in a subshell—while leaving the variables and environment of the main shell untouched.

By using the **export** command, however, variables can be made available to all of your shells. The commands:

```
export p1 p2
p1='see me' p2='hello'
pars
```

will receive the reply:

```
1
2
p1 see me
p2 hello
```

thus indicating that after the **export** of **p1** and **p2**, these two variables are available to all commands. A variable that has already been exported can be changed and will still be known to all commands without having to be exported again.

Command substitution

By enclosing a command in backwards single quote characters,`, or graves, you can substitute the output of one command into a variable or another command.

This can be a handy way to generate parameters for a command file from a prepared file. Assume the file **listf** contains a list of parameters. These can be passed to the command file **show2** thus:

```
show2 `cat listf`
```

Suppose that you want to use the name of the current directory in a shell script. Write

```
dir='pwd'
```

Special shell variables

The shell automatically sets certain variables to determine the environment of the user, such as the home directory, the file for incoming mail, which directories to search for commands, etc.

When you log in to the COHERENT system, the shell variable **HOME** is set to your *home* or default directory name. If your user name is **martha**, and users are given directories in the directory **/u**, then the command:

```
echo $HOME
```

will reply:

```
/u/martha
```

The change directory command, **cd**, sets the working directory to the pathname described by **HOME** if no parameter is given.

The shell prompt is normally '$'. If more input is needed to complete the command, it is '>'. To see it type:

```
echo 'This is a text that will continue
on the next line'
```

The values of these two prompts are in the variables **PS1** and **PS2**, which can be changed if you want different prompts. For example:

```
PS1='! '
PS2=': '
```

To make these take effect each time you log in, put these in your **.profile** file.

The shell variable **PATH** contains a list of directories that the shell searches to find commands. The contents of **PATH** is shown when you type:

```
echo $PATH
```

PATH is typically:

```
:/bin:/usr/bin
```

The directories are separated by colons, and a null string means the current directory. In this example, the shell will first look in the current directory, then in **/bin** then in **/usr/bin**. You can set **PATH** so that it will also search a **bin** directory of your own for commands. To do this, put the following line in your **.profile**:

```
export PATH=$PATH:$HOME/bin
```

This takes the value of the default **PATH** variable and adds the name of the **HOME/bin** to the end. If your user name is **albert** and your home directory is in **/u**, then the value of **$PATH** will become

```
:/bin:/usr/bin:/u/albert/bin
```

Another variable commonly set in **.profile** is **MAIL**. For the user **henry**, this would likely be set by:

```
export MAIL=/usr/spool/mail/henry
```

This file is used to store incoming mail messages.

The **.profile** can also be used to run commands when you log in. For example, if you had the shell script **good.am** mentioned above, in your **.profile**, it would be run automatically as part of your login procedure.

The features mentioned in this section increase the flexibility of the shell, allowing for more generalized commands. This, combined with the shell programming techniques explained in the following

sections, mean that you can tailor the COHERENT shell to suit your needs.

5. Basic shell programming

This section shows how you can write commands that act differently under different circumstances.

Values returned by commands

Most COHERENT commands return a value called the *exit status* indicating success or failure. You can examine this value by typing the command

```
echo $?
```

which tells you the value returned by the last command executed. The value zero indicates success or truth, while a non-zero value indicates failure or falsehood. Commands that return a failure value usually also give an error message.

Create a file called **test1** with the contents

```
This is a test file
```

Be sure there is no file named **test2**. Now type

```
rm test2; echo $?; cat test1; echo $?
```

You will get

```
rm: test2: no such file or directory
1
This is a test file.
0
```

The '1' in the second line means that the **cat** command failed, while the '0' in the last line means that **cat** succeeded. You can use the exit value of a command to determine what your shell script will do next.

The exit status of a shell command can be set by using the **exit** command. The format of the command is simple:

```
exit n
```

where *n* is an optional number giving the value of the exit status. If *n* is omitted, then the exit status will be unchanged from the exit status of the last command executed.

test—condition testing

The test command's only task is to return an exit status. You can use it to test for the existance and mode of files and the equality or inequality of strings and numbers.

