5

The Shell

The Function of the Shell

The shell is the OS-9 command interpreter program. The shell translates the commands you enter into commands the operating system understands and executes. This allows you to use commands such as dir, copy, and procs without knowing the complex machine language OS-9 understands.

The shell also provides a user-configurable environment to personalize the way OS-9 works on your system. You can use the shell to change the shell prompt, send error messages to a file, or backup your disk before you log out.

The shell command starts the shell program. This command is automatically executed following system startup or after logging on to a timesharing terminal. When the shell is ready for commands, it displays the prompt:

```
$
```

This prompt indicates that the shell is active and waiting for a command from your keyboard. You can now type a command line followed by a carriage return.

A number of options are available to the shell. By default, some are automatically turned on following startup or log on. The available shell options are:

Option	Description
-e=<file>	Prints error messages from <file>. If no file is specified, /dd/sys/errmsg is used. Without this option, the shell prints only error numbers with a brief message description. Each error is described in the appendix on error codes.

Option	Description
-ne	Prints no error messages. This is the default option.
-l	The logout built-in command is required to terminate the login shell. <eof> does not cause the shell to terminate.
-nl	<eof> terminates the login shell. <eof> is normally caused by pressing the <Esc> key. This is the default option.
-p	Displays prompt. The default prompt is a dollar sign ($).
-p=<string>	Sets the current shell prompt equal to <string>.
-np	Does not display the prompt.
-t	Echoes input lines.
-nt	Does not echo input lines. This is the default option.
-v	Verbose mode: displays a message for each directory searched when executing a command.
-nv	Turns off verbose mode. This is the default option.
-x	Aborts process upon error. This is the default option.
-nx	Does not abort process upon error.

You can change shell options with either of two methods. The first method involves typing the option on the command line or after the shell command. For example:

$ -np	Turns off the shell prompt.
$ shell -np	Creates a new shell that does not prompt. When the new shell is exited, the original shell will prompt.

The second method uses set, a special shell command. To set shell options, type set, followed by the options desired. When using the set command, a hyphen (-) is unnecessary before the letter option. For example:

$ set np	Turns off the shell prompt.
$ shell set np	Creates a new shell that does not prompt. When the new shell is exited, the original shell will prompt.

As you can see, the two methods accomplish the same function. They are both provided for your convenience. You should use the method that is clearer to you.

The Shell Environment

The shell maintains a unique list of ***environment*** variables for each user on an OS-9 system. These variables affect the operation of the shell or other programs subsequently executed and can be set according to your preference.

You can access all environment variables by any process called by the environment's shell or by descendant shells. This essentially allows you to use the environment variables as ***global*** variables.

+ If a subsequent shell redefines an environment variable, the variable is only redefined for that shell and its descendents. The environment variable is not redefined for the parent shell.

Four environment variables are automatically set up when you log on to a time-sharing system:

Variable	**Description**
PORT	This specifies the name of the terminal. An example of a valid name is /t1. PORT is automatically set up by the tsmon utility.
HOME	This specifies your ***home*** directory. The home directory is specified in your password file entry and is your current data directory when you first log on the system. This is also the directory used when the command chd with no parameters is executed.
SHELL	This is the first process executed when you log on to the system.
USER	This is the user name you typed when prompted by the login command.

On single user systems, you can set these variables with the setenv command. You can also set up a procedure file with your normal configuration of these variables. This procedure file could then be executed each time you startup your terminal.

There are four other important environment variables:

Variable	**Description**
PATH	This specifies any number of directories. A colon (:) must separate directory paths. The shell uses PATH as a list of commands directories to search when executing a command. If the default commands directory does not include the file/module to execute, each directory specified by PATH is searched until the file/module is found or the list is exhausted.

Variable	**Description**

PROMPT	This specifies the current prompt. By specifying an “at” sign (@) as part of your prompt, you may easily keep track of how many shells you have running under each other. @ is a replaceable macro for the shell level number. The base level is set by the environment variable _sh.
_sh	This specifies the base level for counting the number of shell levels. For example, set the shell prompt to “@howdy: ” and _sh to 0:

```
$ setenv _sh 0
$ -p="@howdy: "
howdy: shell
1.howdy: shell
2.howdy: eof
1.howdy: eof
howdy:
```

TERM	This specifies the type of terminal being used. TERM allows word processors, screen editors, and other screen dependent programs to know what type of terminal configuration to use.

NOTE: Environment variables are case sensitive. OS-9 does not recognize a variable if you do not use the proper case.

Changing the Shell Environment

Three utility programs are available to use with environment variables: setenv, unsetenv, and printenv.

setenv:	Declares the variable and sets the value of the variable.
unsetenv:	Clears the value and removes the variable from storage.
printenv:	Prints the variables and their values to standard output.

setenv declares the variable and sets its value. The variable is put in an environment storage area accessed by the shell. For example:

```
$ setenv PATH ..:/h0/cmds:/d0/cmds:/dd/cmds
$ setenv _sh 0
```

These variables are only known to the shell in which they are defined and any descendant processes from that shell. This command does not change the environment of the parent process of the shell which issued setenv.

unsetenv clears the value of the variable and removes it from storage. For example:

```
$ unsetenv PATH
$ unsetenv _sh
```

printenv prints the variables and their values to standard output. For example:

```
$ printenv
PATH=..:/h0/cmds:/d0/cmds:/dd/cmds
PROMPT=howdy
_sh=0
```

These three commands are described in the ***OS-9 Utilities*** section.

Built-In Shell Commands

The shell has a special set of commands, or option switches, built-in to the shell. You can execute these commands without loading a program and creating a new process. You can execute them regardless of your current execution directory.

