The Helios Shell

PERIHELION SOFTWARE LTD

September 1991

Copyright

This manual is Edition 1.3, September 1991.

Printed in the UK.
PDF generated in AT (Vienna).

Acknowledgements

The Helios Shell was written by members of the Helios group at Perihelion Software Limited. Helios software is available for multi-processor systems hosted by a wide range of computer types. Information on how to obtain copies of Helios software is available from Distributed Software Limited, The Maltings, Charlton Road, Shepton Mallet, Somerset BA4 5QE, UK. (Telephone: 0749 344345.)

Contents

Chapter 1

Introducing the Helios shell

This guide is divided into two parts: the first part describes simple use of the Helios shell, including the most common activities; the second part describes more advanced use of the shell.

Helios provides a task called the Helios shell, which acts as a command line interface to the operating system. Within the Helios shell the user can give all the Helios commands, including the one-line, built-in shell commands. These shell commands were intentionally designed to be similar enough to Unix shell commands for Unix users to feel at home. However, Unix users should note that not all Unix shell commands are available in Helios and that commands with identical names in both shells may not act in exactly the same way.

The Helios shell also includes various shorthand methods to save typing, and personalisation methods to help users to create their own commands. Several shells can be used simultaneously, if this is desirable.

Chapter 2

Using the shell

This section provides an introduction to the most commonly used features within the Helios shell. It is intended that this section will be sufficient for the novice user. Anyone who requires a more detailed explanation is advised to consult chapter 3.

2.1 Giving commands to the shell

The simplest kind of command is a single word, where a word is one or more characters delimited by blank space, tabs, or by the beginning or end of the line. The entire line is known as a **command line**. The simple command:

```
ls
```

tells the shell to provide a list of the contents of the current directory.

A command may include a whole sequence of words, where the first word specifies the command and the rest qualify its action. These qualifying words or **arguments** include **options**. Options are built-in alternatives associated with a particular command. They are usually keywords of the form '-x' that determine what optional action a command must take. Options are usually given after the command. For example,

```
ls -l
```

tells the shell to provide a list as before, but with even greater detail. Other arguments may include the name of a file to be edited. For example,

```
ls -l myfile
```

tells the shell to give the same sort of details as above, only this time just for the file named 'myfile'. Notice the option '-l' goes before the other argument; this is always the order.

2.2 Filename/command completion

There is a useful feature of the Helios shell that saves a lot of typing at the keyboard. Type the first few letters of a filename, or of a command name, and then press the escape key. The shell automatically completes the name, if it can, by matching the characters typed to the range of names available. Although this feature can be used for names of any length, it is most useful for long names. If the key sequence CTRL-D is used instead of pressing the escape key, a list of possible options or matches for the specified command or directory name is produced.

2.2.1 Name completion using ESC

Press the escape key (ESC) after typing one or more characters, and this will cause the shell to search for a name that begins with that sequence of characters. The shell can determine whether to look for a command name or a filename or directory name by its context and relative position on the command line. If the shell finds a file with a name which begins with the characters typed, it completes the filename and displays the result on the screen. If it finds that more than one file shares the same initial characters, it appends as many characters as it can to form a stub that is common to all the possible files and then signals an error by sounding the terminal's bell. When this situation arises, either complete the file name, or add a few extra characters before re-using the escape key. For example, instead of typing:

```
cd examples
```

the user could enter the following:

```
cd ex<ESC>
```

The escape key then forces the shell to complete the name, `examples`. Similarly, give:

```
da<ESC>
```

and the shell supplies the last two letters to complete the command name **date**. The user can then press the return key to execute **date**. However, give:

```
d<ESC>
```

and the shell sounds a terminal bell to show that it does not know how to complete the name because there is more than one name that starts with the letter 'd'.

2.2.2 Name completion using CTRL-D

As an alternative to the escape key, use the control combination CTRL-D. This combination causes the shell to list all the files with names that begin with the specified stub. It then redisplays the text entered, enabling the user to complete the name with its correct ending.

Suppose, for example, the directory 'fred' contains two files: one is called **test.tst** and the other is **test.txt**. In this case the command line:

```
cat /fred/teESC
```

will be ambiguous as the stub, 'te', refers to more than one file. The shell will attempt to complete the name, adding as many characters as it can before signalling an error:

```
/fred/test.t
```

At this point there are three options: complete the name, add an 's' and use the escape key to force the shell to complete the name, or use CTRL-D to list all the files with names which begin with 'test.t'.

When using CTRL-D, there is no need to worry about using characters that might match more than one name. Take an example used earlier: the letter 'd' followed by ESC. Suppose that CTRL-D had been used instead of ESC:

```
dCTRL-D
```

the shell would have provided a list something like this:

```
date* df* diff* dump*
```

enabling the user to find the correct command, complete it, and execute it.

2.3 Simple shell input and output

This section describes the control of input and output through the use of redirection and pipes.

2.3.1 Redirection

When using the shell, commands are generally issued by typing them in at the keyboard. The keyboard is the default method of input; it is what is meant by **standard input** (**stdin**). The shell usually returns any output by 'printing' it on the screen, unless it is requested to do otherwise. This default output is known as the **standard output** (**stdout**).