To determine if a file exists, use the command

```
test -f <file>
```

where *<file>* is the name of the file you want. It will return a true value (0) if *<file>* exists and is not a directory, and a false value (1) otherwise. To see the value returned by a command type:

```
echo $?
```

To check if *file* is a directory, use this command form:

```
test -d <file>
```

Strings and numbers can also be examined by test, which is useful when parameter substitution is used. To illustrate, create a shell script test.ed containing:

```
test $1 = $2
echo 'test 1 & 2 for equal:' $?
test $1 != $2
echo 'test 1 & 2 for not equal:' $?
```

Be sure that the '=' in the test command is preceded by and followed with a space, since it is a parameter.

test.ed will test two strings for equality, using the relational operators. Create file1 and put in it:


```
line one
line two
line three
```

and create file2 with:

```
line one
two is different
line three
```

Now try out the command file test.ed. Type:

```
test.ed file1 file2
test.ed file1 file1
```

The first line will produce:

```
test 1 & 2 for equal:   1
test 1 & 2 for not equal:   0
```

and the second line will produce:

```
test 1 & 2 for equal:   0
test 1 & 2 for not equal:   1
```

test.ed could also be written as

```
[ $1 = $2 ]
echo 'test 1 & 2 for equal:' $?
[ $1 != $2 ]
echo 'test 1 & 2 for not equal:' $?
```

The line

```
[ $1 = $2 ]
```

is equivalent to

```
test $1 = $2
```

The '[' and ']' are usually used with **if** (see below). You *must* preceed and follow each of the '[' and ']' with a space, a tab, or a newline.

The comparisons available with the test command are:

s1 = s2	string s1 is equal to string s2
s1 != s2	string s1 is not equal to string s2
n1 -eq n2	number n1 is equal to number n2
n1 -ne n2	number n1 is not equal to number n2
n1 -gt n2	number n1 is greater than number n2
n1 -ge n2	number n1 is greater than or equal to n2
n1 -lt n2	number n1 is less than number n2
n1 -le n2	number n1 is less than or equal to n2

The expressions given above can also be used in conjunction with the following logical operators:

! exp	NOT - negates the logical value of *exp*
exp1 -a exp2	AND - true if both expressions true
exp1 -o exp2	OR - true if either expression true

These expressions can also be grouped using parentheses, which must be enclosed in double quotation marks. For example,

```
test "(" s1 = s2 -a s1 = s3 ")" -o "(" s1 != s4 ")"
```

will return 0 if string s1 is the same as string s2 and string s3 *or* string s1 is not the same as string s4.

Conditional command processing

Now use the two files, file1 and file2, to do a comparison:

```
cmp -s file1 file2
echo $?
```


cmp -s compares two files and returns an exit status of 0 if their contents are the same or an exit status of 1 if the files are not the same. The value of '$?' will be 1 since the files are not the same. The shell can use the exit status to determine what command should be executed next by using the symbols '||' and '&&'.

```
cmp -s file1 file2 || cat file2
```

The characters '||' signify that the following command should be executed if and only if the first command returns a false (non-zero) value, which it will in this example.

The symbol '&&' will execute the following command only if the preceding command returns a true (0) value. This sequence of commands:

```
cp file1 file3
cmp -s file1 file3 && rm file3
```

will remove **file3** if the compare command indicates no differences. Since the **cmp** command is preceded by a copy command, **file1** and **file3** have no differences, and **file3** will be removed.

In general, the commands preceding the '&&' or '||' operators may be other commands separated by ';', '&&' or '||'. Look at this series of commands:

```
cmd1; cmd2 & cmd3; cmd4 || cmd5 && cmd6
```

The above sequence of commands will be executed in the order **cmd1**, **cmd2** which will execute simultaneously with **cmd3** since it is a background command as indicated by the '&', then **cmd4** and **cmd5** if **cmd4** fails, and, if **cmd5** executes successfully, lastly **cmd6**.