The built-in commands and their functions are:

Command	**Description**
* <text>	Indicates a comment: <text> is not processed. This is especially useful in procedure files.
chd <path>	Changes the current data directory to the directory specified by the path.
chx <path>	Changes the current execution directory to the directory specified by the path.
ex <name>	Directly executes the named program. This replaces the shell process with a new execution module.
kill <proc ID>	Aborts the process specified by <proc ID>.
logout	Terminates the current shell. If the login shell is to be terminated, the .logout file in the home directory is executed and then the login shell is terminated.
profile <path>	Reads input from a named file and then returns to the shell's original input source.
set <options>	Sets options for the shell.
setenv <env var> <value>	Sets environment variable to the specified value.
setpr <proc ID> <priority>	Changes the process's priority.
unsetenv <env var>	Deletes an environment variable from the environment.
w	Waits for a child process to terminate.
wait	Waits for all child processes to terminate.

Shell Command Line Processing

The shell reads and processes command lines one at a time from its input path, which is usually your keyboard. Each line is first scanned, or ***parsed***, to identify and process any of the following parts which may be present:

keyword	A name of a program, procedure file, built-in command, or pathlist.
parameters	The names of files, programs, values, variables, constants, etc. to pass to the program being executed.
execution modifiers	These modify a program's execution by redirecting I/O or changing the priority or memory allocation of a process.
separators	When multiple commands are placed on the same command line, separators specify whether they should execute sequentially or concurrently.

The shell can process a command line with only the keyword present. Parameters, execution modifiers, and separators are optional. After it identifies the keyword, the shell processes any execution modifiers and separators. The shell assumes that any text not yet processed are parameters; they are passed to the program called.

The keyword must be the first word in the command line. If the keyword is a built-in command, it is immediately executed.

If the keyword is not a built-in command, the shell assumes it is a program name and attempts to locate it. The shell searches for the command in the following sequence:

¿ The shell checks the memory to see if the program is already loaded into the module directory. If it is already in memory, there is no need to load another copy. The shell then calls the program to be executed.

¡ If the program was not in memory, your current execution directory is searched. If it is found, the shell attempts to load the program. If this fails, the shell tries to execute it as a procedure file. If this fails, the shell attempts the same procedure using the next directory specified in the PATH environment variable. This continues until the command is successfully executed or the list of directories is exhausted.

Ç¬ The shell searches your current data directory. If it finds the specified file, it is processed as a procedure file. Procedure files are assumed to contain one or more shell command lines. These command lines are processed by a newly created, or ***child***, shell as if they had been typed in manually. After all commands from the procedure file execute, control returns to the old, or ***parent***, shell. Because the child shell processes the commands, all built-in commands in the procedure file such as chd and chx only affect the child shell.

The shell returns an error if the program is not found. If the program is found and executed, the shell waits until the program terminates. When the program terminates, it reports any errors returned. If there are more input lines, the shell gets the next line and the process is repeated.

This sample command line calls a program:

```
$ prog #12K sourcefile  -l -j >/p
```

In this example:

prog	Is the keyword.
#12K	Is a modifier which requests that an alternate memory size be assigned to this process. In this case, 12K is used as memory.
sourcefile -l -j	Are parameters passed to prog.
>	Is a modifier, which redirects output to a file or device. In this case, > redirects the output to the printer (/p).
/p	Is the system's printer.

Special Command Line Features

In addition to basic command line processing, the shell facilitates:

- Memory allocation
- I/O redirection, including filters
- Process priority
- Wildcard pattern matching
- Multi-tasking: concurrent execution

These functions are accessed through the use of execution modifiers, separators, and wildcards. There are virtually unlimited combinations of ways to use these capabilities.

Characters which comprise execution modifiers, separators, and wildcards are stripped from the part(s) of the command line passed to a program as parameters. These characters cannot be passed as parameters to programs unless contained in quotes:

Modifiers:	#	Additional memory size
	^	Process priority
	>	Redirect output
	<	Redirect input
	>>	Redirect error output

Separators:	;	Sequential execution
	&	Concurrent execution
	!	Pipe construction
Wildcards:	*	Matches any character
	?	Matches a single character

Execution Modifiers

The shell processes execution modifiers before the program is run. If an error is detected in any of the modifiers, the run is aborted and the error reported.

Additional Memory Size Modifier

Every executable program is converted to machine language for storage. During the conversion process, a ***module header*** is created for the program. A module header is part of all executable programs and holds the program's name, size, memory requirements, etc. You can find a complete explanation of module headers in the ***OS-9 Technical Manual***.

When the shell processes an executable program, it allocates the minimum amount of working memory specified in the program's module header. To increase the default memory size, you can assign memory in 1K increments using the pound sign modifier (#), followed by a number of allocated kilobytes: #10k or #10. If the specified memory allocation is smaller than would otherwise be used, the modifier is ignored.

The increase in memory allocation only affects one command. If you want to increase the allocation for the next command, you must add the modifier (#) again.

NOTE: Programs written in C use the additional memory for stack space only.

I/O Redirection Modifiers

Redirection modifiers redirect the program's standard I/O paths to alternate files or devices. Usually, programs do not use specific file or device names. This makes the redirection of standard I/O to any file or device fairly simple, without altering the program.

Programs which normally receive input from a terminal or send output to a terminal use one or more of these standard I/O paths:

- ***Standard Input Path:*** Normally passes data from a terminal's keyboard to a program.
- ***Standard Output Path:*** Normally passes output data from a program to a terminal's display.
- ***Standard Error Path:*** Can be used for either input or output, depending on the nature of the program using it. This path is commonly used to output routine status messages such as prompts and errors to the terminal's display. By default, the standard error path uses the same device as the standard output path.

A new process can only be created by an existing process. The new process is known as the ***child process***. The process that created the child process is known as the ***parent process***. Each child process inherits the parent process's standard I/O paths.

When the shell creates a new process, it inherits the shell's standard I/O paths. Upon startup or logging in, the shell's standard input is the terminal keyboard. The standard output and standard error are directed to the terminal's display. Consequently, the child's standard input is the terminal keyboard. The child's standard output and standard error are directed to the terminal's display.

When a redirection modifier is used on a shell command line, the shell opens the corresponding paths and passes them to the new process as its standard I/O paths.

