CHAPTER 2

USING OS-9

2.1 BOOTING OS-9

In most computers the main memory (Random Access Memory - RAM) loses its data when the power is turned off. At power-on or reset the CPU cannot assume that it has any valid programs in RAM. Therefore every computer has a small amount of ROM (Read Only Memory) - memory that cannot be written, but does not lose its contents on power-off.

The ROM contains a "bootstrap" program that (in general) reads a known part of a disk to load the operating system into RAM. The bootstrap program then jumps to the coldstart routine of the operating system, which does the rest of the startup procedure. This operation is known as "bootstrapping", from the analogy of someone lifting themselves up by their own bootlaces - how can the operating system start up the computer if the operating system itself is not in memory?

The term "bootstrapping" is usually abbreviated to "booting", and "bootstrap program" to "boot program".

The boot program is not itself part of the operating system, nor can it use any of the facilities of the operating system. It is a self-contained program whose job is to load the operating system into the computer's memory.

Because the boot program must initialize and use the hardware of the particular computer it is running on, it is not provided as a compiled program by Microware. Instead, Microware gives the implementor example source code which the implementor must adapt. In consequence, the booting procedure is not the same on all OS-9 computers, so the description here is necessarily a bit vague.

Microware supplies the implementor with a simple debugger program to go with the boot program, known as the ROM-based debugger. This, too, is a stand-alone program that does not use any of the operating system facilities. The implementor has the option of including this debugger with the boot program. Also, from OS-9 version 2.4, Microware supplies the implementor with a much improved debugger, known as ROMbug. The implementor has the option of including this with the boot program, in place of the old ROM-based debugger.

If one of these debuggers is included with the boot ROM, then normally the system will enter the debugger on power-on or reset. The user must then enter the "go" command:

```
    debug: g[CR]
```

to continue with the booting procedure. (Note: [CR] means press the **RETURN** or **ENTER** key). On some systems, the implementor will have written the boot program to bypass entering the debugger if a switch or link on the CPU board is appropriately configured.

Prior to OS-9 version 2.4, the example source code provided by Microware could only boot from one disk drive, or (if assembled differently) search for the OS-9 kernel in ROM. As most systems want to boot from hard disk normally, but from floppy disk when needed, many implementors have modified the example source code to allow booting from multiple drives. Because this is not in the Microware-supplied code, it varies from one manufacturer to another. Some implementors also incorporated the search for the kernel in ROM. This type of boot program will typically search for the kernel in ROM first. If the kernel is found, the boot program jumps to the coldstart routine of the kernel in ROM. Otherwise, the boot program attempts to boot from each disk drive it knows about in turn, until it is successful with one of them.

From OS-9 version 2.4, Microware provides the implementor with an additional set of example source code known as "CBOOT" (because it is written in C, rather than assembly language). This has the capability to boot from multiple drives, including tape drives, and even across a network.

So, a typical boot-up sequence would be:

- Power-on or hit the reset button! A "booting" message may be displayed on the terminal.

- The ROM-based System Debugger may be present (giving a display of the processor's internal registers):
  ```
  debug: g[CR]
  ```
- You may be prompted to select which drive you want to boot from.
- The boot program will now find the OS-9 kernel in ROM, or read the operating system boot file from the (selected) drive. If you had "enabled" the ROM-based debugger with the **e** command, the boot program will re-enter the debugger. The "go" command will continue the boot. This allows the implementor to check that the boot has worked correctly. The boot program then locates the kernel module, and jumps into its coldstart routine.
- The kernel searches the ROM and the boot file (if any) for modules, and enters them in the module directory. If you had enabled the ROM-based debugger, the kernel will re-enter the debugger. This allows the implementor to set breakpoints in any of the modules in the boot file - the kernel has already checked their CRCs, and will not check them again. The **g** command will continue the OS-9 coldstart.
- The kernel links to the configuration module **init** (to get the user-configurable initialization parameters), initializes all its tables and other memory structures, and opens the default input and output paths (usually the terminal '/term') and default directories (usually the hard disk root directory). It then starts (forks up) the program whose name is given in the **init** module - the initial program to fork. This program is usually called **sysgo**.
- The last operation the kernel does in its coldstart routine is to fork the "system process". This is a program whose code is contained within the kernel module. Its job is to manage the wakeup of programs that are in timed sleep, or have set alarms. This system process is not visible to the user.
- Having forked the system process, the kernel coldstart ends. The operating system executes no further code unless called by a program or an interrupt.
- The **sysgo** program provided by Microware first changes its execution directory to 'CMDS' (on the drive whose name is given in the **init** module). It then forks up the **shell** program - the OS-9 command line interpreter - with its input directed to come from a file called 'startup', rather than from the keyboard. When that **shell** has finished processing the instructions in 'startup', **sysgo** enters a loop,

forking a **shell**, then waiting for it to die. Thus if the user terminates the **shell** program, **sysgo** will start up another one.