To describe what happens in the command line above, the system will:

execute cmd1
execute cmd2 in background
execute cmd3
execute cmd4
if cmd4 failed, execute cmd5
if cmd5 succeeded, execute cmd6

Control flow

The shell is a programming language. It provides the conditional and looping constructs **for**, **if**, **while**, **until**, and **case**.

The **for** construct can be used to process a set of commands once for each of a list of items. A common use is to provide a list of iterative values for parameters.

To illustrate the use of **for**, type the following commands:

```
for i in a b c
do
        echo $i
done
```

The commands within the **for** structure (and the **while** and **until**) must be set off by the word **do** at the beginning and the word **done** at the end. The items **a**, **b**, and **c** form a list of values to be taken on by the *index variable* **i**. The command **echo** will be executed with **i** set to each value in the list in turn. The output will be:

```
a
b
c
```

Notice that after you type the line containing **for**, the shell will prompt with the value of PS2 to remind you that there is more input to be typed in. After you type the line containing **done**, the **for** command will be executed and the prompt will revert to PS1. The **for** command is most often used within a shell script.

The list of values for the index variable can be left off, in which case the list is presumed to be all the parameters in the command line. To illustrate, create this shell script and call it **fortest**:

```
for i
do
        echo $i
        echo '---'
done
```

Notice that there are two commands to be repeated for each value of **i**. Call **fortest**:

```
fortest 1 2 3 4 test
```

and the result will be:

```
1
---
2
---
3
---
4
---
test
---
```

The **for** command can also be used on a single command line:

```
for i in 1 2 3 4 test;do;echo $i;echo '---';done
```

This produces the same results as **fortest**.

Conditional command processing is provided with the **if** shell command. **if** will test the result of a command and conditionally execute other commands based upon that result. **if** can be used in place of '&&' and '||'. Instead of:

```
cmp -s file1 file2 && cat file2
```

you can use:

```
if cmp -s file1 file2
then cat file2
fi
```

with the same result. Note that the **if** command will prompt you for further input until it receives the **fi** just as the **for** command prompted you for input until it received a **done**. The command line:

```
cat file2
```

is executed only if the cmp command returns a zero or true value.

To use the **if** statement to get the same result as:

```
cmp -s file1 file3 || rm file3
```

you will need to use the **else** statement as well:

```
if cmp -s file1 file3
then
else rm file3
fi
```

The commands between **else** and **fi** will be executed if the result of the command following the **if** is false or non-zero. Notice that the **then** part of the **if** command is empty.

Another form of the **if** statement will allow you to test several conditions with one **if** statement, and act on the one that is true. You do this using the **elif**:


```
if command1
then action1
elif command2
then action2
elif command3
then action3
...
else action4
fi
```

The items labeled *command* and *action* are commands.

First, *command1* is executed. If the return is true, *action1* is performed. If the exit code from *command1* is non-zero, then *command2* is executed. If its result is true, then *action2* is performed. This process continues until one of the *commands* returns a true result. If none of the *command* returns is true, then *action4* following the **else** is executed.

To illustrate, create a shell script that takes a file name and, if it is a directory, lists the contents of the directory, or if it is a file that has something in it, or it is a file of zero length will list its name, or else will give a message. The command

```
test -d name
```

returns a value of true if *name* is a directory,

```
test -r name
```

returns a value of true if *name* is an existing file on non-zero length, and

```
test -f name
```

returns a value of true if *name* is an existing file. Create the following executable command file and name it **filecheck**:

```
if test -d $1
then ls $1
elif test -r $1
then cat $1
elif test -f $1
then echo $1
else echo '$1 is not a file or a directory'
fi
```