For example, type the command:

```
ls
```

and the contents of the current directory appear on the screen like this:

file1 file2 file3

followed by the current prompt on a line by itself. If there is no output, the shell indicates that it has executed the command by returning a prompt to the screen. The default prompt is a percentage sign:

```
%
```

The user may wish to redirect the output to somewhere other than the standard output. This can be done by including a **greater than** symbol > within the command, followed by the name of the destination. To send the contents of the current directory to the file **contents**, type:

```
ls > contents
```

It does not matter if the >symbol is surrounded with blank space or not. The shell treats >as a separate word with a meaning of its own and not as part of the command or its argument. For example,

```
ls>contents
```

would be equally acceptable. If the file **contents** does not exist, it is created; otherwise, this command line will overwrite everything in **contents** with the output of **ls**. To ensure that the output is added to the end of an existing file instead of overwriting it, give a **double greater than** symbol >>instead.

To take input from somewhere other than the standard input, precede the name of the input source (often this is the name of a file, the contents of which are to be directed to the **stdin** of the command), with a **less than** symbol <. Again, it can be given immediately before the name or separated from it it with blank space.

2.3.2 Pipes

Whereas redirection can be used to force a process to communicate with files instead of with a terminal, pipes can be used to make processes communicate directly with other processes. This is particularly useful when running a command with an output which needs to be sent directly to the input of a second program for further processing. As an example of pipelining, consider the following:

```
ls | wc -w
```

In this case, the output from **ls** (a list of all the files in the current directory) is being used as the input to **wc** (*word count*). **wc** with the **-w** option counts the number of words and displays the result on the screen. This executes **ls** and **wc**, simultaneously piping the output (**stdout**) of **ls** to the input (**stdin**) of **wc**. The overall effect of this command line is to count the number of files in the current directory.

2.4 The history facilities

The shell stores in memory each command typed into it, and places these commands on a list of past events. The shell allows the user to recall this list to review previous actions, to repeat a command, or to edit a previous command before executing it again. The user can even specify the number of previous commands which will be retained in the list by assigning a value to the shell variable **history**. If **history** has not been set, or has been assigned a value of 0, only the most recent command will be retained.

Each event on the history list is associated with a number, starting with 1 and increasing sequentially. As each event occurs, it is added to the list. The number appears to the left of the event when the events are listed with **history**. For example, the last four events might look like this when listed:

```
100 date
101 ls -l
102 cd newfile
103 history
```

(Notice that **history** is included as the most recent event, number 103). To find out the current event number without resorting to the **history** command, place a `!` character in the prompt string. The prompt string is merely the first word of the shell variable **prompt**. For example, type:

```
set prompt="!  :  "
```

and the shell prompts with:

```
104 :
```

or whatever the event number is at present, followed by a colon. The shell then continues to increment this number with each new event.

To access an event and re-introduce it as current input, use the exclamation mark (!) character. A pair of these characters followed by **newline** tells the shell to access the last event, **echo** it to the terminal, substitute it as the current line of input and repeat its action. To do this with any other event on the list, be more specific. The ! character can be followed by the number of the event to be accessed. To re-execute the **date** command, type:

```
!100
```

The shell will display the event associated with that number and execute it:

```
date
FRI JAN 29 13:22:07 1991
105:
```

An event can be referred to by its position in relation to the current event using a minus sign followed by an integer, where the integer refers to the number of events to go back on the list. For example, suppose the user knows that the current event number is 106 (because the current prompt is set up to reflect event numbers) and the user types `!-6`:

```
106 :  !-6
```

The shell will act as if they had typed `!100` (in other words, 6 from 106 translates as event number 100).

An event can be retrieved from the history list by following ! with as many characters of the command name as will specify it exactly. The shell searches backwards from the current event until it finds the first match in the history. It then repeats that command. For example, to repeat the last command beginning with 'd', type:

```
!d
```

Take care that the last command with this initial letter is the command sought. It may be useful to use two or more letters after the ! to ensure that the correct event is pinpointed and retrieved.

The above examples simply show how to capture a past event and re-introduce it verbatim as current input. The user can, however, retrieve a past event from the history list, make some changes and then re-execute the edited command line. Certain control-key combinations allow the retrieval of command lines from the history list and then position the cursor for inserting text into the line or deleting text from it. A list of these control keys is given below.

Note: the 'up' and 'down' arrow keys can also be used to scroll through the history list.

2.5 Wildcards

Whenever a filename can be given, wildcards can be used as a shorthand method for entering the name. One of these wildcards is `*`, which instructs the shell to replace that character with as many characters as are necessary to produce a valid filename. Another wildcard character, `?`, has a similar meaning, but will only be replaced by a single character. [*abc*] can also be used to specify the match of a single character with one of a group of characters, but this type of wildcard will only match with a character if it is listed within the brackets.

When substituting for wildcards, the shell will use any characters which are adjacent to the wildcard to generate a pattern which will be compared with each of the filenames in a directory. A list of filenames which match this pattern is then generated and inserted into the command line in place of the word which contained the wildcard.

To determine which directories are searched during wildcard substitution, consider the wildcard's position within the command line. If the character appears within a command name, each of the directories which have been assigned

CTRL-A Moves the cursor to the start of a line.