- The 'startup' file contains **shell** command lines to initialize the system. For example, it may call the **tsmon** utility to log in other terminals.
- If the computer hardware does not have a battery-backed calendar/clock chip, 'startup' will have a line to call the **setime** utility, and you will be prompted for the date and time:

```
    yy/mm/dd hh:mm:ss [am/pm]
    Time: 92/08/15/09/30
```

- Once **shell** has processed the 'startup' file, **sysgo** forks up a new shell, which presents you with its default prompt:

```
    $
```

- This ends the startup procedure.

Because the incarnation of **shell** that processes the 'startup' file is not the same incarnation as the one **sysgo** subsequently forks to give you the command line prompt, "private" changes made in instructions in the 'startup' file are not passed on to the **shell** that gives you the prompt. Examples of changes that are "private" to the executing program are changes to the default directories (**chx** and **chd**), and changes to the **shell** prompt string (**-p=...**).

Some users will modify the 'startup' file so that its last instruction is to login the system console (terminal). In this case the **shell** processing the 'startup' file will never finish (because the **tsmon** program used to log in the system console never finishes). The 'ex' built-in command of the **shell** can be used to transform the shell into an incarnation of **tsmon** as the last act of the 'startup' file, to avoid an unnecessary incarnation of shell. If **tsmon** has been called, you must press [CR], which will give you the **login** prompt, rather than the **shell** prompt.

The **shell** program "inherits" the current directories of the **sysgo** program . After bootup, the "current data directory" is the root directory of the "initial device", as specified in the **init** module (unless the 'startup' file calls **tsmon** to log in the system console, in which case the login procedure may change the current data and execution directories). Typical device names are:

/d0	floppy disk drive 0
/h0	hard disk

/dd	"default device" – usually the hard disk

and the "current execution directory" is the 'CMDS' directory within that root directory:

/d0/CMDS	floppy disk drive 0
/h0/CMDS	hard disk
/dd/CMDS	"default device"

Notice that in OS-9 all device names start with a '/' character. A typical system might use the following device names:

/d0	floppy disk drive 0
/d1	floppy disk drive 1
/fh0	hard disk without format protection
/h0	hard disk drive
/h0fmt	same as '/fh0'
/mt0	tape drive
/nil	"null" device – data sent here is lost
/p	parallel port (Centronics) configured for use with a printer
/p1	second serial port configured for use with a printer
/p2	third serial port configured for use with a printer
/r0	RAM disk
/term	first serial port (system console)
/t1	second serial port
/t2	third serial port
/u0	floppy disk drive 0 configured for Microware Universal form if '/d0' is some other format

- **Figure 2 – Typical device names**

2.2 SHELL - THE COMMAND LINE INTERPRETER

shell is a program – it is not built into the operating system. No programs or utilities are built into the operating system itself. **shell** reads in a line (typically from the keyboard), and processes it. The line may contain:

- the name of a program to fork, with parameters:

```
$ dir CMDS -e
```

- the name of a text file containing shell command lines:

```
$ my_proc_file
```

- shell "built-in" commands:

```
$ chd /dd/USER
```

shell accepts several special characters to modify its default behaviour – see figure 3.

Many **shells** may be running concurrently, for the same or different users. Each user has at least one **shell**. Because **shell** is just a program, a user can run a different command line interpreter, in place of **shell** (for example, **mshell**, the advanced command line interpreter from Microware). Also, **shell** may be called (forked) from within another program. For example, both **basic** and **debug** have a **'$'** command to fork a **shell**, and the same can be achieved from **umacs** with the key sequence [^X][C] ([^X] means "hold [CTRL] and press [X]").

shell implements "wild carding" on file names, using any combination of the characters '*' (for "any number of characters or none") and '?' (for "any single character"). **shell** actually performs wild card comparison of names using the "compare names" system call (**F$CmpNam**).

```
$ dir *.c
```

will display the names of all files that end in '.c'.

```
$ list fred?.c
```

will list all files whose names start with 'fred', followed by any single character, followed by '.c'.

Be aware that it is the **shell** that reads the current data directory to resolve wild-carded file names, not the program that is being forked. The program is passed the expanded file names as parameters just as if you had typed them in full.

Command line parameters are separated by spaces. If a parameter is to contain spaces or **shell** special characters, it must be enclosed in single or double quotes. For example:

```
$ echo *
```

will print the names of all files in the current data directory, while:

```
$ echo "*"
```

will print a single '*' character.