In order to use the if statement to compare two strings, you must use the test command. To illustrate, create the following command file, listdir, which performs different commands depending upon the parameters it is given:

```
if test $1 = a
then ls -l $2
elif test $1 = b
then lc $2
elif test $1 = c
then pwd
else echo "unknown parameter: $1"
fi
```

Now, when you type:

```
listdir a
```

it is the equivalent of

```
ls -l
```

And when you type:

```
listdir b
```

it is the equivalent of

```
lc
```


The test command checks to see if parameter $1 is equal to a, b, c, or is unrecognized; and then the if command executes the appropriate command.

while is another looping or repetitive shell statement. The commands:

```
while command1
do
        command2
done
```

will first perform *command1*. If its result is true, then *command2* is executed, and *command1* is again executed. This process repeats until the return from *command1* is no longer true.

Note that the value of *command1* is only tested at the beginning of each loop of the while statement. Consequently, if the value of *command1* is false inside the while loop, but is true at the end of the loop, then the while loop will continue to execute.

An example of how the while command can be used is the following script, movepairs, which renames (using the mv command) files:

```
if test $# -lt 2
then echo "usage: movepairs file1 file2 ..."
     echo "move file1 to file2, file 3 to file4, ..."
     exit 1
fi
while test $2 != ""
do
        mv $1 $2
        shift
        shift
done
if test $1 != ""
then echo "movepairs: odd number of arguments"
     exit 1
fi
```

"" is used to denote the null string.

The first thing **movepairs** does is check to see that there are at least 2 arguments. If there is not, **movepairs** prints out its usage instructions, then exits with a non-zero return. Printing a usage message is often done, especially if the script is to be used by others.

The error message about the improper number of parameters starts with the name of the command, and the parameter **$0** is often used to give it. This is a standard practice, so as to let you know where the error occurred—it might happen, for example, in the middle of a pipeline. Note that the error message

```
echo "movepairs: odd number of arguments"
```

begins with the name of the command **movepairs**, so that a user knows what command found an error. You can also use the shell parameter **$0** to tell the name of the command (unless a **shift** has been done):

```
echo $0: odd number of arguments
```

The **shift** command shifts positional arguments to the left; that is, the value of **$1** is discarded, the value of **$2** is put in **$1**, the value of **$3** is put in **$2**, and so on. The highest numbered positional parameter is unset; that is, if **$3** was the highest positional parameter, there is no longer a value assigned to **$3**.

until resembles **while**. The commands

```
until command1
do
        command2
done
```

will first do *command1* but will only execute *command2* as long as the return from *command1* is false. The loop will terminate when *command1* returns true. To illustrate, create the shell script **splitfile** as:


```
until test $# -eq 0
do
        echo $1
        shift
done
```

If **splitfile** is called with:

```
splitfile four score and seven years ago
```

it would print

```
four
score
and
seven
years
ago
```

and then stop.

The **true** and **false** commands perform the obvious functions of exiting with **0** (true) and **1** (false) status. **true** is often used with **while** to set up an unconditional loop.

It is helpful to be able to control loops from inside the loop rather than just at the top. The **break** and **continue** commands were taken from the C programming language: **break** stops execution of a **for**, **while**, or **until** loop; **continue** goes to the next iteration of the loop. These commands must be between a pair of **do** and **done** statements, or else the shell will ignore them.

To show the use of **break** and **continue**, write a shell script called **cmpfiles** to compare a set of files with a standard file called **stdfile**. **cmpfiles** will quit looking after it finds the first match for **stdfile**, and will print the name of the matching file. If **cmpfiles** finds no file matching **stdfile**, it will print

```
matching file is none
```

If one of the files is non-existant or unreadable, **cmpfiles** will go on to the next file.

```
filename='none'
for i
do
        if test ! -r $i
        then continue
        fi
        if cmp -s $i stdfile
        then filename=$i
             break
        fi
done
echo "matching file is $filename"
```

The case statement resembles the if statement in that it offers a multiple decision. The case statement checks the value of a shell variable, and performs the commands called for when it finds a match for that variable. Just as the if construct must be closed by a fi statement, a case must be closed by an esac.