CTRL-B Moves the cursor to the left by one character.

CTRL-E Moves the cursor to the end of a line.

CTRL-F Moves the cursor to the right by one character.

CTRL-L Moves the cursor to the left by one word.

CTRL-N Replaces the current line with the next line in the history list. This can be used repeatedly to display each line in the list.

CTRL-P Replaces the current line with the preceding line in the history list. This can be used repeatedly to display each line in the list.

CTRL-R Replaces the current line with one from the history list which begins with the same word as at the cursor's current position. When CTRL-R is used on a blank line the last command is fetched from the history list.

CTRL-U Deletes the entire line.

CTRL-W Moves the cursor to the right by one word.

CTRL-X Deletes the word at the cursor.

to the shell variable **path** will be searched. If, however, the wildcard appears within a command's argument, only the current directory will be searched. Listed below are a few examples of wildcard usage.

`*`

General wildcard. Represents a series of characters. For example, `rm *` deletes all the files in the current directory, `rm *.c` deletes all the files in the current directory with names which end in *.c* and `rm Pe*.c` deletes all the files in the current directory with names which begin with *Pe* and end with *.c*, such as *Perihelion.c*.

`?`

Wildcard for one character only. For example, `ls ?a.c` lists all the files with names which consist of four characters, where the last three characters are known to be *a.c*. This could match *aa.c, ba.c,* etc.

`[...]`

Wildcard for one character, but it only matches with the specified characters. For example, `od t.[abc]` displays the contents of the files *t.a, t.b* and *t.c*.

2.6 Running tasks in the background

The Helios shell allows the running of a task, or pipeline of tasks, without the need to wait for its completion, by using the ampersand character, **&**. This metacharacter, when appended to a pipeline, causes the entire pipeline to be run in the background, which frees the shell so that it can process the next command line, which may also be run in the background.

Whenever a task is executed in the background, the shell assigns a job number to that task. This job number is displayed on the standard output device when the task begins to execute and it is also displayed alongside a suitable message when the task has completed. As each task has a unique job number, many commands use it to identify a particular task. The **jobs** command for example, lists all of the current tasks and their job numbers.

It should be noted that the output of tasks running in the background is directed to **stdout** unless otherwise redirected, so there is a possibility of interference with the output of tasks running in the foreground. As an example of how to use the background facilities, consider the following:

```
fgrep Helios huge > Helios &
```

In this example **fgrep** is being used to search for all occurrences of the word 'Helios' within the file 'huge'. As this task may take some time to complete, it is useful to be able to run it in the background, so that other tasks can be completed simultaneously. When this task begins executing, the shell will display the following type of message on the screen, followed by a shell prompt:

```
[1] 1234
```

This shows that the task has been assigned the job number 1 and that it has a process identification number of 1234. When the task terminates, the following message will be displayed:

```
[1]      Done    fgrep Helios huge
```

indicating that job number 1 completed successfully. Several tasks can be executed simultaneously either by using many command lines, such as that described above, or by constructing a single compound command line. The following example shows how two background processes and one foreground process can be invoked from within a single command line:

```
fgrep Helios huge > helios & fgrep Perihelion huge >
perihelion & echo hello
```

In this example, two instances of the **fgrep** command are being executed in the background (the >symbol shows that all output is to be sent to a file instead of to the screen) and one **echo** command is being executed in the foreground. The shell lists the job and the process identification for each of the two background tasks and then executes each command:

```
[1] 1234
[2] 1235
```

Similar messages to that shown above will be displayed when each process terminates.

2.7 Aliases

In its simplest form, the aliasing mechanism allows alternative names to be associated with a command name. This is particularly beneficial to users who are not familiar with Unix-style commands, as each command can be assigned an alternative name which more closely matches that of a more familiar operating system. A particularly good example of this is the **ls** command. Users who are more familiar with the MS-DOS equivalent, **dir**, may therefore alias **dir** to be equivalent to **ls**, so that either name can be used:

```
alias dir ls
```

The command **dir** can now be used to list the contents of the current directory. An extension of this idea is to assign an alias to a command and its arguments. By default, the **ls** command produces a list of the files in the current directory. However, a command that lists all the files along with their creation dates and other information may be preferable to some users. This effect can be achieved by invoking **ls** with the **-l** option, or by assigning an alias to produce a command with default behaviour which gives this more detailed list of files:

```
alias ls 'ls -l'
```

The use of the above alias will cause all references to the command called **ls** to be replaced with **ls -l**. (Notice the use of the quote characters to ensure that the command **ls -l** is treated as a single word.)