+ The three redirection modifiers are:

<	Redirects the standard input path.
>	Redirects the standard output path.
>>	Redirects the standard error path.

When you use redirection modifiers on a command line, they must be immediately followed by a path describing the file or device to or from which the I/O is to be redirected.

Each physical input/output device supported by the system must have a unique name. Although the device names used on a system are somewhat arbitrary, it is customary to use the names Microware assigns to standard devices in OS-9 packages. They are:

Device	Description
TERM	Primary System Terminal
t1, t2, etc.	Other Serial Terminals
p	Parallel Printer
p1	Serial Printer
dd	Default Disk Drive
d0	Floppy Disk Drive Unit 0
d1, d2, etc.	Other Floppy Disk Drives
h0, h1, etc.	Hard Disk Drives (Format-inhibited)
h0fmt, h1fmt, etc.	Hard Disk Drives (Format-enabled)
n0, n1, etc.	Network Devices
mt0, mt1	Tape Devices
r0	Ram Disk
pipe	Pipe Device
nil	Null Device

NOTE: The h0fmt, h1fmt, etc. device descriptors have a bit set that allows you to use the format and os9gen utilities on them. To prevent accidentally formatting a hard disk, you should normally use the device names h0, h1, etc.

You may only use device names as the first name of a pathlist. The device name must be preceded by a slash (/) to indicate that the name is an I/O device. If the device is not a mass storage multi-file device like a disk drive, the device name must be the only name in the path. This restriction is true for devices such as terminals and printers.

For example, you can redirect the standard output of list to write to the system printer instead of the terminal:

```
$ list correspondence >/p
```

The shell automatically opens or creates, and closes (as appropriate) files referenced by I/O redirection modifiers. In the next example, the output of dir is redirected to the path /d1/savelisting:

```
$ dir >/d1/savelisting
```

If list is used on the path /d1/savelisting, output from dir is displayed as follows:

```
$ List /d1/savelisting

  directory of .   10:15:00
file1      myfile      savelisting
```

You can use redirection modifiers before and/or after the program's parameters, but you can use each modifier only once in a given command line. You can use redirection modifiers together to cause more than one redirection of the standard paths. For example, shell <>>>/t1 causes redirection of all three standard paths to /t1.

The addition and hyphen characters (+ and -) can be used with redirection modifiers. The ">-" modifier redirects output to a file. If the file already exists, the output overwrites it. The ">+" modifier adds the output to the end of the file. The following example overwrites dirfile with output from the execution directory listing:

```
dir -x >-dirfile
```

The next example adds the listing of newfile to the end of oldfile.

```
list newfile >+oldfile
```

NOTE: Spaces may not occur between redirection operators and the device or file path.

Process Priority Modifier

On multi-user systems or when multi-tasking, many processes seem to execute simultaneously. Actually, OS-9 uses a scheduling algorithm to allocate execution time to active processes.

All active processes are sorted into a queue based on the ***age*** of the process.

+ The age is a number between 0 and 65535 based on how long a process has waited for execution and its initial priority.

On a timesharing system, the system manager assigns the initial priority for processes started by each user. This priority for the initial process is listed in the password file. The initial process is usually the shell. On a single user system, processes have their priority set in the Init module. All child processes inherit their parent process's priority.

NOTE: Password files are discussed later in this chapter.

When a process enters the active queue, it has an age set to its initial priority. Every time a new active process is submitted for execution, all earlier processes's ages are incremented. The process with the highest age executes first.

If you want a program to run at a higher priority, use the caret modifier (^). By specifying a higher priority, a process is placed higher in the execution queue. For example:

```
$ format /d1 ^255
```

In this example, the process format is given the assigned priority of 255. By assigning a lower number, you can specify a lower priority.

> ⚠ **WARNING:** Specifying too high of a priority for a process can cause all other processes to be locked out until their ages mature. For example, if you specify a priority of 2000 for a large program and all the other processes have an age of less than 100, your program is the only process executed on the system until either your program terminates or another process's age reaches 2000. If another process's age reaches 2000, it is run once and entered back in the queue at its initial priority. Once again, your program either runs until it terminates or until another process's age reaches 2000.

Wildcard Matching

The shell uses some alternate ways to identify file and directory names. It accepts wildcards in the command line. The two recognized wildcard characters are the asterisk and the question mark (* and ?).

An asterisk (*) matches any group of zero or more characters. A question mark (?) matches any single character. The shell searches the current data directory or the directory given in a path for matching file names.

For the following examples, a directory containing the following files is used:

```
  directory of FILES  14:45:20
diary          diary2         form           form.backup    forms
login.names    logistics      logs           old            oldstuff
setime.c       shellfacts     sizes          sizes.backup   utils1
```

The command list log* lists the contents of login.names, logistics, and logs. The pattern log* matches all file names beginning with log followed by zero or more characters. The following commands demonstrate the function of this wildcard.

list s*	Lists all files in the current data directory beginning with s: Shellfacts, setime.c, and sizes.
del *	Deletes every file in the directory FILES.
dir ../*.backup	Lists all files in the parent directory that end with .backup.
dir -x d*	Lists all files in the current execution directory that start with the letter d. This can be helpful if you are unsure of the spelling of a particular utility.

The question mark (?) matches any single character in the position where the wildcard character is located. For example, the command line list log? only lists the contents of the file logs. The following commands demonstrate the function of this wildcard.

del form?	Deletes the file forms but not form.
list s????	Lists the contents of sizes, but not setime.c or shellfacts.

In both examples, the shell only searches for names with five characters.

Wildcards may also be used together. For example, the command list *.? lists any files that end in a period followed by any letter, number, or special character, regardless of what comes before the period. In this case, list *.? lists the contents of the file setime.c.

The shell only attempts to expand a character string containing a wildcard if the character string could be a pathlist. The shell does not expand wildcards used in the keyword of a command line. For example, the shell does not expand the asterisk in the following:

```
d* forms
```

NOTE: The shell disregards wildcard characters enclosed in double quotes. For example:

```
echo "*"
```

This echoes an asterisk (*) to standard output, which is usually the terminal. If you left out the double quotes around the asterisk, the shell would expand the wildcard to include every file name in the current directory and output each name to the terminal. Try it.