The present OS-9 **shell** does not have UNIX-like parameter substitution or flow control "language" features - it is a simple command line interpreter. Microware also sells a much more sophisticated command line interpreter - **mshell**.

;	separates commands to execute sequentially: `$ dir -e ; mdir`

&	separates commands to run concurrently - **shell** does not wait for the child to finish before executing the next part of the command, or giving a new prompt if the '&' is at the end of the line:
	`$ list fred & dir`
	`$ tsmon /t1 &`

!	"pipes" the output of the first program into the input of the second program: `$ echo fred ! list -z`
^	sets the priority of the program being forked: `$ dir ^200`
#	sets the size of the data space of the program, in kilobytes. This modifier is rarely used - utilities dynamically allocate buffer memory themselves: `$ basic #20k`
()	forks up a separate **shell** to execute the commands between the parentheses. The separate **shell** can change its private parameters (such as current data directory) without affecting those of the parent **shell**: `$ (chd /dd/SYS ; list errmsg)`
<	redirects the standard input path of a program being forked to any device or file: `$ list -z <file_list`
>	redirects the standard output path of a program being forked: `$ list fred >/p`

>>	redirects the standard error path of a program being forked:
	`$ dsave /d1 >>err_log`
	the redirection modifiers can be combined to redirect any two or all of the standard paths to another device or file:
	`$ r68 fred.a -ql >>>/p`
	`$ shell <>>>/t1`

• **Figure 3 - shell special characters**

2.3 SHELL BUILT-IN COMMANDS

Certain desirable commands cannot be separate utility programs, as they change private properties of this "incarnation" of **shell**, or operate on "children" of the **shell** (programs forked by the **shell**). Programs forked up by **shell** receive a copy of its environment, but if they change their own environment this has no effect on the environment of the parent **shell**.

chd	change **shell**'s data directory: `$ chd /dd/USER`
chx	change **shell**'s execution directory: `$ chx /dd/USER/ETC/CMDS`
kill	kill another process (by process ID): `$ kill 7`
w	wait for one (any) child of the **shell** to die: `$ w`
wait	wait for all children of the **shell** to die: `$ wait`
setenv	set (or change) an environment variable: `$ setenv TERM vt100`
unsetenv	forget an environment variable: `$ unsetenv PATH`
setpr	set execution priority of a process: `$ setpr 6 200`
logout	exit this shell (same as "end-of-file" key): `$ logout`
profile	execute the commands in a text file (does not fork another **shell** to process the text file - contrast just typing the file name): `$ profile fred`
ex	"chain" rather than "fork" another program - the **shell** effectively dies after forking the program: `$ ex tsmon /term`
-e	enable full error message printing
-ne	print only error numbers
-l	only allow exit via "logout", not "end-of-file" key
-nl	allow exit via "logout" or "end-of-file" key
-p	enable display of prompt

-p=<str>	set new prompt string: `$ -p="George: "`
-np	disable display of prompt
-t	echo input lines – useful for procedure files
-nt	do not echo input lines
-v	display directory searching
-nv	do not display directory searching
-x	abort on error (**shell** program terminates) – the default if processing a procedure file
-nx	do not abort on error – the default if taking input from the keyboard (SCF or GFM device).

2.4 ENVIRONMENT VARIABLES

When a process is forked by the **shell** (or by the **os9exec()** C library function), it is passed a parameter string composed of two parts: the command line parameters, and the environment variables. Each environment variable is a character string with an associated name. For example, the environment variable **TERM** may be assigned the string **vt100**. The **shell** built-in commands **setenv** and **unsetenv** are used to create and delete environment variables local to that **shell**. Any name can be invented, and assigned any character string. When the **shell** forks a process, it passes a copy of all the environment variable names and strings that it currently has. If that process then forks another process itself, it will pass on a copy of the same environment variables (provided it uses the **os9exec()** C library function to do the fork), unless it has changed its copy of the environment variables.

It is important to bear in mind that the environment variables do not exist globally in the system. Each process has, in its static storage memory, a copy of the environment variables passed to it when it was forked. The process can modify, delete, or add to them, and pass them on to any process it forks. This means that different processes can have the same environment variable name with a different character string.

The environment variables are effectively implicit command line parameters. They save you the trouble of typing in many additional parameters with each command line. A program can look through the list of environment variables it has been passed, to see if there are any names it recognizes. For example, the **umacs** screen editor looks for the environment variable **TERM** to tell it which type of terminal is being used. The shell looks for the environment variable **PATH** to tell it what directories to search when forking a program, in addition to the current execution directory (which it searches first), and the environment variable **HOME** to tell it which directory to change to if the **chd** command is used without a pathlist.

