Introducing sh, the Bourne Shell

sh is the command that invokes the Bourne shell, which is the COHERENT system's default command interpreter. The Bourne shell interpets commands, and much more. It is, in effect, both a programming language and an interpreter.

At least one writer has noted that the shell is the original "fourth-generation language" — that is, a powerful programming language that is straightforward enough to be programmed by non-programmers. You will find that taking a little time to master the rudiments of the shell programming language will pay enormous benefits in making best use of your COHERENT system.

Simple Commands

The shell command language is built around simple commands. For example, the following command lists all files in the current directory:

```
lc
```

You can combine several simple commands on one line by separating them with semicolons:

```
who;du;mail
```

The shell executes the commands in sequence as if they had been typed:

```
who
du
mail
```

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

Special Characters

The shell treats the following characters specially; if you want to use them without their special meaning, you must precede them with the backslash character ****, or enclose them within quotation marks:

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

The function of these characters will be explained later in this section. To use one of these characters in a command, for example '?', type:

```
echo \?
```

In addition, the shell treats the following words in a special way when they appear as the first word of a command:

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

Running Commands in the Background

The shell can execute commands simultaneously as well as sequentially. This means that while the shell is executing one command, it lets you type and execute another command. Under the shell, the number of commands you can execute at the same time is limited mainly by the amount of memory and disk space on your system.

If a command is followed by the special character '&', the shell begins to execute it immediately, and prompts you to enter another command. For example, if you need to **sort** a large file but want to continue with other commands while the sort is executing, you can type:

```
sort >bigfile.sorted bigfile.unsorted &
ed prog
```

This allows you to edit file **prog** while your computer quietly executes the sort in the background.

When you run a command with **&**, the shell types the *process id* of the command started in background. When the COHERENT system runs a command, it assigns that command a *process id*, which is a number that uniquely identifies that command to COHERENT. Normally, there is no need to be concerned about these numbers. However, when you run commands in the background, the shell tells you the id of the background process so you can keep track of its execution.

The command

```
ps
```

lists the processes you are currently running. If you have no background jobs, the response is:

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

The first column shows the number that COHERENT has assigned to your terminal. This is the same terminal number printed out by **who**. The second column shows the process id; the third column shows the program or command executing. The characters **-sh** in the third column means the login shell. There are two processes because the shell is running the **ps** command as a separate process.

Once you have started a background command, **ps** shows you the process entry, which has the process id that the shell typed out for you.

If you need the results from a background job, you can wait for it to finish by issuing the command:

```
wait
```

The shell will then accept no further commands until all your background jobs are finished. If there are no background jobs, there will be no delay.

Scripts

Many of the commands that you use in COHERENT are *programs*, such as **ed**. Others, like the **man** command, are *scripts*, or files that merely contain calls to other commands. You can write scripts on your own, simply by using a text editor to type into a file the commands you wish to execute. If you frequently use a set of commands, you can save yourself from having to type them over and over by simply typing them once into a script.

For example, suppose that you wish to check periodically the amount of disk space that you have used, the amount of disk space still available, and see who is using the system. You can write a script to do all of this automatically. Create the script **good.am** by typing the following commands:

```
ed
a
du
df
who | sort
mail
.
w good.am
q
```

From now on, to execute the above-listed commands, you need only type:

```
sh good.am
```

where **sh** is a command that means: read commands from a file, in this case **good.am**. If you can issue a command from your terminal, you can also execute it from within a script.

You can make a command file directly executable by using the command **chmod**. For example, the command

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

lets you execute the script **good.am** by typing

```
good.am
```

and leaving off the **sh**. Once you have done the **chmod** command, you can still issue the commands by typing:

```
sh good.am
```

as well as use **ed** or MicroEMACS to change the contents of the script.

Notice that the commands called by a script may themselves be scripts. This is illustrated by the following script, **second.sh**:

```
ed
a
good.am
lc
.
w second.sh
q
```

Thus, typing:

```
sh second.sh
```

calls the script **good.am**, and also calls the command **lc**.

.profile: Login Shell Script

When you log into the system and before you are issued your first prompt, COHERENT checks your home directory for a file named **.profile**; if it is present, the shell executes the commands it contains.