WARNING: You must be careful when using wildcards with utilities such as del and deldir. You **should not** use wildcards with the -x or -z options of most utilities.

Command Separators

A single shell input line can include more than one command line. These command lines may be executed sequentially or concurrently. Sequential execution causes one program to complete its function and terminate before the next program is allowed to begin execution. Concurrent execution allows several command lines to begin execution and run simultaneously.

Execute commands sequentially by separating the command lines with a semicolon (;). Execute commands concurrently by separating the command lines with an ampersand (&).

Sequential Execution

When you enter one command per line from the keyboard, programs execute one after another, or sequentially. All programs executed sequentially are individual processes created by the shell. After initiating execution of a program to be executed sequentially, the shell waits until the program it created terminates. The command line prompt does not return until the program has finished.

For example, the following command lines are executed one after another. The copy command is executed first, followed by the dir command.

```
$ copy myfile /D1/newfile
$ dir >/p
```

Specify more than one program on a single shell command line for sequential execution by separating each program name and its parameters from the next one with a semicolon (;). For example:

```
$ copy myfile /D1/newfile;  dir >/p
```

The shell first executes copy and then dir. No command line prompt appears between the execution of the copy and dir commands. The command line executes exactly as the previous two command lines, unless an error occurs.

If any program returns an error, subsequent commands on the same line are not executed regardless of the -nx option. In all other regards, a semicolon (;) and a carriage return act as identical separators.

The following example copies the contents of oldfile into newfile. When the copy command is finished, oldfile is deleted. Then, the contents of newfile are listed.

```
$ copy oldfile newfile; del oldfile; list newfile
```

In the next example, the output from dir is redirected into myfile in the d1 directory. The output from list is then redirected to the printer. Finally, temp is deleted.

```
$ dir >/d1/myfile ; list temp >/p; del temp
```

Multi-tasking: Concurrent Execution

Use the ampersand (&) separator to execute programs concurrently. This allows programs to run at the same time as other programs, including the shell. The shell does not wait to complete a process before processing the next command. Concurrent execution is how a background program is started.

Use the concurrent execution separator for multi-tasking. The number of programs that can run at the same time is not fixed; it depends on the amount of free memory in the system and the memory requirements of the specific programs.

Here is an example:

```
$ dir >/P&  list file1&  copy file1 file2 ; del temp
```

The dir, list, and copy utilities run concurrently because they are separated by an ampersand (&). del does not run until copy terminates because sequential execution (;) was specified.

If you add an ampersand (&) to the end of a command line, regardless of the type of execution specified, the shell immediately returns command to the keyboard, displays the $ prompt, and waits for a new command. This frees you from waiting for a process or sequence of processes to terminate.

This is especially useful when making a listing of a long text file on a printer. Instead of waiting for the listing to print to completion, you can use the concurrent execution separator to use your time more efficiently.

If you have several processes running at once, you can display a ***status summary*** of all your processes with the procs utility. procs lists your current processes and pertinent information about each process. The procs utility is discussed later in this chapter.

Pipes and Filters

The third kind of separator is the exclamation point (!) used to construct ***pipelines***. Pipelines consist of two or more concurrent programs whose standard input and/or output paths connect to each other using ***pipes***.

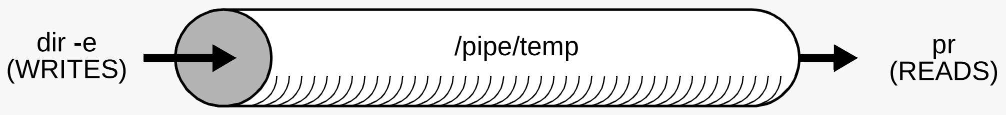
A pipe is simply a way to connect the output of a process to the input of another process, so the two run as a sequence of processes: a pipeline. Pipes are one of the primary means by which data is transferred from process to process for interprocess communications. Pipes are first-in, first-out buffers. They may hold up to 90 bytes of data at a time.

All programs in a pipeline are executed concurrently. The pipes automatically synchronize the programs so the output of one never gets ahead of the input request of the next program in the pipeline. This ensures that data cannot flow through a pipeline any faster than the slowest program can process it.

Any program that reads data from standard input can read from a pipe. Any program that writes data to standard output can write data to a pipe. Several utilities are designed so that the standard output of one can be piped to the standard input of another. For example:

```
$ dir -e ! pr
```

This example causes the standard output of dir to be piped to the standard input of the pr utility instead of on the terminal screen. pr reads the output of dir even though pr reads standard input by default. pr then displays the result.



The standard output of the dir -e command is piped to the standard input of the pr command through an un-named pipe. The pr utility displays the results of the dir -e command.

OS-9 uses two types of pipes: named pipes and un-named pipes.

Un-named Pipes

Un-named pipes are created by the shell when an input line with one or more exclamation point (!) separators is processed. For each exclamation point, the standard output of the program named to the left of the exclamation point is redirected by a pipe to the standard input of the program named to the right of the exclamation point. Individual pipes are created for each exclamation point present. For example:

```
$ update <master_file ! sort ! write_report >/p
```

In this example, the input for the program update is redirected from master_file. update's standard output becomes the standard input for the program sort. sort's output, in turn, becomes the standard input for the program write_report. write_report's standard output is redirected to the printer.

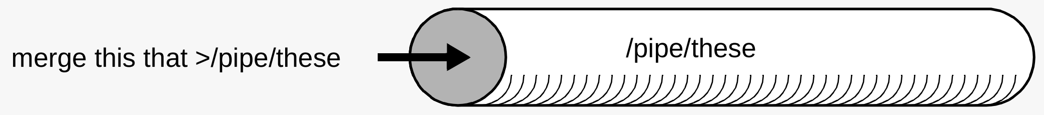
Named Pipes

Named pipes are similar to un-named pipes with one exception: a named pipe works as a holding buffer that can be opened by another process at a different time.