To illustrate, create a shell script named **listdir1** that gives a choice of listing your directory in different ways:

```
case $1 in
     a) ls -l $2;;
     b) lc $2;;
     c) pwd;;
     *) echo unknown parameter $1;;
esac
```

listdir1 performs the same function as the command file **listdir** in an earlier example. **listdir** uses the if and test commands to make the same choices as the case command does in **listdir1**. The effect of the command:

```
listdir1 b
```

is equivalent to:


```
lc
```

while this command:

```
listdir1 a file
```

is the same as:

```
ls -l file
```

Each choice within the case statement is a string followed by ')'. For example:

```
b)
```

indicates the choice for $1 having the value b.

The first choice which matches has its command executed, then the case construct is exited.

The strings selecting the choices may be patterns. The '*)' choice signifies that a match is made on any string. Notice that this resembles the use of '*' to substitute any filename. If one of the other choices is matched first, then the '*' choice will not be executed. It is generally a good programming practice to include '*)' as the last choice in a case statement, to catch bad data.

In a case statement, an expression of the form:

```
[1-9])
```

will match any digit from 1 through 9.

A list of alternative choices may be presented by separating the choices with vertical bars:

```
a|b|c)
```

Notice that each command or command list in the case choices must be terminated by the double character ';;', except for the last

one, where it is optional. If you would like no action to be taken in one choice, just follow the ')' by ';;'. Thus

```
d);;
```

would do nothing if the selection was 'd'.

These control structures make the shell into a true computer language, capable of being used for structured programming.

6. Advanced shell programming

There are additional features to make the shell more powerful and serviceable.

Conditional substitution

As you have seen in previous sections, the shell provides variables or *parameters* to give more flexibility to shell programming. You can change the value of these parameters; for example, by substituting a standard value if the parameter is undefined. To do so, use *conditional substitution* to set values for parameters:

${parameter-word} If *parameter* is set, then substitute its value; otherwise substitute *word*.

${parameter=word} If *parameter* is not set, then set the value of *parameter* to *word*. This is not applicable to positional parameters.

${parameter?word} If *parameter* is set, then substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, then a standard message is printed.

${parameter+word} If *parameter* is set, then substitute *word*; otherwise, substitute nothing.

As an example of the use of conditional substitution, create the shell script **pslax**, which reports on processes by the **ps** command. If your user name is **mimi** and you usually just want to know about processes on the system which are owned by you, but sometimes you want to know about other users' processes, write:

```
ps -lax | grep ${1-henry}
```

If you do not give **pslax** a positional parameter, it will write something like

```
20:  22   1   henry 32K    1 W          -sh
20:  24  22   henry 35K    1 W          pslax
20:  25  24   henry 35K    1 W          pslax
20:  26  25   henry 25K    1 R          ps -lax
20:  27  25   henry 10k    1 S 037312   grep henry
```

if you give pslax a user's name as a parameter, it will report on that user.

Arithmetic operations

The expr command lets you do arithmetic and logical operations on integers and also has some pattern matching capabilities. It evaluates an expression, and writes the result on standard output.

One thing you should be careful about when using expr is that some of the operators used by expr have meaning to the shell:

```
* ? [ ] | ; { } ( ) $
= : ' ' " < > << >>
```

These characters must be escaped using the backslash character, '\'., if the expr command is not within single quotes. These characters must be within double quotes if the expression is inside back quotes. Thus, you should write

```
sum='expr a "*" b'
```

if you wanted to multiply a by b using expr.

Another use for expr is the control of loops. For example, consider the following shell script, lprmany, which will let you print multiple copies of the same file:


```
if test $# -lt 2
then echo "usage: lprmany n file ..."
     exit 1

copies=$1
shift
while test $# -gt 0
do
        i=1
        while test $i -le $copies
        do
                lpr $1
                i='expr $i + 1'
        done
        shift
done
```

Note the nested while loops. Here, expr is used to add one to the current value of i, thus setting a loop which executes lpr *copies* times for each file.