The **printenv** utility is used to list all the currently defined environment variables of the process that forks it (usually your command line **shell**).

2.5 PATHLISTS

All I/O devices and files are accessed by means of pathlists. A pathlist is a text string identifying the device or file.

If the pathlist starts with a '/', the first name element is a device name:

/p

/term

/dd

otherwise the pathlist is relative to the current data directory or current execution directory (depending on whether the file is opened with "execute mode", which is a function of the program using the pathlist). For example, the **load** utility opens the file with the execute mode (unless the '-d' option is used), and so the pathlist is relative to the current execution directory:

```
   $ load mdir
```

whereas the **list** utility opens the file without the execute mode, so the pathlist is relative to the current data directory:

```
   $ list myfile
```

Note: a device name is the name of the device descriptor module describing the device.

The disk file manager of OS-9 ("RBF") supports hierarchical directories. Therefore a pathlist may have any number of name elements. Name elements are separated by '/':

/dd/CMDS/BOOTOBJS/h0fmt

MYDIR/myfile

Letter case is not significant in device names (or any module names), or in **RBF** pathlists. By convention, directories are created with upper case names, to be easily visible in a directory listing:

```
$ makdir NEWDIR
$ copy /dd/startup newdir/startup
```

Each directory contains an entry '.', referring to itself, and '..', referring to its parent. For example:

```
$ dir ..
```

displays a directory listing of the directory one level above this – the parent directory of the current data directory.

```
$ dir ../BROTHER
```

displays a directory listing of the directory 'BROTHER', which is a directory with the same parent as the current data directory – a "sibling" of the current directory.

OS-9 also permits multiple '.' to go further up the hierarchy. Thus '....' is equivalent to '../../..' (both refer to three levels up). The '..' entry of the root directory refers to itself (it is identical to the '.' entry). If your pathlist has more '.' than there are levels to go up the directory hierarchy, this peculiarity of the root directory avoids any problems – the extra '.' are effectively discarded, as going to the parent of the root directory returns to the root directory.

Note: anything in the pathlist beyond the initial device name is a function of the file manager, not the kernel. For example, the '.' convention described above is a function of the disk file manager **RBF**.

2.6 CURRENT DIRECTORIES

There are separate current data and execution directories for each process (running program).

If a pathlist starts with a '/', the first pathlist name element is a device name, and the "current directory" feature is not invoked:

```
$ list /h0/SYS/password
```

The "current directory" feature is invoked if a pathlist does not start with a '/'. The current execution directory is used if the path is opened with the "execute" mode, otherwise the current data directory is used. For example, the **dir** utility opens the target directory without the execute mode (unless the '-x' option is used), so a pathlist that does not start with '/' is relative to the current data directory:

```
    $ dir USER/ROBERT
    $ dir -x BOOTOBJS
```

The **load** utility opens the file to load with the execute mode (unless the '-d' option is used), so the pathlist is relative to the current execution directory:

```
    $ load BOOTOBJS/h0fmt
    $ load -d OBJS/myprog
```

A child process inherits the current directories from the parent process. It can change them, but this does not affect the current directories of the parent (or of any other process). In the above examples, **dir** and **load** inherited the current directories of the **shell** that forked them.

The **shell**'s current directories may be changed using the built-in commands **chx** and **chd**:

```
    $ chx /h0/CMDS
    $ chd .../PROJECT/SOURCE
```

2.7 INPUT LINE EDITING

Line editing is a function of the I/O subsystem handling the input device - the file manager and device driver - not a function of **shell** (although **mshell** does its own line editing). For terminals, the "Sequential Character File manager" (SCF) is normally used. The same line editing is therefore available for most keyboard line entry.

SCF uses a line buffer for the editing, passing the finished line (when [CR] is pressed) to the calling program. The buffer is 512 bytes, so this is the maximum length of a line typed in, including the [CR]. A separate buffer is allocated for each path opened, so line input on one path does not affect another path.

The editing keys are all customizable (using the **tmode** and **xmode** utilities). Setting a key code to zero suspends the feature. The usual key assignments are as follows ([^X] means hold [CTRL] and press the [X] key):

Key	Action
[BS] or [BkSp]	delete one character left
[^X]	delete the whole line
[^D]	reprint the whole line (useful on teletypes!)