This enables you to have COHERENT execute commands as soon as you log in. Check if your installation provides one for you by doing an **lc** (be sure that your current directory is the home directory). If the file is there, print it by saying:

```
cat .profile
```

Some of the commands may be of the form:

```
PATH=':/bin:/usr/bin'
```

This sort of command will be discussed below.

Substitutions

Scripts of the form shown above are processed by the COHERENT shell without change. However, the COHERENT shell increases the power of commands by performing three kinds of substitutions within commands before it executes them.

First, it replaces special characters in commands with file names from the current or other directories. This allows you to issue a single command that processes several files.

Second, you can give a script *arguments*, much like arguments that are passed to a Pascal, Algol, or C procedure. This lets you target the action of the script to a specific file name specified when you call it.

Third, the output of one command can be "piped" into another command to serve as its input.

We will use the command **echo** to illustrate these kinds of substitution. Remember that substitutions take place for all commands in the same way that they do for **echo**.

File Name Substitution

File names are often used as command parameters. That is, you will often tell a command to do something to one or more files. By using special shell characters, you can substitute file names in commands. These special characters describe file name *patterns* for the shell to look for in the directory. When the shell finds the file names, it replaces the pattern with them.

The asterisk ***** matches any number of any characters in file names. Thus,

```
echo *
```

echoes all the file names in the current directory, whereas

```
echo f*
```

gives all file names that begin with the letter **f**, and

```
echo a*z
```

lists all names that begin with **a** and end with **z**.

To illustrate more clearly, create two files by typing

```
cat >zz1
<ctrl-D>
cat >zz2
<ctrl-D>
```

Then the **echo** command

```
echo zz*
```

produces the output:

```
zz1 zz2
```

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

```
echo zz*
```

is equivalent to

```
echo zz1 zz2
```

If no file names fit the pattern, the special characters are not changed, but left in the command exactly as you typed them. To illustrate, type the command

```
rm zz*
echo zz*
```

The first command will remove all files whose names begin with **zz**, and is therefore equivalent to:

```
rm zz1 zz2
```

The **echo** command that follows, however, echoes

```
zz*
```

because no files begin with **zz**; they were just removed.

Enclosing command words within apostrophes prevents the shell from matching file names with the enclosed characters. In the unlikely event that you have a file whose name is

```
zz*
```

that you want to remove, use the command

```
rm 'zz*'
```

The * is enclosed within apostrophes, and therefore is not changed by the shell.

Another special character **?** match any one letter. To see how this works, create empty files **file1**, **file2**, and **file33** by typing:

```
>file1
>file2
>file33
```

The command

```
echo file?
```

replies

```
file1 file2
```

because **?** does not match **33**.

You can use brackets **[** and **]** to indicate a choice of single characters in a pattern:

```
echo file[12]
```

This command replies:

```
file1 file2
```

To match a range of characters, separate the beginning and end of the range with a hyphen. The command

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

prints any file name beginning with a lower-case letter from the first half of the alphabet, and is exactly equivalent to:

```
echo [abcdefghijklm]*
```

When such patterns find several file names, they are inserted in alphabetical order.

Because the character **/** is important in path names, the shell does not match it with * or **?** in patterns. The slash must be matched explicitly; that is, it is matched only by a **/** itself. Therefore, to find all the files in the **/usr** directories with the name **notes**, type:

```
echo /usr/*/notes
```

The asterisk matches all the subdirectories of **/usr** that contain a file named **notes**.

In addition, a leading period in a file name must be matched explicitly. If you have a file in your current directory with the name **.profile**, the command

```
echo *file
```

does not match it.

These patterns can appear anywhere within a command or a command file.

Parameter Substitution

Each shell script can have up to nine *positional parameters*. This lets you write scripts that can be used for many circumstances. Recall that command parameters follow the command itself and are separated by tabs or spaces. An example of a command reference with two parameters is:

```
show first second
```

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

To substitute the positional parameters in the script, use the character **$** followed by the decimal number of the parameter. Consider the following example. First, create two sample files:

```
cat > first
line 1
line two
line 3
<ctrl-D>
cat > second
line 1
line 2
line 3
<ctrl-D>
```

Then, issue the commands

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

Inspect the output carefully. The command **diff** compares two files and prints all lines that differ. In this case, it prints **line two** and **line 2**.