Create named pipes by re-directing output to /pipe/<file>, where <file> is any legal OS-9 file name. For example:

```
$ list letters >/pipe/letters
```

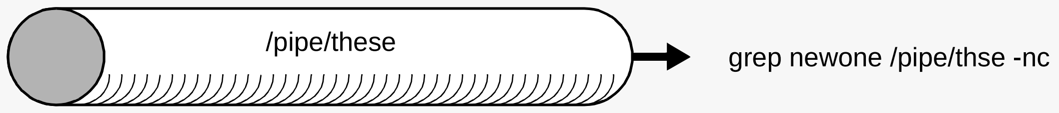
The output from the list command is redirected into a named pipe, /pipe/letters. The information remains in the pipe until it is listed, copied, deleted, or used in some other manner.



The output from the merge command is redirected to the named pipe, /pipe/these. /pipe/these remains open until the contents are used in some way. In this example, another user could later grep for a word in the file /pipe/these:

```
grep newone /pipe/these -nc
```

Once the file has been used, the named pipe is deleted.



You can also create named pipes by writing to the named pipe from a program. Named pipes are similar to mass-storage files. Named pipes have attributes and owners. They may be deleted, copied, or listed using the same syntax one would use to delete, copy, or list a file. You may change the attributes of a named pipe just as you would change the attributes of a file.

dir works with /pipe. This displays all named pipes in existence. A dir -e command may be deceiving. If any utility other than copy creates a named pipe, the pipe size equals 90 bytes. copy expands the size of the pipe to the size of the file. This indicates that the first 90 bytes of the output are in the named pipe. However, if the procs utility is executed, you will see that a path remains open to /pipe. If you were to copy or list the pipe, for example, the pipe would continue to receive input and pass it to its output path until the input process is finished. When the pipe is empty, the named pipe is deleted automatically.

Some of the most useful applications of pipelines are character set conversion, data compression/decompression and text file formatting. Programs which are designed to process data as components of a pipeline are often called ***filters***.

Command Grouping

You can enclose sections of shell input lines in parentheses (()). This allows you to apply modifiers and separators to an entire set of programs. The shell processes them by calling itself recursively as a new process to execute the enclosed program list. For example, the following commands produce the same result:

```
$ (dir /d0; dir /d1) >/p
$ dir /d0 >/p; dir /d1 >/p
```

However, one subtle difference exists. The printer is continuously controlled by one user in the first example, while in the second case, another user could conceivably use the printer in between the dir commands.

Command grouping can be used to cause a group of programs to be executed sequentially with respect to each other and concurrently with respect to the shell that initiated them. For example:

```
$ (del *.backup; list stuff_* >p)&
```

This command begins to sequentially delete all files ending in .backup and then list to the printer the contents of any files that start with stuff_. At the same time, a $ prompt will appear, indicating that the shell is waiting for a new command.

A useful extension of this form is to construct pipelines consisting of sequential and/or concurrent programs. For example:

```
$ (dir CMDS; dir SYS) ! makeuppercase ! transmit
```

This command line outputs the dir listings of CMDS and SYS, in that order, through a pipe to the program makeuppercase. The total output from makeuppercase is then piped to the program transmit.

It is important to remember that OS-9 processes commands from left to right. In the following example, the dir command is executed first, instead of the procs and del commands located inside the parentheses.

```
$ dir& (procs; del whatever)
```

Shell Procedure Files

A procedure file is a text file containing one or more command lines that are identical to command lines manually entered from the keyboard. The shell executes each command line in the exact sequence given in the procedure file.

A simple procedure file could consist of dir on one line and date on another. When the name of this procedure file is entered from the command line, dir would run, followed by date.

Procedure files have a number of valuable applications. They can:

- Eliminate repetitive manual entry of commonly used command sequences.
- Allow the computer to execute a lengthy series of programs in the background unattended, or while you are running other programs in the foreground
- Initialize your environment when you first log in.

In addition, you can use a procedure file to redirect the standard input, standard output, and standard error paths from programs and utilities to procedure files. This has many useful purposes. For example, instead of receiving the sometimes annoying output of shell messages to your terminal at random times, you could redirect the shell's output to a procedure file and review the messages at a more convenient time.

You can also run procedure files in the background by adding the ampersand operator:

```
$ procfile&
+4
```

WARNING: If a procedure file is run in the background, it should not contain any terminal I/O. Any terminal I/O caused by a background procedure file will minimally cause confusion as two or more processes try to control the same I/O path.

Notice the +4 returned by the shell in the example above. This is the process number assigned to the shell running procfile. You could achieve the same effect by using the <control>C interrupt:

```
$ procfile
[<control>C is typed]
+4
```

Using <control>C to place a procedure in the background only works if the procedure has not yet performed I/O to the terminal. Another limitation of the <control>C interrupt occurs when the shell has not had time to set up the command for execution. If the shell has not loaded files from the disk, established pipelines, etc., the <control>C causes the shell to abort the operation and return the shell prompt. For this reason, you should usually use the ampersand to place a procedure in the background.

OS-9 does not have any limit on the number of procedure files that can be simultaneously executed as long as memory is available.

NOTE: Procedure files themselves can cause sequential or concurrent execution of additional procedure files.

The Login Shell and Two Special Procedure Files: .login and .logout

The ***login shell*** is the initial shell created by the login sequence to process the user input commands after logging in.

+ The .login and .logout procedure files provide a way to execute desired commands when logging on to and leaving the system.

To make use of these files, they must be located in the home directory.

.login is executed each time the login command is executed. This allows you to run a number of initializing commands without remembering each and every command. .login is processed as a command file by the login shell immediately after successfully logging on to a system. After all commands in the .login file are processed, the shell prompts the user for more commands. The main difference in handling .login is that the login shell itself actually executes the commands rather than creating another shell to execute the commands.

You can issue commands such as set and setenv within .login and have them affect the login shell. This is especially useful for setting up the environment variables PATH, PROMPT, TERM, and _sh.