2.8 Shell variables

The Helios shell supports a list of variables which can be assigned values. These variables may then appear within command lines where they will be substituted for their values. Some of these variables are created and maintained by the shell (such as **cwd**, which always contains the name of the current working directory) but others can be created or removed at will by the user. A particularly useful application of shell variables is as a shorthand for long filenames. Consider, for example, the following assignment:

```
set h=/usr/perihelion/helios
```

The above command results in the creation of a shell variable **h**, which is assigned the value **/usr/perihelion/helios**. If the shell finds the character sequence **$h** within any future command line, it performs a variable substitution which causes those two characters to be replaced with the value of the variable **h**. The following two command lines are therefore equivalent:

```
emacs $h/test.c
emacs /usr/perihelion/helios/test.c
```

Variable names may consist of up to 20 alphanumeric characters. If such names are used, it may be necessary to use braces, **{}**, to enclose the name whenever it is used within a command line. This allows the shell to distinguish between characters which form part of the variable name and those which form part of the following text. Braces can be omitted if the variable name is followed by any non-alphanumeric character, including a blank space:

```
set helios=/usr/perihelion/helios

cd ${helios}/bin
ls $helios
```

A shell variable can be assigned new values by reapplying the **set** command, and they can be deleted with the **unset** command.

2.9 Using shell windows

A child shell is created from within the original parent shell with **wsh**. This command creates a window and then executes a shell within it. This newly created shell is independent of other shells in other windows and it can be used to execute commands concurrently with them.

The way in which these windows are displayed on the screen is dependent upon which I/O processor is being used, and so is their user interface. Transputer systems that use an IBM PC (or equivalent) may not be able to display more than one window at a time. For these systems a virtual windowing system is used whereby each window occupies the entire screen. A process running in any of these windows may write to the screen at any time, but only the currently active window will receive input from the keyboard. To change the currently active window and so redirect the standard input to another window, use ALT-F1. Repeated use of this key combination will cycle through the available windows.

For systems that are capable of displaying windows, each call to **wsh** results in a new window being created on the screen. The window that is active can be identified by a highlighted title. A window can be selected by moving the mouse over it or by clicking the mouse on the window, depending on the window manager used.

Chapter 3

An advanced description of the shell

The Helios shell is executed whenever a user logs into the system, or it can be started from within an existing shell by using either of the commands **shell** or **wsh**. The first stage is to execute each of the commands that are listed in the shell script, **cshrc**, which should be in the user's home directory. If the shell is a login shell, the commands listed in the file 'login' will also be executed. When creating a child shell by using the command **shell**, various options and arguments can be specified, as follows:

```
shell [ -cefinstvxVX ] [ arg1 ...argn ]
```

The argument strings generally do not affect the behaviour of the shell and are stored in the shell variable **argv**. The option flags are used to modify the behaviour of the shell and are listed below.

Option **Action**

c

Assumes the (single) following argument to be a command line which is to be executed. Any arguments which follow are copied into the shell variable **argv**.

e

Causes the shell to terminate if any command executed from within the shell produces an error.

f

Prevents the shell from reading and executing the commands in the file **cshrc** when it is invoked.

i

Causes the shell to run interactively and to display prompts. Command line editing is then available and the prompt is displayed. Without this

option the shell tries to discover what type of input has been specified. This option is an override mechanism.

n

Causes all commands entered into the shell to be parsed but not executed. This facility is useful for checking the syntax of shell scripts.

s

Instructs the shell to take input from the standard input device.

t

Causes the shell to read just one line of commands, and then to terminate.

v

Causes the shell variable **verbose** to be set. All command lines will be **echoed** to the screen after history substitution has been performed.

x

Causes the shell variable **echo** to be set. All commands will be **echoed** to the screen before execution.

V

Causes the shell variable **verbose** to be set. It differs from **v** (see above) in that this option sets the verbose mode before the shell reads **cshrc**; the other option does not.

X

Causes the shell variable **echo** to be set. It differs from **x** (see above) in that this option sets the **echo** mode before the file **cshrc** is read; the other option does not.

3.1 Characters and words

The shell parses a command line by first splitting the line into its constituent words. Words, with certain exceptions, are characters that are delimited by blank spaces or tabs, or enclosed in pairs of double quotes.

The exceptions are special characters that form separate words even if they are not delimited by blanks or tabs. These characters, known as the parser metacharacters, are as follows: `& | ; < > ( )`. When doubled into metacharacter pairs such as the following, `&& || << >>` each pair forms a single, separate word.

To make one of these metacharacters part of another word and thereby remove its special meaning, place the special backslash character \ immediately before it. This character makes the shell treat a subsequent character as literal text without any special meaning. In fact, it has this effect on any single character that follows it. For example, place a \ before a **newline** character

(carriage return / linefeed), and the shell translates that character as a blank space because its usual meaning has been removed.

The Helios shell also understands the concept of a string, which it treats as part of a word. A string can be made up of one or more characters, including white space, enclosed in single or double quotes. Because the shell treats strings as part of a single word, metacharacters in a string cannot form separate words and spaces and tabs fail to have their usual effect. As explained previously, cancel the effect of a **newline** with \; nevertheless, attempt to use **\newline** in a string and a real **newline** character is produced. The differences between strings enclosed in single quotes and strings enclosed in double quotes are explained in section 3.2.

When the user inputs to the shell indirectly through a shell script (a shell script is a text file of shell commands), rather than directly from the terminal, the special character # can be used to introduce a comment. When the shell finds a #, it treats any subsequent text on the same line of input as a comment and ignores it. To turn off the special meaning of this character, place the literal character \ immediately before it.