Key	Action
[^A]	redisplay line buffer from cursor to end-of-line character (useful for repeating a command, perhaps with editing)
[ESC]	end-of-file if the first character on the line

2.8 OTHER SPECIAL KEYS

Certain other input keys have special effects. These are a function of the device driver. As with the line editing keys, the utilities **xmode** and **tmode** can be used to change the special keys:

Key	Action
[^E]	abort - kill the last process to use the terminal
[^C]	interrupt - normally kills the last process to use the terminal

[^E] causes the device driver to send a "quit" signal to the last process that used the input device. [^C] causes the device driver to send an "interrupt" signal to the last process that used the input device. As with other signals, if the process has not installed a signal handler function (and most utilities do not), the kernel will terminate the process.

shell does install a signal handler, and so receives and handles the signals. If **shell** was the last process to use the terminal, [^E] causes it to kill the last child forked (by sending it the "quit" signal) and return to the prompt. [^C] causes it to return to the prompt without killing the child, effectively putting the child into the "background". So if the user enters a command line, and the program has done no input or output to the terminal, the user can decide to put the program into the "background" by pressing [^C], as if the command line had been terminated by the '&' character:

```
$ pr fred >/p
  [^C]
$
```

If a process is waiting for an I/O operation to complete when it is killed, the I/O operation is not corrupted - the operating system completes the I/O operation before terminating the process. However, if the I/O operation is a read or write of a terminal or printer (through the **SCF** file manager), the operation is aborted, as **SCF** does not support a filing system that could be corrupted. Note that if a write operation is aborted in this way, any characters that have already been written to the device driver's buffer, but are still waiting to be sent to the device, will be transmitted despite the abort, unless the device was about to be shut down in any case (no paths are open on the device). Therefore, unless special care is taken, it is difficult to know whether such characters will be transmitted after an abort (but this is not usually important).

The third special key causes the device driver to request **SCF** to pause output at the end of the next line - any other key restarts the output:

Key	Action
[^W]	pause output at end-of-line

2.9 MULTIPLE PROCESSES ACCESSING THE TERMINAL

SCF queues concurrent accesses to the terminal, whether for input or output. For example, if, when **shell** is waiting for a line of input, a background process attempts to write to the screen (perhaps to display an error message), **SCF** puts it to sleep. The error message will come out when the **shell**'s request is finished - an input line has been typed - and the background process is woken up by the operating system.

In particular, the **shell** built-in command 'w' causes **shell** to wait for a child process to die. This releases the terminal (because the **shell** is waiting for the process to die, rather than requesting keyboard input), and allows the background process to print its error message and die.

2.10 A TYPICAL DIRECTORY STRUCTURE

OS-9 is very customizable, and has been implemented on a wide range of hardware, so many things will vary from system to system. This includes the directory arrangement on the main disk drive. However, most implementors try to retain the example arrangement from Microware, the main elements of which are described here.

Most OS-9 systems will have a primary mass storage device, either a floppy disk or a hard disk. The first floppy disk drive is usually known as '/d0', while a hard disk is usually known as '/h0'.

An "alias" (in the form of an additional device descriptor) is usually provided for each of the devices, so that each may be known as '/dd' (default device). Only one **dd** device descriptor can be loaded at any one time. Usually this will be the hard disk drive if the system has one, but some configurations use a "RAM disk" as the default device.

Many programs use '/dd' as the route to definitions files and other program-specific data. Therefore you should load a "dd" appropriate to the device you want used for such purposes. For example, a typical command to load the **dd** device descriptor for the hard disk is:

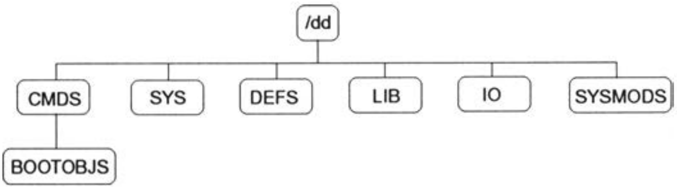
```
    $ load BOOTOBJS/dd.h0
```

Such a load command can be placed in the 'startup' file, although some implementors name '/dd' as the initial device in the **init** module. In such configurations the **dd** module must be in the boot file, or ROM, so to change the default device a new boot file or ROM must be made (or a **dd** device descriptor with a higher revision number can be loaded).

The root directory of this device will usually contain at least the following directories:

CMDS	utilities execution directory
DEFS	header (definitions) source files for assembly language and C programs
LIB	library files (for use by the linker)
SYS	system management text files
SYSMODS	system customization source files
IO	device descriptor (and device driver) source files

and 'CMDS' will also normally contain the directory 'BOOTOBJS'. This directory contains all the OS-9 modules used to make up the operating system, which allows you to create customized boot files, plus any operating system modules not in the boot file, for loading as required.



• **Figure 4 – A typical directory structure**