Now, build the script **show**, which uses parameter substitution:

```
ed show
a
cat $1
cat $2
diff $1 $2
.
wq
chmod +x show
```

To demonstrate the effet of **show**, type:

```
show first second
```

Inspect the output and compare it with the output you received earlier.

If you issue the **show** command with fewer than the required number of parameters, the shell substitutes an empty string in its place. For example, using the command

```
show first
```

is equivalent to

```
cat first
cat
diff first
```

where the null string has been substituted for **$2**.

The example above shows the parameter references separated from each other by a space. In some uses, you may wish to prefix a substituted parameter to a name or a number. When more than one digit follows a **$**, The shell picks up the first digit as the number of the parameter. To illustrate, build the shell file **pos**:

```
ed
a
echo $167
.
w pos
q
chmod +x pos
```

Then call the script with

```
pos five
```

and the result will be:

```
five67
```

Shell Variable Substitution

In addition to positional parameters, the shell provides *variables*. You can assign values to variables, test them, and substitute them in commands.

The variable name can be built from letters, numbers, and the underscore character; for example:

```
high_tension
a
directory
167
```

Note that keywords must not be single digits, because the shell then treats them as positional parameters. Be aware that the shell treats upper-case and lower-case letters differently in variable names.

An assignment statement gives a value to a shell variable:

```
a=welcome
```

You can inspect their value with the **echo** command:

```
echo $a
```

The shell substitutes the value of the variable **a** in the **echo** command, which then appears as

```
echo welcome
```

COHERENT responds to this command by printing:

```
welcome
```

Don't forget the **$** when referring to the value.

Notice that the shell looks for special characters in any command that it sees — this includes the *space* character. To avoid problems, enclose the value to be assigned in apostrophes:

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

There are several uses for variables. One is to hold a long string that you expect to type repeatedly as part of a command. If you are editing files in a subdirectory like

```
/usr/wisdom/source/widget
```

you can abbreviate if you set a variable **pw** to:

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

Then simply using **$pw** in a command

```
echo $pw
```

substitutes the long path name.

Another use of shell variables is as keyword parameters to commands. These then can be used the same way as positional parameters. To see how this works, create another script resembling **show**:

```
ed
a
cat $one
cat $two
diff $one $two
.
w show2
q
chmod +x show2
```

To use **show2**, issue:

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

This is equivalent in effect to:

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

Unlike positional parameters, keyword parameters may be several characters in length. If you want some text to follow immediately a keyword parameter, enclose the keyword parameter in braces. To illustrate this, build a command file called **brace**, as follows:

```
ed
a
echo 'with brace:' ${a}bc
echo 'without brace:' $abc
.
w brace
q
chmod +x brace
```

Call the command file with **a** set:

```
a=567 brace
```

The result is:

```
with brace: 567bc
without brace:
```

When used in this way, the keyword parameters must be assigned before the command and on the same line as the command. 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
```

echo produces:

```
value of one is ordinal
```

Variables set other than on the line of a command are not normally accessible to a script. To illustrate, build a parameter display script:

```
ed
a
echo 1 $1 2 $2 p1 $p1 p2 $p2
.
w pars
q
chmod +x pars
```

This will be used to show the behavior of parameters. The parameters to **echo** without a **$** help to read the output. To pass positional parameters, type:

```
pars ay bee
```

The output is:

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

To pass keyword parameters, type:

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

The result is:

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

To illustrate that the setting of **p1** and **p2** did not "stick", type:

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

echo replies:

```
to show
```

This indicates that **p1** and **p2** are not set.

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

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

This replies:

```
1 2 p1 p2
```

By using the **export** command, however, such variables can be made available to commands. The commands

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

produce:

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

This indicates that after the **export** of **p1** and **p2**, they are available to other commands. Once a variable has appeared in an **export** command, its value can be changed without a need to **export** it again.

Command Substitution

By enclosing a command between **`** characters, you can feed its output onto the command line of another command. For example

```
echo `ls`
```

echoes the output of the **ls** command.

Special Shell Variables

When you log into the COHERENT system, it sets the shell variable **HOME** to your *home* or default directory path. If your user name is **henry**, then the command

```
echo $HOME
```

on most systems prints:

```
/usr/henry
```

The change directory command **cd** sets the working directory to the path found in **HOME** if no argument is given.

The shell normally prompts you with **$** for commands, and with **>** if more information is needed. These two prompts are taken by the shell from the variables **PS1** and **PS2**. You can change these if you want different prompts, for example