3.2 Strings

Character sequences enclosed in single or double quotes form part of a word. A complete word can be formed entirely from a string delimited by white space, or from character sequences which contain a string. The quote characters will not form part of the word unless they are preceded by a backslash character (\). Character sequences enclosed in single quotes are subject to history substitution only. History substitution can be used anywhere on a command line and always occurs in strings, unless the history substitution characters are preceded by a backslash. Variable substitution will only occur in double-quoted strings and alias substitution never occurs in a string. Notice that **!** must be preceded by a backslash to nullify its effect. The only exceptions are that **!** does not take effect when followed by **space**, **end-of-line**, or `=`. For example, the **!** would not generate any history substitutions in `set prompt="!  "`.

3.3 Metacharacters

The following metacharacters are listed from left to right in order of precedence:

```
| ; || && & ( ) < > << >>
```

such that `|` is the most senior (that is, takes precedence over everything else) and `&` is the most junior. `(` and the following metacharacters do not have any precedence. These characters are listed below, together with their functions.

Character **Action**

`|`

Pipes the output from one process to the input of another. For example, the command `sort datalist | cat` sorts the file **datalist** and then it displays the output.

`;`

Used to separate sequences of pipelines (or single commands). The shell will fully execute each pipeline before executing the next. For example, the line `ls ; cat text` will cause **ls** to be executed, followed by **cat**.

`||`

Used to separate two pipelines (or single commands). The second pipeline will only be executed if the first fails. This metacharacter is analogous to the logical OR operator. For example, in `ls || echo ls failed`, the **echo** command will only be executed if **ls** fails. It exits with a non-zero return code.

`&&`

Used to separate two pipelines (or single commands). The second pipeline will only be executed if the first succeeds. This metacharacter is analogous to the logical AND operator. For example, in `ls test && cat test`, **cat** will only be executed if **ls** succeeds. It exits with a zero return code.

`&`

Puts the task into the background. For example, `sort bigfile &` runs the sort program on the file **bigfile** as a background task. It does not wait for it to terminate.

`(`

Used with a terminating `)` to group together a sequence of commands to be run in a child shell. For instance, `(cd;pwd)` lists the full pathname of the user's home directory, but since it is run in a child shell, the current directory of the parent shell is unchanged.

`)`

Used with an opening `(` to group together a series of pipelines. See the entry for `(`.

`>`

Redirects output from standard output (**stdout**). `date > now` for example, redirects the output of **date** from **stdout** to a file called **now**. If **now** does not exist, it is created. If it does exist, it is overwritten.

`<`

Redirects input from standard input (**stdin**). For example, `cat < text` redirects the input of **cat** so that it is taken from the file called **text**. Since **cat** takes a filename, the same effect can be achieved with **cat** directly.

`>>`

Redirects output from **stdout** in the same way as >, except that it appends files instead of overwriting them. For example, if the file called **now** exists, `date >> now` redirects the output from **stdout** and adds it to the end of **now**.

`<<`

Causes the command to read its input from the input of the shell until the word following the <<symbol in the command line is encountered. For example, `cat << end` makes **cat** read its input from the shell's input until it finds the word **end** on a line by itself.

3.4 Expressions

A number of commands that are built into the shell can take expressions as arguments. These expressions may be constructed using arithmetic and logical operators which are similar to those used in the C programming language. In the following list the order of operator precedence increases from left to right, with square brackets indicating groups of operators of equal precedence:

```
|| && | ^ & [== != =~ !~][<= >= < >][<< >>][+ -][* / %] ! ~ [( )]
```

In cases where operators of the same precedence occur several times, they are evaluated left to right. A full list of valid operators can be found in Table 3.1.

3.5 Command execution

The shell takes input, acts upon any commands it finds and produces output. In doing so, it may perform certain substitutions and transformations. These actions are listed below in the order in which they may occur.

1. (**History substitution**.) Using previous command input to produce new input.
2. (**Alias substitution**.) Checking alternative command names for built-in commands.
3. (**Variable substitution**.) Substituting a value for a named variable.
4. Expanding the name of a command.
5. Expanding the name of a filename given as an argument.

Each of the actions listed is described in the sections to follow.

Operator	Description
`||`	logical OR
`|`	bitwise OR
`^`	bitwise exclusive OR (XOR)
`&&`	logical AND
`&`	bitwise AND
`==`	equivalent
`!=`	not equivalent
`=~`	tests for match with specified pattern
`!~`	tests for no match with specified pattern
`!`	logical NOT
`~`	bitwise NOT
`<=`	less than or equal to
`>=`	greater than or equal to
`<`	less than
`>`	greater than
`>>`	shift right
`<<`	shift left
`+`	addition
`-`	subtraction
`/`	division
`*`	multiplication
`%`	modulus

Table **3.1** Operators

3.6 The history list and history substitution

The Helios shell places words from previous commands into a list so that they can be retrieved, edited and executed at a later stage. This is particularly useful for repeating previous commands, correcting typing errors in commands or executing commands which are similar to those which have already been used.

The Helios shell provides two levels of access to the history list. Firstly there is an interactive history mechanism which allows the list to be accessed and edited by using various control key combinations. The list can also be accessed and manipulated by using special characters which are placed in a command line.