Another way of controlling loops is with the *from* command. This takes the form

from *start* to *stop* [by *increment*]

where *start*, *stop*, and *increment* are integers. The by *increment* may be omitted if the *increment* is 1. As an example of how from is used, the main loop of lprmany could be rewritten as:

```
copies=$1
shift
while test $# -gt 0
do
        for i in 'from 1 to $copies'
        do
                lpr $1
        done
        shift
done
```

Interrupt handling

Signals may be sent to a process from another process, from the shell, from the terminal, or from COHERENT. You can have the process stop at the signal, ignore the signal or execute a command. Not all signals can be ignored. To do this, you use the **trap** command.

trap tells the current shell to execute a specified command when it gets one of a list of signals. If the command is the null string, then the signal will be ignored. If no signal is specified, then the command is executed when the process exits. For a complete list of signals, see **signal** in the *COHERENT System Manual*.

If you have a shell procedure which creates temporary files, all of which have the string **tmp** as the first characters of their names and you want to make sure these files are removed in case of an exit (not really a signal, but a trap treated as 0), a hangup (signal 1), an interrupt (signal 2), or a termination (signal 5); you put into your script the line:

```
trap 'rm tmp*' 0 1 2 5
```

Shell invocation arguments

There are several parameters you can use when invoking the **sh** command itself. These parameters are:

−c *string*
: Read shell commands from *string*. Use double quotes to enclose the string if there are any spaces in it.

−e Exit from the shell on any error, if the shell is not interactive.

−i Make the shell interactive, and prompt for input. In a shell script, ignore interrupt (signal 2) and termination signals (signal 5).

−k Export keyword arguments.

−n Read and expand commands, but do not execute them. Useful in checking the shell syntax.

−s Read commands from standard input, and write shell output to standard error. This is what the shell does by default.

−t Read and execute one command rather then the entire file, then exit.

−u If a shell variable has no value, treat it as an error. Ordinarily, the shell will substitute a null string.

−v Print on standard error shell input lines as they are read.

−x Print command and its arguments to standard error as they are executed. This parameter is very useful in debugging shell scripts.

− Cancel the **−x** and **−v** parameters. This is used if you have a long shell script and only want debugging information about part of it. This is used in conjunction with the **set** command described below.

Thus, if you have a shell script called **mvmount** that you want to check to make sure it does what you want, invoke it with

```
sh -x mvmount
```

If you want to change the shell invocation flags, use the **set** command. For example, if you want to make a shell script interactive, instead of having to call it using **sh -i** all the time, put the line

```
set -i
```

in the file.

To see what the current shell options are, type

```
echo $-
```

The set command can also be used to assign values to positional parameters. For example, the command

```
date
```

writes the date to standard output:

```
Thu Feb  7 16:30:27 1985 CST
```

Writing the commands

```
set 'date'
echo $1
```

will give

```
Thu
```

Conclusion

You will probably find that you use the shell more than anything else the COHERENT system has to offer, with the possible exception of a text editor. The COHERENT system has many useful programs and utilities, each one of which is designed to do one job well. The shell is a convenient way of binding these components to work together in an efficient manner. Often, you can write a shell script more easily than you can write a C program. This ease and power is one of the main reasons the COHERENT system is so productive.

Index

User Reaction Report

To keep this manual and COHERENT free of bugs and facilitate future improvements, we would appreciate receiving your reactions. Please fill in the appropriate sections below and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name: _____

Company: _____

Address: _____

_____

Phone: _____ Date: _____

Version and hardware used: _____

Did you find any errors in the manual? _____

_____

_____

Can you suggest any improvements to the manual? _____

_____

_____

Did you find any bugs in the software? _____

_____

_____

Can you suggest improvements or enhancements to the software?

_____

_____

_____

Additional comments: (Please use other side.)