Here is an example .login file:

```
setenv PATH ..:/h0/cmds:/d0/cmds:/dd/cmds:/h0/doc/spex
setenv PROMPT "@what next: "
setenv _sh 0
setenv TERM abm85h
querymail
date
dir
```

.logout is executed to exit the login shell and leave the system. .logout is processed before the login shell terminates. It only processes the .logout file when given to the login shell; other subsequent shells simply terminate. You could use this to execute any cleaning up procedures that should be performed on a regular schedule. This might be anything from instigating a backup procedure of some sort to printing a reminder of things to do. Here is an example .logout file:

```
procs
wait
echo "all processes terminated"
* basic program to instigate backup if necessary *
disk_backup
echo "backup complete"
```

The Profile Command

The profile built-in shell command can be used to cause the current shell to read its input from the named file and then return to its original input source, which is usually the keyboard. To use the profile command, enter profile and the name of a file:

```
profile setmyenviron
```

The specified file (in this case, setmyenviron) may contain any utility or shell commands, including commands to set or unset environment variables or to change directories. These changes will remain in effect after the command has finished executing. This is in contrast to calling a normal procedure file by name only. If you call a normal procedure file without using the profile command, the changes would not affect the environment of the calling shell.

Profile commands may be nested. That is, the file itself may contain a profile command for another file. When the latter profile command is completed, the first one will resume.

A particularly useful application for profile files is within a user’s .login and .logout files. For example, if each user includes the following line in the .login file, then system-wide commands (common environments, news bulletins, etc.) can be included in the file /dd/SYS/login_sys:

```
profile /dd/SYS/login_sys
```

A similar technique can be used for .logout files.

Setting up a Time-Sharing System Startup Procedure File

OS-9 systems used for timesharing usually have a procedure file that brings the system up by means of one simple command or by using the system startup file. This procedure file initiates the timesharing monitor for each terminal. It begins by starting the system clock and initiating concurrent execution of a number of processes that have their I/O redirected to each timesharing terminal.

+ tsmon is a special program which monitors a terminal for activity. Typically, tsmon is executed as part of the start-up procedure when the system is first brought up and remains active until the system shuts down.

tsmon is normally used to monitor I/O devices capable of bi-directional communication, such as CRT terminals. However, you can also use tsmon to monitor a named pipe. If you do this, tsmon creates the named pipe and then waits for data to be written to it by some other process.

You can run several tsmon processes concurrently, each one watching a different group of devices. Because tsmon can monitor up to 28 device name pathlists, you must run multiple tsmon processes when more than 28 devices are to be monitored. Multiple tsmon processes can be useful for other reasons. For example, you may want to keep modems or terminals suspected of hardware trouble isolated from other devices in the system.

Here is a sample procedure file for a timesharing system with terminals named T1, T2, T3, and T71:

```
* system startup procedure file
echo Please Enter the Date and Time
setime
tsmon /t1 /t2 /t3&
tsmon /t71                                    * This terminal has been mis-behaving
```

NOTE: This login procedure will not work until a file called /d0/SYS/Password with the appropriate entries has been created.

NOTE: For more information on tsmon, see the ***OS-9 Utilities*** section.

The Password File

A password file is found in the SYS directory. Each line in the password file is a login entry for a user. The line has several fields separated by a comma. The fields are:

¿ **User name**. The user name may contain up to 32 characters including spaces. If this field is empty, any name will match.

¡ **Password.** The password may contain a maximum of 32 characters including spaces. If this field is omitted, no password is required for the specified user.

Ç¬ **Group.user ID number.** Both the group and the user portion of this number may be from 0 to 65535. 0.0 is the super user. The file security system uses this number as the system-wide user ID to identify all processes initiated by the user. The system manager should assign a unique ID to each potential user.

Ð **Initial process priority**. This number may be from 1 to 65535. It indicates the priority of the initial process.

ƒ **Initial execution directory**. This field is usually set to /d0/CMDS. Specifying a period (.) for this field defaults the initial execution directory to the CMDS file.

Ý **Initial data directory**. This is usually the specific user directory. Specifying a period (.) for this directory defaults to the current directory.

ý **Initial Program.** This field contains the name and parameters of the program to be initially executed. This is usually shell.

NOTE: Fields left empty are indicated by two consecutive commas.

The following is a sample password file:

```
superuser,secret,0.0,255,.,.,shell -p="@howdy"
suzy,morning,1.5,128,.,/d0/SUZY,shell
don,dragon,3.10,100,.,/d0/DON,Basic
```

For more information on password files, see the login utility in the ***OS-9 Utilities*** section.

Creating a Temporary Procedure File

You can create temporary procedure files to perform tasks which require a sequence of commands. The cfp utility creates a temporary procedure file in the current data directory and calls the shell to execute it. After the task is complete, cfp automatically deletes the procedure file, unless you use the -nd option to specify that you do not want the procedure file deleted.

The following is the syntax for the cfp utility:

```
cfp [<opts>] [<path1>] {<path2>} [<opts>]
```

To use the cfp utility, type cfp, the name of the procedure file (<path1>), and the file(s) (<path2>) used by the procedure file. The name of the procedure file may be omitted if the -s=<string> option is used.

All occurrences of an asterisk (*) in the procedure file are replaced by the given pathlist(s) unless preceded by the tilde character (~). For example, ~* translates to *. The command procedure is not executed until all input files have been read.

For example, if you have a procedure file in your current data directory called copyit that consists of a single command line: copy *, you could put all of your C programs from two directories, PROGMS and MISC.JUNK, into your current data directory by typing:

```
$ cfp copyit ../progms/*.c ../misc.junk/*.c
```

If you do not have a procedure file, you can use the -s option. The -s option causes the cfp utility to read the string surrounded by quotes instead of a procedure file. For example:

```
$ cfp -s="copy *" ../progms/*.c ../misc.junk/*.c
```

In this case, the cfp utility creates a temporary procedure file to copy every file ending in .c in both PROGMS and MISC.JUNK to the current data directory. The procedure file created by cfp is deleted when all the files have been copied.