When the shell gives a prompt, respond by entering a line of text or use the control keys which are listed below. These control keys allow the current command line to be overwritten with a previous command line from the history list. This line may then be edited by using other control keys to position the cursor ready for text to be inserted or deleted. The edited command line can then be executed by pressing the return key. There is no need to move the cursor to the end of the line.

CTRL-A

Moves the cursor to the start of the line.

CTRL-E

Moves the cursor to end of the line.

CTRL-P

Replaces the current line with the preceding line in the history list. This can be used repeatedly to display each line in the list.

CTRL-R

Replaces the current line with a line from the history list which begins with the same word as at the current cursor position. If CTRL-R is used on a blank line, the last command is fetched from the history list.

CTRL-U

Deletes the entire line.

CTRL-W

Moves the cursor to the right by one word.

CTRL-X

Deletes the word at the cursor's current position.

The shell is instructed that a history substitution is to occur in a command line by using the character !. This character may occur at any point in a line and is followed by one or more qualifying characters. When the command line is submitted for execution, the shell will attempt to replace these characters with a word (or a group of words) extracted from the history list. The command line is then **echoed** to the screen and executed.

There are three types of qualifying character which can be used with !. The first of these is a number which corresponds to the event number of the required entry in the list. All entries in the history list are assigned an event number by the shell, such that the first entry is numbered 1 and subsequent entries are assigned progressively larger event numbers (2,3,4,...). The character sequence !100 therefore refers to event number 100 in the list. Similarly, if the current event number is known, an offset can be specified instead of an absolute event number. If the current event number is 106, !–6 will therefore refer to event 100.

The '!' character itself can also be used as a qualifying character. In this case, whenever the shell encounters the character sequence **!!**, it substitutes it for the last entry in the history list. A command line consisting solely of these two characters is therefore an instruction to the shell that it should repeat the last command.

To select a specific word from a previous command, use the colon character. When used in conjunction with the '!' character and an optional event number, it causes the shell to substitute these characters for the specified sequence of

words which are stored in the history list. A full list of designators which can be used with the colon character is given below, but as an example, consider the following command line:

```
echo !100:2
```

This will **echo** to the screen the second word from event number 100.

Designator	**Meaning**
n	*n*th argument. Argument 0 is the first (command) word.
^	The first argument (equivalent to '1').
$	The last argument.
*	All arguments. If the event relates to a single word, this will be null.
n-*m*	A range of words. For example, 0–2 means words 0 to r2.
-*m*	Equivalent to 0–*m*.
n-	All arguments from argument *n* onwards (excluding the final word).
*n**	All arguments from argument *n* onwards (including the final word).

In addition to this, it is possible to omit the colon character whenever the qualifying character is ^, $, * or -. The previous event can be recalled with one or more words changed by use of ^ alone. This is achieved by ensuring that ^ is the first non-blank character on a command line. The shell then interprets all characters up to the next ^ as a pattern which is to be replaced by the characters which follow the second occurrence of ^. For example,

```
^helois^helios
```

replaces the first occurrence of `helois` in the last command with `helios` and then executes the result. To append additional text to a line, use a third ^, as in:

```
^helois^helios^hello
```

This command will substitute `helois` for `helios` and append `hello`. A complete or partial listing of all events currently in the history list, with their associated event numbers, can be obtained by using the **history** command.

3.7 Aliases

After performing the history substitutions, the shell performs alias substitutions on the command line. Aliases are alternative names which the user may assign to almost all of the commands which are available from within Helios. They are created by using the **alias** command and removed from the shell's list of aliases by using **unalias**. For example,

```
alias dir ls
```

assigns the alias **dir** to **ls** so that all references to the command **dir** will now be treated as an invocation of **ls**. Similarly,

```
unalias dir
```

causes the shell to forget the alias.

When substituting a command name for an alias, the Helios shell allows the full facilities of the history mechanism to be employed. Furthermore, since the shell adds a command line to its history list before it performs alias substitution, all references to the previous command actually refer to the current command before alias substitution was performed. For example, consider the case where **echo2** had been defined as an alias:

```
alias echo2 'echo \!:2'
```

meaning that **echo2** has become an alias for the command which **echoes** the second argument of the previous command to the screen. (The quotes are needed to ensure that **echo** and its argument are treated as one word and the '\' before the '!' character is needed to prevent the shell from performing the history substitution at the time the alias is defined.) Type:

```
echo2 one two three
```

and the shell will enter the command into the history list and substitute **echo2** and its arguments for the aliased command. History substitution will then be reapplied to the resulting command line, so that the command actually executed is as follows:

```
echo two
```

In cases where the alias does not make use of the history facilities, only the command name is substituted. The arguments are left unchanged. As an example, consider the following:

```
alias dir ls
dir /helios
```

which is equivalent to:

```
ls /helios
```

3.8 Variables and variable substitution

When the shell is executed, it receives a list of variables from the operating system, and these provide the shell with information about the environment in which it is running. These environment variables are then copied into the shell's own variables and are passed on to any program that is executed from within the shell. These variables are listed below.

HOME The user's home directory.

USER The user's login name.

PATH The list of directories which are searched when a command is executed.

TERM The terminal type.

SHELL The full pathname for the shell.

EDITOR The editor to be used.