```
PS1="Fred's Software Palace: "
PS2='!'
```

To have these take effect each time you log in, put the assignment statements in your **.profile** file.

The shell variable **PATH** lists the path names of directories that contain commands. To show the contents of **PATH**, type:

```
echo $PATH
```

It typically will show:

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

This means that the shell looks for a command first in the current directory, then in **/bin**, and, if not found there, then in **/usr/bin**. The path names are separated by ':'. This means that an empty string precedes the first ':', the current directory. Another common setting for **PATH** is:

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

This means that the shell seeks commands first in the current directory, then in '..' (the parent directory of the current directory), then in **/bin**, and finally in **/usr/bin**.

dot . : Read Commands

Similar to the command **sh** is the **.** command. The command

```
. cfil
```

causes the shell to read and execute commands from **cfil**.

This differs from the **sh** command in several respects. First, there's no way to pass parameters to **cfil** with the '.' command. Second, the **sh** command executes another shell to read the commands, whereas '.' simply reads the commands directly. Finally, all the string variables and parameters are accessible by **cfil**.

The command file **good.am** created earlier can be executed with:

```
. good.am
```

This has the same effect. Similarly, the '.' can itself be used within a command file:

```
ed
a
. good.am
lc
.
w third.sh
q
```

Then, the command

```
. third.sh
```

has the same result as the command:

```
sh third.sh
```

Values Returned by Commands

Most COHERENT commands return a value that indicates success or failure. For example, if **grep** cannot find your file, it issues a diagnostic message and returns a value that tells the shell that something went wrong. You can examine this value by typing the command:

```
echo $?
```

This tells you the value returned by the last command executed. Zero indicates success (true), whereas a non-zero value indicates failure (false). Note that this convention is the opposite of that in the C language (a fact that has led to confusion on occasion).

You can use the value returned by a command to affect decisions about executing other commands.

test: Condition Testing

For most commands, the return value is a side-effect of their operation. However, the **test** command's only task is to return a value. This command can test many conditions, and return a value to indicate whether the requested condition is true or false.

The command

```
test -f file01
```

returns true (zero) if **file01** exists and is not a directory. To check if a file is a directory, use:

```
test -d file01
```

test can also test strings. This is useful when you are using parameter substitution. To illustrate, build the following command:

```
ed
a
test $1 = $2
echo 'test 1 & 2 for equal:' $?
test $1 != $2
echo 'test 1 & 2 for not equal:' $?
.
w test.ed
q
chmod +x test.ed
```

Because the '=' is a parameter, be sure to surround it with space characters.

This command file tests its two parameters for equality. Try the commands:

```
test.ed one two
test.ed one one
```

The **test** command has many other options; see the Lexicon entry for **test** for details.

Executing Commands Conditionally

Type the following commands to create two files:

```
cat >file1
line one
line two
line three
<ctrl-D>
cat >file2
line one
two is different
line three
<ctrl-D>
```

Now, compare the files and print the return value:

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

The command **cmp** compares two files byte-by-byte; the **-s** option tells **cmp** merely to indicate whether the files were the same. The command

```
echo $?
```

prints **1** (false) because the files are not the same.

To process a second command based on the result returned by the first, type:

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

The characters **||** signify that the following command **cat** should be executed if the **cmp** command returns a non-zero value, which it will for this example.

The two characters **&&** execute the command that follows them only if the preceding command returns true (zero).

To see how this works, create a third file with the command:

```
cp file1 file3
```

Type the command:

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

This command removes **file3** if **cmp** indicates that **file1** and **file3** are identical. Because **cmp** is preceded by the copy command **cp**, the files **file1** and **file3** are identical, and so **file3** is removed.

Control Flow

Because the shell is a programming language as well as a program, it provides constructs for conditional execution and loops. These are **for**, **if**, **while**, **until**, and **case**. Also, a subshell can be executed within '(' and ')'.

for: Execute a Loop

The **for** construct processes a set of commands once for each element in a list of items.

To illustrate **for**, type the following commands to COHERENT:

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

The items **a**, **b**, and **c** form the list of value that the variable **i** assumes. The shell executes **echo** with **i** assuming each value in turn. The result of these commands is:

```
a
b
c
```

Notice that after you type the line containing **for**, COHERENT prompts with a different character **>** (on most COHERENT systems). The shell does this to remind you that you must type more information. After you type the line containing **done**, the prompt again becomes **$**.

The **for** command is usually used within a script. Also, you can leave off the list of value to the index variable; when you do this, the shell by default uses the arguments typed on the script's command line as the values for the index variable. To illustrate, type:

```
ed
a
for i
do echo $i
echo '---'
done
.
w script.for
q
chmod +x script.for
```

The statement **for i** is equivalent to:

```
for i in $*
```

where **$*** means “all positional parameters”. Notice that two commands are repeated for each value of **i**. Now, call **script.for** with the following command line:

```
script.for 1 2 3 4 test
```

The result is:

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

if: Execute Conditionally

if tests the result of a command and conditionally executes other commands based upon that result. It can be used instead of **&&** and **||**, as shown above. To demonstrate this, first type the command:

```
cp file1 file3
```

This creates **file3** (because we deleted it on the previous page). Then type:

```
if cmp -s file1 file2
then cat file3
fi
```

This means that the shell executes

```
cat file3
```

if **cmp** returns zero (true).

To get the same result as given by the previously illustrated command:

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

with the **if** statement, also use **else**:

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

The commands between **else** and **fi** are executed if the result of the command following the **if** is false or non-zero. Note that there is no command following *then*.

The **elif** statement lets you test several conditions with one **if** statement and act on the one that is true. In general terms,

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

The items labeled *command* and *action* are both commands or lists of commands.

First, the shell executes **command1**. If the result is true, it performs **action1**. If the result from **command1** is not true, the shell then executes **command2**. If its result is true, then it performs **action2**. This process continues so long as none of the commands return a true result. If none of the command results are true, the action following the **else** is executed.

To illustrate **elif**, create a shell script that list on your terminal only one of the three file-name arguments. Use the command

```
test -f name
```

which returns true if *name* is an existing non-directory file.

```
ed
a
if test -f $1
then cat $1
elif test -f $2
then cat $2
elif test -f $3
then cat $3
else echo 'None are files'
fi
.
w cat.1
q
chmod +x cat.1
```

Now, let's exercise **cat.1**. Type:

```
cat.1 file1 file2 file3
cat.1 file3 file2 file1
cat.1 foo bar baz
```

Examine the results.

while: Execute a Loop

Another looping or repetitive shell statement is the **while** statement. The commands

```
while command1
do command2
done
```

first performs *command1*. If its result is true, *command2* is executed, and *command1* is again executed. This process continues until *command1* returns false (non-zero).

until: Another Looping Construct

The construct **until** resembles **while**. For example, the commands:

```
until command1
do command2
done
```

execute *command2* until *command1* returns true (zero).

case: Serial Conditional Execution

The **case** statement resembles the **if** statement in that it offers a multiple choice. To illustrate, type the following script, which lets you choose one of several ways to list the contents of a directory:

```
ed
a
case $1 in
     1) ls -l;;
     2) ls;;
     3) lc;;
     *) echo unknown parameter $1;;
esac
.
w dir
q
chmod +x dir
```

The words **case** and **esac** bracket the entire **case** statement. The effect of the command

```
dir 2
```

is equivalent to:

```
ls
```

Each choice within the **case** statement is indicated by a string followed by **)**:

```
2)
```

indicates what is to be executed if argument **$1** has the value **2**.

The strings that select the choices may be patterns. The choice '*)' signifies that a match can be made on any string. Notice that this resembles the use of * to substitute any file name. An expression of the form

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

in a **case** statement matches any digit from 1 through 9. A list of alternatives can be presented by separating the choices with a vertical bar:

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

Each command or command list in the case choice must be terminated by a double semicolon **;;**.

Summary

The shell is a command programming language that handles simple commands as well as complex commands that can iterate as well as make decisions. Three kinds of substitution are provided to increase the power of your commands.

For more information about the shell, see the tutorial for the shell that follows in this manual. For more information about a given command, see its entry in the Lexicon.

Note, too, that the COHERENT system also includes the Korn shell **ksh**. This is a superset of the Bourne shell described here, and has many features that you may find useful. For information about this shell, see the Lexicon entry for **ksh**.