Using the -s option is convenient because you do not have to edit the procedure file if you want to change the copy procedure. For example, if you are copying large C programs, you may want to increase the memory allocation to speed up the process. You could allocate the additional memory on the cfp command line:

```
$ cfp "-s=copy -b100 *" ../progms/*.c ../misc.junk/*.c
```

You can use the -z and -z=<file> options to read the file names from either standard input or a file. The -z option is used to read the file names from standard input. For example, if you have a procedure file called count.em that contains the command count -l * and you want to count the lines in each program to see how large the programs are before you copy them, you could type the following command line:

```
$ cfp -z count.em
```

The command line prompt does not appear because the cfp utility is waiting for input. Type in the file names on separate command lines. For example

```
$ cfp -z count.em
../progms/*.c
../misc.junk/*.c
```

When you have finished typing the file names, press the carriage return a second time to get the shell prompt.

If you have a file containing a list of the files that you want copied, you could type:

```
$ cfp -z=files "-s=copy *"
```

For more information, read the cfp utility discussion in the ***OS-9 Utilities*** section.

Multiple Shells

Like all OS-9 utilities, the shell can be simultaneously executed by more than one process. This means that in addition to each user having their own shell, an individual user can have multiple shells.

New shells can be created with procedure files. For example, to execute a shell whose standard input is obtained from procfile, type:

```
$ shell <procfile
```

The new shell automatically accepts and executes the command lines from the procedure file instead of a terminal keyboard. This technique is sometimes called ***batch*** processing.

Shells can also fork new shells by simply processing the procedure file:

```
$ procfile
```

Basically, both of the above commands execute the commands found in the procfile file.

By creating new shells, you can also move around the file system more efficiently. To demonstrate this concept, the directory system in Figure 5a is used.

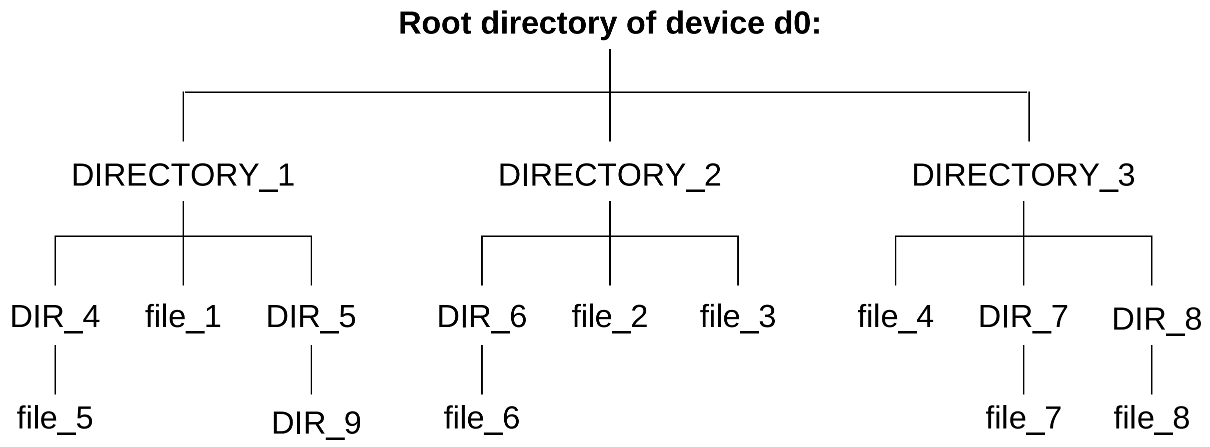


Figure 5a: An example directory

If your current data directory is DIR_9 and you want to work on file_8, you would normally change your current data directory to DIR_8 and access the file by typing:

```
chd /d0/DIRECTORY_3/DIR_8
```

To return to DIR_9 you would execute a similar command. This is somewhat inconvenient and involves always knowing the path to each directory.

Instead, you can create a shell and change directories:

$ (chd /d0/DIRECTORY_3/DIR_8)

This makes your current directory DIR_8, but you can return to DIR_9 by pressing the <escape> (Esc) key. By this method, you may use any directory as a base directory and ***fork*** a shell out to any other directory.

You may continue to imbed as many shells as you like. Each time you press the <Escape> key, you are taken to the previous shell. In this fashion you could conceivably escape from DIRECTORY_2 to DIR_8 to DIR_6 to DIR_9.

REMINDER: Because of the nature of jumping from shell to shell, it is easy to get lost. pd displays a complete pathlist from the root directory to your current data directory. Likewise, when running multiple shells, it is easy to forget how many shells are running. If the _sh environment variable is set to 1 and the shell prompt includes an "at" sign (@), the number of shells replaces the @ in the prompt. For example, if three shells are run under each other, the prompt might look like this:

3.what next:

+ You should experiment with the multiple shell aspects to fully use OS-9.

The Procs Utility

Because of OS-9's multi-tasking abilities, you often have more than one process executing at a time. You may become confused as to which processes are still running and which processes have run to completion. The procs utility displays a list of processes running on the system that are owned by the user running procs. This allows the user to keep track of their current processes.

+ Processes can switch states rapidly, usually many times per second. Therefore, the procs display is a snapshot taken at the instant the command is executed and shows only those processes running at that exact moment.

procs displays ten pieces of information for each process:

Id	The process ID
PId	The parent process ID
Grp.usr	The group and user number of the owner of the process
Prior	The initial priority of the process
MemSiz	The amount of memory the process is using
Sig	The number of any pending signals for the process

S	The process status:

w	Waiting
s	Sleeping
a	Active
*	Currently executing

CPU Time	The amount of CPU time the process has used
Age	The elapsed time since the process started
Module & I/O	The process name and standard I/O paths:

<	Standard input
>	Standard output
>>	Standard error output

If several of the paths point to the same pathlist, the identifiers for the paths are merged.