When the shell variables **user**, **term**, **home** and **path** are changed, the shell changes the corresponding environment variables **USER**, **TERM**, **HOME** and **PATH** to reflect that change. The shell also provides two commands for manipulating these and the other environment variables directly. These commands are described below.

`setenv` *name value*

Sets the named environment variable to the value of the string value.

`unsetenv` *pattern*

Unsets all environment variables with names which match the pattern.

`printenv`

Lists the environment variables and their values.

3.8.1 Shell variables

The Helios shell maintains a list of internal variables which can have one or more words as a value. Some of these variables are maintained directly by the shell, but others can be created or destroyed by the user. The value of any of these variables can be included in a command line at any point. The shell then uses variable substitution to substitute the variable's name for its value. The variables listed below are predefined and have special meaning to the shell.

argv

The arguments of the command line which was used to invoke the shell, where the first argument is *$argv[1]*.

cdpath

Used to specify a list of directories which are used by **cd** when it is searching for a subdirectory.

cwd

The full path name of the current directory.

echo

When set, this will instruct the shell to **echo** all commands on the standard output device before they are executed.

history

The number of commands remembered in the history list. The last command to be executed is always remembered by the shell, regardless of the setting of this variable. Note that large history lists may result in the shell running out of memory.

home

The home directory. The filename expansion of ~ is the value of this variable. **cd** without any argument will arrive in this directory.

ignoreeof

If set, this variable instructs the shell to ignore the end-of-file characters (CTRL-D) which would otherwise terminate the shell.

noclobber

If set, this variable acts as a safeguard during redirection of input and output. It instructs the shell to prevent files from being overwritten by >and ensures that a file is already present before it is used with >>. Suitable error messages are generated if these conditions are not satisfied.

noglob

If set, this variable prevents the shell from performing filename expansions specified by metacharacters such as *.

nonomatch

If set, this variable will prevent the shell from generating an error when a filename expansion fails to match with a file. In these circumstances, the unexpanded pattern is passed on to the command. Note that this variable does not affect the error reporting for badly formed patterns.

path

Each word of the **path** variable specifies a directory in which commands are to be sought for execution. An empty list is assumed to refer to the current directory only. If this variable is not set, commands will only be executed if their full pathname is specified.

prompt

The value of this variable is used as the prompt sign by the shell. For example, the command `set prompt="$ "` sets the prompt to the two-character string '$ '. Two special characters may be used in the prompt string. The character '!' is expanded to the current history event number; the character '?' is expanded to the current window name used by the shell. This is useful when running multiple shells through the **wsh** command on a machine without a graphics screen.

status

The value of this variable is maintained by the shell and will always represent the exit code which was returned by the previous command. A non-zero value shows that the command failed; 0 shows that it terminated correctly.

verbose

If set, this variable causes the shell to **echo** each word of a command line to the standard output device immediately after it has been history-substituted.

3.8.2 Assigning values to shell variables

To assign a value to a shell variable, use the **set** command. This command can be used to create and initialise a new variable or to assign a new value to an existing variable. It can also be used to display the current values of all variables:

set

displays all variables and their values.

set *name*

sets *name* to equal a single word of length *0* (a null word).

`set` *name* `=`*value*

assigns the string *value* to the variable *name*.

Note that the *value* of a variable can be any sequence of characters delimited by white space. If white space is required to form part of the value, quotes must enclose the sequence of characters. This is illustrated below.

set prompt=$

sets prompt to $.

set prompt="$ "

sets prompt to $.

The variables can also be assigned a list of values, by enclosing the entire group of value strings in brackets. The following assignment can therefore be used to assign the three values: **hello**, **hello there** and **world**:

```
set strings=(hello "hello there" world)
```

3.8.3 Variable substitution

The value of a shell variable can be substituted into a command line at any point by using the character $ followed by the variable's name. If the variable's name consists of more than one character and it is not delimited by white space, the name should be enclosed in braces {}. The braces can be used in other circumstances, but they are not strictly necessary. If the variable **path** is assigned the value `/helios/bin/`, these command lines have the following meanings:

```
$path                  = /helios/bin/
${path}command         = /helios/bin/command
hello${path}command    = hello/helios/bin/command
```

If a variable has been assigned a list of values, any one of those values can be selected by using an index with the name of the variable. If **path** is assigned three values (`/helios`, `/helios/bin` and `/helios/bin/cmds`), these command lines have the following meanings:

```
$path
= /helios /helios/bin /helios/bin/cmds

$path[0]
= not valid - shell generates error message

$path[1]
= /helios

$path[3]
= /helios/bin/cmds

$path[4]
= not valid - shell generates error message

hello${path[2]}world
= hello/helios/binworld
```

Other character sequences in the command lines can also be used to access a variable or to obtain information about its current status:

`$#`*name* `${#`*name*`}`

This character sequence is replaced by the number of words that are assigned to the variable **name**. If **path** has been assigned the three values given earlier, `echo $#path` will **echo** 3 to the screen. When the shell is taking its input from a file (instead of from the terminal), the character sequence

```
$0
```

will be replaced by the name of that file. If the input is not being taken from a file, an error will be generated. (See also `$?0`.) The sequence

`$`*number*
`${`*number*`}`

is equivalent to `$argv[` *number* `]`. This character sequence can be used to determine if a variable has been set:

`$?`*name* `$?{`*name*`}`

If the variable has been set, the sequence is replaced by 1; otherwise, it is replaced by 0. This character sequence can be used to determine if the shell is taking its input from a file:

```
$?0
```

If a file is being used, the character sequence is replaced by 1; otherwise, it is replaced by 0. (See also the entry for `$0`.) The following character sequence will be substituted by words read from the standard input device:

```
$<
```

This sequence is particularly useful for obtaining keyboard input from within a shell script. A command line containing shell variables is entered into the history list in its expanded form. Multiple variable substitutions in a single line are allowed, but recursive substitutions are not supported.

3.9 Redirection

Any command which communicates with the user by means of the standard input and output devices can be made to communicate with files instead. This can be achieved by using metacharacters in a command line and it is particularly useful for sending large amounts of output to a file for casual viewing with a text editor. The metacharacters and their filename arguments can usually be placed anywhere on a command line.

`<` *name*

This redirects input to come from the file **filename**, instead of from **stdin**. The filename will be subjected to variable, alias and wildcard substitution. For example,

```
cat < /helios/example/hello.c
```

causes **cat** to read its input from hello.c instead of from the keyboard,

```
cat << word
```

takes input from the standard input device, **stdin**, until the word (**word**) is encountered. For example,

```
cat << end
```

causes **cat** to take its input from the keyboard and to interpret the first occurrence of **end** as the end of input.

>filename *>!filename* *>&filename* *>&!filename*

These redirect output to the named file. Those metacharacter sequences which do not include the ampersand character redirect output that would otherwise be sent to the standard output device, **stdout**; those which include the ampersand character redirect the standard error output, **stderr**. If the named file does not exist, it is created. If it does exist it is overwritten. By setting the shell variable **noclobber**, the shell is prevented from overwriting a file. The `!` character overrides the effects of **noclobber**. The filename is subjected to variable, alias and wildcard substitution. For example,

```
cc -o outfile infile >& errors
```

The file **errors** now contains the compiler's error messages. If the error messages relate to a file, **emacs** could be used to display these messages and to edit the file simultaneously by using the editor's windowing facilities.

>>filename *>>&filename* *>>!filename* *>>&!filename*

The above redirect output and append it to the named file. These metacharacter sequences are similar to those constructed from a single >character, but they append output to the named file instead of overwriting its current contents. If the shell variable **noclobber** is set, an error is generated if the file does not exist. This error can be suppressed by using the metacharacter sequences which use `!`.

3.10 Control flow

A number of the shell's built-in commands are for specifying the order in which commands are executed. These commands are similar to the control flow commands which occur in many high-level programming languages, such as C. They are normally used in shell scripts, but they can be entered from a terminal. If a terminal is used, the commands following the control flow constructs may or may not be executed when they are entered. If the shell will not execute the next command, the prompt will be changed to a question mark (?).

3.10.1 if-then-else

This sequence of commands can be used to select a group of commands for execution. The syntax for this construct is as follows:

```
if ( expression ) then
   command
   ...
   command
else
   command
   ...
   command
endif
```

If *expression* evaluates to any non-zero value, all commands up to the following **else** or **endif** commands will be executed. If the expression evaluates to zero, all commands between **else** and **endif** will be executed. None of the commands in this construct can be pipelines or lists of commands separated by a semi-colon. If nested if-then-else constructs are used, only one **endif** is needed as a terminator:

```
   if ($term == "ibm") then
         echo This is an IBM
      else
         echo This is not an IBM
   endif
```

3.10.2 switch

This construct can be used to select one of many commands for execution. The syntax is as follows:

```
switch (string)
case <string-1>:
        command
        ...
        command
breaksw..
case <string-n>:
        command
        ...
        command
breaksw
default:
        command
        ...
        command
breaksw
```

```
endsw
```

The construct allows a unique label (case prefix) to be applied to each group of commands so that the group is only executed if the label matches the string. If none of the case labels match with the argument string, the commands following the optional default label will be executed. The argument string will be subjected to both variable and wildcard expansion, but if wildcards are used and the pattern matches with more than one file, an error is generated. The case labels may also contain wildcards and shell variables, but wildcards in this context will be tested for a match with the argument string and not with filenames. For example:

```
switch (hello)
   case hi:
      echo this should not appear
   breaksw
   case *llo:
      echo this should appear
   breaksw
endsw
```

3.10.3 while

This construct is used to repeat a group of commands while the specified expression evaluates to zero. The syntax for this construct is as follows:

```
while (expression)
      command-1
      ...
      command-n
end
```

Consider the following as an example of its use:

```
set i=10
   while ($i != 0)
   @i = $i - 1
   echo Current value of i is $i
end
```

3.10.4 foreach

This command assigns a series of values to a shell variable and executes a sequence of commands between each new assignment. The syntax for this construct is as follows:

`foreach` *name* (*wordlist*)
        command-1
        ...
        command-n
`end`

As an example of its use, consider the following:

```
foreach person (andy alan paul tim)
   echo $person
end
```

where each of the four names is written to the standard output device.

Index