The following is an example of procs:

```
$ procs

Id PId Grp.Usr Prior MemSiz Sig S   CPU Time   Age Module & I/O
 2   1   0.0   128    0.25k   0 w       0.01   ??? sysgo <>>>term
 3   2   0.0   128    4.75k   0 w       4.11 01:13 shell <>>>term
 4   3   0.0     5    4.00k   0 a   12:42.06 00:14 xhog <>>>term
 5   3   0.0   128    8.50k   0 *       0.08 00:00 procs <>>term
 6   0   0.0   128    4.00k   0 s       0.02 01:12 tsmon <>>>t1
 7   0   0.0   128    4.00k   0 s       0.01 01:12 tsmon <>>>t2
```

procs -a displays nine pieces of information: the process ID, the parent process ID, the process name and standard I/O paths, and six new pieces of information:

Aging	The age of the process based on the initial priority and how long it has waited for processing
F$calls	The number of service request calls made
I$calls	The number of I/O requests made
Last	The last system call made
Read	The number of bytes read
Written	The number of bytes written

The following is an example of procs -a:

```
$ procs -a

Id PId  Aging  F$calls I$calls Last      Read Written Module & I/O
```

```
  2  1  129     5   1 Wait     0    0 sysgo <>>>term
  3  2  132   116 127 Wait   282  129 shell <>>>term
  4  3   11     1   0 TLink    0    0 xhog  <>>>term
  5  3  128     7   4 GPrDsc   0    0 procs <>>>term
  6  0  130     2   7 ReadLn   0    0 tsmon <>>>t1
  7  0  129     2   7 ReadLn   0    0 tsmon <>>>t2
```

The -b option displays all information from procs and procs -a. The -e option displays information for all processes in the system.

For more information on procs, see the ***OS-9 Utilities*** section.

Waiting For The Background Procedures

If you use OS-9's multi-tasking ability, there will be times when a number of procedures are running in the background. If it is important to wait for these tasks to finish before running a new procedure, use the w or wait built-in shell command.

+ w waits for a child process to be executed to finish.
wait waits for all child processes running in the background to finish.

REMINDER: A child process is a process that the current shell or a child of the shell is executing.

For example, if a document needs to be created from three different files and each file has to be sorted by different fields, the following procedure files can be used to create the same result:

```
*start of first procedure file*
qsort -f=1 file1&
qsort -f=2 file2&
qsort -f=3 file3&
wait
merge file1 file2 file3 >report

*start of second procedure file*
qsort -f=1 file1
qsort -f=2 file2
qsort -f=3 file3
merge file1 file2 file3 >report
```

The first procedure file is much quicker because each of the files are processed concurrently.

Stopping Procedures

You can use two methods to stop a procedure. The first method involves the <control>C or <control>E signals. The second method uses the kill utility.

The shell handles these keyboard generated signals in the following manner. If either of these signals are received while the shell is waiting for keyboard input, the following messages are issued:

```
$ Read I/O error - Error #000:002   [ ^E typed ]
$ Read I/O error - Error #000:003   [ ^C typed ]
```

These are the standard messages given whenever an I/O error occurs when reading command input data. These keyboard signals are useful to get the shell’s attention while it is waiting for a process to terminate.

If the shell is waiting for keyboard input and <control>E is typed, the shell forwards the keyboard abort signal to the current process and immediately prompts for command input:

```
$ sleep 500
[ ^E is typed]
abort
$
```

The abort message is typed by the shell to acknowledge receipt of the interrupt.

If the shell is waiting for keyboard input and <control>C is typed, the shell stops waiting for the current process to terminate and prompts for command input. This action is similar to using an ampersand on the command line. For example:

```
$ sleep 500
[ ^C is typed]
+8
$
```

It is important to remember that using <control>C in this fashion is possible only if the command in question has not yet performed I/O to the terminal. The signal is only received by the last process to perform I/O. If the shell has not yet finished setting up the command for execution, the signal causes the shell to abort the operation and return the prompt.

> <control>C stops the shell from waiting for the process to terminate and returns a prompt for a command.
>
> <control>E forwards the keyboard abort signal to the process and immediately prompts for input.

You can also use the kill utility to terminate background processes by specifying the process number of the process to kill. Obtain the process number of the process to kill from procs. kill is used in the following manner:

```
kill <proc num>
```

For example, if you want to terminate a process called xhog, you would first execute a procs:

```
$ procs

Id PId Grp.Usr Prior MemSiz Sig S   CPU Time   Age Module & I/O
 3   2   7.03   128    4.75k   0 w       4.11 01:13 shell <>>>term
 4   3   7.03     5    4.00k   0 a   12:42.06 00:14 xhog <>>>term
 5   3   7.03   128    8.50k   0 *       0.08 00:00 procs <>>term
```

From procs, you can see that the process number for xhog is 4. You then type:

```
$ kill 4
```

When you execute procs again, xhog is no longer shown.

+ To use the kill utility:
  - Get the process number using the procs utility
  - Type kill <proc num>

Either of these methods terminate any process running in the background with one exception: if a process is waiting for I/O, it may not die until the current I/O operation is complete. Therefore, if you terminate a process and procs shows it still exists, it is probably waiting for the output buffer to be flushed before it can die.

NOTE: You must either own the procedure or be the super user to kill a specified process.

Error Reporting

Many programs, including the shell, use OS-9's standard error reporting function. This displays a brief description of the error and an error number on the standard error path. An appendix listing all of the standard error codes is included with this manual.

If a longer description of errors is desired, set the -e and the -v shell options. This prints error messages from /dd/SYS/errmsg on standard output.

Running Compiled Intermediate Code Programs

Before the shell executes a program, it checks the program module's language type. If its type is not 68000 machine language, the shell calls the appropriate run-time system for that module. Versions of the shell supplied for various systems are capable of calling different run-time systems

For example, if you wanted to run a BASIC I-code module called adventure, you could type either of the two commands given below; they accomplish exactly the same thing:

```
$ BASIC adventure
$ adventure
```

End of Chapter 5